CARL VON OSSIETZKY universität OLDENBURG

Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Techniques for the Verification of Dynamically Typed Programs

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von
Dipl.-Inform. Björn Engelmann

Gutachter:
Prof. Dr. Ernst-Rüdiger Olderog
Prof. Dr. Frank S. De Boer

Oldenburg, 30. Mai 2016
Verteidigt am 26. Oktober 2016

Acknowledgements

I would like to thank Prof. Dr. Ernst-Rüdiger Olderog for his excellent support and guidance during my time as a doctoral researcher in his group. He patiently listened to even my most digressive lines of thought, helped me in dissecting each and every fallacious idea, supported me both with words and with deeds and was – also due to his immense expertise – an irreplaceable companion along my journey. Like a skilled gardener, he expertly trimmed my proliferating ideas and channeled them into productive directions (some branches that did not bear fruit within the period allotted may be found in Part V of this thesis). I thank Prof. Dr. Frank de Boer, whose work provided the foundations both for my program logic and for its completeness result and who supported my endeavor in various ways. Also, I thank Shuling Wang who – during a research stay at Oldenburg University – introduced me to algebraic specifications as well as Juurrian Rot, Michiel Helvenstein, Robert "Corky" Cartwright, Moshe Y. Vardi, Eike Best, Martin Bodin and Dennis Kregel for critical questions and thought-provoking discussions that guided me into the direction leading to the results at hand, Nils Erik Flick, for numerous pleasant train-rides shared, which in hindsight were much more productive than one would have suspected during the process.

I thank my wife Luyue for her support, compromise and great creativity in dealing with even the most impossible day-to-day situations. On the one hand, it was due to this creativity that substantial parts of this thesis were written on trains or even in cars[1], on the other, I must add that without this creativity, there would not be any thesis at hand.

I also thank my family, which jointly supported my "curiosity-driven world-betterment project". In particular I thank my children Mia and Leo, who, much to my surprise, often have let me work concentrated for hours before storming the study under the battle cry "Feierabend!". Last, but not least I would like to thank my mother, who actively supported us not only in times of illness, but often was the only reason a deadline could be reached despite adverse conditions.

Of course, I am deeply indepted to the DFG-funded Research Training Group "System Correctness under Adverse Conditions" (SCARE) which provided both the organisational framework neccessary for conducting productive research and a set of like-minded peers for fruitful intellectual exchange.

Finally, I would also like to extend my gratitude to the Pianochocolate project for composing an excellent piece of music that accompanied me countless hours on the train and enabled keeping my concentration and serenity even amidst noisy school classes.

[1] while parking, of course.

Zusammenfassung

In dieser Dissertation wird die Art der Verbindung zwischen statischen Typsystemen und Programmverifikation untersucht, oder – andersherum ausgedrückt – es wird der Frage nachgegangen, warum es keine verifizierbaren, dynamisch getypten Programmiersprachen gibt. Der Kern dieser Frage ist kontrovers – die meisten Informatiker assoziieren statische Typsysteme sowohl mit Korrektheit als auch mit Verifikation – wir hingegen werden in dieser Dissertation zeigen, dass die häufig anzutreffende Kombination aus statischem Typsystem und Programverifikation nicht auf Notwendigkeit beruht, sondern lediglich auf Bequemlichkeit.

In diesem Sinne werden wir eine dynamisch getypte Programmiersprache definieren und sie mit einer Programmlogik ausstatten, deren Korrektheit und (relative) Vollständigkeit wir beweisen werden, genau wie es heutzutage auch bei Programmlogiken für statisch getypte Sprachen gängig ist.

Um die Verifikations-Kluft zwischen dynamisch und statisch getypten Sprachen vollends zu überbrücken, werden wir des Weiteren die Unterschiede zwischen der Verifikation von statisch getypten und dynamisch getypten Programmen analysieren, Ursachen, die zu einem erhöhten Verifikationsaufwand in letzterem Fall führen können, identifizieren und einen Ansatz entwickeln, derartige Probleme zu beseitigen. Das Ziel ist es, nur dann zusätzlichen Verifikationsaufwand zu haben, wenn das Programm die dynamische Typisierung sehr stark ausnutzt und den Pfad dessen, was in einer statisch getypten Sprache möglich wäre, deutlich verlässt.

Anders ausgedrückt: Solange das Programm prinzipiell auch statisch getypt sein könnte, sollte es bei der Verifikation keinerlei Unterschied machen, ob man es in einer statisch getypten oder einer dynamisch getypten Sprache geschrieben hat. Je mehr das Programm jedoch davon abweicht, desto mehr müssen Abweichungen erst manuell "repariert" werden (indem man manuell ihre Typ-Sicherheit beweist), bevor die erwähnte Methodik angewendet werden kann.

Zu guter Letzt werden wir die gewonnenen Erkenntnisse nutzen, um eine neuartige Typisierung namens "Consensual Typing" für verifizierbare Programmiersprachen zu definieren, die statische und dynamische Typisierung auf eine Art kombiniert, die es ermöglicht, die Typ-Sicherheit von dynamisch getyptem Programmcode zu garantieren, ohne ihn in statisch getypten umschreiben zu müssen und dabei sowohl Soft- als auch Gradual-Typing generalisiert.

Summary

This thesis investigates the connection between static typing and program verification, or - phrased from the opposite direction - seeks to answer the question "why are there no verifiable, dynamically typed programming languages?". While this question is a controversial one – for many people static type systems are strongly associated with both correctness and verification – we will demonstrate in this thesis that although common, the combination of static typing and program verification is not based on neccessity, but on convenience. To that end, we will define a dynamically typed programming language along with a program logic that we demonstrate to be both sound and (relative) complete, just like its state-of-the-art counterparts for statically typed languages are.

Furthermore, in an attempt to close the gap between the two typing disciplines with respect to verification even further, we analysed the differences in terms of verification between dynamically typed programs and statically typed ones, identified sources for inconvenience in the former case and developed an approach to mitigate these by making the overhead experienced during verification proportional to the program's deviations from what would be allowed in static typing.

In other words: whenever a program written in a dynamically typed programming language is statically typable, there is no difference in terms of verification to the statically typed case. However, when the program deviates from being statically typable, these deviations must first be "repaired" by proving them type-safe, before the methodology from the previous case can be applied.

Finally, we leverage above insights to define the novel typing discipline "consensual typing" for verifiable programming languages, which combines static and dynamic typing in a way that allows deriving type safety guarantees for dynamically typed code without rewriting it into statically typed code and thereby generalizes both soft- and gradual typing.

Contents

I. Introduction **11**

1. Preliminaries **15**
1.1. Notation . . . 15
1.2. Programming Languages . . . 15
1.2.1. Semantics . . . 18
1.2.2. Structural Operational Semantics . . . 18
1.3. Types, Type Systems and Approaches to Typing . . . 20
1.3.1. Types and Type Information . . . 20
1.3.2. Static and Dynamic Typing . . . 22
1.3.3. Pluggable Type Systems, Soft and Gradual Typing . . . 24
1.4. Program Analysis . . . 25
1.4.1. Abstract Interpretation . . . 26
1.4.2. Type Inference . . . 26
1.4.3. Galois Connection . . . 27
1.5. Program Verification . . . 28
1.5.1. Hoare Logic . . . 28
1.5.2. Predicate Transformer Semantics . . . 29
1.5.3. Assertion Language and Decidability . . . 30
1.5.4. Weak Second-Order Logic . . . 31
1.5.5. Weak Second-Order Logic with Arithmetic . . . 33
1.6. Theorem Proving . . . 36
1.6.1. Interactive Theorem Proving . . . 36
1.6.2. Automated Theorem Proving . . . 37
1.7. Algebraic Specifications . . . 37

2. Setting: The Model Languages **39**
2.1. The Model Languages Dyn and Stat . . . 39
2.1.1. Syntax . . . 39
2.1.2. Operational Semantics . . . 40
2.1.3. Type Inference . . . 45

II. Program Logic 55

3. Starting Point: Tagged Hoare Logic for Statically Typed Programs 57
3.1. Assertion Language . . . 57
3.2. Notation: Tagged Hoare Logic . . . 58
3.3. Statically Typed Hoare Logic . . . 61
3.4. Auxiliary Rules . . . 64

4. Tagged Hoare Logic for Dynamically Typed Programs 65
4.1. Shared Type System . . . 66
4.2. Side-Effects . . . 66
4.3. Shortcut Rules . . . 67
4.4. The Transition To Dynamic Typing . . . 67
4.5. Dynamically Typed Hoare Logic . . . 68

5. Proof Theory for Dynamically Typed Programs 71
5.1. Soundness . . . 71
5.2. Completeness . . . 72
5.2.1. Non-Recursive Programs . . . 73
5.2.2. Recursive Programs . . . 73
5.2.3. Object-Oriented Programs . . . 73
5.2.4. Freezing the Initial State . . . 74
5.2.5. Expressivity . . . 76
5.2.6. Completeness for Statements . . . 77
5.2.7. Completeness for Recursive Methods . . . 79

III. Type Information 83

6. Comparison of Statically Typed- with Dynamically Typed Verification 85
6.1. Type Safety . . . 85
6.2. Mapping Objects to Values . . . 85
6.3. Side-effecting Expressions . . . 86
6.4. Specialized Data Types . . . 86

7. Layer of Abstraction 89
7.1. Type Safety Preconditions . . . 89
7.2. Mapping Objects to Values . . . 90
7.3. Pure Expressions . . . 92
7.4. Extending the Assertion Language . . . 94

8. Interactive Type Inference 97
8.1. Example: Evaluator . . . 97
8.2. Interactive Type Inference . . . 99
8.3. Typing Assertions . . . 101

8.4. Galois Connection . . . 102
8.5. Translating Abstract States into Assertions . . . 104
8.6. Translating Typings into Proofs . . . 104
8.7. Trusted Assumptions . . . 105
8.7.1. Conjunctive Refinement . . . 105
8.8. Two-Layered Proofs . . . 107
8.9. Interactive Type Inference . . . 109
8.10. Properties . . . 109
8.10.1. Soundness . . . 109
8.10.2. Completeness Relative to the Program Logic . . . 110
8.10.3. Automation . . . 111
8.11. Consensual Typing: Reconciling Static and Dynamic Typing via Verification . . . 111

9. Related Work **113**

IV. Applications **117**

10. Implementation **121**
10.1. Annotations . . . 121
10.2. Weakest Precondition Calculus . . . 122
10.3. Solving Verification Conditions with Recursive Predicates Using Off-the-Shelf SMT Solvers . . . 124
10.4. Practical Issues with the Rule of Adaptation . . . 127
10.4.1. Simplifying Intermediate Results . . . 127
10.4.2. Detecting Ill-Suited Method Specifications . . . 128
10.4.3. Invariance . . . 130

11. Case Studies **131**
11.1. Dynamic Typing . . . 131
11.2. Coerce Protocol . . . 132
11.3. In-Place Value Switching . . . 136
11.4. Nested Lists as Abstract Syntax Trees . . . 137

12. Extensions **141**
12.1. Non-Fatal Typeerrors . . . 141
12.1.1. Soundness . . . 143
12.1.2. Completeness . . . 144
12.2. Optional Variables . . . 144
12.3. Method Update . . . 147
12.4. The Connection to Higher-Order Programs . . . 148
12.4.1. Discussion . . . 148
12.5. Reflection and Introspection . . . 149

V. Further Avenues of Research **151**

13. Modularity **153**
13.1. Reasoning about Unknown Types . . . 153
13.1.1. Interfaces . . . 155
13.2. Algebraic Properties . . . 156
13.2.1. Algebraic Specifications . . . 157
13.2.2. Fusing Hoare Triples . . . 158
13.2.3. Quantification over computable functions . . . 159
13.3. Higher-Order Behaviour . . . 160

14. Conclusion **161**
14.1. Future Work . . . 163

Bibliography **165**

Symbol Index **171**

Concept Index **173**

VI. Appendices **177**

A. Substitutions **179**

B. Simplification Rules **183**

C. Pure Expressions **185**

D. Unsovable Verification Conditions **187**

Part I.

Introduction

"Where we are going always reflects where we came from"
– *Clone Wars Season 4 Episode 11*

Dynamically typed programming languages do not rely on a static type system to ensure that operations are only applied to suitable operands. Instead, they allow every syntactically correct program to be executed and check the suitability of operands at runtime. Detected type errors hence cause runtime exceptions rather than compile-time errors. There is a controversial discussion about the issue of typing in the programming community. Some consider dynamically typed languages as "too slow", "unsafe" and hence "unfit for any serious programming", while others consider them as superior for writing elegant, reusable and extensible programs. Bracha [16] for example noted that dynamic typing allows a much cleaner conceptual distinction between programming language semantics and program analyses used for bug detection.

However, in safety-critical application scenarios, statically typed languages were predominant and this is most probably the reason why the verification community largely ignored dynamically typed languages. As a result, most program verification methodologies (with the notable exception of ACL2 [15]) nowadays are tailored to statically typed languages.

This situation is ironic for multiple reasons:

- The argument that anyone who seeks to engineer a safety-critical application will most probably do so in a statically typed language assumes that all software is written from scratch and with safety in mind. However, experience from Software Engineering clearly suggests that code written today as a temporary fix in a dynamically typed language might very well end up as a permanent solution in a safety-critical application scenario.

- Contemporary software is increasingly polyglot. Hence, as hybrid languages like Objective-C are put forward and frameworks like the JVM or .NET allow mixing languages with static- and dynamic typing, the probability of dynamically typed code being used as part of safety-critical systems increases.

- Program verification is about correctness, not performance. Hence the performance advantage of static typing is irrelevant in this context.

- Program verification is able to guarantee undecidable program properties. This is in contrast to the type checkers used for static typing which restrict the programming language to make the type safety problem decidable. It does, however, make only little sense to restrict a verifiable programming language for this purpose, since in this case a more powerful mechanism (program verification) is readily available and used for all other properties.

With the goal of showing avenues for consolidation of this rather odd state-of-the-art and closing the gap between the two typing disciplines with respect to verification, this thesis makes the following contributions:

- Analyzing the role of type information in program verification (mostly Hoare logic) with a focus on establishing that it is not required for a sound and (relative) complete application of the methodology, but rather used for various optimizations,

- Demonstrating the previous point by providing a sound and (relative) complete Hoare logic for a dynamically typed programming-language,
- Showing that it is possible to derive type information also for dynamically typed programs,
- Proposing *Interactive Type Inference*, combining a Type Inference Algorithm and a Program Logic into a semi-automatic procedure that makes the effort required for deriving type information of a dynamically typed program proportional to its deviations from what would be allowed in static typing,
- Proposing a methodology using above ideas to make verification of dynamically typed programs equivalent to that of statically typed programs both in terms of effort and convenience, by first establishing their type safety, and
- Proposing *Consensual Typing*, a typing discipline for verifiable programming languages combining the benefits of static- and dynamic typing and thereby generalizing both Soft- and Gradual Typing.

This thesis is organized into two main parts: The First Part titled "Program Logic" is concerned with analyzing the impact of Dynamic Typing on the more theoretical aspects of verification like proof systems and their proof-theoretical properties such as soundness and (relative) completeness. The Second Part titled "Type Information" is concerned with its impact on the more practical aspects of a verification system such as verbosity, the size of verification conditions as well as their solvability.

Afterwards, a third part, titled "Applications" will describe our efforts to implement the concepts mentioned, their application to a number of case studies and how else they can be of use. A Fourth Part, titled "Further Avenues of Research" will elaborate on the limits of the developed formalisms and outline promising directions for future research.

1. Preliminaries

"The Internet is the largest equivalence class in the reflexive, transitive, symmetric closure of the relationship "can be reached by an IP packet from""

– *Seth Breidbart*

1.1. Notation

In this thesis, we denote quantification in a style commonly used for algebraic specifications:

$\forall x : \mathbb{N} \bullet p(x)$ and $\exists x : \mathbb{N} \bullet p(x)$,

abbreviate finite, consecutive subsets of the natural numbers as

$\mathbb{N}_m^n = \{n, ..., m\} \subset \mathbb{N}$, $\mathbb{N}_m = \mathbb{N}_m^0$, and

use the notation 2^M for the powerset of a set M.

Furthermore, we denote sequences as $\overrightarrow{x} = x_1...x_n$, $\overrightarrow{x_1}\overrightarrow{x_2}$ denotes the concatenation of the sequences $\overrightarrow{x_1}$ and $\overrightarrow{x_2}$, and $\{\overrightarrow{x}\}$ is the set of all elements of the sequence $\overrightarrow{x}$. The set of all sequences of elements of a set M is denoted as M^* if it includes the empty sequence and as M^+ otherwise.

1.2. Programming Languages

The main topic of this thesis is program verification, a method for establishing the correctness of programs by means of formal reasoning. Hence, the objects we will be studying are programs and their properties. Since programs are written in programming languages, every formal study of their correctness will have to start with a rigorous account of some programming language. In preparation for this, we will shortly recall some basic notions of programming languages required to follow the arguments in this thesis.

Programming Languages are (usually context-free) formal languages for describing algorithms. The set of syntactically valid programs of a particular programming language (= its syntax) can hence be specified using a context-free grammar. In this thesis, the syntax of all formal languages will be given in Backus-Naur-Form, a common notation for context-free grammars. Such grammars consist of multiple rules like the following

$$N ::= Nt \mid t$$

where N stands for a non-terminal symbol and t stands for a terminal symbol. In order to provide convenient syntax for the programmer, and at the same time simplify the parse-trees for implementation of compilers/interpreters, it is customary to have an additional layer of rewriting rules operate directly on the parse-tree after it was produced from the program text by a parser. These rules are often called *syntactic sugar* and usually remove certain redundant syntax elements by reducing them to others. An example of this would be

$$\textbf{if } e \textbf{ then } S \textbf{ end} \Rightarrow \textbf{if } e \textbf{ then } S \textbf{ else null end}$$

which offers the possibility to omit the else-branch in conditionals by automatically adding one containing the **null** statement.

In this thesis, we will be exclusively discussing imperative, object-oriented programming languages. *Imperative* means that the programming language provides read- and write access to memory in the form of mutable variables and that its programs consist of a sequence of commands, called "statements" that are executed one after the other, each reading those variables and modifying them by assigning new values, overwriting the old.

In order to formalize such imperative programs, we fix a *value domain* $\mathcal{D}$, modeling the values that variables of our program may contain / refer to. Also, we introduce an infinite set of local variables $\mathfrak{V}_L$ that any finite program can draw its finitely many variables from and formalize program states $\sigma \in \Sigma = \mathfrak{V}_L \mapsto \mathcal{D}$ as functions mapping variables to values. Statements can then be interpreted as functions mapping states to states.

However, our programs are also *object-oriented*. This means that there is a special kind of value called "objects". Objects can refer to other values (including objects) through instance variables (sometimes called "fields"), hence forming arbitrary data-structures like linked lists, trees or graphs. In fact, states of object-oriented programs are best viewed as directed graphs whose nodes are the objects and whose edges are the instance variables. To allow setting some of those edges to undefined values, there is usually an object called *null*.

Object-oriented (OO) programming languages in which objects are the only values are called *purely object-oriented*. Note that this is not a restriction as objects are versatile enough to encode any other data-type like numbers, lists or strings.

But object-oriented programs are more than just "programs with objects". They organize the code around the object-structure by providing a notion of a "current object" (often called **self**). The instance variables of **self** are the only ones a statement is permitted to read and assign to, the instance variables of all other objects are hidden (a concept called *encapsulation*).

Also, in object-oriented programs the statements of a program are organized into *methods*, each belonging to a *class*. In a method call $x.\mathrm{m}(x_1, ..., x_n)$, x is passed as an implicit argument called the *receiver* and the object it refers to is made the new current object for the duration of the method call.

Furthermore, each object is assigned to a particular class and said to be its *instance*. Since this relationship is permanent, note that only the methods of a class are per-

mitted to modify the instance variables of its instances. Now each class is given a special method called its *constructor* for creating and initializing new instances of this class. Hence classes manage the entire life-cycle (creation and modification) of their instances.

To be precise, this model is called *class-based object-orientation*. Ruby and Python are both class-based object-oriented languages. JavaScript uses a slightly different model called prototype-based object-orientation. However, in this thesis we will concentrate on the more common class-based variant.

Note that unlike with procedure calls $P(x_1, ..., x_n)$, where the procedure name P must uniquely determine the procedure to be called, a method call $x.\mathrm{m}(x_1, ..., x_n)$ is ambiguous as multiple classes $C_1, C_2, ...$ might implement a method m of arity n. In order to determine which of these is actually called, most languages use *dynamic dispatch*, which means that for all $i \in \mathbb{N}_n$, if the receiver x refers to an instance of class C_i, then C_i's implementation of $m(x_1, ..., x_n)$ is called. Note that while being an instance of a class C could be considered type information (see Section 1.3.1), it is the *runtime type* that is used to disambiguate the call, not some static approximation of it. In object-oriented programs control flow and data flow are therefore interdependent and hence highly precise type information is required to statically reason about their control flow.

To facilitate code reuse, object-oriented programming languages usually provide an inheritance mechanisms. *Inheritance* means that classes may inherit methods (and their implementation) from other classes. When a class A inherits all methods from a class B, it is called a *subclass* of B. Since the canonical use case for this is specialization (Square being a subclass of Rectangle), type systems for object-oriented languages are designed to allow passing instances of subclasses (Square) wherever instances of the parent class (Rectangle) are expected, which is called *subtyping*. A formalization of this principle in the context of program verification is called the *Liskov substitution principle* [53] and the subtyping relation it induces is called *behavioral subtyping*.

To formalize object-oriented programs, we extend above definitions by introducing a set of *classes* $\mathfrak{C}$, a set of *methods* $\mathfrak{M} = \bigcup_{C \in \mathfrak{C}} \mathfrak{M}_C$, where $\mathfrak{M}_C$ denotes the methods of class C, a set of *variables* as $\mathfrak{V} = \mathfrak{V}_L \uplus \mathfrak{V}_I$, where $\mathfrak{V}_L$ denotes the (infinite) set of *local variables* and $\mathfrak{V}_I$ denotes the (also infinite) set of *instance variables*. Then, *program states* $\sigma \in \Sigma$ are tuples $\sigma = (\sigma_l, \sigma_i)$, where σ_l is a function of type $\mathfrak{V}_L \mapsto \mathcal{D}$ mappings local variables to values (which usually include objects $\mathbb{O} \subseteq \mathcal{D}$) and σ_i is a function of type $\mathbb{O} \times \mathfrak{V}_I \mapsto \mathcal{D}$ mapping an object o and an instance variable @v to the value referenced by $o.@v$ (thus modeling the internal states of all objects). However, we overload notation and provide shorthands $\sigma(\mathrm{u}) \equiv \sigma_l(\mathrm{u})$ and $\sigma(o.@\mathrm{v}) = \sigma_i(o, @\mathrm{v})$.

For a given program state σ, the *state update* $\sigma[\mathrm{u} := v]$ is a program state that is identical to σ except that the value of $\mathrm{u} \in \mathfrak{V}_L$ has been updated to $v \in \mathcal{D}$. Similarly, we denote by $\sigma[o.@\mathrm{v} := v]$ a program state that is identical to σ, except that the value of $o.@\mathrm{v}$ for $o \in \mathbb{O}, @\mathrm{v} \in \mathfrak{V}_I$ has been updated to $v \in \mathcal{D}$.

1.2.1. Semantics

Before going into detail about different approaches for establishing the correctness of programs, let me first introduce some basic notions and notations for formalizing the meaning of programs.

While in practice the meaning of most programming languages and hence the behavior of their programs is given in the form of an executable compiler / interpreter for the language, formal reasoning about programs in a particular programming language requires modeling the language mathematically – such a model is called a *semantics* of the programming language.

There are multiple different approaches for defining the semantics of programming languages. We will only list those of relevance to this thesis.

- *Operational Semantics* formally defines a machine-model executing the programs of the language. For instance, giving a Turing Machine executing the programs would constitute an operational semantics. The most popular variant are *structural operational semantics* due to Hennesy and Plotkin [40, 66] that define the behavior in terms of a label-transition system defined inductively over the structure of the programs. Operational semantics are very popular as they are very intuitive and are often even executable [49], which allows for testing them (for instance against a previously implemented interpreter or on example programs). Structural Operational Semantics will be introduced in more detail in Section 1.2.2.

- *Axiomatic Semantics* uses a formal inference system to logically derive program properties. This type of semantics is intended for use in program verification. Unfortunately, it is perceived as unintuitive by many programming language designers who are more familiar with the control-flow-centric view useful for implementing compilers/interpreter than with logical inference systems. Also, lacking direct executability, axiomatic semantics are often hard to debug. It is hence customary in program verification to give both an operational and an axiomatic semantics for a programming language and to establish a close relationship between the two by proving the axiomatic semantics sound and complete relative to the operational semantics. Hoare logic is an example of an axiomatic semantics (see Section 1.5.1).

- *Denotational Semantics* is given by functions mapping programs of the programming language to mathematical objects representing their meaning (called *denotations*). When choosing the right kind of objects as denotations, this style of semantics can be very useful. A well-known example of a denotational semantics is the predicate transformer semantics due to Dijkstra (see Section 1.5.2).

1.2.2. Structural Operational Semantics

Structural Operation Semantics, as introduced by Hennessy and Plotkin [40, 66], describes the behaviour of programs as a transition relation on configurations. Thus,

in this formalism, every execution of a given program corresponds to a path through a directed graph whose nodes are the configurations of a machine and whose edges are the possible computation steps performed by said machine. The path itself hence corresponds to the sequence of configurations and computation steps encountered by the machine when executing the program. Since in this view a program corresponds to a directed graph containing all its possible execution sequences, this modelling style naturally extends to non-deterministic programs.

Often, the *proper configurations* $\langle S, \sigma\rangle$ consist of a program state $\sigma \in \Sigma$ and a subprogram S that is "left to execute". Such configurations can be regarded as tuples from $\mathit{Stmt} \times \Sigma$. Additionally, we define a set of *final configurations* Conf_{final} of the form $\mathbf{final}\langle\sigma\rangle$ for some $\sigma \in \Sigma$ as well as a set of *error configurations* Conf_{error} of the form $\mathbf{fail}\langle S, \sigma\rangle$ or $\mathbf{typeerror}\langle S, \sigma\rangle$ for some $S \in \mathit{Stmt}$ and $\sigma \in \Sigma$. The reason why we give error states the same information as proper states is that we want to include programming languages featuring error recovery into our discussion (see Section 12.1). Hence, the set of *configurations* Conf is defined as

$$\begin{aligned}
\mathit{Conf}_{proper} &\triangleq \mathit{Stmt} \times \Sigma,\\
\mathit{Conf}_{final} &\triangleq \{\mathbf{final}\langle\sigma\rangle \mid \sigma \in \Sigma\},\\
\mathit{Conf}_{error} &\triangleq \{\mathbf{fail}\langle S, \sigma\rangle, \mathbf{typeerror}\langle S, \sigma\rangle \mid S \in \mathit{Stmt}, \sigma \in \Sigma\},\\
\mathit{Conf} &\triangleq \mathit{Conf}_{proper} \uplus \mathit{Conf}_{final} \uplus \mathit{Conf}_{error}.
\end{aligned}$$

We also introduce the abbreviations

$$\begin{aligned}
\mathbf{final}\langle\sigma\rangle &\equiv \langle \mathbf{r}, \sigma\rangle\\
\mathbf{typeerror}\langle\sigma\rangle &\equiv \mathbf{typeerror}\langle \mathbf{r}, \sigma\rangle\\
\mathbf{fail}\langle\sigma\rangle &\equiv \mathbf{fail}\langle \mathbf{r}, \sigma\rangle
\end{aligned}$$

Proper configurations can occur everywhere in finite or infinite computations, while final and error-configurations may only occur as the last configuration of finite computations. As the behaviour of programs is usually compositional, the transition relation can be defined inductively over the structure of said programs.

A *transition* between configurations $\langle S, \sigma\rangle$ and $\langle S', \sigma'\rangle$ is then denoted as

$$\langle S, \sigma\rangle \;\rightarrow\; \langle S', \sigma'\rangle.$$

While each of the styles for formalizing semantics focuses on different aspects of program behaviour (for instance operational semantics retain every single program step, while denotational semantics abstracts those away), in operational semantics it is customary to define the Input/Output semantics of programs on top of the underlying transition system as a function

$$\mathcal{M}[\![S]\!] : \Sigma \mapsto 2^{\Sigma}$$

with

$$\mathcal{M}[\![S]\!](\sigma) = \{\tau \mid \langle S, \sigma\rangle \;\stackrel{*}{\rightarrow}\; \mathbf{final}\langle\tau\rangle\}.$$

where $\mathbf{final}\langle\tau\rangle$ denotes a configuration with state τ and no program left to execute.

Note that this definition is in essence a denotational semantics using functions between (sets of) states as denotations. Also note that this definition does not capture the information whether a program terminates or not. A program S', that does not terminate from the given start state σ is simply mapped to a empty set of final states: $\mathcal{M}[\![S']\!](\sigma) = \emptyset$. Hence, this definition is only useful when termination should not be guaranteed (partial correctness) and must be accompanied by similar definitions for other notions of correctness. The one including termination (total correctness) would be

$$\mathcal{M}_{tot}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) \cup \{\bot \mid S \text{ can diverge from } \sigma\}.$$

1.3. Types, Type Systems and Approaches to Typing

1.3.1. Types and Type Information

A "type" is a set of values sharing a common property. The idea of partitioning the value space $\mathcal{D}$ into types $\tau_1, ..., \tau_n$ with $\mathcal{D}_{\tau_1} \uplus ... \uplus \mathcal{D}_{\tau_n} = \mathcal{D}$ is incredibly old. In fact, its origins can be traced back as far as to the endeveours of the mathematicians Bertrand Russell and Gottlob Frege to formalize all of mathematics. In order to avoid Russell's paradox, an inconsistency discovered in their formalization of set theory, they introduced a type hierarchy and allowed sets of type A to only contain sets of preceding types (lower than A in the hierachy), thus avoiding loops.

Since then, types and type systems have been used in various contexts and for various purposes. In programming, they are mostly used for ruling out certain kinds of bugs and for compile-time performance optimization. However, over time the terms "type" and "type system" have taken on a broader meaning of "value-centic information that can be statically derived using a program analysis and helps preventing bugs".

Since in this thesis we concern ourselves mainly with correctness, the more restricted context of type safety will suffice. More formally, we define:

A program π is called *type-safe* if no execution of π can result in a type error. *Type safety* is the problem of deciding whether a given program is type-safe. Since type errors can be regarded as a form of output, type safety is a nontrivial semantic property and hence undecidable for Turing-complete languages by Rice's theorem [71].

Our model language **dyn** (see Section 2.1) is a pure and class-based object-oriented language. As usual in such languages, "type errors" are defined as calling a method on a receiver that does not support it. We hence need to define our types in such a way that they are useful for detecting such errors. Since the methods supported by an object are those defined in its respective class, classnames are a natural choice for types. Since in dynamically typed languages the inheritance relation is usually of minor importance as subtyping does not neccessarily imply subclassing, we opt for union types instead of including the inheritance relation into our types. Union Types are naturally represented as sets of classnames. For a given program π we thus define:

$$\mathcal{T} = 2^{\mathfrak{C}_\pi \cup \{C_{null}\}}$$

$\mathfrak{C}_\pi$ is the finite set of class names occuring in the program π and the class C_{null} is defined to be the class of the value *null*, which is the usual object-oriented null value. Explicitly representing it in our union types allows for our types to serve the additional purpose of keeping track of *null* values.

For a given program π, we regard a *type* T as an element of the set $\mathcal{T}$. However, since a type can be regarded as an abstraction of a single value, analysing a program π will require to keep track of (the types of) all values in π.
Recall that *concrete states*

$$\Sigma = (\mathfrak{V}_L \mapsto \mathcal{D}) \times (\mathbb{O} \mapsto \mathfrak{V}_I \mapsto \mathcal{D})$$

map local variables ($\mathfrak{V}_L$) to values = domain elements ($\mathcal{D}$) = objects (in the case of **dyn**) as well as objects ($\mathbb{O}$) to their inner states ($\mathfrak{V}_I \mapsto \mathcal{D}$) in turn mapping instance variables ($\mathfrak{V}_I$) to objects ($\mathcal{D}$). When replacing the concrete value domain $\mathcal{D}$ by our type domain $\mathcal{T}$ and explicitly adding the instance variables of the current object (in order to allow our to-be-defined analysis to track them flow-sensitively, see Section 1.4.1) we reach the following definition of *abstract states*:

$$\mathring{\Sigma} = (\mathfrak{V}_L \mapsto \mathcal{T}) \times (\mathfrak{V}_I \mapsto \mathcal{T}) \times (\mathfrak{C} \mapsto \mathfrak{V}_I \mapsto \mathcal{T}).$$

Note that the terminology "abstract state" is in-tune with abstract interpretation, which will be used throughout this thesis to formalize type inference algorithms (see Section 1.4.1).

A *typing ty* of a program π is a mapping from program locations $\mathbf{Loc}_\pi$ to abstract states ($ty : \mathbf{Loc}_\pi \mapsto \mathring{\Sigma}$). Program locations will be formally defined for our programming language in Section 2.1.3. They can be understood as parse-tree nodes of the program π. It is important to stress that two syntactically equivalent statements of π nevertheless have distinct program locations. We use the notation ${}_\circ S$ (resp. $S_\bullet$) to denote the program locations before (resp. after) the evaluation of the statement S.

Definition 1. *A typing ty for a program π is called* sound *iff in every execution of π, whenever a statement S of π is evaluated to a value v, then*

- *v is of a type $T \sqsubseteq ty(S_\bullet)(\mathbf{r})$,*
- *all local variables $u \in \mathfrak{V}_L$ reference values of a type $T_u \sqsubseteq ty(S_\bullet)(u)$,*
- *all instance variables $@v \in \mathfrak{V}_I$ of the current object* **self** *reference values of a type $T_{@v} \sqsubseteq ty(S_\bullet)(\mathbf{self}.@v)$, and*
- *all instance variables $@v \in \mathfrak{V}_I$ of all instances $o \in \mathbb{O}$ of class $C \in \mathfrak{C}$ reference values of type $T_{C.@v} \sqsubseteq ty(S_\bullet)(C, @v)$.*

Definition 2. *A typing ty is at least as* precise *as another typing ty′, written $ty \sqsubseteq ty'$, iff for all program locations $L \in \mathbf{Loc}_\pi$ it holds that $ty(L) \sqsubseteq ty'(L)$.*

Definition 3. *For a program π, the* least precise type-safe typing *$ty^\dagger_\pi$ is a typing where*

- *for every method call* $e_0.m(e_1, ..., e_n)$,

$$ty^\dagger_\pi(e_{0\bullet})(\mathbf{r}) = \{C \mid C \in \mathfrak{C} \textit{ supports method } m \textit{ of arity } n\},$$

- *in the case of* ***stat****, for every operation*[1] $e_1 \oplus e_2$ *of type* $\mathbb{T}_1 \times \mathbb{T}_2 \mapsto \mathbb{T}$,

$$ty^\dagger_\pi(e_{1\bullet})(\mathbf{r}) = \mathbb{T}_1, ty^\dagger_\pi(e_{2\bullet})(\mathbf{r}) = \mathbb{T}_2$$

- *for every conditional or while loop with condition* e,

$$ty^\dagger_\pi(e_\bullet)(\mathbf{r}) = \{bool\}, \textit{ and}$$

- *for all other program locations* $L \in \mathbf{Loc}_\pi$,

$$ty^\dagger_\pi(L) = \top.$$

By definition, a program π *is type-safe iff it has a sound*[2] *typing* ty *that is precise enough to establish type safety (*$ty \sqsubseteq ty^\dagger_\pi$*).*

Type safety verifiers (type inference algorithms) derive a typing for a given program by over-approximating its behavior. A verifier is *sound* iff the typings it derives are.

Note that given a typing ty for a program π, it is straightforward to decide $ty \sqsubseteq ty^\dagger_\pi$. However, deciding soundness usually requires more information. For this reason, sound type safety verifiers usually a) assign non-$\top$ types to all program locations and b) provide a set of inference rules (commonly called a "type system") allowing to check safety of their derived typings using this additional type information. A soundness proof for these rules with respect to the semantics of the programming language is a crucial part of proving such algorithms sound.

1.3.2. Static and Dynamic Typing

A *statically typed* programming language uses a program analysis to determine the types of all values flowing through its programs. Often, the user is additionally required to annotate his/her program with information supporting this analysis. Should this analysis fail – which can be either because the annotations are inconsistent or because the program is not typesafe or due to the inherent limits of program analyses – then the program is rejected and cannot be compiled. In essence, *static typing* is the idea of restricting the valid programs of a programming language to those whose type safety can be guaranteed by a particular program analysis.

Dynamic typing, on the other hand, was introduced much later with the advent of LISP and found its way into many contemporarily popular languages like Ruby, Python and JavaScript. Contrary to static typing, *dynamically typed* languages do not restrict the valid programs in any way, but rather check the validity of operands

[1] In this case, we consider $e_1 == e_2$ an operation of type $\mathbb{O} \times \mathbb{O} \mapsto \mathbb{B}$.

[2] if a method call, conditional or while loop is unreachable, sound typings may assign the type $\bot$ to its receiver / condition.

for each executed operation at runtime. This, of course, introduces a runtime overhead and has the drawback that it is in general not possible to automatically derive type information for such programs, as their typing problem is undecidable.

Controversy: The question which of these two approaches to typing is preferrable is dividing programmers around the globe like few others. As part of the general "which programming language is the best?"-debate, countless battles have been fought between proponents of the two approaches and even reputed members of the scientific community often seem inclined to fuel the heated debate rather than to perpetuate their scientific objectivity. Interestingly, in this matter both sides have claimed their approach to be the more general one and the respective other to be merely a special case. While I can follow this argument for the dynamically typed side (Every statically typed program can be trivially translated into a dynamically typed language, but not vice versa due to the restrictions imposed by static typing), Harper [39] for instance claims that dynamically typed languages would be statically typed languages whose type-system has only one type (he thus calls them "unityped"). To substantiate this, he employs one of the following arguments:

1. Using the over-simplified example of the λ-calculus, where a type-system with recursive types allows to give every λ-term the type $\mu X.X \to X$, which basically states "everything is a function". This example is over-simplified as the λ-calculus has only one type: function. And as function application is the only operation, the λ-calculus can never produce a type error, even without a type system – Obviously the distinction between static and dynamic typing does not make much sense in this setting.

2. Stating that in dynamically typed languages everything is given one giant sum type over all existing types. This in fact is pretty much equivalent to the way type information is stored in dynamically typed languages, but there is a difference: Would one do this in a statically typed language (having sum types), then the type system would allow for using exactly one operation on each data value: a (giant) case distinction. By contrast, in a dynamically typed language one can apply any operation directly to any data value (of course, if the data value does not support the operation, one will get a runtime type error). In my humble opinion, there seems to be a conceptual difference as emulating the behaviour of a dynamically typed language in a statically typed one this way would require inserting a giant case distinction before each and every operation in which all cases but the valid ones would yield a typeerror.

3. Formalizing a translation of dynamically typed programs into a "statically typed" language with typecasts by inserting a typecast to a type that supports the operation before every operand of every operation. The problem here is that although many statically typed languages (C, Java) support typecasts, typecasts actually break static type systems by introducing a (small, controlled) amount of dynamic typing by delaying type checking to runtime. In order to be clear with terminology, it is hence much better to call such languages hybridly typed instead of

"statically typed"[3]. Hence, keeping in mind that hybrid typing is a combination of static- and dynamic typing it is much less surprising that dynamically typed programs can be translated into such a language.

Note that in both translations of dynamically typed programs into a statically or hybridly typed language, all benefits static typing might bring are lost. Both with sum types and with typecasts, the static type system is neither able to establish type safety of the resulting program, nor will it run any faster than in the dynamically typed language, as all runtime typechecks are also performed in the translated version. In contrast, the type information derived as described in Chapter 8 could be used to translate dynamically typed programs into statically typed languages such that the respective type system could actually work its magic.

History: In the first decades of computing, computers where incredibly slow compared to today's standards. Hence performance was considered crucial when writing and compiling programs. Also, RAM memory was costly and hence scarce. Classifying the value space into types of values whose binary representation was to be interpreted in the same way and assigning these types to variables referencing them allowed for a) programs to keep track of the various ways in which each chunk of bits could be interpreted without needing additional memory to store this information. b) compilers to generate code optimized for the type of data it operated on. Furthermore, in low-level languages like C, misinterpreting an integer as a string (whose end was marked by the first `\0`-byte after its start address in memory) or even worse as a memory address to jump to, could have serious consequences like memory corruption or segmentation faults. As these type errors were often very hard to reproduce and debug, avoiding them at the language-level was perceived as of utmost importance. For these reasons, all major programming languages since then have been statically typed.

This is probably the reason why most prior research in program verification concentrated on statically typed languages.

Note that most of the issues that led to the creation of statically typed languages are getting less and less serious as both computational power and memory are becoming cheaper and cheaper. Also, modern high-level languages usually handle memory-management automatically, thereby significantly reducing the impact of type errors.

1.3.3. Pluggable Type Systems, Soft and Gradual Typing

Since the advent of LISP and maybe motivated by the heated dabate in the programming community, many researchers have explored ways to combine static- and dynamic typing in beneficial ways. We already discussed typecasts in the last section. There are also other forms of hybrid typing, for instance by using an explicit type **Dynamic** [1].

Another approach is *Soft Typing* [18], the idea of using a type inference algorithm on a dynamically typed program to find as much type information as possible and then insert run-time checks only where the type information was insufficient to guarantee type safety.

[3] For this very reason there are advocates of static typing (esp. the Haskell community) that resent the idea of typecasts altogether.

Gradual Typing [75, 9] also starts from a dynamically typed program, but adds a static type system and lets the user gradually add type information to it – those values annotated are typed statically, the remaining values dynamically. Both approaches can of course also be combined [69].

According to Bracha [16], the combination of a dynamically typed language with (multiple) optional type systems (like in Soft or Gradual Typing) is conceptually cleaner due to the clear distinction between programming language semantics and program analyses. He calls this concept "Pluggable Type Systems" and argues that it is preferrable to statically typed ones as it "can provide most of the advantages without most of the drawbacks".

Note, however, that static type systems provide a type-safety guarantee for the entire program while both soft- and gradual typing exclude those regions of code that are too hard to analyse. In contrast, *consensual typing* as explained in Section 8.11 allows for such guarantees to be provided for entire dynamically typed programs as long as they are actually typesafe.

1.4. Program Analysis

The Theorem of Rice [71] states that all non-trivial semantic properties of programs written in a Turing-complete programming language are undecidable. Since these days, all approaches developed by the correctness research community can be categorized by which of two paths were used to circumvent this problem: Those aproaches that chose to accept approximate answers are nowadays known as *program analysis* and those that chose to include the user into the process in order to still gain precise results are today called *program verification*.

This section is concerned with the former. The latter, which is the main topic of this thesis, will be given a detailed account in Section 1.5.

In this thesis, program analysis will be applied to establish type-safety. Type-safety is a safety property and expresses the absence of a certain kind of unwanted behaviour (type errors). This kind of property can be soundly approximated using over-approximation, since the absence of the unwanted behaviour in a (safe) over-approximation of the program behaviour implies its absence in the actual program behaviour.

Traditionally, the theoretical background for program analysis can be either phrased in terms of data-flow analysis, constraint-based analysis or abstract interpretation. For a thorough introduction to all of these as well as their various similarities and differences, we refer the interested reader to [61].

For the formal developments in this thesis, the notion of a Galois Connection from Abstract Interpretation turned out to be particularily intuitive and hence this style was chosen also for the remaining material related to program analysis.

1.4.1. Abstract Interpretation

Abstract Interpretation is the theoretical basis underlying most semantics-based program analyses. It was introduced by Cousot and Cousot [23] to provide a semantic foundation for program analysis. Essentially, one replaces the concrete program by an abstract model which can be feasibly analysed, a process which is called "abstraction". While the program semantics (see Section 1.2.2) describes the program statement S as a transformer on (concrete) states $\sigma_1, \sigma_2 \in \Sigma$

$$\langle S, \sigma_1 \rangle \quad \stackrel{*}{\rightarrow} \quad \mathbf{final}\langle \sigma_2 \rangle,$$

abstract interpretation would model the same statement S in terms of abstract states $\mathring{\sigma}_1, \mathring{\sigma}_2 \in \mathring{\Sigma}$:

$$S \vdash \mathring{\sigma}_1 \rhd \mathring{\sigma}_2$$

In order to ensure soundness of the resulting analysis, care has to be taken that the abstraction safely approximates the program. We hence introduce a *correctness relation* $R \subseteq \Sigma \times \mathring{\Sigma}$ such that

$$\sigma_1 R \mathring{\sigma}_1 \wedge \langle S, \sigma_1 \rangle \quad \stackrel{*}{\rightarrow} \quad \mathbf{final}\langle \sigma_2 \rangle \wedge S \vdash \mathring{\sigma}_1 \rhd \mathring{\sigma}_2 \Rightarrow \sigma_2 R \mathring{\sigma}_2$$

holds for all statements S as well as all $\sigma_1, \sigma_2 \in \Sigma, \mathring{\sigma}_1, \mathring{\sigma}_2 \in \mathring{\Sigma}$.

1.4.2. Type Inference

Type Inference is a program analysis with the goal of deriving sound and precise type information for a given program. We already defined a notion of types suitable for such an analyses in Section 1.3.1.

To meet the requirements of abstract interpretation, we extend the previously defined set $\mathcal{T}$ of types to a *complete lattice* $\mathfrak{L}_{\mathcal{T}} \stackrel{\Delta}{=} (\mathcal{T}, \subseteq, \top, \bot, \cup, \cap)$ with

- $\mathcal{T}$ as our *Abstract Domain* of *Union Types* (represented as sets of class names),
- The usual subset relation $\subseteq$ as *lattice pre-order* ($\sqsubseteq$),
- $\mathcal{T} \ni \top = \mathfrak{C} \cup \{C_{null}\}$ as largest lattice element,
- $\mathcal{T} \ni \bot = \{\}$ as smallest lattice element,
- $\sqcup = \cup$ (set-union) and $\sqcap = \cap$ (set-intersection) as *join* and *meet* operations respectively.

As usual, we extend the lattice operations on our type domain pointwise for all $\mathring{\sigma} \in \mathring{\Sigma}$

- $\mathring{\sigma} \sqsubseteq \mathring{\sigma}' \Leftrightarrow \forall \mathrm{x} \in \mathfrak{V}_L \bullet \mathring{\sigma}(\mathrm{x}) \subseteq \mathring{\sigma}'(\mathrm{x}) \wedge \forall @\mathrm{v} \in \mathfrak{V}_I \bullet \mathring{\sigma}(\mathbf{self}.@\mathrm{v}) \subseteq \mathring{\sigma}'(\mathbf{self}.@\mathrm{v}) \wedge \forall C \in \mathfrak{C}, @\mathrm{v} \in \mathfrak{V}_I \bullet \mathring{\sigma}(C, @\mathrm{v}) \subseteq \mathring{\sigma}'(C, @\mathrm{v})$.

- $(\mathring{\sigma} \sqcup \mathring{\sigma}')(X) \triangleq \mathring{\sigma}(X) \cup \mathring{\sigma}'(X)$ for all $X \in \mathfrak{V}_L \cup \{\textbf{self}.@\text{v} \mid @\text{v} \in \mathfrak{V}_I\} \cup \{(C, @\text{v}) \mid C \in \mathfrak{C}, @\text{v} \in \mathfrak{V}_I\}$

- $(\mathring{\sigma} \sqcap \mathring{\sigma}')(X) \triangleq \mathring{\sigma}(X) \cap \mathring{\sigma}'(X)$ for all $X \in \mathfrak{V}_L \cup \{\textbf{self}.@\text{v} \mid @\text{v} \in \mathfrak{V}_I\} \cup \{(C, @\text{v}) \mid C \in \mathfrak{C}, @\text{v} \in \mathfrak{V}_I\}$

1.4.3. Galois Connection

In Abstract Interpretation, the notion of a Galois Connection is used to connect different Abstract Domains and mathematically capture the notion of "abstraction".

Definition 4. *A (monotone)* Galois Connection *connecting two Abstract Domains* $(A, \sqsubseteq_A)$ *and* $(B, \sqsubseteq_B)$ *is a quadruple* (A, α, γ, B) *that, in addition to the domains, contains a monotone abstraction function* $\alpha : A \mapsto B$ *and a monotone concretization function* $\gamma : B \mapsto A$*, such that*

$$\forall a \in A, b \in B \bullet a \sqsubseteq_A \gamma(\alpha(a)) \wedge \alpha(\gamma(b)) \sqsubseteq_B b.$$

Definition 5. *A* Galois Insertion *is a Galois Connection with* $\forall b \in B.b = \alpha(\gamma(b))$.

In Abstract Interpretation, Galois Insertions are used to connect Abstract Domains with the Concrete Domain of program states $(2^\Sigma, \subseteq)$.

For our type inference above, we hence define a Galois Insertion $(2^\Sigma, \alpha_{TI}, \gamma_{TI}, \mathring{\Sigma})$ with

$\alpha_{TI} : 2^\Sigma \mapsto \mathring{\Sigma}$ defined by $\alpha_{TI}(\sigma)(\text{x}) \triangleq \{C\}$ iff $\sigma(\text{x})$ is an instance of class C for all $\text{x} \in \mathfrak{V}_L, C \in \mathfrak{C}$ and $\alpha_{TI}(\sigma)(o.@\text{v}) \triangleq \{C\}$ iff $\sigma(o.@\text{v})$ is an instance of class C for all $o \in \mathbb{O}, @\text{v} \in \mathfrak{V}_I, C \in \mathfrak{C}$.

$\alpha_{TI}(\{\sigma_1, ..., \sigma_n\}) \triangleq \alpha_{TI}(\sigma_1) \sqcup ... \sqcup \alpha_{TI}(\sigma_n)$ and

$\gamma_{TI} : \mathring{\Sigma} \mapsto 2^\Sigma$ is (uniquely) defined by $\gamma_{TI}(\mathring{\sigma}) \triangleq \{\sigma \mid \alpha_{TI}(\sigma) \sqsubseteq \mathring{\sigma}\}$

Lemma 1 (Galois Insertion for Type Abstraction)**.** $(2^\Sigma, \alpha_{TI}, \gamma_{TI}, \mathring{\Sigma})$ *is a Galois Insertion between the concrete domain* $(2^\Sigma, \subseteq)$ *and the abstract domain* $(\mathring{\Sigma}, \sqsubseteq)$.

Proof. 1. $\forall S \subseteq \Sigma \bullet S \subseteq \gamma_{TI}(\alpha_{TI}(S))$. Let $\sigma \in S$, then there is some abstract state $\mathring{\sigma} = \alpha_{TI}(\sigma)$ with $\mathring{\sigma} \sqsubseteq \alpha_{TI}(S)$, since it is one of its components. Hence, since $\alpha_{TI}(\sigma) \sqsubseteq \alpha_{TI}(S)$, we have $\sigma \in \gamma_{TI}(\alpha_{TI}(S))$ by definition of γ_{TI}.

2. $\forall \mathring{\sigma} \in \mathring{\Sigma} \bullet \mathring{\sigma} = \alpha_{TI}(\gamma_{TI}(\mathring{\sigma}))$. Since a concrete state may assign an object of arbitrary (singleton) type to each variable it is easy to construct a state σ exactly matching an abstract state that only assigns singleton types to variables. Since furthermore $\mathring{\sigma}$ can be decomposed into a join of finitely many abstract states $\mathring{\sigma} = \mathring{\sigma}_1 \sqcup ... \sqcup \mathring{\sigma}_n$ such that each $\mathring{\sigma}_i$ only assigns singleton types, there surely is a set of concrete states S such that $S = \gamma_{TI}(\mathring{\sigma})$ and $\mathring{\sigma} = \alpha_{TI}(S)$. □

1.5. Program Verification

As stated before, *program verification* refers to all non-automatic approaches to establish program correctness (the fact that a program satisfies a specification). These approaches are subsumed by the more general term *formal methods* as they require the user to establish correctness in some formal system amenable to mechanized correctness checking. Formal methods hence acknowledge that the problem of program correctness can only be solved by humans, but seek to mitigate the error rate inevitable in human manual work by letting a computer check the solution. It is this checking that necessitates establishing the correctness argument in a formal system amenable to automatic proof-checking – hence the name "formal methods".

While a large number of approaches to program verification have been proposed, we will in this thesis concentrate on Hoare logic (Section 1.5.1), the predominant approach for verifying imperative programs. Hoare logic has close ties with Weakest Precondition Calculus (WPC) (Section 1.5.2), another well-known approach for imperative programs initiated by Dijkstra. For instance, both for implementation purposes (see Chapter 10) and in completeness arguments (see Section 5.2), it is customary to use a WPC derived from it instead of the Hoare logic itself. As both approaches are parametric in their assertion language – a logic used to express assertions about program states, we will also introduce Weak Second-Order logic (Section 1.5.4) as well as its extension with Arithmetic (Section 1.5.5), which our assertion language **AL** (see Section 3.1) is based on.

1.5.1. Hoare Logic

Hoare logic was introduced by C.A.R. (Tony) Hoare [41] to logically reason about partial correctness of sequential, non-recursive while programs. It is an example of an axiomatic semantics and its basic statements are called *Hoare Triples*

$$\{p\}S\{q\}$$

for a program statement S and two assertions p and q called the precondition (p) and the postcondition (q) of S. *Validity* of such Hoare Triples is defined as

$$\models \{p\}S\{q\} \quad \text{iff} \quad \mathcal{M}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!],$$

where $\mathcal{M}[\![S]\!]$ denotes the *input-output-semantics* of the statement S (a function mapping (sets of) initial states $\sigma \in \Sigma$ to sets of final states) and $[\![p]\!] \subseteq \Sigma$ denotes the semantics of the assertion p, that is a set of states, such that $\sigma \models p$ holds for each $\sigma \in [\![p]\!]$. Intuitively, $\{p\}S\{q\}$ interpreted in the sense of partial correctness means that if the statement S is executed in a state satisfying p and terminates, then the result state will satisfy q.

Formally, a *Hoare logic* is a Tuple (H, A) with H being a proof system for reasoning about program correctness by deriving the Hoare triples mentioned above and A being a (complete) proof system for the assertion language used. An *Assertion Language* is a logic interpreted over program states. Hence program states $\sigma \in \Sigma$ are the models

of its logical formulas and as explained above, the semantics of a formula (called an *assertion*) p is a set of program states denoted as $[\![p]\!]$.

Hoare logic is a well-studied formalism and was extended multiple times. Contemporary Hoare logics exist for a much wider variety of program classes (recursive programs [33], object-oriented programs [6], parallel programs [64], nondeterministic programs [48], distributed programs [7], etc.) as well as for different notions of correctness (total correctness, strong partial correctness, etc. [6]) given as separate proof systems.

Also, theoretical studies have shown Hoare logic to not only be sound, but also complete relative to the assertion language used [22, 33, 14], a property that distinguishes it from many other approaches to program verification.

1.5.2. Predicate Transformer Semantics

Another approach to program verification strongly related to Hoare logic is Dijkstra's Predicate Transformer Semantics [26], with its concrete instantiations Weakest Precondition- and Strongest Postcondition Calculus. Both can be seen as strategies for applying the proof rules of Hoare logic that allow for reducing Hoare Triples for non-cyclic statements to implications of the Assertion Language. Both variants can be understood as mapping programs to functions between assertions that are interpreted as state-sets, and hence as a form of denotational semantics.

The weakest precondition $WP(S, q)$ of a statement S relative to the postcondition q is an assertion p satisfying the Hoare Triple $\{p\}S\{q\}$, such that for all assertions p' having this property, $p' \to p$ holds. $WP(S, q)$ can hence be understood as the minimal requirement for S to terminate in a state satisfying q. From this definition, it immediately follows that $\{p\}S\{q\}$ is valid iff $p \to WP(S, q)$ holds.

Similarly, the strongest postcondition is the maximal guarantee than can be given about a final state after executing the statement and the Strongest Postcondition Calculus exhibits a similar relationship with Hoare logic:

$$\models \{p\}S\{q\} \quad \text{iff} \quad SP(p, S) \to q.$$

As these relationships allow not only for switching between the formalisms, but also to derive them from each other, we take the standpoint that they are merely different ways to formalize the same idea and hence exchangeable for one another. With this, we consider ourselves in good company, as it is for instance customary to switch to a Predicate Transformer Semantics within completeness proofs for Hoare logics [22, 33, 14].

In this sense, Predicate Transformer Semantics can be regarded as the operational[4] side of Hoare logic. Hoare logic, on the other hand, stands in the tradition of logical inference systems and is focused on the derivation of properties, which for instance

[4] Here, "operational" does not refer to the operation of an abstract machine executing the program like in operational semantics. Rather, it refers to the operation of a verification tool checking the program. In this respect, WPC and SPC are preferable because they remove any nondeterminism inherent in inference systems like Hoare logic and thus provide complete strategies for reducing Hoare triples to formulas of the Assertion Language, which is very valuable for implementing verification tools.

allows its REC rule to handle recursive method calls elegantly. In a Weakest Precondition Calculus, it has long been a problem how to calculate the weakest precondition of a method call in a way that is complete for recursive programs (which requires to take applications of the adaptation-rules (CONS, INV, CONJ, DISJ, etc.) into account) until Hoare proposed the Rule of Adaptation [42], which solves this problem:

Definition 6 (Rule of Adaptation for WPC)**.** *Given a specification*

$$\{p'\}P(v_1, ..., v_n)\{q'\}$$

for a procedure P of arity n, the Weakest Precondition of a call to the procedure $S \equiv P(v_1, ..., v_n)$ with respect to a postcondition q is given by

$WP(P(v_1, ..., v_n), q) \triangleq \exists \overline{c} \bullet (p' \wedge \forall \overline{v} \bullet q' \rightarrow q)$ *where $\overline{v} = v_1, ..., v_n$ is a list of all variables free in S (the formal parameters of the call) and $\overline{c}$ is a list of all variables free in p' or q', but not in S.*

In Section 10.2, we will translate our previously developed Hoare logic for **dyn** into the Weakest Precondition Calculus (using the Rule of Adaptation) that we used as a basis for implementing a verification tool for **dyn**. However, in Section 10.4 we will elaborate on a number of practical issues with the Rule of Adaptation that we experienced when using the calculus to verify our case studies.

1.5.3. Assertion Language and Decidability

Recall that the problem of program verification that Hoare logic and Predicate Transformer Semantics are attempting to solve is undecidable by the Theorem of Rice [71]. Since both formalisms are sound and complete, they obviously cannot be decidable. Also, since both are parameterized with an Assertion Language, note that we are not discussing a single formalism, but a class of formalisms in this respect.

1. With an Assertion Language whose satisfiablity is decidable (like propositional logic), applying a Weakest Precondition Calculus would allow us to reduce Hoare Triples for non-cyclic programs (i.e. without loops and without recursion) to implications of this assertion language, which could in all cases be decided. However,
   - non-cyclic programs are not Turing-complete as they all terminate for all inputs,
   - only propositional properties could be verified using this approach, and
   - a propositional assertion language is not expressive (not able to express the weakest preconditions of all statements relative to all postconditions) and hence the Hoare logic would not be complete relative to such an assertion language as there would be no guarantee that all intermediate assertions as well as all loop invariants and method contracts necessary could be expressed in the assertion language.

2. Applying above approach to cyclic programs (with loops or with recursion) will encounter the problem that in Hoare logic the user has to supply the loop invariants (LOOP rule) and the method contracts (REC rule). Hence a Weakest Precondition Calculus can only calculate a weakest precondition for a loop given its loop invariant and can only calculate one for a method call given the method's contract. Automatically deriving loop invariants or method contracts is an undecidable problem on its own as for instance a loop invariant that implies the loop condition ensures that the loop will not terminate and hence solves the halting problem for the respective program.

3. Using an expressive assertion language would guarantee that all intermediate assertions and all loop invariants can be expressed in it and hence would allow for the Hoare logic to be complete relative to the assertion language. However, note that

   - loop invariants and method contracts must still be supplied by the user (see above), and
   - an assertion language that is expressive for a Turing-complete programming language must also be able to express all recursively enumerable sets (or all μ-recursive functions) and hence is also able to express the Halting Problem. For such a logic, satisfiablity is surely undecidable.

We follow [6] (and most of the verification community) in choosing expressiveness over automation (the latter option). Similar to [14], our Assertion Language is hence based on Weak Second-Order Logic with Arithmetic (Section 1.5.5).

1.5.4. Weak Second-Order Logic

First-Order predicate calculus (also called "First-Order logic" or FOL) extends propositional logic with quantification over elements of a (countable) domain of discourse $\mathcal{D}$ (called *individuals*). Second-Order predicate calculus (also called "Second-Order logic" or SOL) extends FOL by allowing quantification over elements of uncountable domains, like (possibly infinite) sequences of (countable) individuals. Weak second-order predicate calculus (also called "Weak second-order logic" or WSOL) lies between these two: it allows quantification over finite sequences of (countable) individuals. Since it excludes the case of infinite sequences, it is stricly less expressive than second-order logic. Also, since it allows to quantify over sequences, it is strictly more expressive than first-order logic. However, this relationship becomes more complicated when Arithmetic comes into play, as will be discussed in the next section.

Weak Second-Order Logic is not a single formal system, but (just like FOL and SOL) can be instantiated with different signatures and interpretations. A *signature* is a triple $\mathfrak{S} \triangleq (Pred, Func, ari)$ with $Pred$ being a set of predicate symbols, $Func$ being a set of function symbols and $ari : Pred \cup Func \mapsto \mathbb{N}$ being a function assigning an arity to each predicate and each function symbol.

Given such a signature, the syntax of Weak Second-Order Logic can be defined as

$$f ::= \exists v_i.f \mid f \wedge f \mid \neg f \mid t = t \mid P(t_1, ..., t_n)$$
$$t ::= f(t_1, ..., t_n) \mid v_i$$

where $v_i \in \mathfrak{V}_{WSO}$ denotes a variable from the (countably) infinite set $\mathfrak{V}_{WSO} = \{v_1, v_2, ...\}$, $P \in \mathit{Pred}$ denotes a predicate of arity n and $f \in \mathit{Func}$ denotes a function of arity n.

There are numerous common abbreviations:

$$\forall v_i.f \equiv \neg\exists v_i.\neg f, f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2), f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2,$$
$$f_1 \leftrightarrow f_2 \equiv f_1 \rightarrow f_2 \wedge f_2 \rightarrow f_1, \mathit{true} \equiv v_i = v_i, \mathit{false} \equiv \neg \mathit{true}.$$

An *interpretation* of a Weak Second-Order Logic with signature $\mathfrak{S}$ is a triple $\mathfrak{I} \triangleq (\mathcal{D}, i_{\mathit{Pred}}, i_{\mathit{Func}})$, where $\mathcal{D}$ is a (countable) domain of discourse, i_{Pred} is a function mapping each predicate symbol $P \in \mathit{Pred}$ to a function $\mathcal{D}^{ari(P)} \mapsto \mathbb{B}$, and i_{Func} is a function mapping each function symbol $f \in \mathit{Func}$ to a function $\mathcal{D}^{ari(f)} \mapsto \mathcal{D}$.

Definition 7. *A* model *m of a formula f is a mapping from $\mathfrak{V}$ to $\mathcal{D}$ such that $m \models f$ holds. Terms are evaluated in a model as follows:*

- $m(v_i) = d \in \mathcal{D}$
- $m(f(t_1, ..., t_n)) = i_{\mathit{Func}}(f)(m(t_1), ..., m(t_n))$

The satisfaction relation $\models$ *for formulas is then defined as follows:*

- $m \models t_1 = t_2$ *iff* $m(t_1) = m(t_2)$
- $m \models P(t_1, ..., t_n)$ *iff* $i_{\mathit{Pred}}(P)(m(t_1), ..., m(t_n))$
- $m \models \neg f$ *iff* $m \not\models f$
- $m \models f_1 \wedge f_2$ *iff* $m \models f_1 \wedge m \models f_2$
- $m \models \exists v_i.f$ *iff* $\exists d \in \mathcal{D}.m[v_i := d] \models f$

A variable v_i occurs *bound* in a formula f iff $\exists v_i.f'$ for some f' is a subformula of f. A variable v_i occurs *free* in a formula f iff it occurs in f, but does not occur bound in f. We define $\mathit{free}(f) = \{v_i \mid v_i \text{ occurs free in } f\}$.

Definition 8. *A formula f is called* closed *or a* sentence *iff* $\mathit{free}(f) = \emptyset$.

Definition 9. *The* existential closure *of a formula f with* $\mathit{free}(f) = \{v_{i_1}, ..., v_{i_n}\}$ *is the sentence* $\exists v_{i_1} ... \exists v_{i_n}.f$.

Definition 10. *The* universal closure *of a formula f with* $\mathit{free}(f) = \{v_{i_1}, ..., v_{i_n}\}$ *is the sentence* $\forall v_{i_1} ... \forall v_{i_n}.f$.

Definition 11. *We call a formula f* satisfiable *iff there is a model m such that $m \models f$.*

Definition 12. *We call a formula f* valid *iff $m \models f$ holds in all models m.*

Observe the following:

- For sentences, satisfiability and validity are equivalent.
- A formula f is satisfiable iff its existential closure is.
- A formula f is valid iff its universal closure is.
- A formula f is valid iff $\neg f$ is unsatisfiable.

1.5.5. Weak Second-Order Logic with Arithmetic

Our Assertion Language **AL** (see Section 3.1) is based on Weak Second-Order Logic with Arithmetic ($WSOL_{arith}$), that is Weak Second-Order Logic as described in the last section with the signature $(\{<\}, \{0, 1, +, -, *, div, mod\}, ari)$ in its standard interpretation (with $\mathcal{D} = \mathbb{N}$). ari assigns the arity 0 to the function (constant) symbols $0, 1$ and the arity 2 to all other function/predicate symbols. Note that $WSOL_{arith}$ includes FOL_{arith}, which is sometimes also refered to as "Number Theory".

While being the minimal logic that is expressive enough for our programming language and hence allows our Hoare logic to be (relative) complete, its expressiveness already causes a number of drawbacks:

1. Gödels incompleteness theorem [32, Theorem VI] states that there cannot be a complete and consistent proof system for Number Theory (or any formal system including it), since a technique developed by him ("Gödelization" – explained in the next section) can be used for expressing a sentence R (called the Gödel sentence), stating "This sentence is unprovable in the proof system P", in Number Theory. Hence, for any sound and complete proof system P, if P could prove R, then soundness of P implies that R must be true – which contradicts the assumption. If, on the other hand, P can prove $\neg R$, then soundness of P implies that $\neg R$ is true, which states that R must be provable in P. Hence, both R and $\neg R$ would be provable in P and P would hence be inconsistent.

2. Using Gödelization, it is also possible to express μ-recursive functions (Section 1.5.5), which are extensively used in the (relative) completeness proof for our Hoare logic. However, since μ-recursive functions are Turing-complete, they allow for expressing the Halting problem in Number Theory. Hence, satisfiability of Number Theory (and any logic containing it, like **AL**) must be undecidable. Fortunately, there are nevertheless theorem provers (see Section 1.6) and even automatic ones (see Section 1.6.2) able to decide satisfiability of large classes of formulas by reducing them to decidable subsets of the logic.

Gödelization

The technique called Gödelization consists of two basic ideas. First, the following formula establishes a bijection between natural numbers and finite sequences of natural numbers:

$$gödel(n_1...n_k, n) \equiv n = 2^{n_1} * 3^{n_2} * ... * p_k^{n_k}$$

where p_k is the kth prime number. Also note that this formula is expressible in Number Theory.

The second idea is that most object sets in discrete mathematics (Strings, Trees, Graphs, Computable Functions, Programs, Formulas, Proofs, etc.) are countable and can hence be mapped into the natural numbers using a bijection (derivable using the above formula).

While this technique was developed by Gödel to encode formulas and formal proofs into natural numbers and then express their derivability (= provability in a formal proof system) as a formula (resulting in the Gödel sentence – see last section), it is also useful for encoding programs and program states into natural numbers and thus for simulating their execution using μ-recursive functions (next section).

In general, it is convenient to introduce the notation $|n|$ to denote the length of the sequence encoded in the natural number n as well as $n[k]$ to denote the kth element in the sequence encoded in the natural number n.

Encoding μ-Recursive Functions

Notation: In this section, we sometimes use $p\frac{v}{t}$ to denote the result of substituting a term t for a variable v in a formula p. We will also be using the notation for Gödelization introduced in the last section.

μ-recursive functions are the recursion-theoretical equivalent to Turing Machines. Like *primitive recursive functions*, they are of type $\mathbb{N}^k \mapsto \mathbb{N}$, but contrary to them, they are partial and hence are not necessarily defined for all arguments.

Definition 13. *The* μ-recursive functions *are the smallest class of partial functions such that*

- ***Constants:*** *for every* $n, k \in \mathbb{N}$, $f(x_1, ..., x_n) = k$ *is a* μ*-recursive function.*
- ***Successor:*** $f(x) = x + 1$ *is a* μ*-recursive function.*
- ***Projections:*** *for every* $i, k \in \mathbb{N}$ *with* $1 \leq i \leq k$, $f(x_1, ..., x_k) = x_i$ *is a* μ*-recursive function.*
- ***Composition:*** *for every* m*-ary* μ*-recursive function* $h(x_1, ..., x_m)$ *and every* m k*-ary* μ*-recursive functions* $g_1(x_1, ..., x_k), ..., g_m(x_1, ..., x_k)$,

  $$f(x_1, ..., x_k) = h(g_1(x_1, ..., x_k), ..., g_m(x_1, ..., x_k))$$

  is also a μ*-recursive function.*

- ***Primitive Recursion:*** *for every μ-recursive functions $g(x_1, ..., x_n)$ and $h(x, x', x_1, ..., x_n)$,*

$$f(i, x_1, ..., x_n) = \begin{cases} g(x_n, ...x_n) & \text{if } i = 0 \\ h(i-1, f(i-1, x_1, ..., x_n), x_1, ..., x_n) & \text{otherwise} \end{cases}$$

is also a μ-recursive function.

- ***Minimization:*** *for every total μ-recursive function $g(i, x_1, ..., x_n)$, the function f such that $f(x_1, ..., x_n) = z$ iff $g(z, x_1, ..., x_n) = 0$ and $g(z', x_1, ..., x_n) > 0$ for all $z' \in \mathbb{N}^0_{z-1}$ is also a (partial) μ-recursive function.*

.

Lemma 2. *For every k-ary, μ-recursive function f, there exists a formula p in $WSOL_{arith}$ with free variables $r, x_1, ..., x_k$, such that*

$$f(a_1, ..., a_k) = z \text{ iff } \models p\frac{r, x_1, ..., x_k}{z, a_1, ..., a_k}$$

Proof. By induction over the structure of μ-recursive functions.

- If f is a constant function $f(x_1, ..., x_k) = n$, then the formula $p \equiv r = n$ satisfies the Lemma.
- If f is the successor function $f(x_1) = x_1 + 1$, then the formula $p \equiv r = x_1 + 1$ satisfied the Lemma.
- If f is the projection $f(x_1, ..., x_n) = x_i$, then the formula $p \equiv r = x_i$ satisfies the Lemma.
- If f is a composition of a k-ary function h and k n-ary functions $g_1, ..., g_k$, then by the induction hypothesis, there are formulas $p_h, p_{g_1}, ..., p_{g_k}$ corresponding to the functions $h, g_1, ..., g_k$ as described in the Lemma. Then,

$$p \equiv \exists v_1, ..., v_k : \mathbb{N} \bullet p_h\frac{x_1, ..., x_k}{v_1, ..., v_k} \wedge p_{g_1}\frac{r}{v_1} \wedge ... \wedge p_{g_k}\frac{r}{v_k}$$

  satisfies the Lemma.

- If f is a primitive recursion with a n-ary function g and a $n+2$-ary function h, then by the induction hypothesis, there are formulas p_g and p_h corresponding to the functions g and h as described in the Lemma. Then,

$$p \equiv \exists s : \mathbb{N} \bullet |s| = x_1 \wedge r = s[x_1] \wedge p_g\frac{r, x_1, ..., x_n}{s[0], x_2, ..., x_{n+1}}) \wedge$$
$$\forall i : \mathbb{N} \bullet 0 \leq i < x_1 \rightarrow p_h\frac{r, x_1, ..., x_{n+2}}{s[i+1], i, s[i], x_1, ..., x_n}$$

  satisfies the Lemma.

- If f is the minimization of a $n+1$-ary function g, then according to the induction hypothesis, there is a formula p_g corresponding to g as described in the Lemma. Then,

  $$p \equiv \exists v : \mathbb{N} \bullet p_g \frac{r, x_1, ..., x_{n+1}}{0, v, x_1, ...x_n} \wedge \forall v' : \mathbb{N} \bullet v' < v \rightarrow \exists v_r : \mathbb{N} \bullet p_g \frac{r, x_1, ..., x_{n+1}}{v_r, v', x_1, ..., x_n} \wedge v_r > 0$$

  satisfies the Lemma.

□

Theorem 1. *Every μ-recursive function is expressible in $WSOL_{arith}$.*

Proof. Direct consequence of Lemma 2. □

Theorem 2. *Every μ-recursive function is expressible in **AL**.*

Proof. **AL** subsumes $WSOL_{arith}$ (see Section 3.1). Hence Theorem 2 follows directly from Theorem 1. □

1.6. Theorem Proving

Theorem proving is an area of theoretical computer science concerned with the construction of algorithms for the automatic (or at least machine-checked) construction of formal proofs for mathematical theorems (usually expressed using a logic like FOL, WSOL or HOL). The underlying problem is closely related to program verification (Predicate Transformer Semantics can be seen as a way to reduce program verification to theorem proving) and is hence also undecidable (see Section 1.5.5) for many expressive logics.

Just like with program verification, approaches developed by the theorem proving community can be categorized into those that favor automation over completeness and the converse flavor. Those approaches that accept incompleteness in favor of automation are called *automatic theorem proving* or *proof searching* and will be detailed in Section 1.6.2. Those requiring manual effort are usually able to prove more theorems, are refered to as *interactive theorem proving* or *proof checking* and will be detailed in Section 1.6.1.

As our goal is to close the gap between static- and dynamically typed programming languages in terms of verification technology, we follow the popular paradigm of auto-active program verification [51, 80] in combining user-specified loop conditions/method contracts with automatic theorem proving.

1.6.1. Interactive Theorem Proving

Interactive theorem provers or *proof checkers* like Isabelle [62] provide convenient environments for humans to conduct formal proofs in. All statements are automatically scrutinized by a model checker in order to detect typos [12] and even some degree of automation is provided using automated theorem provers [11]. The main objective of

proof checkers however, is – as the name states – , to rule out the possibility of human error by mechanized proof checking. Such tools often start with a minimal core of axioms and trust only what can be derived from them. Also, since the tool does not need to find proofs itself, undecidability is not an issue in this setting and hence extremely expressive logics like Higher-Order Logic (HOL) may be used and inductive arguments may be derived, which is usually well beyond the capabilities of automated theorem provers [58].

1.6.2. Automated Theorem Proving

Automated Theorem Provers, *proof searchers* or *Satisfiability Modulo Theories(SMT)-Solvers* are tools that attempt to decide satisfiability of formulas given in an expressive logic like $WSOL_{arith}$ by employing decision procedures for various fragments of said logic, extending their reach with translations and reductions and combining them using heuristics.

Usually based on a resolution-based solver for Propositional Logic [25], these tools started by extending SAT-solvers with decision procedures for decidable theories like Real Arithmetic, Linear Arithmetic, Presburger Arithmetic (Arithmetic without Multiplication), etc., but have matured to also solve first-order formulas as they often support quantifier elimination procedures (even for quantification over Arrays, Bitvectors, Regular Expressions, etc.).

A state-of-the-art example of such an SMT-solver is Z3 [10], developed by Microsoft Research. It received the 2015 ACM SIGPLAN Software System award [3]. We follow the verification tools for statically typed programming languages Boogie [8] and Dafny [2] in using Z3 as our theorem-proving backend.

1.7. Algebraic Specifications

Algebraic Specifications were first introduced in the context of functional programming, are related to *algebraic data types* and are useful for specifying program properties in terms of relationships between different functions/methods rather than on the level of the input-output behaviour of a single function/method [53].

Given an algebraic data type, that is

- a set of constructors, creating values of the data type and
- a set of operations, operating on values of the data type,

an algebraic specification is a set of (conditional) equations between expressions using the constructors and operations as functions. An example of such a specification for lists is the following:

```
sort list {
  new(e:elem) -> list
  cons(l:list,e:elem) -> list
  size(l:list) -> integer
```

```
  get(l:list,index:integer) -> elem
}
spec(list) { forall e:elem,l:list,i:int
  size(new(e)) = 1,
  size(cons(l,e)) = size(l) + 1,
  get(new(e),0) = e,
  get(cons(l,e),size(l)) = e,
  get(cons(l,e),i) = get(l,i) if i < size(a),
  get(l,i) = null if i < 0
}
```

Guttag and Horowitz [36] extended algebraic specifications to side-effecting procedures/methods by logically splitting the methods up into several side-effect-free functions returning different aspects of the original side-effect, i.e, in our example, one could illustrate this by adding a side-effecting operation `add(l:list,e:elem) -> integer` that returns the index of the newly added element, but has the side-effect of adding it to the end of the list.

```
spec(list) {
  add(l,e)_ret = size(l),
  add(l,e)_arg1 = cons(l,e)
  get(add(a,e)_arg1,add(a,e)_ret) = e
}
```

As one can see, the side-effecting operation `add(l,e)` is logically split up into two side-effect-free functions `add(l,e)_ret`, which returns the index of the added element and `add(l,e)_arg1`, which returns the altered list.

Shuling Wang et al. [74] proposed an approach using this idea for applying Algebraic Specifications to Object-Oriented Programs and verifying them using Separation Logic [70]. However, as their methodology crucially relies on the separating conjunction ($*$), it is not directly applicable in the context of Hoare logic.

2. Setting: The Model Languages

"[Speaking about Algol 60] Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors."

– *C.A.R. Hoare*

2.1. The Model Languages Dyn and Stat

To explain our methodology in a setting facilitating formal proof, we introduce a pair of minimalistic model programming languages that differ only in the fact that one is dynamically typed (**dyn**) while the other uses a static type system with type inference (**stat**). In order to make **dyn** resemble its real-world siblings Ruby, Python and JavaScript, the two are imperative, class-based (purely) object-oriented languages including inheritance, method renaming, dynamic dispatch and constructors. However, to focus our inquiry on dynamic typing, we will for now not model other features commonly found in these languages like method update, closures or eval(). See Sections 12.3 and 12.4 for information on how to extend our formalism with method update and closures, respectively.

2.1.1. Syntax

The syntax of both **dyn** and **stat** is depicted in Figures 2.1 and 2.2. In **dyn**, method bodies consist of statements (S) which in contrast to expressions (e) can contain sequential composition. Expressions are composed of the only constant **null**, local- and instance variables (the latter prefixed by @), the self-reference **self**, operators for object identity and dynamic type checks, method- and constructor calls, assignments, conditionals and while loops.

As customary in dynamically typed languages, **dyn** desugars operations to method calls. The only built-in operation in **dyn** is object identity (==), everything else is defined in the language itself (see Figure 2.1). Note that this has the unfortunate consequence that the convenient distinction between side-effect-free expressions and side-effecting statements as used in statically typed Hoare logic does not uphold. Not only can operations and method calls not be distinguished syntactically, but the type information necessary to tell whether or not a method call is side-effecting is also not available statically.

Of course, there are also side-effecting expressions in statically typed languages. C and Java for instance support operations `x++` and `++x`, that are clearly side-effecting. However, in the literature on statically typed Hoare logic, it is assumed that all expressions are side-effect-free. Note that this is not a limitation as it is always possible

to transform an expression e containing side-effecting subexpressions $e_0, ..., e_n$ into a sequence of statements $\mathrm{u}_0 := e_0; ...; \mathrm{u}_n := e_n; e[e_1, ..., e_n := \mathrm{u}_0, ..., \mathrm{u}_n]$ where the u_i are fresh local variables for all $i \in \mathbb{N}_n$, by "prepending" the side-effects. Applying this prepending-transformation recursively and in a way that preserves the order of evaluation, it is possible to transform every program containing side-effecting expressions into an equivalent one where all expressions are side-effect-free.

Of course, this prepending-transformation is also applicable to dynamically typed programs. However, this would require syntactically defining a subset of side-effect-free expressions. For **dyn**, the best one could do in this respect would be

$$e_\varepsilon ::= \textbf{null} \mid \mathrm{u} \mid @\mathrm{x} \mid \textbf{self} \mid e_\varepsilon == e_\varepsilon \mid e_\varepsilon \text{ is\_a? } \mathrm{C}$$

Since such extremely restricted expressions are hardly useful, this would have the effect that in practice (nearly) all expressions occurring in a program would need to be prepended. Noticing that this a) voids all benefits one could hope to gain from using shortcut rules (see Chapter 4), and b) can instead also be built directly into the rules, we opted to instead consider all expressions as side-effecting and do the prepending logically, thus removing the necessity for this intermediate step in the verification process.

For this reason, **dyn** expressions may contain method calls and assignments. For example, a := b := 5 is a valid **dyn** expression with the side-effect of assigning 5 to both a and b.

Note that equality (=) is desugared to a (class-specific) method call, while object identity (==) is a build-in operation yielding *true* iff the two expressions refer to the same object (We stipulate null == null yields *true*).

Furthermore, each class except the predefined class *object* must specify a parent class whose methods are inherited. The inheritance relation must be acyclic. Every class thus transitively inherits from *object*. Inherited methods may be overwritten or renamed (using *rename*). Like in other dynamically typed languages, inheritance is mere code reuse and can be removed using an automatic expansion step [65]. Furthermore, we will assume this step to be completed and not concern ourselves any further with inheritance or renaming.

Since types in **dyn** are a property of values rather than variables, there is no need to declare the latter. However, as all local variables are initialized to *null* at the beginning of a method and all instance variables are initialized to *null* on object creation, no uninitialized variable access can occur in **dyn**. For information on how to extend the formalism to also prevent uninitialized variable access, see Section 12.2.

The only reasons for type errors are hence non-boolean conditions in conditionals or while-loops and method call receivers whose class does not support a method matching name and arity of the call (MethodNotFound).

2.1.2. Operational Semantics

Both **dyn** and **stat** programs consist of a main statement S and sets of classes $\mathfrak{C}$ with their respective methods $\mathfrak{M}_C$ for $C \in \mathfrak{C}$. $\mathfrak{V}_S = \{\textbf{self}, \mathbf{r}\} \subset \mathfrak{V}_L$ is a set of special

Syntax of **dyn**:

$Prog_d \ni \pi ::= \overrightarrow{class}\ S$
$Class_d \ni class ::= \textbf{class}\ \mathrm{C} < \mathrm{C}\ \{\overrightarrow{meth}\}$
$Meth_d \ni meth ::= \textbf{method}\ \mathrm{m}(\overrightarrow{\mathrm{u}})\{S\}$
$\quad \mid \textbf{rename}\ \mathrm{m}\ \mathrm{m}$
$Stmt_d \ni S ::= S; S \mid e$
$Expr_d \ni e ::= \textbf{null} \mid \mathrm{u} \mid @\mathrm{v} \mid \textbf{self} \mid e == e$
$\quad \mid e\ \text{is\_a?}\ \mathrm{C} \mid e.\mathrm{m}(\overrightarrow{e}) \mid \textbf{new}\ \mathrm{C}(\overrightarrow{e})$
$\quad \mid \mathrm{u} := e \mid \textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end}$
$\quad \mid @\mathrm{v} := e \mid \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}$
$(\mathrm{u} \in \mathfrak{V}_L, @\mathrm{v} \in \mathfrak{V}_I, \mathrm{C} \in \mathfrak{C}, \mathrm{m} \in \mathfrak{M})$

Syntactic sugar:

$e_1 \oplus e_2 \equiv e_1.\mathrm{m}_{\oplus}(e_2)$
$\textbf{if}\ e\ \textbf{then}\ S\ \textbf{end} \equiv$
$\textbf{if}\ e\ \textbf{then}\ S\ \textbf{else null end}$
$false \equiv \textbf{new}\ \mathrm{bool}(\textbf{null}) \qquad [\,] \equiv \textbf{new}\ \mathrm{list}()$
$true \equiv false.\mathrm{not}() \qquad [..., o] \equiv [...].\mathrm{add(o)}$
$0 \equiv \textbf{new}\ \mathrm{num}(\textbf{null})$
$n \equiv (n-1).\mathrm{succ}()$ for $n \in \mathbb{N}$
"" $\equiv \textbf{new}\ \mathrm{string}(\textbf{null}, \textbf{null})$,
"...a" $\equiv$ "...".addchar(n_a) where $n_\mathrm{a} \in \mathbb{N}$ is the ASCII-code of character a.

Figure 2.1.: Syntax of **dyn**

Syntax of **stat**:

$Prog_s \ni \pi ::= \overrightarrow{class}\ S$
$Class_s \ni class ::= \textbf{class}\ \mathrm{C} < \mathrm{C}\ \{\overrightarrow{meth}\}$
$Meth_s \ni meth ::= \textbf{method}\ \mathrm{m}(\overrightarrow{\mathrm{u}})\{S\}$
$\quad \mid \textbf{rename}\ \mathrm{m}\ \mathrm{m}$
$Stmt_s \ni S ::= S; S \mid \mathrm{u} := e \mid @\mathrm{v} := e$
$\quad \mid \mathrm{u} := e.\mathrm{m}(\overrightarrow{e}) \mid \mathrm{u} := \textbf{new}\ \mathrm{C}(\overrightarrow{e})$
$\quad \mid \textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end}$
$\quad \mid \textbf{while}\ e\ \textbf{do}\ S\ \textbf{done}$
$Expr_s \ni e ::= \textbf{null} \mid \mathrm{u} \mid @\mathrm{v} \mid \textbf{self} \mid e == e$
$\quad \mid op(\overrightarrow{e}) \mid cnst \mid e\ is\_a?\ C$
$\quad \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e\ \textbf{end}$
$(\mathrm{u} \in \mathfrak{V}_L, @\mathrm{v} \in \mathfrak{V}_I, \mathrm{C} \in \mathfrak{C}, \mathrm{m} \in \mathfrak{M})$

Operations $\in op$:

$+, -, *, div, mod : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$,
$<, >, =: \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B}$,
$\wedge, \vee : \mathbb{B} \times \mathbb{B} \mapsto \mathbb{B}, \neg : \mathbb{B} \mapsto \mathbb{B}$, and
$[\![\ .\ ]\!] \in \{\overrightarrow{C}\} : \mathbb{O} \mapsto \mathbb{B}$

Constants $\in cnst$:

$1, 2, 3, ... : \mathbb{N}$,
$true, false : \mathbb{B}$

Figure 2.2.: Syntax of **stat**

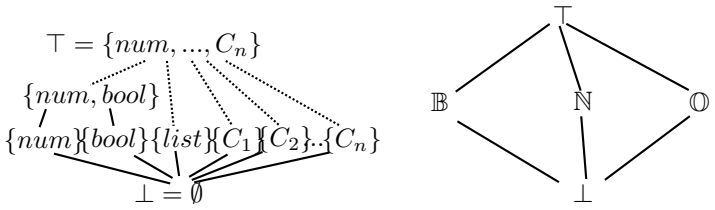


Figure 2.3.: Type lattices of **dyn** (left) and **stat**/**AL** (right)

variables. While **self** references the current object and is not allowed to be assigned to in programs, **r** holds the result of the last evaluated expression and cannot be used in programs.

dyn's value domain $\mathcal{D}_d$ is the set of all objects and for a given program π, its type system is the lattice of union types represented as sets of class names $\{\mathrm{C}_1, ..., \mathrm{C}_n\} \in \mathcal{T}_d = 2^{\mathfrak{C}_\pi \cup \{C_{null}\}}$ with the subset-ordering $\subseteq$ (see Figure 2.3). **stat** on the contrary distinguishes basic data types $\mathcal{T}_s = \{\mathbb{O}, \mathbb{N}, \mathbb{B}, ...\}$ and its value domain $\mathcal{D}_s \cong \biguplus_{\tau \in \mathcal{T}_s} \mathcal{D}_\tau$ includes objects, numbers and booleans. To keep track of instance-class relationships we use class references and for every class $\mathrm{C} \in \mathfrak{C}$ introduce a distinct object θ_C as well as a special instance variable **@c** such that $o.@\mathbf{c} = \theta_\mathrm{C}$ iff o is an instance of class C. Using **@c** in programs is not permitted.

Comparing Dyn with Stat: Dyn is a pure object-oriented language (objects are the only values) while **stat** has basic data types besides objects. However, both provide the same constants and pure (i.e. side-effect-free) operations on them. Dyn desugars them to constructor and method calls (see Figure 2.1), while **stat** (like usual in statically typed languages) provides them build-in (c_ε and $op(\overrightarrow{e_\varepsilon})$ in Figure 2.2).

Also, **stat** expressions are pure. Side-effects are only allowed in statements, which must only have pure subexpressions. This is not a limitation, as explained in the last section.

Every **stat** program is also a **dyn** program that evaluates to (an object-oriented version of) the same result. The only reason that the opposite direction does not hold is the language restriction imposed by **stat**'s static type system.

In Figure 2.4, we define a structural operational semantics of **dyn**. Usually, in a structural operational semantics, expressions are assumed to be side-effect-free and the effect of assignments can hence be expressed as an axiom

$$\langle \mathrm{u} := e, \sigma \rangle \;\rightarrow\; \mathbf{final}\langle \sigma[\mathrm{u} := \sigma(e)] \rangle.$$

As explained in the last section, **dyn** expressions are side-effecting. We hence need to evaluate the assignment $\mathrm{u} := e$ in two steps: first evaluating the expression e and then assigning its resulting value to the variable u. Furthermore, we need an interface between these two steps: A way by which the assignment can determine the result of the previously evaluated expression e. For this purpose, we introduce a special variable **r** of type $\mathbb{O}$ as well as the convention that every expression or statement will

store its result in **r**. Note that this construction works only due to dynamic typing: In a statically typed programming language, expressions would evaluate to values of different types which could not well be assigned to a single variable. The choice of object as the unifying supertype of all values is common in pure OO-languages: When everything is an object, clearly every expression will evaluate to one. Furthermore, as **r** is the only statement that does not change anything (not even **r**), we define the empty program as **r**, stipulate $(\mathbf{r}; S) \equiv (S; \mathbf{r}) \equiv S$ for all statements S and call the configurations $\langle \mathbf{r}, \sigma \rangle \equiv \mathbf{final}\langle\sigma\rangle$ for some state σ final.

For **dyn**, we use class-based OO and model object creation as activation[1]. We introduce a "representative" object θ_C for each class C as well as a special instance variable @**c** not allowed to occur in programs for maintaining both the instance-class relation and the activation state of each object.

We call an object o with $o.@\mathbf{c} = \mathit{null}$ *inactive*, meaning it is "not yet created". Initially, all objects (except *null* and the representatives θ_C for each class C) are inactive. We suppose an infinite enumeration of objects $o_1, o_2, ...$ containing every object (both active and inactive) exactly once and introduce a function $\gamma : \Sigma \mapsto \mathbb{O}$ mapping every state $\sigma \in \Sigma$ to the object o_k with the least index k that is inactive in σ.

Upon its creation, an object o is assigned a class C and is henceforth regarded an instance of C. Technically, this is achieved by resetting the value of $o.@\mathbf{c}$ to θ_C (see the rule for object creation). We use init_C to denote the initial (internal) state of an object of class C: $\mathit{init}_C.@\mathbf{c} = \theta_C$ and $\mathit{init}_C.@v = \mathit{null}$ for all $@v \in \mathfrak{V}_I \setminus \{@\mathbf{c}\}$.

We can then formally define the predicates $\mathit{bool}(o)$ and $\mathit{bool}(o, b)$ used in Figure 2.4 to check for boolean values as

$\mathit{bool}(o) \equiv o.@\mathbf{c} = \theta_{\mathit{bool}}$ for all $o \in \mathbb{O}$ and
$\mathit{bool}(o, b) \equiv \mathit{bool}(o) \wedge b \leftrightarrow o.@\mathit{to\_ref} \neq \mathit{null}$[2] for all $o \in \mathbb{O}, b \in \mathbb{B}$.

[1] Assuming an infinite sequence of already existing, but deactivated objects, object creation boils down to picking the next one and marking it as "activated".
[2] Other methods to distinguish the values true and false are conceivable.

1. $\langle \mathbf{null}, \sigma\rangle \;\to\; \mathbf{final}\langle\sigma[\mathbf{r} := \mathit{null}]\rangle$
2. $\langle \mathrm{v}, \sigma\rangle \;\to\; \mathbf{final}\langle\sigma[\mathbf{r} := \sigma(\mathrm{v})]\rangle$ where $\mathrm{v} \in \mathfrak{V}$
3. $$\frac{\langle e, \sigma\rangle \;\overset{*}{\to}\; \mathbf{final}\langle\tau\rangle}{\langle \mathrm{v} := e, \sigma\rangle \;\to\; \mathbf{final}\langle\tau[\mathrm{v} := \tau(\mathbf{r})]\rangle}$$ where $\mathrm{v} \in \mathfrak{V}, T \in \{\mathbf{typeerror}, \mathbf{fail}\}$

   3b. $$\frac{\langle e, \sigma\rangle \;\overset{*}{\to}\; T\langle\tau\rangle}{\langle \mathrm{v} := e, \sigma\rangle \;\to\; T\langle\tau\rangle}$$
4. $$\frac{\langle S_1, \sigma\rangle \;\to\; \langle S_2, \tau\rangle}{\langle S_1; S, \sigma\rangle \;\to\; \langle S_2; S, \tau\rangle}$$ where $T \in \{\mathbf{typeerror}, \mathbf{fail}\}$

   4b. $$\frac{\langle S_1, \sigma\rangle \;\overset{*}{\to}\; T\langle S_2, \tau\rangle}{\langle S_1; S, \sigma\rangle \;\to\; T\langle S_2; S, \tau\rangle}$$
5. $$\frac{\langle e, \sigma\rangle \;\overset{*}{\to}\; \mathbf{final}\langle\tau\rangle,\ \mathit{bool}(\tau(\mathbf{r}), \mathrm{true})}{\langle S, \sigma\rangle \;\to\; \langle S_1, \tau\rangle} \qquad \frac{\langle e, \sigma\rangle \;\overset{*}{\to}\; \mathbf{final}\langle\tau\rangle,\ \mathit{bool}(\tau(\mathbf{r}), \mathrm{false})}{\langle S, \sigma\rangle \;\to\; \langle S_2, \tau\rangle}$$
   where $S \equiv \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}$
6. $$\frac{\langle e, \sigma\rangle \;\overset{*}{\to}\; \mathbf{final}\langle\tau\rangle \qquad \tau(\mathbf{r}) = \mathit{null}}{\langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}, \sigma\rangle \;\to\; \mathbf{fail}\langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}, \tau\rangle}$$
7. $$\frac{\langle e, \sigma\rangle \;\overset{*}{\to}\; \mathbf{final}\langle\tau\rangle \qquad \neg \mathit{bool}(\tau(\mathbf{r}))}{\langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}, \sigma\rangle \;\to\; \mathbf{typeerror}\langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}, \tau\rangle}$$
8. $$\frac{\langle e, \sigma\rangle \;\overset{*}{\to}\; T\langle\tau\rangle}{\langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}, \sigma\rangle \;\to\; T\langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}, \tau\rangle}$$
   where $T \in \{\mathbf{typeerror}, \mathbf{fail}\}$.
9. $\langle S', \sigma\rangle \;\to\; \langle \mathbf{if}\ e\ \mathbf{then}\ S; S'\ \mathbf{else\ null\ end}, \sigma\rangle$ where $S' \equiv \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}$
10. $\langle \overrightarrow{\mathrm{u}} := \overrightarrow{\mathrm{v}}, \sigma\rangle \;\to\; \mathbf{final}\langle\sigma[\overrightarrow{u} := \sigma(\overrightarrow{v})]\rangle$ where $\overrightarrow{u}, \overrightarrow{v} \in \mathfrak{V}_L^+$
11. $$\frac{\langle S, \sigma[\overrightarrow{\mathrm{u}}\,\overrightarrow{\mathrm{u}} := \overrightarrow{\mathrm{v}}\,\overrightarrow{\mathit{null}}]\rangle \;\overset{*}{\to}\; \mathbf{final}\langle\tau\rangle}{\langle \mathbf{begin\ local}\ \overrightarrow{\mathrm{u}} := \overrightarrow{\mathrm{v}}; S\ \mathbf{end}, \sigma\rangle \;\to\; \mathbf{final}\langle\tau[\overrightarrow{\mathrm{u}}\,\overrightarrow{\mathrm{u}} := \sigma(\overrightarrow{\mathrm{u}}\,\overrightarrow{\mathrm{u}})]\rangle}$$
    where $\{\overrightarrow{u}\} = ((\mathit{var}(S) \cup \mathit{change}(S)) \cap \mathfrak{V}_L) \setminus (\{\overrightarrow{u}\} \cup \mathfrak{V}_S)$ and $\overrightarrow{\mathit{null}}$ is a sequence of *null* values of fitting length.
12. $$\frac{\langle S, \sigma[\overrightarrow{\mathrm{u}}\,\overrightarrow{\mathrm{u}} := \overrightarrow{\mathrm{v}}\,\overrightarrow{\mathit{null}}]\rangle \;\overset{*}{\to}\; T\langle\tau\rangle}{\langle \mathbf{begin\ local}\ \overrightarrow{\mathrm{u}} := \overrightarrow{\mathrm{v}}; S\ \mathbf{end}, \sigma\rangle \;\to\; T\langle \mathbf{begin\ local}\ \overrightarrow{\mathrm{u}} := \overrightarrow{\mathrm{v}}; S\ \mathbf{end}, \tau[\overrightarrow{\mathrm{u}}\,\overrightarrow{\mathrm{u}} := \sigma(\overrightarrow{\mathrm{u}}\,\overrightarrow{\mathrm{u}})]\rangle}$$
    with $T \in \{\mathbf{typeerror}, \mathbf{fail}\}$, $\overrightarrow{u}$ and $\overrightarrow{\mathit{null}}$ as in the previous rule.
13. $$\frac{\langle e_i, \sigma_i\rangle \;\overset{*}{\to}\; \mathbf{final}\langle\sigma_{i+1}\rangle \text{ for all } i \in \mathbb{N}_n}{\langle e_0.m(e_1, ..., e_n), \sigma_0\rangle \;\to\; \langle S', \sigma_{n+1}\rangle}$$
    where $\sigma_1(\mathbf{r}) \neq \mathit{null}$, $\mathbf{method}\ m(u_1, ..., u_n)\{S\} \in \mathfrak{M}_C$ and $\sigma_1(\mathbf{r}.@\mathbf{c}) = \theta_C$, $S' \equiv \mathbf{begin\ local\ self}, \overrightarrow{u} := \sigma_1(\mathbf{r}), ..., \sigma_{n+1}(\mathbf{r}); S\ \mathbf{end}$.

Figure 2.4.: **dyn**'s structural operational semantics (Continued on page 46).

Note how the rule for assignment uses the two-step idea to handle side-effecting expressions. The rules for conditionals and while loops also use it to first evaluate the condition and then branch on its result. Since no type system guarantees this result to be boolean, further distinguished behaviors for failures and type errors are necessary. The same holds for receivers of method calls.

Following [5], method calls are handled using a block construct (**begin local**-blocks) instead of introducing an explicit call-stack into the program states. Note how all local variables are reset to their previous values when the block ends. This reset is also done when a failure or typeerror occured within the block, as otherwise the state resulting from such an failed method call would leak the values of method-internal local variables to the caller[3].

The rules for method call (or better: **begin local**-blocks) and object creation instantiate all local- and instance variables to *null*.

Note also the handling of special variables in method calls: on entry, **self** is set to the receiver of the method call while on exit **r** intentionally remains unmodified to pass the return value back to the caller.

Constructors are normal methods conventionally named *init* that are called on newly created instances directly after they were created. The instance creation (activation) itself is called $\mathbf{new}_C$. Note that **new** $C(...)$ returns the constructor's return value which is not necessarily the newly created instance. Also note that calling **new** $C'(...)$ for a class C' that does not have a method *init* results in a **typeerror**.

2.1.3. Type Inference

Contrary to **stat**, which rejects programs deemed unsafe at compile time, **dyn** allows every syntactically correct program to be executed and raises type errors at runtime when

- a method call is not supported by its receiver (in this arity) or
- a condition of a conditional or while loop is not boolean.

While "message not understood"-errors are fundamentally linked to type-checking in class-based OO-languages, dynamically typed languages often allow conditions to be of arbitrary type. Nevertheless, the second error condition models a common error class where a built-in operation supports a fixed set of types.

Many dynamically typed languages raise type errors when accessing variables prior to assignment. See Section 12.2 for information on how our system can be extended to handle this kind of type errors as well. For now we will consider all local (instance) variables to be initialized to *null* prior to method executions (on instantiation). Also, type errors are often treated as exceptions, allowing for interception and handling. For simplicity, we will for now consider them as fatal and will discuss an extension with non-fatal type errors later in Section 12.1.

[3] While this would not cause any problems in this version of the semantics, the extension discussed in Section 12.1 would provide the means to exploit such an issue.

14. $$\frac{\langle e_0, \sigma_0\rangle \xrightarrow{*} \textbf{final}\langle\sigma_1\rangle \qquad \sigma_1(\mathbf{r}) = null}{\langle e_0.m(e_1,...,e_n), \sigma_0\rangle \rightarrow \textbf{fail}\langle e_0.m(e_1,...,e_n), \sigma_1\rangle}$$

15. $$\frac{\langle e_0, \sigma_0\rangle \xrightarrow{*} \textbf{final}\langle\sigma_1\rangle \qquad \sigma_1(\mathbf{r}) \neq null \quad \sigma_1(\mathbf{r}.@\mathbf{c}) = \theta_C \qquad \nexists \textbf{method}\ m(u_1,...,u_n)\{S\} \in \mathfrak{M}_C}{\langle e_0.m(e_1,...,e_n), \sigma_0\rangle \rightarrow \textbf{typeerror}\langle e_0.m(e_1,...,e_n), \sigma_1\rangle}$$

16. $\langle \textbf{new}\ \mathrm{C}(e_1,...,e_n), \sigma\rangle \rightarrow \langle \textbf{new}_C.\mathrm{init}(e_1,...,e_n), \sigma\rangle$

17. $\langle \textbf{new}_C, \sigma\rangle \rightarrow \textbf{final}\langle\sigma[o := init_C][\mathbf{r} := o]\rangle$ where $o = \gamma(\sigma)$

18. $$\frac{\langle e_i, \sigma_i\rangle \xrightarrow{*} \textbf{final}\langle\sigma_{i+1}\rangle \text{ for all } i \in \{0,1\} \quad \exists b : \mathbb{B} \bullet b \leftrightarrow \sigma_1(\mathbf{r}) = \sigma_2(\mathbf{r}) \wedge B}{\langle e_0 == e_1, \sigma_0\rangle \rightarrow \langle S_r, \sigma_2\rangle}$$
where $B \equiv (S_r \equiv true \wedge b \vee S_r \equiv false \wedge \neg b)$.

19. $$\frac{\langle e, \sigma\rangle \xrightarrow{*} \textbf{final}\langle\sigma_1\rangle,\ \exists b : \mathbb{B} \bullet b \leftrightarrow \sigma_1(\mathbf{r}.@\mathbf{c}) = \theta_C \wedge B}{\langle e\ is\_a?\ C, \sigma\rangle \rightarrow \langle S_r, \sigma_1\rangle}$$
where B is defined as in the previous rule.

20. $\textbf{fail}\langle S, \sigma\rangle \rightarrow \textbf{fail}\langle\sigma\rangle$ and $\textbf{typeerror}\langle S, \sigma\rangle \rightarrow \textbf{typeerror}\langle\sigma\rangle$ for all S, σ where above rules do not imply otherwise.

Figure 2.4.: **dyn**'s structural operational semantics (Continuation from page 44).

1. $\langle \mathrm{u} := e_\varepsilon, \sigma\rangle \;\rightarrow\; \textbf{final}\langle\sigma[\mathrm{u} := \sigma(e_\varepsilon)]\rangle$, $\quad \sigma(op(\overrightarrow{e_\varepsilon})) \triangleq f_{op}(\overrightarrow{\sigma(e_\varepsilon)})$,
   $\sigma(e_{\varepsilon 0} == e_{\varepsilon 1}) \triangleq \sigma(e_{\varepsilon 0}) = \sigma(e_{\varepsilon 1})$, $\quad \sigma(e_\varepsilon \; is\_a?\; C) \triangleq \sigma(e_\varepsilon.@\mathbf{c}) = \theta_C$
2. $$\frac{\sigma(e_\varepsilon) = \text{true}}{\langle \textbf{if } e_\varepsilon \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}, \sigma\rangle \;\rightarrow\; \langle S_1, \tau\rangle}$$
3. $$\frac{\sigma(e_\varepsilon) = \text{false}}{\langle \textbf{if } e_\varepsilon \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}, \sigma\rangle \;\rightarrow\; \langle S_2, \tau\rangle}$$
4. $\langle \overrightarrow{u} := \overrightarrow{e_\varepsilon}, \sigma\rangle \;\rightarrow\; \textbf{final}\langle\sigma[\overrightarrow{u} := \sigma(\overrightarrow{e_\varepsilon})]\rangle$ where $\overrightarrow{u} \in \mathfrak{V}_L^+$, $\overrightarrow{e_\varepsilon} \in Expr_s^+$
5. $\langle \textbf{begin local } \overrightarrow{u} := \overrightarrow{e_\varepsilon}; S \textbf{ end}, \sigma\rangle \;\rightarrow\; \langle \overrightarrow{u}\,\overrightarrow{u} := \overrightarrow{e_\varepsilon}\,\overrightarrow{null}; S; \overrightarrow{u}\,\overrightarrow{u} := \sigma(\overrightarrow{u}\,\overrightarrow{u}), \sigma\rangle$
   where $\{\overrightarrow{u}\} = ((var(S) \cup change(S)) \cap \mathfrak{V}_L) \setminus (\{\overrightarrow{u}\} \cup \mathfrak{V}_S)$ and $\overrightarrow{null}$ is a sequence of *null* values of fitting length.
6. $\langle e_{\varepsilon 0}.\mathrm{m}(e_{\varepsilon 1}, ..., e_{\varepsilon n}), \sigma\rangle \;\rightarrow\; \langle \textbf{begin local self}, \overrightarrow{u} := e_{\varepsilon 0}, ..., e_{\varepsilon n}; S \textbf{ end}, \sigma\rangle$
   where **method** $m(u_1, ..., u_n)\{S\} \in \mathfrak{M}_C$ and $\sigma(e_{\varepsilon 0}.@\mathbf{c}) = \theta_C$.

Figure 2.5.: **stat**'s structural operational semantics coincides with **dyn**'s except for those rules.

Recall that **stat** is statically typed. However, to be syntactically equivalent to its dynamically typed twin **dyn**, its syntax does not contain any type information. It follows that for static type checking of **stat** programs it is necessary to first derive the type information using a program analysis (type inference). Note however, that type checking additionally requires checking the type inference results for type safety, which is accomplished by comparing them with the least precise typesafe typing $ty_\pi^\dagger$, i.e. when ty is the result for a program π, then π is typesafe iff $ty \sqsubseteq ty_\pi^\dagger$. It is important to note that the analysis formalized in this section only deals with the derivation of type information (type inference) while the restrictions (type checking) imposed by the static type system were already formalized in definition 3. Among these restrictions are for instance the fact that the receiver of a method call must support the respective method and that the operands of operations like = or == need to be of compatible types.

In this section, we will thus introduce a simple flow-sensitive type inference algorithm for **stat**. Like many program analyses, our algorithm is based on the theory of Abstract Interpretation [23]. Keep in mind that in static typing only well-typed programs are considered valid and this algorithm hence represents an additional criterion that **stat** programs have to meet as opposed to **dyn** programs. To see that this is indeed a restriction, observe that despite their syntactic similarity, there are **dyn** programs that cannot be translated into **stat** without significant alteration. The case study given in Section 11.1 is an example of this.

Instead of the type domain defined in Section 1.4.2 (which is for **dyn**), we use the lattice depicted in Figure 2.3 for **stat**. Our abstract domain hence is $T = \{\mathbb{N}, \mathbb{B}, \mathbb{O}, \top, \bot\}$ with the depicted partial order. We also introduce abstract states $\mathring{\Sigma} \triangleq \mathfrak{V}_L \mapsto T \times \mathfrak{V}_I \mapsto T \times \mathfrak{C} \mapsto \mathfrak{V}_I \mapsto T$, which also form a complete lattice with the point-wise lifted lattice operations.

In order to define our type inference as an Abstract Interpretation-based program analysis, we introduce the following notation:

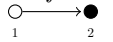
Definition 14. *A* directed, labeled, flow-graph over a set L *is a 5-tuple* $G \triangleq (N, E, s, e, M)$ *consisting of*

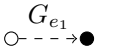
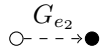
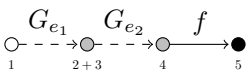
- *a finite set* N *of nodes,*
- *a finite set* $E \subseteq N \times N$ *of edges,*
- *a start node* $s \in N$*,*
- *an exit node* $e \in N$ *and*
- *an edge-labeling function* M*, mapping edges from* E *to* L*.*

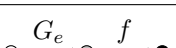
Since start- and exit-nodes are both unique in a flow-graph, we depict flow-graphs like $\overset{1}{\circ}\xrightarrow{l_1}\overset{2}{\circ}\xrightarrow{l_2}\overset{3}{\bullet}$ where $\overset{1}{\circ}$ is the start-node, $\overset{3}{\bullet}$ is the exit-node and $l_1, l_2 \in L$ are labels. We use dashed edges and G-labels to denote sub-flow-graphs like $\overset{1}{\circ}\overset{G}{\dashrightarrow}\overset{2}{\bullet}$ where G stands for the entire flow-graph from $\overset{1}{\circ}$ to $\overset{2}{\bullet}$.

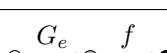
Then, for a given **stat**-program π, we define the set of program locations $\mathbf{Loc}_\pi$ as well as the control flow $\mathbf{F}_\pi \subseteq \mathbf{Loc}_\pi \times \mathbf{Loc}_\pi$ by inductively defining a function $FG()$ mapping statements and expressions to directed, labeled flow-graphs over the set of functions from $\mathring{\Sigma}$ to $\mathring{\Sigma}$.

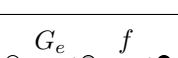
- $$\frac{FG(S_1) = \overset{1}{\circ}\overset{G_{S_1}}{\dashrightarrow}\overset{2}{\bullet} \qquad FG(S_2) = \overset{3}{\circ}\overset{G_{S_2}}{\dashrightarrow}\overset{4}{\bullet}}{FG(S_1; S_2) = \overset{1}{\circ}\overset{G_{S_1}}{\dashrightarrow}\overset{2+3}{\circ}\overset{G_{S_2}}{\dashrightarrow}\overset{4}{\bullet}}$$
- $FG(\mathbf{null}) = \overset{1}{\circ}\xrightarrow{f}\overset{2}{\bullet}$ with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\mathbf{r} := \mathbb{O}]$
- $FG(\mathrm{u}) = \overset{1}{\circ}\xrightarrow{f}\overset{2}{\bullet}$ with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\mathbf{r} := \mathring{\sigma}(\mathrm{u})]$
- $FG(@\mathrm{v}) = \overset{1}{\circ}\xrightarrow{f}\overset{2}{\bullet}$ with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\mathbf{r} := \mathring{\sigma}(\mathbf{self}.@\mathrm{v})]$

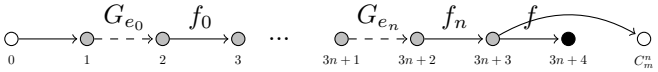
- $FG(\mathbf{self}) =$  with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\mathbf{r} := \mathring{\sigma}(\mathbf{self})]$

- $$\frac{FG(e_1) = \text{  } \qquad FG(e_2) = \text{  }}{FG(e_1{==}e_2) = \text{  }}$$ with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\mathbf{r} := \mathbb{B}]$

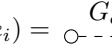
- $$\frac{FG(e) = \text{  }}{FG(e \text{ is\_a? } C) = \text{  }}$$ with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\mathbf{r} := \mathbb{B}]$

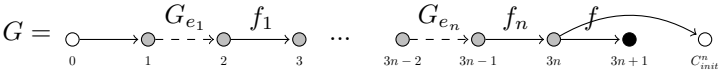
- $$\frac{FG(e) = \text{  }}{FG(\text{u} := e) = \text{  }}$$ with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\text{u} := \mathring{\sigma}(\mathbf{r})]$

- $$\frac{FG(e) = \text{}}{FG(@\text{v} := e) = \text{  }}$$ with $f \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[\mathbf{self}.@\text{v} := \mathring{\sigma}(\mathbf{r})]$

- $$\frac{FG(e_i) = \text{} \text{ for } i \in \mathbb{N}_n^0}{FG(e_0.\text{m}(e_1, ..., e_n)) = \text{  }}$$

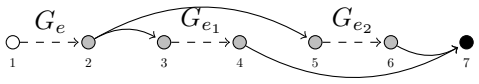
  with $f_i \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[a_i := \mathring{\sigma}(\mathbf{r})]$ for $i \in \mathbb{N}_n^0$, $f \triangleq \lambda\mathring{\sigma}. \bigsqcup_{C \in \mathfrak{C}_m^n \cap \mathring{\sigma}(a_0)} R_{C.m}^n$ where $\mathfrak{C}_m^n = \{C \mid \mathit{method}\ m(p_1, ..., p_n) \in \mathfrak{M}_C\}$ and $a_1, ..., a_n$ are intermediate variables used to store the arguments of the method call.

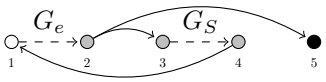
- $$\frac{FG(e_i) = \text{  } \text{ for } i \in \mathbb{N}_n^1}{FG(\mathbf{new}\ \text{C}(e_1, ..., e_n)) = G}$$

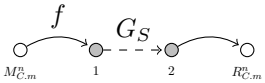
  with $G =$  ,

  $f_i \triangleq \lambda\mathring{\sigma}.\mathring{\sigma}[a_i := \mathring{\sigma}(\mathbf{r})]$ for $i \in \mathbb{N}_n^0$, $f \triangleq \lambda\mathring{\sigma}.R_{C.init}^n$.

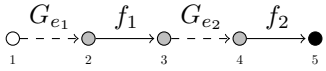
- $$\frac{FG(e) = \overset{1}{\circ} \xrightarrow{G_e} \overset{2}{\bullet} \qquad FG(e_1) = \overset{3}{\circ} \xrightarrow{G_{e_1}} \overset{4}{\bullet} \qquad FG(e_2) = \overset{5}{\circ} \xrightarrow{G_{e_2}} \overset{6}{\bullet}}{FG(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ end}) = G}$$

  with $G =$ 

- $$\frac{FG(e) = \overset{1}{\circ} \xrightarrow{G_e} \overset{2}{\bullet} \qquad FG(S) = \overset{3}{\circ} \xrightarrow{G_S} \overset{4}{\bullet}}{FG(\textbf{while } e \textbf{ do } S \textbf{ done}) =}$$
  

- $$\frac{FG(S) = \overset{1}{\circ} \xrightarrow{G_S} \overset{2}{\bullet}}{FG(\text{method C.m}(p_1, ..., p_n)\{S\}) =}$$
  

  with $f \triangleq \lambda\mathring{\sigma}.\bigsqcup\{f'_C(C^n_m) \mid C \in C^n_m(a_0)\}$ where $f'_C(\mathring{\sigma}) = \mathring{\sigma}[self, p_1, ..., p_n, \overrightarrow{v} := \mathbb{O}, \mathring{\sigma}(a_1), ..., \mathring{\sigma}(a_n), \overrightarrow{\mathbb{O}}]$, $\{\overrightarrow{v}\} = ((var(S) \cup change(S)) \cap \mathfrak{V}_L) \setminus \{\overrightarrow{p}\}$ and $\overrightarrow{\mathbb{O}}$ is a sequence of $\mathbb{O}$-types of fitting length.

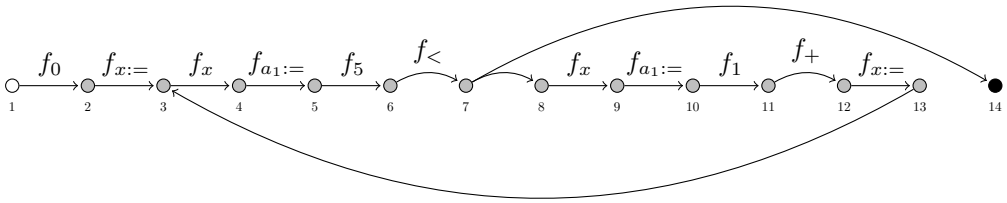
- $$\frac{\oplus : \mathbb{T}_1 \times \mathbb{T}_2 \mapsto \mathbb{T} \in op \qquad FG(e_1) = \overset{3}{\circ} \xrightarrow{G_{e_1}} \overset{4}{\bullet} \qquad FG(e_2) = \overset{5}{\circ} \xrightarrow{G_{e_2}} \overset{6}{\bullet}}{FG(e_1 \oplus e_2) =}$$
  

  where $f_1(\mathring{\sigma}) = \mathring{\sigma}[a_1 := \mathring{\sigma}(\mathbf{r})]$ and $f_2(\mathring{\sigma}) = \mathring{\sigma}[\mathbf{r} := \mathbb{T}]$

For example, the following simple program π':

```
x := 0;
while x < 5 do
  x := x + 1;
done
```

is mapped to the following flow-graph $FG(\pi')$:



where
$f_n(\mathring{\sigma}) = \mathring{\sigma}[\mathbf{r} := \mathbb{N}]$ for all $n \in \mathbb{N}$,
$f_x(\mathring{\sigma}) = \mathring{\sigma}[\mathbf{r} := \mathring{\sigma}(x)]$ for all $x \in \mathfrak{V}_L$,
$f_{x:=}(\mathring{\sigma}) = \mathring{\sigma}[x := \mathring{\sigma}(\mathbf{r})]$ for all $x \in \mathfrak{V}_L$, and
$f_\oplus$ is defined like the function f_2 for the operation $\oplus$ in the rule for operations.

Note that given a **stat**-program π, $FG()$ provides a way to extract both the set of program locations $\mathbf{Loc}_\pi$ and the control flow $\mathbf{F}_\pi \subseteq \mathbf{Loc}_\pi \times \mathbf{Loc}_\pi$ from the parse tree. Additionally, the edges in this directed flow-graph are labeled with so-called *transfer functions* computing the respective steps in an abstract computation on abstract states from $\mathring{\Sigma}$.

$$FG(\pi) = (\mathbf{Loc}_\pi, \mathbf{F}_\pi, tf_\pi)$$

We use the notations ${}_\circ S$ and $S_\bullet$ to refer to the respective unique start and exit locations of the flow-graph corresponding to the statement or expression S. It is important to note that we think of program locations as identified with parse-tree nodes and hence consider it possible that two syntactically identical statements $S \equiv S'$ within a program π differ in their program locations ${}_\circ S \neq {}_\circ S'$ and $S_\bullet \neq S'_\bullet$.

Note that all transfer functions above are monotone ($x \sqsubseteq y \to f(x) \sqsubseteq f(y)$) with respect to the partial order $\sqsubseteq$ of our type domain. Hence, they fulfill all requirements for a Monotone Framework:

Definition 15 (Monotone Framework)**.** *A* Monotone Framework *is a 5-tuple*

$$\mathfrak{F} = (L, F, E, \iota, f)$$

with

- $(L, \sqsubseteq)$ *being a complete lattice serving as an Abstract Domain,*
- $F \subseteq \mathbf{Loc} \times \mathbf{Loc}$ *is a finite flow,*
- $E \subseteq \mathbf{Loc}$ *a finite set of extremal labels,*
- ι *a mapping from* E *to extremal values,*
- f *a mapping from flow-edges* $\in F$ *to montone transfer functions from* $L \mapsto L$.

Every instance $\mathfrak{F}$ of the monotone framework implicitly defines the following set of constraints $\mathfrak{F}^{\sqsupseteq}$ with solutions $A \in \mathbf{Loc} \mapsto L$:

$$A(l) \sqsupseteq f_{(l',l)}(A(l')) \text{ forall } (l',l) \in F$$

$$A(l) \sqsupseteq \iota(l) \text{ if } l \in E$$

We can then define an associated function $\overrightarrow{f}$ as

$$\overrightarrow{f_{\mathfrak{F}}} = \lambda l. \bigsqcup \{f_{(l',l)}(A(l')) \mid (l',l) \in F\} \sqcup \begin{cases} \iota(l) \text{ if } l \in E \\ \bot \text{ otherwise} \end{cases}$$

Note that every solution $A \models \mathfrak{F}^{\sqsupseteq}$ must also be a fixpoint of $\overrightarrow{f_{\mathfrak{F}}}$. Since all our transfer functions are monotone, so is $\overrightarrow{f_{\mathfrak{F}}}$ and Tarski's fixpoint theorem [78] hence guarantees that a solution can be obtained using the usual least fixpoint algorithm.

Using the program locations, control flow, transfer functions and above abstract domain of types, we can instantiate the monotone framework as follows

$$\mathfrak{F}_\pi = (\mathring{\Sigma}, \mathbf{F}_\pi, \{_\circ\pi, \pi_\bullet\}, \bot, tf_\pi)$$

and obtain the desired flow-sensitive type information about our program in the form of a solution $A \models \mathfrak{F}_\pi^{\sqsupseteq}$. To discuss soundness of our analysis, we introduce the following correctness relation

$R \subseteq \Sigma \times \mathring{\Sigma}$ such that

$$\begin{aligned} \sigma R \mathring{\sigma} \triangleq & \forall \mathrm{u} \in \mathfrak{V}_L \bullet \sigma(\mathrm{u}) \text{ is of type } \mathring{\sigma}(\mathrm{u}) \wedge \\ & \forall @\mathrm{v} \in \mathfrak{V}_I \bullet \sigma(\mathbf{self}.@\mathrm{v}) \text{ is of type } \mathring{\sigma}(\mathbf{self}.@\mathrm{v}) \wedge \\ & \forall o \in \mathbb{O} \bullet \sigma(o.@\mathbf{c}) = \theta_C \rightarrow \forall @\mathrm{v} \in \mathfrak{V}_I \bullet \sigma(o.@\mathrm{v}) \text{ is of type } \mathring{\sigma}(C, @\mathrm{v}) \end{aligned}$$

along with the following notations to relate our program analysis to the operational semantics:

$$\begin{aligned} S \vdash \sigma \rightsquigarrow \sigma' &\triangleq \langle S, \sigma \rangle \ \stackrel{*}{\rightarrow} \ \mathbf{final}\langle \sigma' \rangle \\ S, \mathring{\sigma} \blacktriangleright ty &\triangleq \mathfrak{F}_B = (\mathring{\Sigma}, \mathbf{F}_S, \{_\circ S, S_\bullet\}, \{_\circ S \mapsto \mathring{\sigma}, S_\bullet \mapsto \bot\}, tf_S) \wedge ty \models \mathfrak{F}_B^{\sqsupseteq} \\ S \vdash \mathring{\sigma} \rhd \mathring{\sigma}' &\triangleq S, \mathring{\sigma} \blacktriangleright ty \wedge ty(S_\bullet) = \mathring{\sigma}' \end{aligned}$$

Intuitively, $\sigma R \mathring{\sigma}$ states that $\mathring{\sigma}$ is an abstraction of σ, $S \vdash \sigma \rightsquigarrow \sigma'$ means that the statement S when executed in the state σ will terminate after finitely many steps in the final state σ', $S, \mathring{\sigma} \blacktriangleright ty$ states that above type inference algorithm derives the typing ty when applied to the statement S and the abstract start state $\mathring{\sigma}$ and finally, $S \vdash \mathring{\sigma} \rhd \mathring{\sigma}'$ states that in the abstract model of our type inference, when the statement S is (abstractly) executed in the (abstract) start state $\mathring{\sigma}$, it yields the (abstract) final state $\mathring{\sigma}'$.

Theorem 3 (Soundness of Type Inference)**.** *For all* $S \in \mathit{Stmt}, \sigma \in \Sigma, \mathring{\sigma} \in \mathring{\Sigma}$*,*

$$\sigma R \mathring{\sigma} \wedge S \vdash \sigma \rightsquigarrow \sigma' \wedge S \vdash \mathring{\sigma} \rhd \mathring{\sigma}' \Rightarrow \sigma' R \mathring{\sigma}'$$

Hence, we consider the analysis *sound* when executing the statement S and abstracting its final state always yields the same result like abstracting its start state and executing S abstractly from it.

Proof. The analysis can easily be checked for soundness by comparing above rules with their respective counterparts from the operational semantics given in Section 2.1.2 and checking that Theorem 3 holds in each case. □

Naming the solution of $\mathfrak{F}_{\overline{B}}^{\sqsupseteq}\ ty$ already indicated that our type inference algorithm is deriving typings in the sense of Section 1.3.1. Its soundness furthermore implies that these typings safely over-approximate all possible runtime behaviors (are sound in the sense of Section 1.3.1). However, recall that to establish type-safety, ty must also be of sufficient precision. Formally, we require $ty \sqsubseteq ty_\pi^\dagger$ for when this is the case, we can be sure that

- the receivers of all method calls will support the called method and
- the conditions of all conditionals and loops will be boolean

in all executions of S.

Definition 16. *A* ***stat****-program* π *is called* well-typed *iff* $\pi, \bot \blacktriangleright ty$ *and* $ty \sqsubseteq ty_\pi^\dagger$*.*

Since **stat** is statically typed, we furthermore impose the restriction that all **stat**-programs have to be well-typed.

Note that by using the type domain for **dyn** as defined in Section 1.4.2 instead of T and some minor modifications in the rules above (for instance substituting $\{C_{null}\}$ instead of $\mathbb{O}$ for the type of **null** and $\{bool\}$ for $\mathbb{B}$ whereever appropriate), one can turn this analysis into one for **dyn**. In the following we will hence regard it as applicable to both languages.

Part II.

Program Logic

"Logic will get you from A to B. Imagination will get you anywhere!"
– *Albert Einstein*

3. Starting Point: Tagged Hoare Logic for Statically Typed Programs

In this chapter, we will give a standard (statically typed) Hoare logic as a point of reference. Before introducing the program logic itself (Section 3.3), we will first define the assertion language **AL** (Section 3.1) and introduce a notation we call "Tagged Hoare Logic" allowing for the concise treatment of multiple notions of correctness in Hoare logic-like proof rules without the need for duplicating our proof system (Section 3.2).

3.1. Assertion Language

Before going into details of the program logic, we introduce the assertion language **AL**. Its syntax is depicted in Figure 3.1. Essentially, it is predicate logic with quantification over finite sequences of basic types – weak second order logic. Note that **AL** is statically typed, as usual. Its type system however is simplistic: The basic types $\mathbb{T} = \{\mathbb{N}, \mathbb{O}, \mathbb{B}\}$ form a flat lattice with $\top$ and $\bot$ and a type constructor τ^* for finite sequences of elements of type τ .

Assertions (a) are constructed from equations between logical expressions of identical type, boolean connectives, quantification over finite sequences, typechecks and tags whose role will become clear in the next chapter. (Typed) logical expressions (l) in turn consist of logical variables of some type $t \in \mathbb{T}$, local program variables of type $\mathbb{O}$, conditionals with a condition of type $\mathbb{B}$ and branches of equal type, instance variables $l.@x$ where both l and the result are of type $\mathbb{O}$, the representative objects (θ_C) of all classes $C \in \mathfrak{C}$, the objects *null* and **self** as well as constants (*cnst*) and operations ($op(\overrightarrow{l})$), which just like **stat** expressions include the usual operations on booleans and natural numbers and thus allow integer arithmetic. Note that contrary to programming expressions, logical expressions are able to access instance variables of objects other than **self**.

Following [14], undefined operations like dereferencing a *null* value or accessing a sequence with an index that is out of bounds ($l[n]$ with $n \geq |l|$) yield a *null* value and equality is non-strict with respect to such values ($null = null$ is *true*) in order to keep assertions two-valued. Also, for logical expressions $l \in LExp$, we extend the state-access to $\sigma(l)$ in the canonical way.

Note that the syntax $[\![l]\!] \in \{Cl\}$ is redundant. It can be regarded as syntactic sugar

$$[\![l]\!] \in \{C_1, ..., C_n\} \triangleq l.@\mathbf{c} = \theta_{C_1} \vee ... \vee l.@\mathbf{c} = \theta_{C_n}$$

and also note that since *null* is the only instance of class C_{null}, it holds that

$$l.@\mathbf{c} = \theta_{C_{null}} \Leftrightarrow l = null.$$

To link programming language-objects with assertion-values, we define

Definition 17 (Mapping Predicates)**.** [1] [2] *For* $o : \mathbb{O}$*,* $n : \mathbb{N}$*, and* $b : \mathbb{B}$

$\mathbb{N}(o) \triangleq o \neq null \wedge o.@\mathbf{c} = \theta_{num}$*,*

$\mathbb{N}(o, n) \triangleq \mathbb{N}(o) \wedge n = 0 \rightarrow o.@\text{pred} = null \wedge n > 0 \rightarrow \mathbb{N}(o.@\text{pred}, n - 1)$*,*

$\mathbb{B}(o) \triangleq o \neq null \wedge o.@\mathbf{c} = \theta_{bool}$*, and*

$\mathbb{B}(o, b) \triangleq \mathbb{B}(o) \wedge b \leftrightarrow o.@\text{to\_ref} \neq null$*.*

To see that mapping predicates are necessary for completeness, consider the intermediate assertion p in the following program

$P \equiv$ **if** b **then** $x := 5$ **else** $x :=$ *true* **end**$\{p\}$;
if x is_a? *bool* **then if** x **then** $x := 10$ **end else** $x := x * 2$ **end**

Since **AL** is statically typed, we must also give a type to the program variable x. Now, giving it the type $\mathbb{N}$ would allow us to express $x = 5$, but not $x = true$ while giving it the type $\mathbb{B}$ raises the converse problem. However, using mapping predicates, it is possible to accurately describe the set of intermediate states as $\mathbb{N}(x, 5) \vee \mathbb{B}(x, true)$. From this observation it is not hard to see that $\{true\}P\{x = 10\}$ (or $\{true\}P\{\mathbb{N}(x, 10)\}$) is not derivable without mapping predicates.

In assertions, tags may appear, e.g., **typesafe** $\rightarrow v \neq null$. We use the notation $\sigma, \mathbf{tags} \models a$ to denote the fact that the assertion a is true in the state σ under the tags **tags**. The definition of $\models$ is standard except for the case

$$\sigma, \mathbf{tags} \models \mathbf{tag} \text{ iff } \mathbf{tag} \in \mathbf{tags}.$$

Those tags will be explained in the next chapter. For now, it is sufficient to regard them as an additional kind of variable.

3.2. Notation: Tagged Hoare Logic

Hoare Logic usually comes in a number of variants called the "notions of correctness". Hoare Triples

$$\{p\}S\{q\}$$

hence always need to be accompanied by a statement of the form "holds in the sense of so-and-so correctness" to allow proper interpretation. For instance, in the sense of partial correctness, above triple would express a safety property (from all start states satisfying p, executing S will result in a state satisfying q) while in the sense

[1] The predicate $\mathbb{N}(o, n)$ is recursive. However, the technique discussed in Section 1.5.5 allows expressing it in **AL**.

[2] @pred and @to_ref are instance variables of the classes num and bool, respectively.

$$
\begin{aligned}
Asrt \ni a ::= & \overbrace{l = l \mid \neg a \mid a \wedge a \mid \exists v : t^* \bullet a}^{Weak\ Second{-}Order\ Logic} \mid \overbrace{[\![l]\!] \in \{Cl\}}^{Types} \mid \overbrace{\mathbf{tag}}^{Tags} \\
LExp \ni l ::= & \ v \mid \underbrace{\mathrm{u} \mid \mathbf{if}\ l\ \mathbf{then}\ l\ \mathbf{else}\ l\ \mathbf{fi}}_{Program\ Logic} \mid \underbrace{l.@x \mid \theta_C \mid \mathbf{null} \mid \mathbf{self}}_{Objects} \mid \underbrace{|l| \mid l[l_n]}_{Sequences} \\
& \mid \underbrace{cnst \mid op(\overrightarrow{l})}_{Arithmetic} \\
Cl ::= & \ \epsilon \mid CL,\ CL ::= C \mid C, CL
\end{aligned}
$$

with $t \in \mathbb{T}$, $\mathbf{tag} \in \mathcal{T}ags$, $C \in \mathfrak{C}$, op and $cnst$ like defined in Figure 2.2. The abbreviations introduced in Section 1.5.4 also apply here. Brackets are used for disambiguation. Additionally: $Qv : t \bullet a \equiv Qv : t^* \bullet |v| = 1 \wedge a[v[0]/v]$ for $Q \in \{\forall, \exists\}$,

Figure 3.1.: Syntax of the assertion language **AL**

of total correctness it would additionally express the liveness property that S always terminates from states satisfying p and hence a final state satisfying q will eventually be reached. In scientific papers dealing with Hoare Logic it is customary to choose one of these interpretations and furthermore stick to it to avoid confusing the reader. However, since in the remainder of this thesis, we will be often be jumping back and forth between typesafe- and type-unsafe notions of correctness, allow me to introduce a notation I call *Tagged Hoare Logic* to address this issue and furthermore express myself more clearly.

Further benefits of this notation are

- Fuses the abundance of proof sytems due to different "notions of correctness" into one, which is beneficial from a tool design perspective as it moves the choice of which properties the tool should verify from the tool designer to the user writing the specification.
- Allows soundness and completeness arguments to be derived once for a single proof system instead of a multitude of them thus reducing duplicated work.
- Saves valuable space in scientific papers ☺.

The basic idea behind Tagged Hoare Logic is to disassemble Hoare Triples into their semantic components and then introduce tags like **terminates** to clearly state in the postcondition what property is expressed by each of them. Above Hoare Triple in the sense of total correctness hence becomes two triples

$$\{p\}S\{q\} \qquad and \qquad \{p\}S\{\mathbf{terminates}\}.$$

Since in Tagged Hoare Logic, there is only one proof system, those two triples are statements in the same proof system and a trivial application of the conjunction rule yields

$$\{p\}S\{\mathbf{terminates} \wedge q\}$$

which makes the total correctness interpretation explicit. Similarly we introduce the tags **typesafe** for type-safe partial correctness (the absence of type errors) and **failsafe** for strong partial correctness (the absence of faults). Since pre-and postconditions are assertions and assertions are expressing sets of states, philosophically, one may think of those tags as a state-centric way of expressing liveness properties. The tag **terminates** hence means something like "this state is reachable eventually".

A (big step) program semantics maps programs and initial states to sets of final states. Traditionally, each notion of correctness needs its own program semantics as they differ in what characteristics of a computation they guarantee. We define the (infinite) set of (finite or infinite) computations as

$$Comp = Conf^*_{proper}\, Conf_{final} \cup Conf^*_{proper}\, Conf_{error} \cup Conf^\omega_{proper}$$

and those of a program S starting in an initial state σ as

$$Comp(S,\sigma) = \{C_0, C_1, ... \mid C_0 = \langle S,\sigma\rangle \wedge \forall i \bullet C_i \ \rightarrow\ C_{i+1}\} \subset Comp\ .$$

Let the symbol ρ denote elements of $Comp$. We now formally introduce the following tags along with their respective error states:

$$\mathcal{T}ags = \{\mathbf{terminates}, \mathbf{typesafe}, \mathbf{failsafe}\}.$$

Each tag stands for a notion of correctness that in addition to partial correctness avoids one type of error: **terminates** avoids divergence (infinite computations), **typesafe** avoids type errors (e.g., non-boolean expressions as loop conditions), and **failsafe** avoids runtime failures (e.g., *null* as condition or method call receiver). The tagged program semantics $\mathcal{M}_{\text{tags}}$ defined below will record any occurring error by a special *error state* of the set $\Sigma_\perp = \{\perp, \mathbf{typeerror}, \mathbf{fail}\}$. Let $\Sigma_+ = \Sigma \uplus \Sigma_\perp$, where Σ is the set of program states. To define the tagged program semantics $\mathcal{M}_{\text{tags}}$, we need appropriate selectors:

$$\mathcal{S} : \mathcal{T}ags \cup \{\emptyset\} \mapsto Comp \mapsto 2^{\Sigma_+}$$

with

$$\mathcal{S}_\emptyset(\rho) = \begin{cases} \{\tau\} & \text{if } \rho = C_0, ..., C_n \wedge C_n = \mathbf{final}\langle\tau\rangle^1 \wedge \tau \in \Sigma \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\text{terminates}}(\rho) = \begin{cases} \{\perp\} & \text{if } \rho \text{ is infinite} \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathbf{typesafe}}(\rho) = \begin{cases} \{\mathbf{typeerror}\} & \text{if } \rho = C_0, ..., C_n \wedge C_n = \mathbf{typeerror}\langle\tau\rangle^1 \text{ for } \tau \in \Sigma \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathbf{failsafe}}(\rho) = \begin{cases} \{\mathbf{fail}\} & \text{if } \rho = C_0, ..., C_n \wedge C_n = \mathbf{fail}\langle\tau\rangle^1 \text{ for some } \tau \in \Sigma \\ \{\} & \text{otherwise} \end{cases}$$

Finally, we are able to define tagged program semantics

$$\mathcal{M} : 2^{\mathcal{T}ags} \mapsto Prog \mapsto \Sigma \mapsto 2^{\Sigma_+}$$

[1] as defined in Section 1.2.2

allowing arbitrary combinations of correctness notions. Let $\mathbf{tags} \subseteq \mathcal{T}ags$, then the input-output semantics of S respecting tags is defined as follows:

$$\mathcal{M}_{\text{tags}}[\![S]\!](\sigma) \triangleq \bigcup\{\mathcal{S}_{\text{tag}}(\rho) \mid \rho \in Comp(S,\sigma), \mathbf{tag} \in \mathbf{tags} \cup \{\emptyset\}\}.$$

However, we first need to extend the semantics of our assertions (see Figure 3.1) to also include tags

$$[\![p]\!]_{\text{tags}} = \{\sigma \mid \sigma \in \Sigma \wedge \sigma, \mathbf{tags} \models p\}$$

and extend the input-output semantics to sets of states

$$\mathcal{M}_{\text{tags}}[\![S]\!](\{\sigma_1, \sigma_2, ...\}) \triangleq \bigcup_{\sigma \in \{\sigma_1, \sigma_2, ...\}} \mathcal{M}_{\text{tags}}[\![S]\!](\sigma)$$

before we can properly define the meaning of our Tagged Triples:

Definition 18 (Tagged Hoare Triples)**.** *For a* Tagged Hoare Triple
$\models \{p\}S\{\mathbf{tags} \wedge q\}$ *iff* $\mathcal{M}_{\text{tags}}[\![S]\!]([\![p]\!]_{\text{tags}}) \subseteq [\![q]\!]_{\text{tags}}$.

Note that the semantics $\mathcal{M}_{\text{tags}}[\![S]\!]$ of a program S can produce error states, but the semantics $[\![p]\!]_{\text{tags}}$ and $[\![q]\!]_{\text{tags}}$ of the assertions p and q do not tolerate any error states. Thus $\models \{p\}s\{\mathbf{tags} \wedge q\}$ formalizes program correctness in the sense of the tags **tags** as desired.

3.3. Statically Typed Hoare Logic

In this chapter, we will give a standard, (statically typed) Hoare logic $\mathcal{H}_s$ for **stat**. It will serve as a starting point for our further developments as well as a point of reference to compare our to-be-defined dynamically typed Hoare logic against. Although the rules given are a tagged Hoare logic, all definitions and rules are taken more or less directly from standard literature (mostly [5]) and can hence be safely skipped by readers that are well-versed in Hoare logic or related formalisms. The proof rules of $\mathcal{H}_s$ will use three substitutions on assertions. Proper definitions for all three can be found in appendix A.

We emphasise that this is a *statically typed* Hoare logic. The rules and definitions given in this chapter are suitable only when applied to well-typed **stat** programs and NOT for **dyn** programs.

For a **stat** or **dyn** statement S, let $var(S)$ ($change(S)$) denote the set of variables accessed in S (appearing on the left of an assignment in S). For an assertion p let $free(p)$ denote the set of free variables of p and $p[v := l]$ the result of substituting the logical expression l for all free occurrences of the logical variable v in p.

Definition 19. *The proof system $\mathcal{H}_s$ consists of the following rules:*

AXIOM: ASGN (both normal and instance variables)

$$\{q[u := e]\}u := e\{\mathbf{tags} \wedge q\}$$

Indeed, an assignment is terminating, typesafe and failsafe in this setting.

RULE: SEQ

$$\frac{\{p\}S_1\{\mathbf{tags} \wedge r\} \qquad \{r\}S_2\{\mathbf{tags} \wedge q\}}{\{p\}S_1; S_2\{\mathbf{tags} \wedge q\}}$$

RULE: COND

$$\frac{\{p \wedge e\}S_1\{\mathbf{tags} \wedge q\} \qquad \{p \wedge \neg e\}S_2\{\mathbf{tags} \wedge q\}}{\{p\}\ \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ \{\mathbf{tags} \wedge q\}}$$

RULE: LOOP

$$\frac{\begin{array}{c}\{p \wedge e \wedge \mathbf{terminates} \rightarrow t = z\}\ S\ \{\mathbf{tags} \wedge p \wedge \mathbf{terminates} \rightarrow t < z\} \\ p \rightarrow t \geq 0\end{array}}{\{p\}\ \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}\ \{\mathbf{tags} \wedge p \wedge \neg e\}}$$

where t is an integer expression and z is an integer variable that does not appear in p, e, t or S.
When $\mathbf{terminates} \in \mathbf{tags}$, the implications ensure that S decreases the loop variant t on each iteration. Consequently, there can only be finitely many iterations before $t \geq 0$ would be violated. The loop hence cannot diverge. Typesafety and Failsafety are ensured as long as they hold for S, since e is a **stat**-expressions and thus by definition non-diverging, failsafe and typesafe.

RULE: CONS

$$\frac{p \rightarrow p_1, \{p_1\}S\{\mathbf{tags}' \wedge q_1\}, q_1 \rightarrow q, \mathbf{tags}' \supseteq \mathbf{tags}}{\{p\}S\{\mathbf{tags} \wedge q\}}$$

The well-known consequence rule from traditional Hoare logic has been extended with the ability to omit tags. This is sound for **terminates**, **typesafe** and **failsafe** as all of them restrict the program behaviour and hence $\{p\}S\{\mathbf{tag} \wedge q\}$ implies $\{p\}S\{q\}$ in each case (for all $\mathbf{tag} \in \mathcal{T}ags$).

RULE: BLCK

$$\frac{\{p\}\overrightarrow{u}\overrightarrow{u} := \overrightarrow{e}\overrightarrow{null}; S\{\mathbf{tags} \wedge q\}}{\{p\}\ \mathbf{begin\ local}\ \overrightarrow{u} := \overrightarrow{e}; S\ \mathbf{end}\{\mathbf{tags} \wedge q\}}$$

where $\mathfrak{V}_L \cap \mathit{free}(q) = \emptyset$, $\{\overrightarrow{u}\} \subset \mathfrak{V}_L$ and $\{\overrightarrow{e}\} \subseteq Expr_s$, $\{\overrightarrow{\overrightarrow{u}}\} = ((var(S) \cup change(S)) \cap \mathfrak{V}_L) \setminus \{\overrightarrow{u}\}$ and $\overrightarrow{null}$ is a sequence of *null* constants of fitting length.

AXIOM: PASGN

$$\{p[\overrightarrow{u} := \overrightarrow{e}]\}\overrightarrow{u} := \overrightarrow{e}\{\mathbf{tags} \wedge p\}$$

Note that the parallel assignment assumes all expressions in $\overrightarrow{e}$ to not interact, which trivially holds in **stat** as all expressions are side-effect-free.

RULE: INST

$$\frac{\{p\}v_0.m(v_1,...,v_n)\{\mathbf{tags} \wedge q\}}{\{p[\overrightarrow{v} := \overrightarrow{e}]\}e_0.m(e_1,...,e_n)\{\mathbf{tags} \wedge q[\overrightarrow{v} := \overrightarrow{e}]\}}$$

where $\{\overrightarrow{v}\} \cap var(\mathfrak{M}) = var(\overrightarrow{e}) \cap change(\mathfrak{M}) = \emptyset$

Like PASGN, the INST rule also assumes the expressions in $\overrightarrow{e}$ to not interact, which trivially holds in **stat** as all expressions are side-effect-free.

RULE: REC

$$\frac{\begin{array}{c}A \vdash \{p\}S\{\mathbf{tags} \wedge q\}, \\ A' \vdash \{p_i \wedge t = z\}\mathbf{begin\ local\ self}, \overrightarrow{u_i} := l_i, \overrightarrow{v_i}; S_i\ \mathbf{end}\{\mathbf{tags}_i \wedge q_i\}, i \in \mathbb{N}_n^1 \\ p_i \rightarrow (\mathbf{failsafe} \rightarrow l_i \neq null \wedge \mathbf{typesafe} \rightarrow l_i \neq null \rightarrow l_i.@\mathbf{c} = \theta_{C_i}), i \in \mathbb{N}_n^1\end{array}}{\{p\}S\{\mathbf{tags} \wedge q\}}$$

where $\mathbf{method}\ m_i(\overrightarrow{u_i})\{S_i\} \in \mathfrak{M}_{C_i}$,

$A = \{\{p_i\}l_i.m_i(\overrightarrow{v_i})\{q_i\} \mid i \in \mathbb{N}_n^1\} \cup \{\{p_{\mathbf{tag},i}\}l_i.m_i(\overrightarrow{v_i})\{\mathbf{tag}\} \mid i \in \mathbb{N}_n^1, \mathbf{tag} \in \mathcal{T}ags\}$,

$A' = \{\{p_i \wedge (\mathbf{terminates} \rightarrow t < z\}l_i.m_i(\overrightarrow{v_i})\{q_i\} \mid i \in \mathbb{N}_n^1\} \cup \{\{p_{\mathbf{tag},i} \wedge (\mathbf{terminates} \rightarrow t < z\}l_i.m_i(\overrightarrow{v_i})\{\mathbf{tag}\} \mid i \in \mathbb{N}_n^1, \mathbf{tag} \in \mathcal{T}ags\}$,

z is a logical variable of type $\mathbb{N}$ that does not occur in p_i, q_i, t and S_i for $i \in \mathbb{N}_n^1$ and is treated in the proofs as a constant, and t is a logical expression of type $\mathbb{N}$.

While very similar to the REC rule from traditional Hoare logic introduced by Gorelick [33], adding distinguished tags $\mathbf{tags}_i$ to each method contract permits verifying a program with a mixture of different notions of correctness. For an application scenario, Section 12.1 introduces an extension that allows mixing for instance typesafe and non-typesafe code. Note, however, that the sets A and A' differ from those used traditionally. This is due to a generalization of the concept of most general correctness formulas neccessary for completeness in tagged Hoare Logic. For more information about this, please refer to the discussion before Theorem 7 in Section 5.2. Also note that the recursion bound t is only relevant for methods that actually need to terminate ($\mathbf{terminates} \in \mathbf{tags}_i$).

RULE: CNST

$$\frac{\{p\}\mathbf{new}_C.init(\overrightarrow{e})\{\mathbf{tags} \wedge q\}}{\{p\}\mathbf{new}\ C(\overrightarrow{e})\{\mathbf{tags} \wedge q\}}$$

AXIOM: NEW

$$\{p[\mathbf{r} := \mathbf{new}_C]\}\mathbf{new}_C\{\mathbf{tags} \wedge p\}$$

3.4. Auxiliary Rules

RULE: DISJ

$$\frac{\{p\}S\{\mathbf{tags} \wedge q\} \qquad \{r\}S\{\mathbf{tags} \wedge q\}}{\{p \vee r\}S\{\mathbf{tags} \wedge q\}}$$

RULE: CONJ

$$\frac{\{p_1\}S\{\mathbf{tags} \wedge q_1\} \qquad \{p_2\}S\{\mathbf{tags}' \wedge q_2\}}{\{p_1 \wedge p_2\}S\{\mathbf{tags} \wedge \mathbf{tags}' \wedge q_1 \wedge q_2\}}$$

RULE: $\exists$-INT

$$\frac{\{p\}S\{\mathbf{tags} \wedge q\}}{\{\exists x : \mathbb{N} \bullet p\}S\{\mathbf{tags} \wedge q\}}$$

where $x \notin var(\mathfrak{M}) \cup var(S) \cup \mathit{free}(q)$.

RULE: INV

$$\frac{\{r\}S\{\mathbf{tags} \wedge q\}}{\{p \wedge r\}S\{\mathbf{tags} \wedge p \wedge q\}}$$

where $\mathit{free}(p) \cap (\mathit{change}(\mathfrak{M}) \cup \mathit{change}(S)) = \emptyset$ and p does not contain quantification over objects. Note that $\mathit{change}(\mathfrak{M}) \subseteq \mathfrak{V}_I$.

RULE: SUBST

$$\frac{\{p\}S\{\mathbf{tags} \wedge q\}}{\{p[\overrightarrow{z} := \overrightarrow{t}]\}S\{\mathbf{tags} \wedge q[\overrightarrow{z} := \overrightarrow{t}]\}}$$

where $var(\overrightarrow{z}) \cap (var(\mathfrak{M}) \cup var(S)) = var(\overrightarrow{t}) \cap (\mathit{change}(\mathfrak{M}) \cup \mathit{change}(S)) = \emptyset$.

4. Tagged Hoare Logic for Dynamically Typed Programs

> "Premature optimization is the root of all evil!"
>
> – *C.A.R. (Tony) Hoare*

Traditional Hoare logic, as developed by C.A.R. Hoare, contains implicit assumptions of well-typedness and hence type-safety. For instance, the axiom for assignment

$$\{q[\mathrm{v} := e]\}\mathrm{v} := e\{q\}$$

substitutes the program expression e for all occurrences of the variable v in the assertion q, which will only yield a well-typed assertion $q[\mathrm{v} := e]$ when the types of v and e are compatible, which is exactly what a static type system would enforce when encountering the assignment $\mathrm{v} := e$. Hence, in the case of a dynamically typed language like **dyn**, where no static type-checker ensures the assignment to be well-typed, applying this axiom could yield assertions and verification conditions that are not well-typed and whose verification would hence fail in the automated theorem prover.

Why C.A.R. Hoare cast his proof system in this way can only be speculated about. Most probably he intended to make program verification as simple and convenient as possible and either did not foresee the exclusion of dynamically typed languages that this caused or considered them as of secondary importance. Surely, the fact that the languages he was most familiar with (ALGOL 60) was statically typed will have contributed to this.

In this chapter, we will generalize Hoare logic by removing these implicit assumptions to what we call *dynamically typed Hoare logic*. In order to clearly distinguish these two variants of Hoare logic, we will in this thesis refer to the traditional variant as *statically typed Hoare logic*.

While dynamically typed Hoare logic offers the same proof-theoretic strength as statically typed Hoare logic (see Chapter 5), a direct comparison (Chapter 6) reveals that the implicit assumptions allowed for several optimizations, enabling statically typed Hoare logic to offer a higher degree of convenience when verifying program properties.

The role type information plays in program verification in general and in Hoare logic in particular can be understood by studying a typical statically typed Hoare logic. The

following sections will discuss different aspects of its usage.

4.1. Shared Type System

A first general observation common to all rules is that they are using a statically typed assertion language to denote sets of program states. In fact, the type system used for this assertion language is the same as the one used for the programming language. This way, the types of program variables in the program states can be reused when they appear in assertions. The assertion $x < 5$ is hence well-typed when the variable x is of numeric type. Since in a statically typed language there can never be a program state in which x would have a different type, types can be regarded as constant properties of variables just like in the program.

4.2. Side-Effects

Consider the rule INST for instantiating the arguments of a method call:

$$\frac{\{p\}v_0.m(v_1,...,v_n)\{\mathbf{tags} \wedge q\}}{\{p[\overrightarrow{v} := \overrightarrow{e}]\}e_0.m(e_1,...,e_n)\{\mathbf{tags} \wedge q[\overrightarrow{v} := \overrightarrow{e}]\}}$$

where $\{\overrightarrow{v}\} \cap var(\mathfrak{M}) = var(\overrightarrow{e}) \cap change(\mathfrak{M}) = \emptyset$. In this rule, $\overrightarrow{v}$ is a sequence of program variables denoting the formal arguments of a call to the method m. On the other hand, $\overrightarrow{e}$ is a sequence of (program) expressions that were used as actual parameters in the call. The parallel substitution $p[\overrightarrow{v} := \overrightarrow{e}]$ denotes the simultaneous replacement of all free occurrences of variables from $\overrightarrow{v}$ in p with the respective expression from $\overrightarrow{e}$.

Noteworthy about this rule is that it does not consider any influences evaluated argument expressions might have on each other. Hence, it is only sound when the possibility of an actual parameter in $\{\overrightarrow{e}\}$ changing the state can be ruled out. In fact, in statically typed Hoare logics, programs are comprised of side-effect-free expressions and (potentially) side-effecting statements. This syntactic distinction allows for simple and effective rules like the above. This plays well with the fact that statically typed languages usually supply several basic data-types as well as a large number of build-in side-effect-free operations on them which are compiled directly to machine-code instructions. In dynamically typed languages, however, not only is it uncommon to have basic data-types or supply many side-effect-free build-in operations, but also are operations usually desugared to method calls in order to allow overriding. Hence, in the presence of dynamic dispatch distinguishing between a side-effect-free operation and a side-effecting one requires accurate type information, which is not available statically. In this setting, it is thus not even possible to decide whether the expressions $a + b$ is side-effecting or not, since the variables a and b could refer to instances of user-defined data-types implementing the method m_+ in arbitrary ways. Therefore, in order to make program logics for dynamically typed languages sound, they either have to prepend all operations using a program transformation or have to consider them all as side-effecting and thus work with more complex rules.

4.3. Shortcut Rules

Once again, consider the assignment axiom in statically typed languages:

$$\{p[v := e]\}v := e\{p\}.$$

Note that e is an expression of the programming language rather than a logical expression of the assertion language. The only reason that the substitution $p[v := e]$ results in a syntactically correct assertion is that the programming language expressions are a subset of the logical expressions of the assertion language. Hence not only are variables and their types shared between the two languages, but entire expressions with their (side-effect-free) operations. This convenient construction is so prevalent in statically typed Hoare logics that it was even used by Cook [22] in his seminal completeness proof. It is hence not at all clear whether or not breaking this construction affects Cook's results. For this reason, we will in Chapter 5 carefully study the proof theory of dynamically typed Hoare logic to exclude the possibility that allowing variables to reference values of different types may have adverse effects on the completeness of our Hoare logic.

We call rules like the above *shortcut rules* as reasoning about assignments like $v := e$ in our Hoare logic for **dyn** requires deconstructing the expression e and reasoning about every operation in it as the method call it is desugared to (also considering potential side effects) which takes significantly more effort. However, in Chapter 7 we will introduce a technique allowing the use of shortcut rules also for reasoning about dynamically typed programs.

4.4. The Transition To Dynamic Typing

We already saw that dynamic typing in the programming language will break not only the construction that allowed sharing the type system / expressions between programming- and assertion language, but also the convenient syntactic distinction between side-effect-free expressions and side-effecting statements. Hence a Hoare logic for a dynamically typed language like **dyn** must

- provide a way to map untyped programming language variables to typed values,
- consider all expressions as side-effecting, and
- refrain from using shortcut rules.

One could hence consider dynamic typing as an adverse condition trying to impede the verification of **dyn** programs. Fortunately, it turns out that

- when considering untyped variables as of type object, an assertion language with recursive predicates is able to map them to typed values,
- introducing a special variable **r** and making all expressions/statements conventionally assign their result value to it, enables reformulating shortcut rules as conventional ones, and

- while side-effecting expressions make the rules more complex it is still possible to obtain a sound and complete Hoare logic for **dyn** (see next section).

4.5. Dynamically Typed Hoare Logic

This section will list the rules of a dynamically typed Hoare logic $\mathcal{H}_d$ for **dyn**. Just like the rules for $\mathcal{H}_s$ given in Section 3.3, our exposition of the proof rules of $\mathcal{H}_d$ will use three substitutions on assertions. Proper definitions for all three can be found in Appendix A.

The special variable $\mathbf{r}$ may appear in both pre- and postconditions. In preconditions it references some initial value, in postconditions the result of the last executed expression. Note that it is important that $\mathbf{r}$ can appear in preconditions. Otherwise the weakest precondition $WP(\mathbf{r}, \mathbf{r} = null)$ would not be expressible which would induce incompleteness.

Definition 20. *The proof system $\mathcal{H}_d$ consists of the following rules:*

AXIOM: VAR (includes the case of $v \equiv \mathbf{self}$)

$$\{p[\mathbf{r} := v]\}v\{\mathbf{tags} \wedge p\}$$

AXIOM: IVAR

$$\{p[\mathbf{r} := \mathbf{self}.@\mathrm{v}]\}@v\{\mathbf{tags} \wedge p\}$$

RULE: ASGN (both normal and instance variables)

$$\frac{\{p\}e\{\mathbf{tags} \wedge q[v := \mathbf{r}]\}}{\{p\}v := e\{\mathbf{tags} \wedge q\}} \text{ where } v \in \mathfrak{V}$$

This rule splits the evaluation of an assignment into two parts: First, the expression e is evaluated (premise) resulting in a state $\sigma \models q[v := \mathbf{r}]$ and then, the value of $\sigma(\mathbf{r})$ is assigned to the variable v, resulting in a state $\sigma' \models q$. Since the latter part is terminating, type-safe and fail-safe, establishing these properties for e implies that they hold for the entire assignment.

AXIOM: CONST

$$\{p[\mathbf{r} := null]\}\mathrm{null}\{\mathbf{tags} \wedge p\}$$

Note that the axioms VAR, IVAR and CONST are not part of $\mathcal{H}_s$. Instead of reasoning about variables and constants, statically typed Hoare logic uses the shortcut-rule ASGN to reason about entire expressions.

RULE: COND

$$\frac{\begin{array}{c}\{p\}e\{\mathbf{tags} \wedge r \wedge \mathbf{failsafe} \rightarrow \mathbf{r} \neq \mathit{null} \wedge \mathbf{typesafe} \rightarrow (\mathbf{r} \neq \mathit{null} \rightarrow \mathbb{B}(\mathbf{r}))\} \\ \{r \wedge \mathbb{B}(\mathbf{r}, \mathit{true})\}S_1\{\mathbf{tags} \wedge q\} \\ \{r \wedge \mathbb{B}(\mathbf{r}, \mathit{false})\}S_2\{\mathbf{tags} \wedge q\}\end{array}}{\{p\}\ \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ \{\mathbf{tags} \wedge q\}}$$

Here, the intermediate assertion r describes the state σ reached after evaluating the condition e. This state is important since the value of $\sigma(\mathbf{r})$ decides which branch is evaluated next.

RULE: LOOP

$$\frac{\begin{array}{c}\{p\}e\{\mathbf{tags} \wedge p' \wedge \mathbf{failsafe} \rightarrow \mathbf{r} \neq \mathit{null} \wedge \mathbf{typesafe} \rightarrow (\mathbf{r} \neq \mathit{null} \rightarrow \mathbb{B}(\mathbf{r}))\} \\ \{p' \wedge \mathbb{B}(\mathbf{r}, \mathit{true})\}S\{\mathbf{tags} \wedge p\} \\ \{p' \wedge \mathbb{B}(\mathbf{r}, \mathit{true}) \wedge r(z)\}S; e\{p' \wedge \mathbf{terminates} \rightarrow \forall z' : \mathbb{N} \bullet r(z') \rightarrow z' < z\}\end{array}}{\{p\}\ \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}\ \{\mathbf{tags} \wedge \exists b : \mathbb{O} \bullet p'[\mathbf{r} := b] \wedge \mathbb{B}(b, \mathit{false}) \wedge \mathbf{r} = \mathit{null}\}}$$

where b is a logical variable of type $\mathbb{B}$, z is a logical variable of type $\mathbb{N}$ that does not appear in p, p', e or S, $r(z)$ is a predicate with z among its free variables such that $\forall\sigma \bullet \sigma \models p' \rightarrow \exists z' : \mathbb{N} \bullet r(z')$ and $r(z')$ is the result of substituting z' for z in $r(z)$.

Note that, contrary to statically typed Hoare logic, this rule needs two loop invariants (p and p') to capture the state before and after evaluating the condition e, since e could be side-effecting. Also note that the way termination is ensured using the loop variant $r(z)$ differs significantly from statically typed Hoare logic. The reasons for this will become clear in Chapter 5.

RULE: INST

$$\frac{\begin{array}{c}\{p_i\}e_i\{\mathbf{tags} \wedge p_{i+1}[v_i := \mathbf{r}]\}\text{ for } i \in \mathbb{N}_n \\ \{p_{n+1}\}v_0.m(v_1, ..., v_n)\{\mathbf{tags} \wedge q\}\end{array}}{\{p_0\}e_0.m(e_1, ..., e_n)\{\mathbf{tags} \wedge q\}}$$

where the v_i are fresh local variables that do not occur in any e_j for all $i, j \in \mathbb{N}_n$.

RULE: REC

$$\frac{\begin{array}{c}A \vdash \{p\}S\{\mathbf{tags} \wedge q\}, \\ A' \vdash \{p_i \wedge r_i(z)\}\mathbf{begin\ local\ self}, \overrightarrow{u_i} := l_i, \overrightarrow{v_i}; S_i\ \mathbf{end}\{\mathbf{tags}_i \wedge q_i\}, i \in \mathbb{N}_n^1 \\ p_i \rightarrow (\mathbf{failsafe} \rightarrow l_i \neq \mathit{null} \wedge \mathbf{typesafe} \rightarrow l_i \neq \mathit{null} \rightarrow l_i.@\mathbf{c} = \theta_{C_i}), i \in \mathbb{N}_n^1\end{array}}{\{p\}S\{\mathbf{tags} \wedge q\}}$$

where $\mathbf{method}\ m_i(\overrightarrow{u_i})\{S_i\} \in \mathfrak{M}_{C_i}$, $A = \{\{p_i\}l_i.m_i(\overrightarrow{v_i})\{q_i\} \mid i \in \mathbb{N}_n^1\} \cup \{\{p_{\mathbf{tag},i}\}l_i.m_i(\overrightarrow{v_i})\{\mathbf{tag}\} \mid i \in \mathbb{N}_n^1, \mathbf{tag} \in \mathcal{T}ags\}$, $A' = \{\{p_i \wedge (\mathbf{terminates} \rightarrow \forall z' : \mathbb{N} \bullet r_i(z') \rightarrow z' < z)\}l_i.m_i(\overrightarrow{v_i})\{q_i\} \mid i \in \mathbb{N}_n^1\} \cup \{\{p_{\mathbf{tag},i} \wedge (\mathbf{terminates} \rightarrow \forall z' : \mathbb{N} \bullet r_i(z') \rightarrow z' < z)\}l_i.m_i(\overrightarrow{v_i})\{\mathbf{tag}\} \mid i \in \mathbb{N}_n^1, \mathbf{tag} \in \mathcal{T}ags\}$, z is a logical variable of type $\mathbb{N}$ that does not occur in p_i, q_i and S_i for $i \in \mathbb{N}_n^1$ and is treated in the proofs as a constant, $r_i(z)$ for $i \in \mathbb{N}_n^1$ are predicates with z among their free variables such that $\forall\sigma \bullet \sigma \models p_i \rightarrow \exists z' : \mathbb{N} \bullet r_i(z')$ for all $i \in \mathbb{N}_n^1$ and $r_i(z')$ denotes the result of substituting z' for z in $r_i(z)$.

RULE: EQUAL

$$\frac{\begin{array}{c}\{p\}e_1\{\mathbf{tags} \wedge r[v_1 := \mathbf{r}]\} \\ \{r\}e_2\{\mathbf{tags} \wedge q[v_2 := \mathbf{r}]\}\end{array}}{\{p\}e_1 == e_2\{\mathbf{tags} \wedge q \wedge d\}}$$

where $d \equiv (\mathbb{B}(\mathbf{r}, true) \wedge v_1 = v_2) \vee \ (\mathbb{B}(\mathbf{r}, false) \wedge v_1 \neq v_2)$.

RULE: TYPECHECK

$$\frac{\{p\}e\{\mathbf{tags} \wedge q[v := \mathbf{r}]\}}{\{p\}e \text{ is\_a? } \{C\}\{\mathbf{tags} \wedge q \wedge d\}}$$

where $d \equiv (\exists b : \mathbb{B} \bullet \mathbb{B}(\mathbf{r}, b) \wedge b \leftrightarrow [\![v]\!] \in \{C\})$

The remaining rules are identical to those of the tagged, statically typed Hoare logic $\mathcal{H}_s$ given in Section 3.3.

The fact that dyn-expressions have side effects is mirrored in several rules. Like their corresponding rules in the operational semantics, the usual axiom for assignment is turned into a rule and the COND and LOOP rules both evaluate the condition before branching on its result in an intermediate state.

The rules PASGN, BLCK, INST and REC are needed to handle method calls. After handling side effecting expressions in arguments beforehand (INST) and ensuring that methods are only called on receivers supporting them (last premise of REC), method calls are assumed to satisfy the same properties as a block executing the body of the called method in an environment with local variables suitably initialized (BLCK,PASGN).

The LOOP and REC rules feature a novel form of loop variants / recursion bound. The basic idea is to use a predicate $r(z)$ instead of the usual integer expression t in order to allow quantification within loop variants / recursion bounds. While this was primarily introduced to circumvent a common incompleteness issue in Hoare logics for total correctness (see proof of Theorem 8 for details), note that it also allows using mapping predicates directly in loop variants / recursion bounds, i.e., proving

$$\{\mathbb{N}(i)\}\mathbf{while}\ i > 0\ \mathbf{do}\ i := i - 1\ \mathbf{done}\{\mathbf{terminates}\}$$

with $r(z) \equiv \mathbb{N}(i, z)$.

5. Proof Theory for Dynamically Typed Programs

5.1. Soundness

Soundness of the proof system $\mathcal{H}_d$ for **dyn** follows from a standard inductive argument. We will only present the case for the LOOP rule as the idea of using a predicate r as a loop variant for total correctness is novel.

Induction Hypothesis: $\vdash_{\mathcal{H}_d} \{p\}S\{\mathbf{tags} \wedge q\}$ implies $\models \{p\}S\{\mathbf{tags} \wedge q\}$ for all assertions p and q, all **dyn** statements S and all $\mathbf{tags} \subseteq \mathcal{T}ags$.

Induction Step:
We consider the case of a loop **while** e **do** S **done** and distinguish the following cases according to the chosen notion of correctness.
Partial Correctness: Given $\vdash_{\mathcal{H}_d} \{p\}e\{\mathbf{tags} \wedge p'\}$ and $\vdash_{\mathcal{H}_d} \{p' \wedge \mathbb{B}(\mathbf{r}, true)\}S\{\mathbf{tags} \wedge p\}$, by the induction hypothesis $\models \{p\}e\{\mathbf{tags} \wedge p'\}$ and $\models \{p' \wedge \mathbb{B}(\mathbf{r}, true)\}S\{\mathbf{tags} \wedge p\}$ follow. We have to show that the conclusion of the LOOP rule is valid, i.e.,

$$\models \{p\}\mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}\{\mathbf{tags} \wedge \exists b : \mathbb{O} \bullet p'[\mathbf{r} := b] \wedge \mathbb{B}(b, false) \wedge \mathbf{r} = null\}.$$

Hence, when executing the program **while** e **do** S **done** in a state $\sigma \models p$, the operational semantics will first apply rule 9 yielding the configuration

$$\langle \mathbf{if}\ e\ \mathbf{then}\ S; \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{done}\ \mathbf{else}\ \text{null}\ \mathbf{end}, \sigma\rangle,$$

then apply whatever rules are neccessary to evaluate $\langle e, \sigma\rangle$ to a configuration $\mathbf{final}\langle\tau\rangle$. From $\models \{p\}e\{p'\}$ we deduce $\tau \models p'$. Furthermore, the operational semantics uses rules 5-7 to branch on the value of $\tau(\mathbf{r})$. Since for partial correctness we are only interested in normal program termination, the cases yielding **fail** or **typeerror** will be handled below. Hence we are left with two subcases:

1) $\tau \models \mathbb{B}(\mathbf{r}, true)$: In this case, rule 5a) is the only one applicable and $\langle S, \tau\rangle$ will be evaluated next. From $\{p' \wedge \mathbb{B}(\mathbf{r}, true)\}S\{p\}$ we can deduce that the resulting state σ' will again satisfy p. We are hence again in a configuration $\langle$while e do S done$, \sigma'\rangle$ with $\sigma' \models p$. With σ' taking the role of σ and with regard to the operational semantics, this configuration offers exactly the same options as the one before applying rule 9. Now this loop in the transition system raises the possibility of divergence. However, for partial correctness we may disregard this possibility, as we only provide guarantees for finite computations. The case of divergence will be discussed below.

2) $\tau \models \mathbb{B}(\mathbf{r}, false)$: In this case, rule 5b) is the only one applicable and $\langle \text{null}, \tau \rangle$ is the only statement left to evaluate. Applying rule 1 leaves us in a configuration $\mathbf{final}\langle \tau' \rangle$ with $\tau' \models \exists b : \mathbb{O} \bullet p'[\mathbf{r} := b] \land \mathbb{B}(b, false) \land \mathbf{r} = null$ where b represents the result of e on the last iteration while $\mathbf{r}$ represents the result of the entire loop. As this is the only way for our program to terminate normally, $\models \{p\}\text{while } e \text{ do } S \text{ done}\{\exists b : \mathbb{O} \bullet p'[\mathbf{r} := b] \land \mathbb{B}(b, false) \land \mathbf{r} = null\}$ holds.
Termination: For partial correctness, the premise $\{p' \land \mathbb{B}(\mathbf{r}, true)\}S; e\{p'\}$ can be derived from the two other premises by an application of the SEQ rule. Since the predicate $r(z)$ can be chosen to be $r(z) \equiv z = z$ and the part of the postcondition implied by **terminates** can be omitted when **terminates** $\notin$ **tags**, it follows that the third premise does not strengthen the premises in any way. However, for total correctness we have **terminates** $\in$ **tags** and hence it is mandatory to provide the additional predicate $r(z)$ with z among its free variables, such that $\forall \sigma \bullet \sigma \models p' \rightarrow \exists z' : \mathbb{N} \bullet r(z')$ and $\models \{p' \land \mathbb{B}(\mathbf{r}, true) \land r(z)\}S; e\{p' \land \forall z' : \mathbb{N} \bullet r(z') \rightarrow z' < z\}$. $r(z)$ may be understood as mapping states to sets of natural number values for z. The first requirement thus ensures that the "mapping" $r(z)$ is total on all states in $[\![p']\!]$, while the second requires the loop body S together with the condition e to decrease its supremum. Since the state τ reached after evaluating e the first time satisfied p', by the (conditional) totality of $r(z)$ we deduce that there must be an "initial" non-empty set Z of natural numbers such that for all $z_i \in Z$, $\tau \models r(z_i)$ holds. Let z_{max} be the supremum of Z. Then, since z_{max} is a natural number and it is required to decrease strictly on each loop iteration and all natural numbers are ≥ 0, there can only be a finite number of iterations satisfying the second requirement. Consequently, the loop has to terminate after finitely many iterations.

Failsafety: A failure might occur either in evaluating e or S or by rule 6 when e evaluates to *null*. Requiring e and S to be both **failsafe** as well as $\{p\}e\{\mathbf{failsafe} \rightarrow \mathbf{r} \neq null\}$ (implied by the first premise) hence covers all these cases.

Typesafety: The same argument as for failsafety applies here, only with rule 7 and the requirement $\mathbf{typesafe} \rightarrow (\mathbf{r} \neq null \rightarrow \mathbb{B}(\mathbf{r}))$ (implied by the first premise) instead of $\mathbf{failsafe} \rightarrow \mathbf{r} \neq null$. Note that the case for failure is intentionally left open as typesafe partial correctness only needs to guarantee the absense of type errors and too strong a premise would induce incompleteness. □

5.2. Completeness

In this section, we will prove the axiomatic semantics $\mathcal{H}_d$ of **dyn** (relative) complete [22] with respect to its operational semantics following the seminal completeness proof of Cook and Gorelick [22, 33] as well as its extension to OO-programs due to de Boer and Pierik [14]. That is, given a closed program π with a finite set of class definitions, we prove that $\vDash \{p\}\pi\{q\}$ implies $\vdash_{\mathcal{H}_d, \mathcal{T}} \{p\}\pi\{q\}$ assuming a complete proof system $\mathcal{T}$ for the assertion language **AL**.

We will start by summarizing previous completeness proofs by the authors mentioned above, along with a discussion of the problems encountered when applying those proofs

directly to dynamically typed languages like **dyn**.

5.2.1. Non-Recursive Programs

(Relative) completeness of Hoare logic was first proven by Cook [22] for non-recursive procedural programs. Due to him is the insight that a valid correctness formula $\vDash \{p\}S\{q\}$ may fail to have a derivation in a Hoare logic (H, T), where T is a proof system for the assertion language used, either

- due to incompleteness of T or
- due to an intermediate assertion / loop invariant not expressible in the assertion language or
- due to incompleteness of H.

While the first point is the reason one can only hope to prove the proof system (H, T) complete relative to T, the second is the reason completeness proofs have ever since included an argument for the expressivity of the assertion language.

5.2.2. Recursive Programs

Gorelick [33] extended Cook's work to recursive programs. The central idea behind his proof was the notion of *most general correctness formulas* (MGF's)

$$\{\overline{x} = \overline{z}\}\ P_i\ \{SP(\overline{x} = \overline{z}, S_i)\}$$

for all procedures P_i with procedure bodies S_i where $\overline{x} = x_1, ..., x_n$ denotes the sequence of all program variables, $\overline{z} = z_1, ..., z_n$ is a corresponding sequence of logical variables that do NOT occur in ANY procedure body, thus "freezing" the entire initial state and $SP(p, S)$ is the strongest postcondition of a precondition p with respect to the statement S. Gorelick also introduced the REC rule (in the form commonly used in statically typed Hoare logic (see f.i. [5])) for reasoning about mutually recursive procedures and was the first to notice that achieving completeness requires that the assertion language is able to capture every aspect of a program state in logical variables, in order to "freeze" this information during program execution and allow the postcondition to compare the initial- to the final state (which is the central idea behind his MGFs).

5.2.3. Object-Oriented Programs

Later, de Boer and Pierik [14] generalized the results to programs with dynamically created values (pointers or objects). To this purpose, they introduced the substitution for object creation (see Appendix A). Also, they pointed out that in OO-contexts, freezing a program state additionally requires freezing the internal states of all objects existing in that state, thus necessitating a more sophisticated freezing-strategy.

5.2.4. Freezing the Initial State

While the approach from [14] stores objects and the values of their instance variables class-wise, which is difficult in a dynamically typed language like **dyn**, the basic idea is fortunately still applicable. We use a logical variable *obj* of type $\mathbb{O}^*$ to store a (finite) sequence of all existing objects:

$$all(obj) \equiv \forall o : \mathbb{O} \bullet \exists i : \mathbb{N} \bullet i < |obj| \wedge obj[i] = o$$

Since *obj* establishes a bijection from natural numbers to objects, it allows encoding states as sequences of natural numbers.

For convenience, we introduce a polymorphic[1] function *pos* that maps a sequence and an element to the smallest index (= position) that element occupies within the sequence. It thus satisfies

$\forall \tau : \mathbb{T} \bullet \forall e : \tau, s : \tau^* \bullet s[pos(s, e)] = e$

For a given program π, we introduce an enumeration $ivar \in ((var(\pi) \cup change(\pi)) \cap \mathfrak{V}_I)^*$ of all instance variables occurring in π and define the following predicate for freezing states:

$$\begin{aligned} code(\overline{x}, obj, \varsigma) \equiv\ & |\varsigma| = |ivar| + 1 \wedge |\varsigma[0]| = |\overline{x}| \wedge \\ & \forall i : \mathbb{N} \bullet i < |\overline{x}| \rightarrow \varsigma[0][i] = pos(obj, x_i) \wedge \\ & \forall i, j : \mathbb{N} \bullet (i < |ivar| \wedge j < |obj|) \rightarrow ivar[i] = @v \wedge obj[j] = o \rightarrow \\ & \quad \varsigma[i+1][j] = pos(obj, o.@v) \end{aligned}$$

where $\overline{x} = x_1, ..., x_n$ is a sequence of all local variables. The predicate $code(\overline{x}, obj, \varsigma)$ uses the sequence *obj* to capture the state of all local variables in $\overline{x}$ as well as the internal states of all objects existing in the current state in the frozen state ς of type $\mathbb{N}$ (but encoding a sequence of sequences of natural numbers). The structure of such a frozen state ς is depicted in Figure 5.1: The first number $\varsigma[0]$ encodes a sequence of natural numbers representing the objects referenced by the variables in $\overline{x}$, while the remaining numbers $\varsigma[i]$ for $i \in \mathbb{N}^1_{|ivar|+1}$ encode sequences of natural numbers such that $\varsigma[i][j]$ represents the object referenced by the instance variable $ivar[i-1]$ of the jth object (the unique object o such that $obj[j] = o$). Note that ς captures the internal states of all existing objects without referencing any of them.

Also note that this is indeed satisfiable for all states as $null \in \mathbb{O}$. Furthermore, for a given program π, we say that ς *encodes* σ and write

$$\sigma \sim \varsigma \quad \text{iff} \quad \sigma \models all(obj) \wedge code(\overline{x}, obj, \varsigma)$$

with $\{\overline{x}\} = (var(\pi) \cup change(\pi)) \cap \mathfrak{V}_L$ being a finite sequence of all local variables occuring in π.

Lemma 3 (Left-Totality of $\sim$)**.** $\forall \sigma : \Sigma \bullet \exists \varsigma : \mathbb{N} \bullet \sigma \sim \varsigma$.

Finally, we are ready to define a predicate transformer Θ (called the "freezing function" in [14]). However, while in their work, Θ also bounds all quantification and

[1] We use the polymorphic version for the sake of readability although the type system of **AL** does not allow polymorphism. However, polymorphic functions can be emulated using one version for each element type.

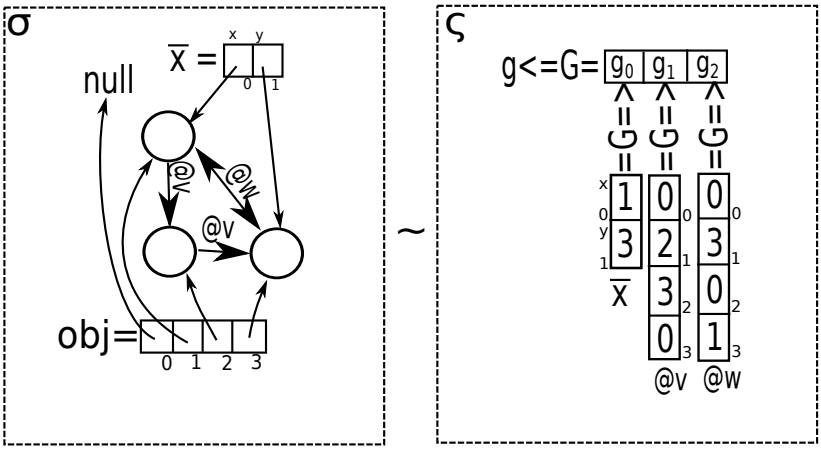


Figure 5.1.: Example of a frozen state ς encoding a state σ ($\sigma \sim \varsigma$). Here, G denotes Gödelization, an operation encoding a finite list of natural numbers into a single one.

replaces instance variable dereferencing by lookups in sequences, we additionally translate all object expressions into expressions of type $\mathbb{N}$ to allow simulating computations directly on the frozen states.

We now define the predicate transformer $\Theta^{\overline{x}}_{obj}(\varsigma)$, which is a postfix-operator applicable to assertions that depends on a sequence of local variables $\overline{x}$ and a frozen state ς to transform a given assertion into a form that operates on ς instead of the actual program variables (freezing). It is hence defined by induction over the structure of assertions. We have the following main cases:

- $(l.@v)\Theta^{\overline{x}}_{obj}(\varsigma) \equiv \varsigma[i_{@v}+1][l\Theta^{\overline{x}}_{obj}(\varsigma)]$ where $@v = iv_{i_{@v}}$
- $\mathrm{u}\Theta^{\overline{x}}_{obj}(\varsigma) \equiv \varsigma[0][i_x]$ where $\mathrm{u} = x_{i_x}$
- $u\Theta^{\overline{x}}_{obj}(\varsigma) \equiv u'$ where u is a logical variable of type $\mathbb{O}$ and u' is a fresh logical variable of type $\mathbb{N}$
- $(l_1 = l_2)\Theta^{\overline{x}}_{obj}(\varsigma) \equiv l_1\Theta^{\overline{x}}_{obj}(\varsigma) = l_2\Theta^{\overline{x}}_{obj}(\varsigma)$ where l_1 and l_2 are of type $\mathbb{O}$.
- $(\exists o : \mathbb{O} \bullet p)\Theta^{\overline{x}}_{obj}(\varsigma) \equiv (\exists o' : \mathbb{N} \bullet 0 \leq o' < |obj| \to p\Theta^{\overline{x}}_{obj}(\varsigma))$

Like the Θ in [14], our $\Theta^{\overline{x}}_{obj}(\varsigma)$ hence satisfies the following property.

Theorem 4 (Invariance)**.** $\vdash \{p\Theta^{\overline{x}}_{obj}(\varsigma)\}S\{p\Theta^{\overline{x}}_{obj}(\varsigma)\}$ *for all statements* S*, assertions* p *and frozen states* ς *as long as* $\overline{x}$ *contains all program variables used in* p *(*$free(p) \subseteq \{\overline{x}\}$*).*

It can hence replace Θ in the remaining argument. Note that the truth value of $p\Theta^{\overline{x}}_{obj}(\varsigma)$ depends only on ς and is hence independent of any particular state. We hence write $\models p\Theta^{\overline{x}}_{obj}(\varsigma)$ if its truth value is true. Also observe

Lemma 4 (Freezing)**.** $\sigma \models q$ *iff* $\sigma \sim \varsigma \wedge \models q\Theta^{\overline{x}}_{obj}(\varsigma)$ *for all* $\sigma \in \Sigma$ *and* $q \in Asrt$*.*

Proof. By induction over the structure of q. □

It follows that a frozen state ς contains all information about σ when $\sigma \sim \varsigma$ holds.

5.2.5. Expressivity

Cook [22] first discussed the importance of an expressive assertion language for the completeness of a Hoare logic. In essence, the assertion language must be able to express the strongest postcondition $SP(p, S)$ for all statements S and preconditions p.

In the last section, we already established that it is possible to capture all information about a state in a single natural number. Then, we consider a predicate $comp_S$ of type $\mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ simulating **dyn** computations on such frozen states and note that, since such computations are by definition computable, they can be defined as a μ-recursive function.

By Theorem 2, it is hence expressible in our assertion language and we can use it within our assertions without any loss of generality. To formalize the idea that $comp_S$ simulates **dyn** computations on frozen states, we stipulate

Definition 21. $comp_S = \{(\varsigma, \varsigma') \mid \forall \sigma, \sigma' \bullet (\sigma \sim \varsigma \wedge \sigma' \sim \varsigma') \rightarrow \sigma' \in \mathcal{M}[\![S]\!](\sigma)\}$

Using $comp_S$ we can show the following:

Theorem 5 (Definability of Weakest Preconditions)**.** *For all postconditions* q *and statements* S*, the precondition*

$$p \equiv \forall \varsigma, \varsigma' \bullet (all() \wedge code(\overline{x}, \varsigma') \wedge comp_S(\varsigma', \varsigma)) \rightarrow q\Theta^{\overline{x}}_{obj}(\varsigma)$$

satisfies $[\![p]\!] = \{\sigma \mid \mathcal{M}[\![S]\!](\sigma) \subseteq [\![q]\!]\}$*.*

Proof. We have to prove the equality $LHS = RHS$ where $LHS \equiv [\![p]\!]$ and $RHS \equiv \{\sigma \mid \mathcal{M}[\![s]\!](\sigma) \subseteq [\![q]\!]\}$. We will first prove the direction $LHS \subseteq RHS$ and then turn to the converse question.

1) $LHS \subseteq RHS$: Assuming a state $\sigma \in LHS$, then by left-totality of $\sim$, there is a ς' such that $\sigma \sim \varsigma'$. Furthermore, from $\sigma \models \forall \varsigma \bullet comp_S(\varsigma', \varsigma) \rightarrow q\Theta^{\overline{x}}_{obj}(\varsigma)$ and Definition 21 we deduce that every state $\sigma' \in \mathcal{M}[\![s]\!](\sigma)$ has a ς satisfying $\sigma' \sim \varsigma$ as well as $comp_S(\varsigma', \varsigma)$. Since all premises of the implication on the left-hand side are satisfied, $\models q\Theta^{\overline{x}}_{obj}(\varsigma)$ must hold as well. Note that the latter two are properties of ς and ς' rather than any particular state. Using Lemma 4 we can then deduce $\sigma' \models q$ and since our only assumption about σ' is that it is a post-state of σ, it follows that $\mathcal{M}[\![s]\!](\sigma) \subseteq [\![q]\!]$ and hence that $\sigma \in RHS$.

2) $RHS \subseteq LHS$: Assume $\sigma \in RHS$. σ is hence an initial state and all its post-states $\sigma' \in \mathcal{M}[\![s]\!](\sigma)$ satisfy the assertion q. Then, by left-totality of $\sim$, there is a

frozen state ς' such that $\sigma \sim \varsigma'$ and for every post-state σ' there is a frozen state ς such that $\sigma' \sim \varsigma$. Now, by Definition 21, every pair of (frozen) pre- and post-state $(\varsigma', \varsigma) \in comp_S$. Also, since the post-states σ' satisfy q and $\sigma' \sim \varsigma$, by Lemma 4 we know that $\models q\Theta^{\overline{x}}_{obj}(\varsigma)$ holds. Therefore, the entire assertion p is true in σ and hence $\sigma \in LHS$. □

Predicates $terminates_S(\varsigma), typesafe_S(\varsigma)$ and $failsafe_S(\varsigma)$ can be defined in a similar manner as $comp_S$ and allow for deriving above result for different notions of correctness (tags). Since definability of weakest preconditions is equivalent to the definability of strongest postconditions [63], we have

Theorem 6 (Expressiveness)**.** *The assertion language **AL** is expressive with respect to its standard interpretation and the programming language **dyn**.*

5.2.6. Completeness for Statements

As in the work of Cook [22] and Gorelick [33], the core of our completeness proof consists of an induction over the structure of a statement S. Since several of our rules deviate from our predecessors, we need to exchange these cases in the argument. We will concentrate on the most interesting cases.

Induction Basis:

- $S \equiv$ *null*: Assume $\models \{p\}\text{null}\{q\}$. Then, by the operational semantics, $p \to q[\mathbf{r} := \mathit{null}]$ must also be true. It is hence derivable in $\mathcal{T}$ and the desired result follows from the CONST axiom followed by applying the rule of consequence (CONS). *Typesafety, Failsafety & Termination:* The CONST axiom allows deriving any combination of tags desired.
- $S \equiv$ u: Assume $\models \{p\}\text{u}\{q\}$. Then, by the operational semantics, $p \to q[\mathbf{r} := \text{u}]$ must also be true. It is hence derivable in $\mathcal{T}$ and the desired result follows from the VAR axiom followed by applying the rule of consequence (CONS).

  Typesafety, Failsafety & Termination: The VAR axiom allows deriving any combination of tags desired.
- $s \equiv$ @v: Just like the case for u, applying IVAR instead of VAR.

Induction Hypothesis: $\models \{p\}S\{q\} \to \vdash_{\mathcal{H}_d,\mathcal{T}} \{p\}S\{q\}$ for all assertions p, q and all statements S of a program π containing no recursive method calls.

Induction Step:

- $S \equiv \text{u} := e$: Assume $\models \{p\}S\{\mathbf{tags} \wedge q\}$. Then, according to the operational semantics, also $\models \{p\}e\{\mathbf{tags} \wedge q[\text{u} := \mathbf{r}]\}$. By the induction hypothesis, it is hence derivable. An application of the rule ASGN derives the desired result.
- $S \equiv S_1; S_2$: Assume $\models \{p\}S_1; S_2\{\mathbf{tags} \wedge q\}$. Then by the expressibility of the assertion language, there is an intermediate assertion r such that $\models \{p\}S_1\{\mathbf{tags} \wedge r\}$ and $\models \{r\}S_2\{\mathbf{tags} \wedge q\}$. Hence by the induction hypothesis, both are derivable and an application of the rule SEQ derives the desired result.

- $S \equiv \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}$: Assume $\models \{p\}\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}\{\textbf{tags} \wedge q\}$. Then, by the expressiveness of the assertion language and the operational semantics, there is an intermediate assertion r such that $\models \{p\}e\{\textbf{tags} \wedge r\}$, $\models \{r \wedge \mathbb{B}(\mathbf{r}, true)\}S_1\{\textbf{tags} \wedge q\}$ and $\models \{r \wedge \mathbb{B}(\mathbf{r}, false)\}S_2\{\textbf{tags} \wedge q\}$ hold and are hence derivable by the induction hypothesis. Now an application of the rule COND derives the desired result.

  Failsafety: Since above argumentation already ensured that e, S_1 and S_2 are all failsafe, the only additional requirement is $\{p\}e\{\mathbf{r} \neq null\}$. However, since the case $\mathbf{r} = null$ leads to failure in the operational semantics, this must hold for any execution of S in order to be failsafe and hence must be derivable by the induction hypothesis.

  Typesafety: The same argumentation as for failsafety applies here, only the additional requirement is $\{p\}e\{\mathbf{r} \neq null \rightarrow \mathbb{B}(\mathbf{r})\}$. Note that the case of $\mathbf{r} = null$ can be deliberately allowed, since it leads to a failure in the operational semantics and thus does not affect typesafety.

- $S \equiv \textbf{while } e \textbf{ do } S_1 \textbf{ done}$: Assume $\models \{p\}\textbf{while } e \textbf{ do } S_1 \textbf{ done}\{\textbf{tags} \wedge q\}$. Then, by the standard argument for while loops due to Cook [22] (and explained particularly well by Apt [4]), the expressiveness of the assertion language and the operational semantics, there are two assertions i and i' such that $p \rightarrow i$, $\models \{i\}e\{\textbf{tags} \wedge i'\}$, $\models \{i' \wedge \mathbb{B}(\mathbf{r}, true)\}S_1\{\textbf{tags} \wedge i\}$ and $\exists b : \mathbb{O} \bullet i'[\mathbf{r} := b] \wedge \mathbb{B}(b, false) \wedge \mathbf{r} = null \rightarrow q$ hold and are hence derivable by the induction hypothesis and the completeness of $\mathcal{T}$. While i is the loop invariant of S, i' is an intermediate state neccessary because in **dyn**, e could have side-effects. Now, an application of the LOOP rule followed by the rule of consequence derives the desired result.

  Termination: Assuming $\models \{p\}\textbf{while } e \textbf{ do } S_1 \textbf{ done}\{\textbf{terminates} \wedge q\}$, then there is a μ-recursive function v simulating the execution of S using $comp_S$ from a (frozen) initial state ς and determining the least number of iterations it takes to reach a frozen state ς' from the ς, such that e evaluates to false in ς'. Note that by Theorem 2 v can be expressed in **AL**. Also, by our assumption that the loop S terminates, the function v is well-defined on all (frozen versions of) states in p' and thus $r(z) \equiv all() \wedge code(\overline{x}, \varsigma) \wedge z = v(\varsigma)$ is a canonical loop variant satisfying $\forall \sigma \bullet \sigma \models p' \rightarrow \exists z' : \mathbb{N} \bullet r(z')$. Since $v(\varsigma)$ determines the number of iterations until reaching a target state, executing $S; e$ clearly decreases it and thus $\models \{p' \wedge \mathbb{B}(\mathbf{r}, true) \wedge r(z)\}S; e\{p' \wedge \forall z' : \mathbb{N} \bullet r(z') \rightarrow z' < z\}$ holds. By the induction hypothesis, it is thus derivable. An application of the LOOP rule derives the desired result.

  Failsafety & Typesafety: Exactly the same argument as for conditionals applies here as well.

Remark regarding Termination: As usual for reasoning about termination, the LOOP rule is equipped with a so-called loop-variant (activated (implied) by

terminates). Usually[2], this variant take the form of an integer expression t whose value a) must be > 0 whenever the loop is entered (thus forcing termination when reaching zero) and b) must decrease on every iteration. Note that this methodology syntactically restricts the loop variant to be an integer expression of the assertion language. Now, as observed by Apt, De Boer and Olderog in [5, page 86], this method introduces incompleteness in the case of total correctness, for the integer expressions of the assertion language **AL** are insufficient for expressing all necessary loop-variants since while-loops and recursive methods allow **dyn** programs to calculate any μ-recursive function and hence obviously also to bound the number of loop iterations by any μ-recursive function, while the set of integer operations available in **AL** is quite limited (for instance lacking exponentiation). We circumvented this problem by using a new form of loop-variants in $\mathcal{H}_d$, which allows the use of quantifiers. The old form (used f.i. in the LOOP rule from $\mathcal{H}_s$) used a logical variable z of type $\mathbb{N}$ to store the value of t before a loop iteration ($t = z$ in the precondition) and compare it to the new value in the postcondition ($t < z$). Our new form in $\mathcal{H}_d$ uses a predicate $r(z)$ with z among its free variables instead of $t = z$ and the logical expression $\forall z' : \mathbb{N} \bullet r(z') \to z' < z$ where $r(z')$ denotes the result of substituting z' for z in r instead of $t < z$. Firstly, observe that this is a conservative extension as one may set $r \equiv t = z$ for some integer expression t. Secondly, note that by Theorem 2, r may compute any μ-recursive function and is thus contrary to integer expressions able to express any function computable by **dyn** programs (including f.i. exponentiation).

5.2.7. Completeness for Recursive Methods

The methodology for proving a Hoare logic complete for recursive procedures by using most general correctness formulas is due to Gorelick [33]. It was extended to OO-programs by De Boer and Pierik [14].

A curious implication of dynamic dispatch under dynamic typing is that the lack of type information prohibits pinpointing the exact target of a method call. For instance, the weakest precondition of the call x.size() with respect to the postcondition $\mathbb{N}(\mathbf{r}, 5)$ must include all possibilities like the case of the variable x referring to a string of length 5 as well as x referring to a list of size 5. In general, the weakest precondition of a method call $l.\mathrm{m}(v_1, ..., v_n)$ is the disjunction of all weakest preconditions derivable as described in the proof of Theorem 7 from the most general correctness formulas of all methods C.m of arity n of all classes $C \in \mathfrak{C}$, each conjoined with the corresponding type assumption $[\![l]\!] \in \{C\}$. Note that this methodology introduces an implicit closed world assumption as it might fail when combining the derived property with a different set of classes. However, we regard this problem as one of modularity rather than completeness and thus as out of scope.

As our tagged Hoare logic incorporates different notions of correctness, we generalize Gorelick's idea to a set of most general correctness formulas. The most general correctness formulas for a statement S are

$$MGF(S) = \{\{WP(S, init(\varsigma))\}S\{init(\varsigma)\}\} \cup \{\{WP_{\mathbf{tag}}(S, true)\}S\{\mathbf{tag}\} \mid \mathbf{tag} \in \mathcal{T}ags\}$$

[2] see for instance $\mathcal{H}_s$ in Section 3.3

with $init(\varsigma) \equiv all(obj) \wedge code(\overline{x}, obj, \varsigma)$. The reason for this is obvious: From $MGF(S)$, we can deduce $\{WP_{\text{tags}}(S, q)\}S\{\mathbf{tags} \wedge q\}$ for all $\mathbf{tags} \subseteq \mathcal{T}ags$ using the conjunction rule. The converse is not in all cases possible.

The results from Section 5.2.6 imply that the above set can be derived for any **dyn** statement S given that the formulas it contains are true. Should, e.g., S raise a type error on all inputs then $WP_{\text{typesafe}}(S, true) \equiv false$ and $\{false\}S\{\mathbf{typesafe}\}$ is derivable.

Theorem 7 (MGFs)**.** *If* $\models \{p\}S\{\mathbf{tags} \wedge q\}$*, then* $MGF(S) \vdash_{\mathcal{H}_d, \mathcal{T}} \{p\}S\{\mathbf{tags} \wedge q\}$*.*

Proof. Assume $\models \{p\}S\{\mathbf{tags} \wedge q\}$. Then $\{p\}S\{q\}$ and $\{p\}S\{\mathbf{tag}\}$ for all $\mathbf{tag} \in \mathbf{tags}$ are also all true.

1) $\vdash \{p\}S\{q\}$: For technical convenience only we assume that p and q do not contain free occurrences of ς. If they do, these need to be renamed using the substitution rule. By Theorem 4, we have $\{q\Theta^{\overline{x}}_{obj}(\varsigma)\}S\{q\Theta^{\overline{x}}_{obj}(\varsigma)\}$ for some frozen state ς. By the definition of WP, we have $\{WP(S, init(\varsigma))\}S\{init(\varsigma)\}$. An application of the conjunction rule yields

$$\{q\Theta^{\overline{x}}_{obj}(\varsigma) \wedge WP(S, init(\varsigma))\}S\{q\Theta^{\overline{x}}_{obj}(\varsigma) \wedge init(\varsigma)\}.$$

Next, we have to prove $p \to q\Theta^{\overline{x}}_{obj}(\varsigma) \wedge WP(S, init(\varsigma))$. Assume $\sigma \models p$. Then by $\models \{p\}S\{q\}$, for all $\sigma' \in \mathcal{M}[\![S]\!](\sigma)$, we have $\sigma' \models q$. By Lemma 4, we have $\sigma' \models q\Theta^{\overline{x}}_{obj}(\varsigma) \wedge init(\varsigma)$.

From this, we will establish $\sigma \models q\Theta^{\overline{x}}_{obj}(\varsigma)$ by showing that $\sigma \not\models q\Theta^{\overline{x}}_{obj}(\varsigma)$ leads to a contradiction. We are thus assuming $\sigma \not\models q\Theta^{\overline{x}}_{obj}(\varsigma)$. This implies $\sigma \models \neg q\Theta^{\overline{x}}_{obj}(\varsigma)$ and since $\vdash \{\neg q\Theta^{\overline{x}}_{obj}(\varsigma)\}S\{\neg q\Theta^{\overline{x}}_{obj}(\varsigma)\}$ by Theorem 4 and $\models \{\neg q\Theta^{\overline{x}}_{obj}(\varsigma)\}S\{\neg q\Theta^{\overline{x}}_{obj}(\varsigma)\}$ by the soundness of our Hoare logic, it follows that $\sigma' \models \neg q\Theta^{\overline{x}}_{obj}(\varsigma)$, which contradicts our earlier assumption.

Hence, $\sigma \models q\Theta^{\overline{x}}_{obj}(\varsigma)$ and by the definition of WP, $\sigma \models WP(S, init(\varsigma))$. Therefore, $p \to q\Theta^{\overline{x}}_{obj}(\varsigma) \wedge WP(S, init(\varsigma))$ holds and since $q\Theta^{\overline{x}}_{obj}(\varsigma) \wedge init(\varsigma) \to q$ follows directly from Lemma 4, an application of the rule of consequence derives $\{p\}S\{q\}$.

2) $\vdash \{p\}S\{\mathbf{tag}\}$: if this is true, then $p \to WP_{\text{tag}}(S, true)$ must also be true by the definition of weakest preconditions, and hence is derivable by the completeness of $\mathcal{T}$. Since $\{WP_{\text{tag}}(S, true)\}S\{\mathbf{tag}\} \in MGF(S)$, an application of the consequence rule derives the desired result.

3) $\vdash \{p\}S\{\mathbf{tags} \wedge q\}$: One application of the conjunction rule per tag in **tags** completes the proof. □

Finally, since our recursion rule is in principle identical to the one devised by Gorelick [33] for this purpose, we are now able to apply the same inductive argument as in [33] for proving our Hoare logic complete for recursive methods.

Lemma 5. *Let $M_i \equiv l_i.m_i(\overrightarrow{v_i})$ denote the ith (possibly recursive) method call occurring in a closed **dyn** program π and let $A = \bigcup_{i=1}^{n} MGF(M_i)$ be the set of most general correctness formulas about all method calls $M_1, ..., M_n$ in π. Then for all statements S of π and all assertions p and q:*

$$\textit{If } \models \{p\}S\{\mathbf{tags} \wedge q\}, \textit{ then } A \vdash_{\mathcal{H}_d,\mathcal{T}} \{p\}S\{\mathbf{tags} \wedge q\}.$$

Proof. By induction over the structure of S. Most cases are as in the proof for the non-recursive case. Most interesting is the new case for method calls: $S \equiv l_i.m_i(\overrightarrow{v_i})$: Assuming $\models \{p\}S\{\mathbf{tags} \wedge q\}$ and S is the ith method call M_i in our program, then $MGF(S) \subseteq A$ and hence $A \vdash \{p\}S\{\mathbf{tags} \wedge q\}$ by Theorem 7. As Gorelick [33] pointed out, this also holds for recursive method calls. □

Theorem 8 (Completeness for Recursive Methods)**.**

$$\textit{If} \qquad \models \{p\}S\{\mathbf{tags} \wedge q\} \qquad \textit{then} \qquad \vdash_{\mathcal{H}_d,\mathcal{T}} \{p\}S\{\mathbf{tags} \wedge q\},$$

for any statement S of a closed program π containing possibly recursive method calls and all assertions p and q.

Proof. Expressiveness of **AL** guarantees the expressibility of $WP_{\mathbf{tags}}(S, q)$ for any statement S, postcondition q and $\mathbf{tags} \in \mathcal{T}ags$. Hence by setting $q \equiv init(\varsigma)$ and $S \equiv M_i$ for any $i \in \mathbb{N}_n^1$, we can see that the set $A = \bigcup_{i=1}^{n} MGF(M_i)$ of most general correctness formulas of all method calls $M_1, ..., M_n$ is expressible in our logic. Now, since by definition of $WP_{\mathbf{tags}}$, these formulas are true, and according to the operational semantics, a method call is equivalent to evaluating its body in a **begin local**-block using appropriate variable substitutions, we have by Lemma 5

$$A \vdash_{\mathcal{H}_d,\mathcal{T}} \{p_i\}\mathbf{begin\ local}\ \text{self}, \overrightarrow{u_i} := l_i, \overrightarrow{v_i}; S_i\ \mathbf{end}\{q_i\} \text{ for all } i \in \mathbb{N}_n^1 \text{ as well as}$$
$$A \vdash_{\mathcal{H}_d,\mathcal{T}} \{p_{\mathbf{tag},i}\}\mathbf{begin\ local}\ \text{self}, \overrightarrow{u_i} := l_i, \overrightarrow{v_i}; S_i\ \mathbf{end}\{\mathbf{tag}\} \text{ for all } \mathbb{N}_n^1, \mathbf{tag} \in \mathcal{T}ags$$

with $p_i \equiv WP(M_i, init(\varsigma))$, $q_i \equiv init(\varsigma)$, $p_{\mathbf{tag},i} \equiv WP_{\mathbf{tag}}(M_i, true)$ and S_i denoting the method body of the method called in M_i for all $i \in \mathbb{N}_n^1$. Note that in the case not concerned with termination (i.e. **terminates** $\notin$ **tags**) we may postulate $r(z) \equiv z = z$ and can then

- derive the set A' (as used in the REC rule in Section 4.5) from A by applying the rule of consequence to each Hoare Triple in A, in essence adding the conjunct ($\mathbf{terminates} \rightarrow \forall z' : \mathbb{N} \bullet r_i(z') \rightarrow z' < z$) to the precondition of each Hoare triple (which is equivalent as **terminates** is false), and
- note that this derivation is reversible and hence A can also be derived from A'.
- Hence we can substitute A' for A in above formulas, and
- apply the rule of consequence to each of them to obtain

$$A \vdash_{\mathcal{H}_d,\mathcal{T}} \{p_i \wedge r_i(z)\}\textbf{begin local } \text{self}, \overrightarrow{u_i} := l_i, \overrightarrow{v_i}; S_i \textbf{ end}\{\textbf{tags}_i \wedge q_i\}$$

for all $i \in \mathbb{N}_n^1$. Note that these are just the premises of the REC rule. Furthermore, assuming $\models \{p\}S\{q\}$, by Lemma 5 we have

$$A \vdash \{p\}S\{\textbf{tags} \wedge q\}$$

which is the last missing premise of the REC rule. Hence, an application of said rule derives the desired result and completes the proof.

Termination: Just like the LOOP rule, the REC rule in $\mathcal{H}_d$ also uses a predicate $r(z)$ as a recursion bound. Hence the very same argumentation as in the remark regarding termination at the end of Section 5.2.6 applies here as well. Then, since the predicate $r(z)$ is able to express any μ-recursive function, μ-recursive functions are able to simulate the execution of **dyn** programs and **dyn** programs are trivially able to calculate the depth of any recursion in them by adding a parameter used as a counter, we conclude that **AL** predicates suffice for expressing all recursion bounds that can possibly occur in **dyn** programs and thus allow deriving $\{p\}S\{\textbf{terminates}\}$ whenever $\models \{p\}S\{\textbf{terminates}\}$ also in the case with recursive methods. □

Part III.

Type Information

"Assumptions are a transparent grid, through which we observe the universe, and sometimes we give in to the illusion that the grid would be the universe itself."

– *Kogitor Eklo - Butlers Djihad*

6. Comparison of Statically Typed- with Dynamically Typed Verification

While theoretically the completeness result for our Hoare logic guarantees the derivablility of all true properties about **dyn**-programs[1] just like it is the case for many statically typed languages, there are still quite a few practical disadvantages in verifying dynamically typed programs compared to statically typed ones.

Comparing the proof rules for **dyn** with those of **stat** is quite revealing regarding those differences. At first sight it is already obvious that the **dyn** rules are more complicated than their **stat** counterparts. Analyzing their differences in more detail, it is possible to identify the core reasons why reasoning about dynamically typed programs is more complex than reasoning about statically typed ones.

6.1. Type Safety

In dynamically typed languages, type errors are runtime events. Like divergence and failures, they give rise to a notion of correctness excluding them that we call *typesafe partial correctness*. In our tagged Hoare Logic, typesafe partial correctness corresponds to the tag **typesafe**. Hence, in the rules given in Section 4.5, the parts responsible for ensuring type safety can be identified as those parts of an assertion that is activated (implied) by **typesafe**. Such *type safety preconditions* are not necessary in statically typed languages.

6.2. Mapping Objects to Values

Hoare logic for dynamically typed languages must rely on predicates to connect (untyped) program objects and (typed) logical values. For instance, the COND rule uses the predicate $\mathbb{B}()$ to establish a correspondence between the result of the program expressions e and the boolean constants $true$ and $false$. This additional layer of indirection not only reduces readability, but also hinders substitutions for pure expressions (see next paragraph). Additionally, like in the case of $x < 5$, which corresponds to $\exists x' : \mathbb{N}.\mathbb{N}(x, x') \wedge x' < 5$, using mapping predicates often entails existentially quantifying the resulting value, which reduces readability even more and makes verification conditions harder to solve as quantifier elimination is usually a costly procedure.

[1] Assuming that the implications used in the rule of consequence are all derivable in the proof system $\mathcal{T}$ for the assertion language, of course.

6.3. Side-effecting Expressions

In the $\mathcal{H}_s$-rules ASGN and COND, the pure program expression e is directly used in logical assertions. Here, the design choice of a shared type system pays off. Unfortunately, dynamic typing forces us to relinquish this benefit, as the inavailability of static type information does not allow for distinguishing side-effecting from side-effect-free method calls in a language featuring dynamic dispatch, and potentially impure expressions are ill-suited for logical reasoning [50]. Observe also how the INST rule from $\mathcal{H}_d$ models the evaluation order using a sequence of intermediate predicates p_i, which would not be necessary for pure expressions. However, since **dyn** treats operations as method calls, the INST rule needs to be applied even for pure operations like $+, <, \wedge$, etc, making it tedious to derive even simple properties of assignments or conditionals. Also recall that the LOOP rule from $\mathcal{H}_d$ requires a second loop invariant because the loop condition might have side-effects.

6.4. Specialized Data Types

Another benefit of having a shared type system between the assertion language and the programming language is that programs operating on complex data types like lists, sets, records or trees are reasoned about in an assertion language supporting these same primitives, which is very useful, for instance when values need to be stored in logical variables (freezing).

Consider a sorting algorithm operating on lists of natural numbers. In order to express the property that the resulting list is a permutation of the one initialy given, we have to make the initial list available in the postcondition. In our assertion language **AL**, the only data-types suitable for this task are $\mathbb{N}$ and $\mathbb{O}$. Of course it would be possible to store a reference to the initial list in an object variable. However, this would not solve the problem since objects are accessible by the program and in this case the list is in fact modified by the algorithm, so following the reference in the postcondition would lead to the modified list instead of the initial one...

Natural numbers offer greater promise as (finite) lists of natural numbers can be encoded as a single natural number using Gödelization (see Section 1.5.5). Hence, the Hoare triple

$$\{\mathit{gödel\_list}(\mathrm{l}, n)\}S\{\exists m : \mathbb{N}.\mathit{gödel\_list}(\mathrm{l}, m) \wedge \mathit{permutation}(n, m)\}$$

expresses the desired property that the statement S permutes the list l by encoding its contents into a natural number and "freezing" it in the logical variable n.

Unfortunately, Gödelization also permits the encoding of arbitrary μ-recursive functions (see Section 1.5.5) and, since μ-recursive functions are able to simulate Turing-Machines, allows for expressing the Halting Problem. Hence, satisfiability of any logic that allows Gödelization must be undecidable – this includes our assertion language **AL** as well as any other Logic subsuming first-order logic with integer arithmetic. The only reason our SMT-solver Z3 is able to decide the satisfiability of many verification conditions is that they often fall into some decidable subset of the logic supported by

Z3. However, using Gödelization in a formula obviously precludes this possibility and thus Z3 is forced to return "unknown" to this kind of query.

Of course, this would also be the case when verifying a statically typed program. However, the shared type system and the shared expressions in statically typed Hoare logics ensure that programs that operate on lists are verified with an assertion language that also supports lists as a data-type. For instance, in Boogie [8] lists are represented as arrays which directly correspond to those of Z3, which even features a specialized decision procedure for arrays [17].

To sum up, in statically typed languages, the statically available type information is passed into the theorem prover and used to guide the application of specialized decision procedures. Hence, although our program logic is (relative) complete, applying it naïvely generates verification conditions that only use integer arithmetic and a self-defined sort for objects, ignoring the rich assortment of more specialized data types build into Z3 and similar solvers, hence forgoing the potential of their respective specialized decision procedures.

As stated initially, all these issues do not prevent the successful verification of a dynamically typed program, but are obstacles that exist only in dynamically typed programs because the solutions provided by verification systems for statically typed languages rely on the static availability of type information and can hence not be directly applied. However, note that it is always possible that a verification condition becomes too complicated for an SMT-solver to handle, especially when it is further complicated by additional quantifiers (for instance, due to method calls) or phrased in terms of less fitting data types. Also note that this usually has a severe practical impact on the verification of the program in question.

In this part of the thesis, we will explore the idea of mitigating these issues under the assumption that type information is available for our program. While this assumption does not hold for dynamically typed programs in general, type inference algorithms are often able to provide suitable type information for large parts of many such programs. Also, in Chapter 8 we will provide the means necessary to extend the reach of our technique to all dynamically typed programs, including those that lie beyond the capabilities of type inference algorithms.

7. Layer of Abstraction

In this chapter, we will introduce the Layer of Abstraction to mitigate the issues discussed in the previous chapters under the assumption that type information is available. Basically, it retrofits the optimizations used in statically typed Hoare logic to the case of dynamically typed programs.

7.1. Type Safety Preconditions

Like already mentioned, the fact that type errors are runtime events in dynamically typed languages gives rise to a notion of correctness that we call *type-safe partial correctness*. In our tagged Hoare logic, we write Hoare triples that hold in the sense of type-safe partial correctness as

$$\{p\}S\{\mathbf{typesafe} \wedge q\}$$

In the proof rules given in Section 4.5, *typesafety-preconditions* are those parts of assertion activated (implied) by **typesafe**.

In statically typed Hoare logic, type-safety preconditions are unnecessary. Regarding such preconditions, correctness proofs in statically typed languages resemble those in dynamically typed languages for type-unsafe correctness notions. Omitting these preconditions hence is a first step in proving dynamically typed programs like statically typed ones. This can be achieved by treating type safety issues separately from other correctness issues. We can adapt the decomposition rule found in [5] to formalize such a separation. However, contrary to our Tagged Hoare Logic, [5] uses different proof systems for each notion of correctness. Hence the added subscripts (like τ and p for "type-safe" and "partial correctness", respectively) to the $\vdash$-symbol to indicate which proof system is referred to.

Definition 22 (Decomposition from [5])**.** *The following rule is added to the proof system for some type-safe notion of correctness (τX) where X stands for a type-unsafe notion of correctness:*

$$\frac{\vdash_X \{p\}S\{q\} \quad \vdash_{\tau p} \{p\}S\{true\}}{\{p\}S\{q\}}$$

where $\vdash_X$ refers to the corresponding type-unsafe variant of the proof system while $\vdash_{\tau p}$ always refers to a proof system for type-safe partial correctness.

Within the framework of Tagged Hoare logic, this rule can be rephrased as

Definition 23 (Decomposition)**.**

$$\frac{\{p\}S\{\mathbf{tags} \wedge q\} \quad \{p\}S\{\mathbf{typesafe} \wedge true\}}{\{p\}S\{\mathbf{typesafe} \wedge \mathbf{tags} \wedge q\}}$$

which is just a special case of the CONJ rule. We conclude that the CONJ rule of Tagged Hoare Logic already allows separating and combining the different aspects of correctness and that – like in [5] – whenever a certain aspect (type-safety, termination, fail-safety) has been established for a statement S, we can subsequently omit the corresponding preconditions when reasoning about S.

7.2. Mapping Objects to Values

Although they often seem verbose, we have seen in Section 3.1 that mapping predicates are necessary for the completeness of our Hoare logic. Fortunately, given the types of all variables used, those predicates can be generated automatically. We will now introduce a "virtual" logical variable û of the corresponding type for each program variable u that may be safely mapped[1].

First, a subset of "pure" (i.e. immutable) classes $\mathfrak{C}_\varepsilon \subseteq \mathfrak{C}$ along with a function Ψ mapping classes from $\mathfrak{C}_\varepsilon$ to corresponding types $\tau \in \mathbb{T}$ of the assertion language must be defined. For **dyn**, this mapping is

$\Psi(C_{null}) = \mathbb{O}$, $\Psi(num) = \mathbb{N}$, $\Psi(bool) = \mathbb{B}$, ...

This mapping can be extended to union types $T \in \mathcal{T}$ by defining

$\Psi(\{\mathrm{C}\}) = \Psi(\mathrm{C})$ for $\mathrm{C} \in \mathfrak{C}_\varepsilon$ and $\Psi(T) = \mathbb{O}$ otherwise.

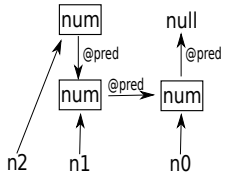
For each type $\tau \in \mathbb{T}$, we assume a mapping predicate $\tau(o, v) : \mathbb{O} \times \mathcal{D}_\tau \mapsto \mathbb{B}$ for mapping objects to values as well as a safety predicate $safe_\tau(o) : \mathbb{O} \mapsto \mathbb{B}$ defining under what condition this mapping is safe. For the type $\mathbb{N}$ these are[2]

$\mathbb{N}(o, n) \triangleq safe_{\mathbb{N}}(o) \wedge (n = 0 \wedge o.@\mathrm{pred} = \mathrm{null} \vee n > 0 \wedge o.@\mathrm{pred} \neq \mathrm{null} \wedge \mathbb{N}(o.@\mathrm{pred}, n-1))$ and

$safe_{\mathbb{N}}(o) \triangleq o \neq null \wedge [\![o]\!] \in \{num\}$.

Concretely, the **dyn**-program on the left produces the object structure depicted in the middle, which satisfies the assertions on the right.

```
n0 = new num(null);
n1 = new num(n0);
n2 = new num(n1);
```



$\neg safe_{\mathbb{N}}(null)$
$\mathbb{N}(\mathrm{n0}, 0)$
$\mathbb{N}(\mathrm{n1}, 1)$
$\mathbb{N}(\mathrm{n2}, 2)$

We then introduce a new assertion language $\mathbf{AL}_\Upsilon$ supporting the use of automatically mapped virtual variables x̂. Its semantics is defined in terms of a mapping $\Upsilon : Asrt_\Upsilon \mapsto Asrt$ from the new to the old assertion language.

Definition 24 (Automatic Variable Mapping)**.** *Let* $\hat{\mathrm{x}}_1, ..., \hat{\mathrm{x}}_n$ *be a sequence of all virtual variables occuring free in* p *and* $\mathrm{x}_1, ..., \mathrm{x}_n$ *a corresponding sequence of program*

[1] $safe_{\mathbb{T}}(\mathrm{u})$ holds for some type $\mathbb{T}$

[2] Expressing them using quantification instead of recursion is possible, but less readable (see Section 1.5.5).

variables that can be safely mapped to types $\tau_1, ..., \tau_n$*. Also, let* $v_{\hat{\mathrm{x}}_1}, ..., v_{\hat{\mathrm{x}}_n}$ *be a corresponding sequence of logical variables of types* $\tau_1, ..., \tau_n$*. Then,*

$\Upsilon(p) \triangleq \exists v_{\hat{\mathrm{x}}_1} : \tau_1, ..., v_{\hat{\mathrm{x}}_n} : \tau_n \bullet \ \Upsilon_S(p) \wedge \Upsilon_M(p)$

$\Upsilon_S(p) \triangleq p[\hat{\mathrm{x}}_1, ..., \hat{\mathrm{x}}_n := v_{\hat{\mathrm{x}}_1}, ..., v_{\hat{\mathrm{x}}_n}]$, $\Upsilon_M(p) \triangleq \tau_1(\mathrm{x}_1, v_{\hat{\mathrm{x}}_1}) \wedge ... \wedge \tau_n(\mathrm{x}_n, v_{\hat{\mathrm{x}}_n})$

For instance, assuming that $safe_{\mathbb{N}}(\mathrm{u})$ could be established, then the Υ-assertion $p \equiv \hat{\mathrm{u}} < 5$ can be used instead of the equivalent $\Upsilon(p) \equiv \exists v_{\hat{\mathrm{u}}} : \mathbb{N} \bullet \ v_{\hat{\mathrm{u}}} < 5 \wedge \mathbb{N}(\mathrm{u}, v_{\hat{\mathrm{u}}})$.

The precise definition of which variables can be "safely mapped" depends on the type information available. For the type inference algorithm discussed in Section 2.1.3, the x_i may be local variables u or instance variables of the current object **self**.@x.

However, given that an assertion p implies $safe_\tau(l)$ for some type τ and some logical expression l of type $\mathbb{O}$, note that it is always possible to apply the same technique to automatically map the object referenced by l to the value of type τ it encodes. We will hence in the following take the freedom to apply the notation $\widehat{l}$ to denote the mapped value of arbitrary logical expressions l of type $\mathbb{O}$.

Also note that $Asrt_\Upsilon$ conservatively extends $Asrt$, as any assertion $p \in Asrt$ is mapped to itself. We hence assume Υ to be implicitly applied to all assertions, enabling the pervasive use of automatic object mapping.

To formally show that the automatic object mapping permits trivially mapping **stat** assertions into **dyn** assertions, we need a mapping Θ between their states.

Translating States: $\Theta(\sigma_s) \triangleq \sigma_d$ where σ_d is derived from σ_s by introducing for every base type $\tau \in \mathbb{T} \setminus \{\mathbb{O}\}$ a (possibly infinite) set of objects $\{o_v \mid v \in \mathcal{D}_\tau \wedge \tau(o_v, v)\}$ and substituting every variable x of base type τ, holding the value $v \in \mathcal{D}_\tau$ by a variable x of type $\mathbb{O}$, referencing the object o_v. Furthermore, for each base type $\tau \in \mathbb{T} \setminus \{\mathbb{O}\}$, we identify any two objects o_1, o_2 iff $\exists v.\tau(o_1, v) \wedge \tau(o_2, v)$. We lift this equivalence to **dyn** states in the natural way.

Translating Assertions: $\Theta(p) \triangleq p[\mathrm{x}_1, ..., \mathrm{x}_n := \hat{\mathrm{x}}_1, ..., \hat{\mathrm{x}}_n]$ where x_i are all variables that can be safely mapped and occur free in p.

Theorem 9. *For all assertions* p *and* ***stat*** *states* σ*:* $\sigma \models p$ *iff* $\Theta(\sigma) \models \Theta(p)$.

Proof. By definition of $\Theta(\sigma)$, for all variables x of a base type τ in σ, $\Theta(\sigma) \models safe_\tau(\mathrm{x})$ holds and x can hence be safely mapped. Under the assumption that for all such variables x it holds that $[\![\hat{\mathrm{x}}]\!](\Theta(\sigma)) = [\![\mathrm{x}]\!](\sigma)$, the following lemma can be established by induction over the structure of the assertion language: $[\![l]\!](\sigma) = [\![\Theta(l)]\!](\Theta(\sigma))$ for all logical expressions l and **stat** states σ. As the assumption is guaranteed by the mapping predicates introduced by Υ_M, the desired result can then be established by induction over the structure of the assertion language. □

The automatic mapping requires the safety predicates to be previously established, which in turn requires type information and tracking of null values. Both is produced by the type inference algorithm discussed in Section 2.1.3. We will in the following refer to both simply as "type information".

7.3. Pure Expressions

Hoare logic for statically typed languages allows highly effective reasoning by including (syntactically identified) pure program expressions into assertions. In this section, we will show that assuming the availability of type information, this concept is also applicable to dynamically typed languages.

To define a pure subset of **dyn** expressions, one complements the set of "pure" classes $\mathfrak{C}_\varepsilon$ with a set of "pure" (i.e. side-effect-free) methods $\mathfrak{M}_\varepsilon \subseteq \mathfrak{M}$ and extends the function Ψ to also map method and constructor calls to corresponding logical expressions. Such an expression $l \in \mathit{LExp}$ of type τ with free variables $v_0, ..., v_n$ of types $\tau_0, ..., \tau_n$ can be interpreted as a function $f_l : \tau_0 \times ... \times \tau_n \mapsto \tau$. We hence denote its type as $\mathit{LExp}(\tau_0 \times ... \times \tau_n \mapsto \tau)$. The extension of the mapping Ψ can then be formalized as follows:

- For every pure operation m of arity n:

$$\Psi : (\tau_0.\mathrm{m}(\tau_1, ..., \tau_n) \to \tau) \mapsto \mathit{LExp}(\tau_0 \times ... \times \tau_n \mapsto \tau)$$

- For every pure constructor new C of arity n:

$$\Psi : (\Psi(C).\mathrm{init}(\tau_1, ..., \tau_n) \to \tau) \mapsto \mathit{LExp}(\tau_1 \times ... \times \tau_n \mapsto \tau)$$

For the type $\mathbb{N}$ these are:

$\Psi(\mathbb{N}.\mathrm{init}(\mathbb{N})) = \mathbf{if}\ v_1 = \mathit{null}\ \mathbf{then}\ 0\ \mathbf{else}\ \hat{v_1} + 1\ \mathbf{fi}$
$\Psi(\mathbb{N}.\mathrm{succ}()) = \hat{v_0} + 1.$
$\Psi(\mathbb{N}.\mathrm{pred}()) = \hat{v_0} - 1.$
$\Psi(\mathbb{N}.\mathrm{add}(\mathbb{N})) = \hat{v_0} + \hat{v_1},$
$\Psi(\mathbb{N}.\mathrm{subtract}(\mathbb{N})) = \hat{v_0} - \hat{v_1},$
...

It is then possible to define a predicate $\mathit{pure}(e)$ automatically identifying pure expressions given type information for all variables free in e. Ψ can be extended to map such pure program expressions to typed logical expressions. We denote the type of a pure expression by $\tau(e)$.

For instance, $\Psi(\mathbf{new}\ \mathrm{num}(\mathbf{new}\ \mathrm{num}(\mathbf{null})).\mathrm{add}(\mathrm{x})) = 2 + \hat{\mathrm{x}}$.

Then, after establishing that

$$\{p[\hat{\mathbf{r}} := \Psi(\tau_0.\mathrm{m}(\tau_1, ..., \tau_n) \to \tau)]\}$$
$$\mathrm{u}_0.\mathrm{m}(\mathrm{u}_1, ..., \mathrm{u}_n)$$
$$\{\mathbf{terminates} \wedge \mathbf{typesafe} \wedge \mathbf{failsafe} \wedge p\}$$

with $\tau_i = \tau(\mathrm{u}_i)$ for all $i \in \mathbb{N}_n$ holds for all methods in $\mathfrak{M}_\varepsilon$, the following axiom can be established by induction over the structure of e

AXIOM: PURE EXPR: $\{p[\hat{\mathbf{r}} := \Psi(e)]\}e\{\mathbf{tags} \wedge p\}$ where $\mathit{pure}(e)$

Combining the axiom with **dyn**-specific proof rules yields simplified rules for pure expressions that closely resemble those for **stat**. For instance:

AXIOM: PURE ASGN

$$\{p[\hat{\mathrm{x}}, \hat{\mathbf{r}} := \Psi(e), \Psi(e)]\}\mathrm{x} := e\{\mathbf{tags} \wedge p\}$$

where $\mathit{pure}(e)$, $\tau(e) \sqsubseteq \tau(\mathrm{x})$.

RULE: PURE COND

$$\frac{\{p \wedge \Psi(e)\}S_1\{\mathbf{tags} \wedge q\} \qquad \{p \wedge \neg\Psi(e)\}S_2\{\mathbf{tags} \wedge q\}}{\{p\} \text{ if } e \text{ then } S_1 \text{ else } S_2 \text{ fi } \{\mathbf{tags} \wedge q\}}$$

where $\mathit{pure}(e)$, $\mathbf{r} \notin \mathit{free}(p)$ and $\tau(e) = \mathbb{B}$.

RULE: PURE LOOP

$$\frac{\{p \wedge \Psi(e) \wedge \mathbf{terminates} \rightarrow r(z)\}S\{\mathbf{tags} \wedge p \wedge \mathbf{terminates} \rightarrow \forall z' : \mathbb{N} \bullet r(z') \rightarrow z' < z\}}{\{p\} \text{ while } e \text{ do } S \text{ od } \{\mathbf{tags} \wedge p \wedge \neg\Psi(e) \wedge \mathbf{r} = \mathit{null}\}}$$

where $\mathit{pure}(e)$, $\tau(e) = \mathbb{B}$.

RULE: PURE INST

$$\frac{\{p\}\mathrm{u}_0.\mathrm{m}(\mathrm{u}_1, ..., \mathrm{u}_n)\{\mathbf{tags} \wedge q\}}{\{p[\overrightarrow{\hat{\mathrm{u}}} := \overrightarrow{\Psi(e)}]\}e_0.\mathrm{m}(e_1, ..., e_n)\{\mathbf{tags} \wedge q[\overrightarrow{\hat{\mathrm{u}}} := \overrightarrow{\Psi(e)}]\}}$$

where $\mathrm{u}_i \in \mathfrak{V}_L$ fresh and $\mathit{pure}(e_i)$ for all $i \in \mathbb{N}_n$.

Definitions for $\mathit{pure}(e)$, $\Psi : \mathit{Expr}_d \mapsto \mathit{LExp}$ and $\tau(e)$ as well as soundness proofs can be found in Appendix C. Finally, we are able to state the main result of this section: in combination with decomposition and automatic object mapping, above rules allow verification just like in statically typed languages. In fact, **dyn** proofs using these techniques resemble **stat** proofs so closely that it is possible to trivially translate every proof for a **stat** program in $\mathcal{H}_s$ into one for the same program in $\mathcal{H}_d$.

Translating Programs: Since **stat** $\subset$ **dyn**, we simply have $\Theta(S) \triangleq S$.

Translating Proofs: $\Theta(\phi) = \varphi$ is defined inductively over the structure of the proof ϕ in Hoare logic for **stat**. Applications of the rules ASGN,COND,LOOP and INST need to be substituted for applications of PURE ASGN, PURE COND, PURE LOOP and PURE INST + PURE ASGN respectively. Note that this is always possible as **stat** expressions are pure and well-typed pure assignments preserve safety predicates. Applications of all other rules can be preserved, as they are identical for **dyn** and **stat**. The only other difference is the loop variant / recursion bound in the rules LOOP and REC. However, note that using a predicate as loop variant / recursion bound is a conservative extension since one can always set $r(z) \equiv z = t$ for some integer expression t.

Theorem 10. *For every **stat** program S and every correctness proof ϕ of a property $\{p\}S\{\mathbf{tags} \wedge q\}$ with* **typesafe** $\notin$ **tags** *in tagged Hoare logic for **stat** programs, $\Theta(\phi)$ is a valid proof of the property $\{\Theta(p)\}S\{\mathbf{tags} \wedge \Theta(q)\}$ in tagged Hoare logic for **dyn** programs.*

Proof. By induction over the structure of the proof ϕ, using Theorem 9 and the fact that the application conditions for the pure expression rules are satisfied when S is a statically typed program and all assertions where translated using Θ. □

Furthermore, since types for **stat** programs can be inferred, their type-safety proofs can be constructed automatically (see Section 8.6). Applying the CONJ rule then yields a proof for $\{p\}S\{\mathbf{tags} \wedge \mathbf{typesafe} \wedge q\}$. It follows that for statically well-typed programs, deriving a proof in $\mathcal{H}_d$ (using the layer of abstraction) does not require any more effort than deriving it in $\mathcal{H}_s$. The next chapter will discuss how the applicability of the layer of abstraction can be extended to arbitrary dynamically typed programs by deriving the necessary type information semi-automatically. In Section 11.4, we will demonstrate both techniques by proving the evaluator example correct.

7.4. Extending the Assertion Language

The merit of the Layer of Abstraction described above is proportional to the number of pure classes and pure methods. However, each pure class also needs a data type in the assertion language that its instances can be mapped to. It is no use to have a pure class string with several pure methods when there is no data type string in the assertion language. Hence, the Layer of Abstraction calls not only for an additional assortment of data types, but also for an extensible assertion language.

For this thesis it will be sufficient to augment **AL** with additional data types $\mathbb{S}, \mathbb{L}, \mathbb{M}$ for strings, lists and finite maps

$$\{\mathbb{S}, \mathbb{L}, \mathbb{M}\} \subset \mathbb{T}.$$

There is no need for type constructors as our lists ($\mathbb{L}$) always map natural numbers ($\mathbb{N}$) to objects ($\mathbb{O}$) and our maps ($\mathbb{M}$) always map objects ($\mathbb{O}$) to objects ($\mathbb{O}$).

However, for a practical implementation it would be advisable to provide a way for the user to import data types and build-in predicates from the SMT-solver into the assertion language. This would have the additional benefits that

- a wide range of data types is readily available in most SMT solvers,
- since the Layer of Abstraction essentially reduces verification conditions over object-structures to statements over native data types of the solver, using more specialized data types would enable the Solver to apply the same specialized decision procedures like in statically typed languages.

Of course, the user should also be allowed to extend the sets $\mathfrak{C}_\varepsilon$ and $\mathfrak{M}_\varepsilon$ of pure Classes and pure Methods in order to make use of those imported data types. In order to declare a method m as pure and associate it with a function f_m of type $\mathbb{T}_0 \times ... \times \mathbb{T}_n \mapsto \mathbb{T}$ on a buildin (or imported) data type,

$$\{\overrightarrow{\mathrm{u}} = \overrightarrow{v}\}\mathrm{u}_0.\mathrm{m}(\mathrm{u}_1, ..., \mathrm{u}_n)\{\mathbf{pure} \wedge \hat{\mathbf{r}} = f_\mathrm{m}(v_0, ..., v_n)\}$$

needs to be established, where $\mathbb{T}_i = \tau(\mathrm{u}_i)$ for all $i \in \mathbb{N}_n$ holds in the precondition and $\tau(\mathbf{r}) = \mathbb{T}$ holds in the postcondition. **pure** is an abbreviation for **terminates** $\wedge$

typesafe $\wedge$ **failsafe** $\wedge$ **sideeffectfree**, and **sideeffectfree** is automatically assigned to any method fulfilling the following criteria

- does not assign to instance variables (except when the method is a constructor) and
- calls only methods that are also **sideeffectfree**.

Note that such methods may create objects (whose constructor is **sideeffectfree**). However, as the reference to the newly created object cannot be persisted, after the method m returned, it is either detached (has no incoming references) and can hence be ignored (or in some languages even garbage-collected) or it is returned from the method and can hence be treated as a value completely independent of the remaining state (no aliasing).

Note that this methodology is a safe over-approximation of side-effect-freedom. It is sound since methods satisfying above criteria can obviously not cause any side-effects. However, it is incomplete as there can be methods violating above criteria that are still side-effect free (their final state always equals their start state except for the value of **r**). Note that this over-approximation is sufficient for our purposes as the Layer of Abstraction is only an optimization for convenience and our program logic is (relative) complete without it.

In order to extend the set of pure classes $\mathfrak{C}_\varepsilon$, the user has to supply a mapping prediate $\mathbb{T}(o, v)$ for some buildin (or imported) type $\mathbb{T}$, that maps objects of a class, whose methods are all pure (including the constructor) and that is hence immutable, to values of the respective data-type and vice versa. It is important to check that

- The mapping $\mathbb{T}(o, v)$ is a bijection between equivalence classes of objects and values of type $\mathbb{T}$. Thus,
  - Injectivity: for any two instances o, o' of class C, such that $o.equals(o') = o'.equals(o) = false$, for all values $v, v' \in \mathbb{T}$, it holds that $\mathbb{T}(o, v) \wedge \mathbb{T}(o', v')$ implies $v \neq v'$.
  - Surjectivity: for every value v of type $\mathbb{T}$, there is an instance o of class C, such that $\mathbb{T}(o, v)$. Also: For every instance of class C, there is a value v such that $\mathbb{T}(o, v)$.
- Homomorphism: For every n-ary method m of class C that is associated with a function $f_{\mathrm{m}}(x_0, x_1, ..., x_n)$ of type $\mathbb{T} \times \mathbb{T}_1 \times ... \times \mathbb{T}_n \mapsto \mathbb{T}_R$, the following holds:

  $$\{\mathbb{T}(v_0, v_0') \wedge \mathbb{T}_1(v_1, v_1') \wedge ... \wedge \mathbb{T}_n(v_n, v_n')\} v_0.\mathrm{m}(v_1, ..., v_n) \{\mathbb{T}_R(\mathbf{r}, f_{\mathrm{m}}(v_0', v_1', ..., v_n'))\}$$

In our example ($\mathbb{N}$), the first criterion ensures that every $n \in \mathbb{N}$ can be mapped to an instance of class `num`, and vice versa. Note, however, that there can be 2 distinct objects, say o_1 and o_2 representing the same natural number n, i.e. $\mathbb{N}(o_1, n)$ and $\mathbb{N}(o_2, n)$ both hold. In this case, however, o_1.equals(o_2) must return true, which shows that o_1 and o_2 are in the same equivalence class. The second criterion then ensure that the methods of class `num` corresponsing to operations on $\mathbb{N}$ are behaviourally equivalent to those operations, i.e. mapping the objects to natural numbers and then adding them should yield the same result as doing it the other way around.

8. Interactive Type Inference

"Paths to victory there are, other than crushing one's enemy."

– *Yoda*

8.1. Example: Evaluator

In this chapter, we will develop an approach to solving typing-problems even for dynamically typed programs. It has already been stated that the typing problems for these programs can be especially difficult to solve. However, in discussions with colleagues I often found that many people have a difficult time imagining a typing problem that is "hard". We will hence start by presenting an example of such a hard typing problem in order to provide our readers with a clear idea of which problem we are attempting to solve. The example was chosen for clarifying the phrase "lies beyond the capabilities of type inference algorithms" as it turns out to be rather ambiguous due to the abundance of quite different type systems with quite different capabilities in practical use. It does so by not only exeeding the capabilities of the simple type inference algorithm introduced in Section 2.1.3, but also by being untypable for every (decidable) type systems I encountered so far.

Figure 8.2 depicts a dynamically typed program evaluating arithmetic expressions. While crafted to provide a hard typing problem, its use of ad-hoc data structures is not uncommon in Ruby, Python or Javascript.

The class `Evaluator` has two methods `parse()` and `calc()`. The former parses a string and returns the respective parse tree, while the latter evaluates a given parse tree over a given environment (a mapping from variable names (strings) to integers).

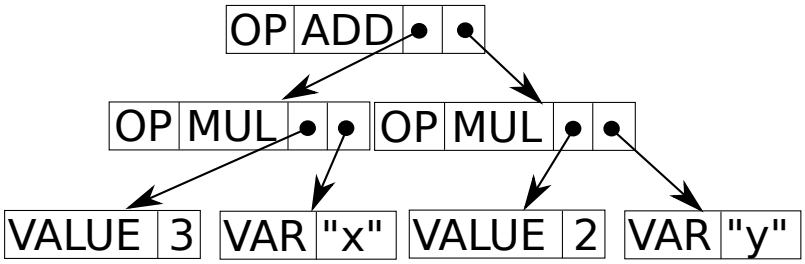


Figure 8.1.: Parse tree of the exemplary term $3 * x + 2 * y$ encoded as nested heterogeneous lists

```
class Evaluator {
  method parse(str) { ... }

  method calc(env, tree) {
    if tree[0] == VALUE then tree[1]
    else
      if tree[0] == VAR then env[tree[1]]
      else
        if tree[0] == OP then
          if tree[1] == ADD then
            calc(env, tree[2]) + calc(env, tree[3])
          else
            if ...
            else null end
          end
        else null end
      end
    end
  }
}

new Evaluator().parse(input).calc(ENV)
```

Figure 8.2.: Relevant part of the evaluator example source code

The hardness of its typing problem is partly due to the way the parse trees are represented as ad-hoc constructions of nested, heterogeneous lists (An example is depicted in Figure 8.1). Numeric constants `VALUE`, `VAR` and `OP` in the first element distinguish the three node kinds of the parse-tree (nodes representing values, variables and operations). The types of the remaining list elements depend on these node kinds: the second element is numeric (the value) for value-nodes, a string (the variable name to be looked up in the environment) for variable-nodes and numeric (representing the operation to be performed) for operation-nodes. Only operation-nodes use nesting: further list elements are sub-parse-trees that are to be recursively evaluated to operands.

Typing this example requires deducing precise types for heterogeneous lists from propositions (like `tree[0] = VALUE`) about their first element. To the best of our knowledge there is no procedure able to establish such implications automatically. Also note that the typing problem can be made even harder: allowing an arbitrary number of operands in operation-nodes, returning strings instead of `null`, etc. In Section 11.4, this example will be used to demonstrate the Layer of Abstraction (Chapter 7) as well as the Interactive Type Inference techniques (below).

8.2. Interactive Type Inference

Sufficient type information for dynamically typed programs is uncomputable in general (see below). However, a number of good approximations exist [44, 30] that we will refer to as *automatic type-safety verifiers*.

It is known that many dynamically typed programs only occasionally deviate from what would also be possible in static typing disciplines[1] and consequently, that the output of type inference algorithms is usually sufficient for typing most of their subexpressions [44, Section 5][30, Section 6].

If the entire program can be typed by a sound automatic verifier, then statically typed Hoare logic can be applied. However, as illustrated by the evaluator example in Section 8.1, the whole point of dynamic typing is the possibility to go beyond the limits of automatically inferrable type systems. Approaches to verifying these languages thus must also be able to operate under less ideal circumstances.

The fact that type-safety is a non-trivial semantic property in the sense of the Theorem of Rice [71] and hence undecidable for Turing-complete languages (like **dyn**) also applies to the derivation of sound and precise type information for such programs as both problems can be reduced to each other. However, this only means that a computer is not able to solve the problem on its own – e.g., there is no type inference algorithm solving it automatically. As common in program verification, humans (the user) are still able to solve the problem by investing manual effort. Now, since we are aiming to reduce the effort required from the user, the next obvious question is: Is there a way to combine an (automatic) type inference algorithm and a (complete) program logic into a semi-automatic procedure that

[1] Advanced dynamic features like mixins, traits, method update and dynamic class hierarchies increase the complexity of type inference. However, in this thesis we aim to study the problem of dynamic typing in isolation and leave them as future work.

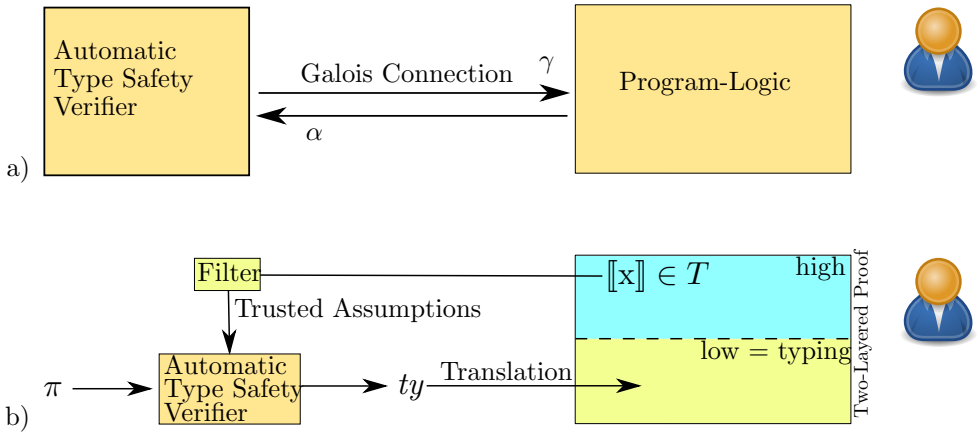


Figure 8.3.: Overview of the concept

1. is complete if the program logic is,

2. is fully automatic whenever the type inference algorithm can solve the problem on its own, and

3. minimizes the effort required from the user otherwise?

Figure 8.3 a) depicts a general concept for such a combination. The basic idea is to interpret the Program Logic's Assertion Language as an Abstract Domain $(Asrt, \rightarrow)$ in the sense of Abstract Interpretation (see Section 1.4.1), connect the two Abstract Domains using a Galois Connection (Section 8.4) and use it to exchange results bidirectionally.

As depicted in Figure 8.3 b), after a first attempt of the type inference to solve the problem automatically, one either has a solution or is able to refine the results obtained using the progam logic. Manual effort is only required for those problems that lie beyond the reach of the automatic analysis hence allowing the user to focus on the hard parts of the problem.

In the concept depicted in Figure 8.3 b), the correctness proof is split into two "layers". While the user (supported by a theorem prover) derives his proof in the higher layer, the lower layer contains type information created and modified solely by the type inference. For this purpose, the typing ty inferred for the program π is translated into a proof (Section 8.6). While the type information contained in this lower layer proof is already useful for supporting the user's higher-layer proof (see Chapter 7), the user may at any time decide to refine it by deriving more precise type information in the higher layer. This information is filtered to make it interpretable for the analysis and then supplied as a trusted assumption to refine the lower-layer typing (Sections 8.7 and 8.8).

8.3. Typing Assertions

A *type-safety proof* for a statement S is a proof of the property $\{p\}S\{\textbf{typesafe}\}$ for some precondition p in tagged Hoare logic. When run from a state satisfying p, it ensures type-safety of S by establishing all type-safety preconditions.

When interpreting our Assertion Language **AL** as an Abstract Domain $(Asrt, \rightarrow)$, we can use a Galois Connection to translate the assertions from Hoare-logic proofs into flow-sensitive program analysis results.

Such proofs constitute a kind of typing as their assertions contain type information that is by definition sufficient to establish type-safety. Soundness of these typings follows from the soundness of the Hoare logic used and can be validated using its proof rules. Before discussing how to extract type information from a Hoare logic proof, one should state that this information needs to be compatible with the type-safety verifier to be useful for our purpose. We hence define *typing assertions* $TAsrt \subset Asrt$ as a subset of the assertion language modeling the capabilities of this verifier.

For instance, the verifier discussed in Section 2.1.3 is based on a flow-sensitive, path-sensitive data flow analysis. As usual, only local variables of the current method and instance variables of the current object are tracked flow-sensitively. The remainder of the heap is abstracted into a finite number of type variables $[\![\mathrm{C.@x}]\!]$ – one for each instance variable @x of each class C.

Logically, the analysis establishes a *global typing invariant* of the form

$$\mathcal{I}(\mathring{\sigma}) \triangleq \forall o \bullet \bigwedge_{\mathrm{C}\in\mathfrak{C}} \left([\![o]\!] \in \{\mathrm{C}\} \rightarrow \bigwedge_{@\mathrm{x}\in\mathfrak{V}_\mathrm{C}} [\![o.@\mathrm{x}]\!] \in \mathring{\sigma}(C, @\mathrm{x}) \right)$$

where $\mathring{\sigma}$ denotes the abstract state associated with the current program location in one of its typings ty, thus stating the fact that the types assigned to the instance variables @x in $\mathring{\sigma}$ are over-approximating the actual types of those instance variables.

Since the instance variables are not tracked flow-sensitively by our analyses, they are independent of the program location and hence do not differ between the abstract states contained in a typing ty. We hence write $\mathcal{I}(ty)$ to denote the global typing invariant derived from any abstract state within the typing ty.

Also, the analysis provides for each program location the return type of the previously executed expression as well as the types of all variables tracked flow-sensitively. Logically, those can be regarded as a conjunction of typing literals (see below). Additionally, path sensitivity allows differentiating different paths leading to a program location and hence requires expressing alternatives, leading us to a disjunctive normal form of *typing literals* ($TLit$). Hence, only the literals allowed in typing assertions are verifier-specific. For our analysis we define the set $TAsrt$ of *typing assertions* as follows:

$$\begin{aligned} TAsrt \ni \tau &::= \tau_1 \wedge \mathcal{I} \mid \tau \vee \tau_1 \\ \tau_1 &::= \mu \mid \neg\mu \mid \tau_1 \wedge \tau_1 \\ TLit \ni \mu &::= [\![\mathrm{u}]\!] \in T \mid [\![\textbf{self}.@\mathrm{x}]\!] \in T \qquad \text{with } T \in \mathcal{T} \end{aligned}$$

8.4. Galois Connection

The first step towards interactive type inference is a Galois connection $(2^{\mathring{\Sigma}}, \alpha, \gamma, Asrt)$ between the two abstract domains $(Asrt, \rightarrow)$ and $(2^{\mathring{\Sigma}}, \sqsubseteq)$[2]. We will establish it using the abstract domain $(TAsrt, \rightarrow)$ of typing assertions as an intermediate step.

These typing assertions correspond exactly to sets of abstract states as used in our type inference TI and can hence be related using a Galois Connection $(2^{\mathring{\Sigma}}, \alpha_1, \gamma_1, TAsrt)$ where $\alpha_1 : 2^{\mathring{\Sigma}} \mapsto TAsrt$ is given by

$$\alpha_1(\mathring{\sigma}) \triangleq \left(\bigwedge_{\mathrm{x} \in dom(\mathring{\sigma}) \cap \mathfrak{V}_L} [\![\mathrm{x}]\!] \in \mathring{\sigma}(\mathrm{x}) \right) \wedge \left(\bigwedge_{@v \in dom(\mathring{\sigma}) \cap \mathfrak{V}_I} [\![\mathbf{self}.@\mathrm{v}]\!] \in \mathring{\sigma}(\mathbf{self}.@\mathrm{v}) \right) \wedge \mathcal{I}(\mathring{\sigma}),$$

$\alpha_1(\{\mathring{\sigma}_1, ..., \mathring{\sigma}_n\}) \triangleq \alpha_1(\mathring{\sigma}_1) \vee ... \vee \alpha_1(\mathring{\sigma}_n)$, and

$\gamma_1 : TAsrt \mapsto 2^{\mathring{\Sigma}}$ is given by $\gamma_1(\tau) \triangleq \{\alpha_{TI}(\sigma) \mid \sigma \in [\![\tau]\!]\}$
where α_{TI} / γ_{TI} are the abstraction / concretization functions from the Galois insertion between concrete states $(2^{\Sigma}, \subseteq)$ and abstract states $(2^{\mathring{\Sigma}}, \sqsubseteq)$ as defined in Section 1.4.3.

Since $TAsrt \subseteq Asrt$, we can of course reuse the satisfaction relation from $Asrt$. However, since typing assertions operate exclusively on the level of types, it makes sense to extend it to abstract states.

$$\mathring{\sigma} \models \tau \text{ iff } \forall \sigma \in \gamma_{TI}(\mathring{\sigma}) \bullet \sigma \models \tau.$$

Lemma 6 (Adequacy of abstraction)**.** $\forall \mathring{\sigma} \in \mathring{\Sigma} \bullet \gamma_{TI}(\mathring{\sigma}) \subseteq [\![\alpha_1(\mathring{\sigma})]\!]$

Proof. Let $\mathring{\sigma} \in \mathring{\Sigma}$, then $\mathring{\sigma}(\mathrm{x}) = \{C\}$ iff $\mathring{\sigma} \models [\![\mathrm{x}]\!] \in \{C\}$ for all $\mathrm{x} \in \mathfrak{V}_L$ and some $C \in \mathfrak{C}$. It follows that $\mathring{\sigma} \models \bigwedge_{\mathrm{x} \in dom(\mathring{\sigma}) \cap \mathfrak{V}_L} [\![\mathrm{x}]\!] \in \mathring{\sigma}(\mathrm{x})$. The same argument also holds for instance variables. It follows that $\mathring{\sigma} \models \bigwedge_{@v \in dom(\mathring{\sigma}) \cap \mathfrak{V}_I} [\![\mathbf{self}.@\mathrm{v}]\!] \in \mathring{\sigma}(\mathbf{self}.@\mathrm{v})$ and $\mathring{\sigma} \models \mathcal{I}(\mathring{\sigma})$. Consequently, $\mathring{\sigma} \models \alpha_1(\mathring{\sigma})$ and by the definition of $\models$ for abstract states, also $\forall \sigma \in \gamma_{TI}(\mathring{\sigma}) \bullet \sigma \models \alpha_1(\mathring{\sigma})$. It follows that $\gamma_{TI}(\mathring{\sigma}) \subseteq [\![\alpha_1(\mathring{\sigma})]\!]$ holds as desired and since we only assumed $\mathring{\sigma}$ to be an abstract state, this holds for all $\mathring{\sigma} \in \mathring{\Sigma}$. □

Lemma 7 (First Galois Connection)**.** $(TAsrt, \alpha_1, \gamma_1, 2^{\mathring{\Sigma}})$ *is a Galois Connection between the complete lattices* $(TAsrt, \rightarrow)$ *and* $(2^{\mathring{\Sigma}}, \subseteq)$*.*

Proof. 1. $\forall S \subseteq \mathring{\Sigma}.S \subseteq \gamma_1(\alpha_1(S))$: Let $\sigma \in S$, then $\alpha_1(\sigma)$ is a disjunct in $\alpha_1(S)$. 2. $\forall \tau \in TAsrt.\tau \rightarrow \alpha_1(\gamma_1(\tau))$: Let $\mathring{\sigma} \in \gamma_1(\tau)$, then $\mathring{\sigma} \models \tau$ and hence $\tau \rightarrow \alpha_1(\mathring{\sigma})$. Since this holds for all abstract states $\mathring{\sigma} \in \gamma_1(\tau)$, τ also implies their disjunction $\alpha_1(\gamma_1(\tau))$. □

We will now define how to extract type information from Hoare logic proofs. In such a proof, each postcondition may contain flow-sensitive type information about

[2]The ordering relation $\sqsubseteq$ is lifted to sets of abstract states in the natural way by treating the sets as disjunctions.

variables as well as the return value **r** of the previous expression. Given such a post-condition (or any other assertion) p, one extracts this information by first converting p into disjunctive normal form, treating typing literals, equations and quantifiers as literals (μ) and then applying a *projection* $pr : Asrt \mapsto TAsrt$ defined as follows:

$$\begin{aligned} pr(\mu) &\triangleq \begin{cases} \mu & \text{if } \mu \in TLit \\ true & \text{otherwise} \end{cases} \\ pr(\neg\mu) &\triangleq \neg pr(\mu) \\ pr(\mu_1 \wedge \mu_2) &\triangleq pr(\mu_1) \wedge pr(\mu_2) \\ pr(\tau_1 \vee \tau_2) &\triangleq pr(TAsrt_1) \vee pr(\tau_2) \end{aligned}$$

The projection pr thus preserves $\wedge, \vee$ and $\neg$ while mapping all literals $\notin TLit$ to $true$ ($\triangleq [\![\mathbf{self}]\!] \in \top$). Every assertion p hence implies $pr(p)$. Note that depending on the structure of p, there might be a significant loss of precision. This is unproblematic, however, as supplying type information is in the user's interest. Furthermore, one can define a *projection* $pr_{\mathrm{x}} : TAsrt \mapsto \mathcal{T}$ further projecting typing assertions to summary types for the variable x such that for all assertions p and all variables x, we have $p \to [\![\mathrm{x}]\!] \in pr_{\mathrm{x}}(p)$. In the case of our type inference TI, $pr_{\mathrm{x}}(p)$ is defined as follows:

- $pr_{\mathrm{x}}([\![\mathrm{x}]\!] \in T) \triangleq T$
- $pr_{\mathrm{x}}([\![\mathrm{x}']\!] \in T) \triangleq \top$ with $\mathrm{x}' \neq \mathrm{x}$
- $pr_{\mathrm{x}}(\neg\mu) \triangleq \top \setminus pr_{\mathrm{x}}(\mu)$
- $pr_{\mathrm{x}}(\tau \wedge \tau') \triangleq pr_{\mathrm{x}}(\tau) \sqcap pr_{\mathrm{x}}(\tau')$
- $pr_{\mathrm{x}}(\tau \vee \tau') \triangleq pr_{\mathrm{x}}(\tau) \sqcup pr_{\mathrm{x}}(\tau')$

We extend the domain of pr_{x} to assertions by defining $pr_{\mathrm{x}} \triangleq pr_{\mathrm{x}} \circ pr$. Using it, every proof ψ for a program π in tagged Hoare logic gives rise to a typing ty_ψ, such that

- $ty_\psi(S_\bullet)(\mathrm{u}) = pr_{\mathrm{u}}(q_1 \wedge ... \wedge q_k)$ for all $\mathrm{u} \in \mathfrak{V}_L \cup \mathfrak{V}_S$, and
- $ty_\psi(S_\bullet)(\mathbf{self}.@\mathrm{v}) = pr_{\mathbf{self}.@\mathrm{v}}(q_1 \wedge ... \wedge q_k)$ for all $@\mathrm{v} \in \mathfrak{V}_I$

hold for every sub-statement S of π where the q_i are the postconditions of all Hoare triples of the form $\{p_i\}S\{q_i\}$ in ψ.

We can then define a Galois Connection $(Asrt, \alpha_2, \gamma_2, TAsrt)$ between Assertions and Typing Assertions with

$\alpha_2 : Asrt \mapsto TAsrt$ defined as $\alpha_2(p) = pr(p)$
$\gamma_2 : TAsrt \mapsto Asrt$ defined as $\gamma_2(\tau) = \tau$ $\qquad (TAsrt \subseteq Asrt)$

Lemma 8 (Second Galois Connection)**.** *$(Asrt, \alpha_2, \gamma_2, TAsrt)$ is a Galois Connection.*

Proof. 1. $\forall p \in Asrt \bullet p \to \gamma_2(\alpha_2(p))$. By the definition of γ_2 and α_2, $p \to \gamma_2(\alpha_2(p))$ can be simplified to $p \to pr(p)$ which holds by definition of pr. 2. $\forall \tau \in TAsrt \bullet \tau \to \alpha_2(\gamma_2(\tau))$. By the definition of γ_2, this can be simplified to $\tau \to \alpha_2(\tau)$ which holds since typing assertions contain only typing information, are therefore invariant under pr and by reflexivity of implication. □

Since Galois Connections can be composed using functional composition $\circ$, $(Asrt, \alpha = \alpha_1 \circ \alpha_2, \gamma = \gamma_2 \circ \gamma_1, \mathring{\Sigma})$ is also a Galois connection between the abstract domains (complete lattices) $(Asrt, \to)$ and $(2^{\mathring{\Sigma}}, \subseteq)$ and hence allows for translating proofs directly into typings and vice versa.

8.5. Translating Abstract States into Assertions

The function Ξ maps the type information contained in a flow-sensitive, path-sensitive typing ty for each program location $L \in \mathbf{Loc}_\pi$ into a typing assertion using the Galois Connection defined above. The typings produced by the type inference algorithm from Section 2.1.3 are flow-sensitive, as $ty([\![\mathrm{x}]\!], L, i)$, the type of the variable x at program location L on path i, takes strong updates into account. They are also path-sensitive as they can differentiate between multiple paths to the same location. Hence, for those typings, the function Ξ is given by

$$\Xi(ty, L) \triangleq \gamma(ty(L)).$$

8.6. Translating Typings into Proofs

Definition 25 (Typing Proof)**.** *A* typing proof ψ *for a typing* ty *of a statement* S *is a minimal*[3] *proof of the property* $\{p\}S\{true\}$ *for some precondition* p *in tagged Hoare logic such that* $ty_\psi = ty$*.*

Technically, ψ only establishes soundness of the typing ty (by being a Hoare logic proof and $ty_\psi = ty$). However, when $ty_\psi \sqsubseteq ty^\dagger_S$, ψ can be turned into a type-safety proof by adding the **typesafe**-tag to the postcondition and trivially establishing the type-safety preconditions. Hence, typing proofs are well-suited as intermediate steps towards type-safety proofs.

Recall that typings can be checked for soundness using a verifier-specific inference system. It is hence possible to extend Ξ to mechanically derive a typing proof $\psi = \Xi(\pi, ty)$ for a sound typing ty of a program π by translating the rules of this inference system into Hoare logic and establishing $\mathcal{I}(ty)$ as a global invariant. In such a proof, each assertion p at program location L is exactly the typing assertion $\Xi(ty, L)$. We hence write $\Xi(\psi, L) \triangleq \Xi(ty, L)$.

The interested reader may find such a translation formalized in [28, appendix D]. Note that Ξ allows using automatically derived type information in Hoare logic proofs

[3] All Hoare triples in ψ must contribute to establishing the conclusion.

even in interactive theorem proving environments like Isabelle that trust only propositions that they verified a proof for.

8.7. Trusted Assumptions

We first extend **dyn** expressions by introducing a type filter operation $e ::= T \sqcap e$ with $T \in \mathcal{T}$, having the following (monotone) typing rule (see Section 2.1.3 for information on flow-graphs and typing rules):

$$\frac{e \gtrdot \overset{1}{\circ} \overset{G_e}{\dashrightarrow} \overset{2}{\bullet}}{T \sqcap e \gtrdot \overset{1}{\circ} \overset{G_e}{\dashrightarrow} \overset{2}{\circ} \overset{f}{\longrightarrow} \overset{3}{\bullet}} \text{ with } f \triangleq \lambda\hat{\sigma}.\hat{\sigma}[\mathbf{r} := T \sqcap \hat{\sigma}(\mathbf{r})]$$

We require automatic type-safety verifiers to provide an interface for supplying trusted assumptions. Abstractly, one defines a refinement relation $ty \overset{\tau,L}{\rightarrow} ty'$ between typings. Here, ty' refines ty by taking the (additional) trusted assumption $\tau \in TAsrt$ at program location L into account.

By inserting type filters, it is possible to refine a typing assertion $\tau \equiv \Xi(ty, L)$ for a program location L to $\tau \wedge \tau'$ for some assumption τ':

8.7.1. Conjunctive Refinement

Definition 26 (Refinement of Typings)**.** *Let ty be a typing derived for a program π ($\pi, \emptyset \blacktriangleright ty$). Then a* conjunctive refinement step *of ty using the trusted assumption $\tau \in TAsrt$ at program location $L \in \mathbf{Loc}_\pi$ is a quadruple (ty, τ, L, ty'), written $ty \overset{\tau,L}{\rightarrow} ty'$ with the typing ty' being derived for a program π' ($\pi', \emptyset \blacktriangleright ty'$) resulting from π by inserting the statement $\mathcal{R}_\tau$ just before L with $\mathcal{R}_\tau$ being defined inductively as*

$\mathcal{R}_{[\![x]\!] \in T} \triangleq x := T \sqcap x$

$\mathcal{R}_{\mu \wedge \mu'} \triangleq \mathcal{R}_\mu; \mathcal{R}_{\mu'}$

$\mathcal{R}_{\nu \vee \nu'} \triangleq$ **if** ? **then** $\mathcal{R}_\nu$ **else** $\mathcal{R}_{\nu'}$ **end**[4]

In order for a conjunctive refinement step to be useful (i.e. $ty' \sqsubset ty$), we stipulate $\Xi(ty, L) \not\rightarrow \tau$. Hence, for the program location L, the trusted assumption τ must contain more precise type information than the typing ty.

In essence, if τ has disjuncts $\nu_1, ..., \nu_n$ then all paths reaching the resulting conditional are split into n paths and for each $j \in \mathbb{N}_n^1$, the types $[\![\mathrm{x}_1]\!], ..., [\![\mathrm{x}_m]\!]$ of all variables $\mathrm{x}_1, ..., \mathrm{x}_m \in free(\nu_j)$ are refined to $[\![\mathrm{x}_i]\!] \sqcap pr_{\mathrm{x}_i}(\nu_j)$ for all $i \in \mathbb{N}_m^1$ and all program locations dominated by L.

The reason why this kind of refinement is called "conjunctive" becomes apparent in the following Lemma and Theorem:

[4] The condition ? denotes non-deterministic choice and is also a flag signaling the type inference algorithm to treat the conditional path-sensitive.

Lemma 9. *For every typing assertion τ, every statement S containing $\mathcal{R}_\tau$ as a subexpression and having a non-cyclic flow-graph, every typing ty and every abstract start state $\mathring{\sigma}$ such that $\mathcal{R}_\tau, \mathring{\sigma} \blacktriangleright ty$, it holds that $\Xi(ty, \mathcal{R}_{\tau\bullet}) \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \tau$.*

Proof. By induction over the structure of τ:
Induction Hypothesis:
Assuming that above lemma holds for all τ with structurally smaller abstract syntax trees.
Induction Base:
Let $\tau \equiv [\![x]\!] \in T$ and, consequently, $\mathcal{R}_\tau \equiv x := T \sqcap x$. Then, the flow-graph of S contains an edge $\underset{1}{\circ} \xrightarrow{f_{\mathcal{R}_\tau}} \underset{2}{\bullet}$ with $\underset{1}{\circ} =_\circ \mathcal{R}_\tau$ and $\underset{2}{\bullet} = \mathcal{R}_{\tau\bullet}$. From the typing rules for $T \sqcap e$ above, assignment and variable access, we can deduce that $f_{\mathcal{R}_\tau} \equiv \lambda\hat{\sigma}.\hat{\sigma}[x := T \sqcap \hat{\sigma}(x)]$. Now, since every solution of the corresponding constraint system will have to satisfy the constraint imposed by $\mathcal{R}_\tau$, and since the flow-graph is free of cycles, we know that for every such solution ty, it holds that $ty[\mathcal{R}_{\tau\bullet}] = \hat{\sigma}[x := T \sqcap \hat{\sigma}(x)]$ where $\hat{\sigma} = ty[_\circ\mathcal{R}_\tau]$. Therefore, since $ty[\mathcal{R}_{\tau\bullet}]$ and $ty[_\circ\mathcal{R}_\tau]$ differ only in the type of x and $ty[\mathcal{R}_{\tau\bullet}](x) = ty[_\circ\mathcal{R}_\tau](x) \sqcap T$, the desired result follows from the definition of Ξ.
Induction Step:
1) $\tau \equiv \mu \wedge \mu'$ and, consequently, $\mathcal{R}_\tau \equiv \mathcal{R}_\mu; \mathcal{R}_{\mu'}$. By the typing rule for sequential composition we have that $ty[\mathcal{R}_{\tau\bullet}] = f_{\mu'}(f_\mu(ty[_\circ\mathcal{R}_\tau]))$. Let $L = TLitX_\bullet$ denote the intermediate location after executing only $\mathcal{R}_\mu$. By the induction hypothesis we have that $\Xi(ty, L) \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \mu$ and $\Xi(ty, \mathcal{R}_{\tau\bullet}) \leftrightarrow \Xi(ty, L) \wedge \mu'$. It follows that $\Xi(ty, \mathcal{R}_{\tau\bullet}) \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \mu \wedge \mu' \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \tau$.
2) $\tau \equiv \nu \vee \nu'$ and, consequently, $\mathcal{R}_\tau \equiv$ **if** ? **then** $\mathcal{R}_\nu$ **else** $\mathcal{R}_{\nu'}$ **end**. Since our type inference algorithm treats the conditional with the ?-condition path-sensitively and since there are two paths p_ν and $p_{\nu'}$ through $\mathcal{R}_\tau$, there also need to be two abstract states $\mathring{\sigma}_\nu = ty[\mathcal{R}_{\tau\bullet}, p_\nu]$ and $\mathring{\sigma}_{\nu'} = ty[\mathcal{R}_{\tau\bullet}, p_{\nu'}]$. From the typing rule for conditionals and the induction hypothesis, we can deduce that $\Xi(\mathring{\sigma}_\nu) \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \nu$ and $\Xi(\mathring{\sigma}_{\nu'}) \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \nu'$. Furthermore, from the definition of Ξ, we know that in cases with multiple paths leading to a program location, their corresponding typing assertions are disjoined. Hence, $\Xi(ty, \mathcal{R}_{\tau\bullet}) \equiv \Xi(\mathring{\sigma}_\nu) \vee \Xi(\mathring{\sigma}_{\nu'}) \leftrightarrow (\Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \nu) \vee (\Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \nu') \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge (\nu \vee \nu') \leftrightarrow \Xi(ty_{,\circ}\mathcal{R}_\tau) \wedge \tau$. □

The effect caused in the abstract domain of types by inserting the refinement expression $\mathcal{R}_\tau$ hence equals the effect of a conjunction with the typing assertion τ on the logical level.

Observe that just like with its set-theoretic parallel intersection, conjoining a formula with another one can only decrease the number of models, but never increase it. Applied to our abstract domain of types, this means that conjunctive refinement can only increase precision, but never decrease it.

Let us now generalize this observation to cases with cyclic flow-graphs.

Theorem 11. *Let $ty \overset{\tau,L}{\rightarrow} ty'$ be conjunctive refinement step. Then it holds that*

$$\Xi(ty', L) \leftarrow \Xi(ty, L) \wedge \tau.$$

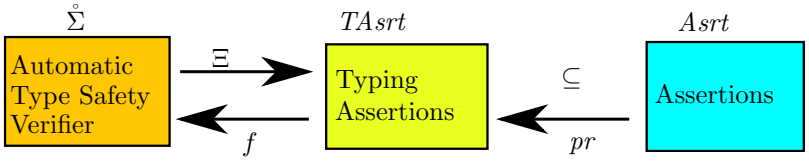


Figure 8.4.: Overview of the abstract domains involved and the mappings between them

Proof. Since in the conjunctive refinement step, ty' is derived for a program π', that is equivalent to π, except for the subexpression $\mathcal{R}_\tau$ being inserted just before L, we can assume L to be the output location $\mathcal{R}_{\tau\bullet}$ of $\mathcal{R}_\tau$ in π'. Should π''s flow-graph be free of cycles, the theorem would follow directly by Lemma 9. In case of cycles, however, it is possible for the increase in precision on the output location $\mathcal{R}_{\tau\bullet}$ to cause an increase in precision at the input location ${}_\circ\mathcal{R}_\tau$, thus creating a feedback loop. However, since all functions along the flow-graph are monotone, it is ensured that the increase in precision at location L due to the conjunctive refinement will only increase but never decrease precision at any other location, including itself. □

As already apparent in the last proof, another important property of conjunctive refinements is their *monotonicity*.

Theorem 12 (Strict Monotonicity)**.** *For all conjunctive refinements* $ty \overset{\tau,L}{\to} ty'$, $ty' \sqsubset ty$ *holds.*

Proof. **Monotonicity:** Let $ty \overset{\tau,L}{\to} ty'$ be a conjunctive refinement step and $\mathrm{x} \in \mathfrak{V}_L$. Then $ty'(L)(\mathrm{x}) = ty(L)(\mathrm{x}) \sqcap pr_{\mathrm{x}}(\tau) \sqsubseteq ty(L)(\mathrm{x})$. This difference is induced by the constraints generated from $\mathcal{R}_\tau$. Since all other constraints are identical between π and π' and all constraints are monotone, $ty'(L') \sqsubseteq ty(L')$ holds for all programs locations L' of π and consequently $ty' \sqsubseteq ty$ follows by induction over the constraint system.

Strictness: Since $\Xi(ty, L) \not\to \tau$ holds, there is at least one variable $\mathrm{y} \in \mathfrak{V}_L$ such that $ty'(L)(\mathrm{y}) = ty(L)(\mathrm{y}) \sqcap pr_{\mathrm{y}}(\tau) \sqsubset ty(L)(\mathrm{y})$. Consequently, $ty' \sqsubset ty$. □

8.8. Two-Layered Proofs

A *two-layered proof* is a proof in tagged Hoare logic for a dynamically typed program, in which every assertion has the form $\tau \wedge p$ for a typing assertion τ and an assertion p. While p is user-editable, the typing assertion τ is meant to be created and modified solely by an automated type-safety verifier. We refer to τ as the "lower layer" and p as the "higher layer" of the proof/assertion. Formally, we define:

Definition 27 (Fusion of Hoare logic Proofs)**.** *Let* ϕ *be a proof in tagged Hoare logic for* $\{p\}S\{\mathbf{tags} \wedge q\}$ *and* φ *be a typing proof for* $\{\tau\}S\{\tau'\}$*. Then, the fusion* $\phi + \varphi$ *is a two-layered proof for* $\{p \wedge \tau\}S\{\mathbf{tags} \wedge q \wedge \tau'\}$*.*

and then give the following theorem about proof fusion:

Theorem 13 (Two-Layered Proof Construction)**.** *Given a typing proof ϕ_l and a proof ϕ_h for the same program π, it is always possible to construct a two-layered-proof ϕ with ϕ_l as lower and ϕ_h as higher layer.*

Proof. Without loss of generality, we assume ϕ_l and ϕ_h to be minimal (all Hoare triples contribute to the proof's conclusion). They hence have a tree-like structure. Their fusion can then be constructed by recursion over this structure.

Induction basis: (Fusing axioms): All axioms in our tagged Hoare logic (Section 4.5), are invariant under conjunction: if $\{p\}S\{q\}$ and $\{\tau\}S\{\tau'\}$ can be derived using this axiom, then $\{p \wedge \tau\}S\{q \wedge \tau'\}$ can also be.

Induction step: (Fusing rules): All rules in our tagged Hoare logic (Section 4.5) have the following properties

- Invariant under fusion: If
  $\dfrac{\{p_1\}S_1\{q_1\}, ..., \{p_n\}S_n\{q_n\}}{\{p\}S\{q\}}$ (X)
  and $\dfrac{\{\tau_1\}S_1\{\tau_1'\}, ..., \{\tau_n\}S_n\{\tau_n'\}}{\{\tau\}S\{\tau'\}}$ (X)
  are valid rule applications, then
  $\dfrac{\{p_1 \wedge \tau_1\}S_1\{q_1 \wedge \tau_1'\}, ..., \{p_n \wedge \tau_n\}S_n\{q_n \wedge \tau_n'\}}{\{p \wedge \tau\}S\{q \wedge \tau'\}}$ (X)
  is also.
- They are either syntax-directed (and hence must appear in both proofs) or have a neutral application $\dfrac{\{p\}S\{q\}}{\{p\}S\{q\}}$ (X) that can be inserted into a proof to make its structure match the other one (having an application of rule X).

For most rules, this is obvious. For applications of CONJ and DISJ, one needs to fuse the proof with both premises. To see that the properties hold for the SUBST rule, consider that all variables occurring in typing assertions are being read in some method of the program (otherwise, typing them is useless). Hence, the side-condition of the SUBST rule does not allow them to be substituted for and all applications of this rule hence are neutral for all typing assertions.

Both proofs can hence be made structurally equivalent by inserting neutral rule applications and then fused using the invariance property. □

Starting from a typing proof $\Xi(\pi, ty)$ in the lower layer and only *true* in the higher layer, proofs in the higher layer are supported by type information from the lower layer (Chapter 7). The type information may also be refined:

Definition 28 (Refinement of Typing Proofs)**.** *Let $\psi = \Xi(\pi, ty)$ be a typing proof generated by a typing ty of a program π. Then each conjunctive refinement step $ty \overset{\tau,L}{\rightarrow} ty'$ gives rise to a* conjunctive proof refinement step *$\psi \overset{\tau,L}{\rightarrow} \psi'$ with $\psi' = \Xi(\pi, ty')$.*

Let $\psi_l = \Xi(\pi, ty)$ be the lower layer of a two-layered proof ψ. Then whenever a new typing literal appears within the higher layer p of an assertion at program location L in ψ, the lower-layer proof ψ_l is substituted by the result ψ_l' of the conjunctive proof refinement step $\psi_l \overset{\alpha_2(p),L}{\rightarrow} \psi_l'$. In such refinements $\Xi(\psi_l', L') \rightarrow \Xi(\psi_l, L')$ holds for all $L' \in \mathbf{Loc}$ due to theorem 12. Higher layer proof steps depending on lower layer information hence remain valid.

8.9. Interactive Type Inference

From the components introduced, we can assemble the process of *interactive type inference* for the semi-automatic verification of type-safety as follows:

Starting from a typing ty_0 derived by the type inference algorithm for a program π $(\pi, \emptyset \blacktriangleright ty_0)$, the user can initiate conjunctive refinement steps $\cdots ty_i \overset{\tau,L}{\rightarrow} ty_{i+1}$ by deriving more precise type information in the higher layer using the program logic, until finally for some typing ty_n, it holds that $ty_n \sqsubseteq ty_\pi^\dagger$ and the program π is hence proven type-safe.

Although cyclic, this process is well-founded as typings of finite programs assign only finitely many types, each containing a combination of finitely many classnames. Since conjunctive refinement steps strictly increase precision, each of them must remove at least one such classname from at least one such type. Hence, it is only possible to do this finitely many times before reaching a typing that is precise enough to establish type safety.

Note that it is nevertheless possible to get stuck in this process when the user does not find any way to derive more precise type information any more. In this case, either the program is not type-safe or the user does not know why it should be. This, however, is a pitfall common to all formal methods.

8.10. Properties

In this section, we will study whether the procedure detailed above indeed satisfies all requirements mentioned at the beginning of this chapter.

8.10.1. Soundness

One may consider trusted assumptions as similar to typecasts. However, note that such assumptions have to be established using a sound program logic before being accepted by a sound type inference. Hence, while typecasts make a type system unsound, our procedure does not allow proving a type-unsafe program type-safe as long as both the type inference and the program logic are sound.

8.10.2. Completeness Relative to the Program Logic

The following theorems show that despite the fact that the projection pr may cause a drastic loss of precision, sufficiently precise type information can always be supplied for type-safe programs.

Lemma 10. *For every assertion p and every typing assertion τ such that $p \to \tau$, there exists an equivalent assertion $p' \leftrightarrow p$ such that $pr(p') \leftrightarrow \tau$.*

Proof. $p' \equiv p \wedge \tau$ has the described properties. □

Definition 29. *A typing assertion τ is* most precise *for an assertion p iff $p \to \tau$ and for all typing assertions τ', $p \to \tau'$ implies $\tau \to \tau'$.*

Theorem 14. *Every type-safety proof ψ has an equivalent proof ψ' such that for every assertion p' in ψ', $pr(p')$ is most precise for p'.*

Proof. For a single assertion, this follows directly from Lemma 10 and the definition of the most precise typing assertion. Applying this to every assertion in the proof ψ derives ψ' and hence proves above theorem. □

Theorem 15 (Completeness relative to Hoare logic)**.** *Given completeness of the Hoare logic, for every[6] type-safe program π there exists a type-safety proof ψ such that ty_ψ is sound and precise enough to establish type-safety: $ty_\psi \sqsubseteq ty^\dagger_\pi$[5].*

Proof. Follows from completeness of the Hoare logic, Theorem 14, and the fact that type-safety proofs must establish the absence of type errors and hence contain sufficiently precise type information. □

Since we established (relative) completeness of our Hoare logic for **dyn** in Chapter 5.2, using it for interactive type inference allows us to derive precise type information for every type-safe **dyn** program[6] and hence to apply the Layer of Abstraction to all of them.

Note that a type-unsafe program that raises a **typeerror** for a certain class of inputs can still be proven conditionally type-safe by equipping it with a precondition excluding this case. The Layer of Abstraction could hence still be applied. Hence, only the verification of programs that raise type-errors on every input cannot benefit from it. However, such programs are argueably not very useful.

[6] assuming that all necessary implications can be established. Using an automated theorem prover to do so this admittedly is a rather unrealistic assumption. However, note that a) using an interactive theorem prover already makes the assumption much more realistic and that b) it is getting more and more realistic at least for the practically relevant cases as theorem proving technology advances.

8.10.3. Automation

Should the initial typing ty_0 derived by the type inference already suffice to prove type-safety ($ty_0 \sqsubseteq ty_\pi^{\dagger}$[5]), then no refinement steps are necessary and type-safety is established fully automatically. This is the case whenever the type inference is able to proof type-safety of a program (part) by itself. Also, when this is not the case, the user can use the type information supplied to find the root of the problem and only has to take care of those problems that the type inference cannot handle, while the remaining program is checked automatically.

8.11. Consensual Typing: Reconciling Static and Dynamic Typing via Verification

The fundamental difference between the mindsets of static and dynamic typing is a question of priorities that can be summarized as "safety vs. freedom". Advocates of static typing perceive type errors (however they may be defined in their language of choice) as threatening[5] and thus consider a restriction of the programming language a reasonable price for securing them from type errors once and for all. On the other hand, advocates of dynamic typing usually value freedom over safety and feel that in such a restricted programming language, they are "constantly working against the type system" or at least "doing so much nonsense just to make the type system happy" while considering type errors as just one out of the numerous kinds of exceptions also occuring in statically typed languages.

While the traditionally heated debates between these two communities have become part of programmer folklore over the last couple of decades, I'd like to take this opportunity to point out that what we learned in the previous chapter enables us to take a third standpoint in this matter.

Recall that type safety is undecidable in general and hence the program analysis used by static typing to safeguard its programs is inherently incomplete. Static typing circumvents this problem by restricting the programming language to those programs that can be safeguarded this way. Hence the root cause for the restriction is the incompleteness of the analysis. Note that in the case of a verifiable programming language (with a formal semantics, an assertion language and a program logic readily available), using an interactive type inference instead would remove the incompleteness and hence the need for a restriction.

So in essence our solution for making type inference complete (Chapter 8) naturally leads to a non-restricted programming language (similar to a dynamically typed one) that automatically safeguards all program parts with decidable type safety problems and provides the user with the means necessary to derive the same guarantees also for

[5] as defined in Section 1.3.1

[5] this can be seen in statements like "without types, programs are meaningless nonsense" which over-exaggerates the disambiguating function types play in statically typed languages to the point of pretending that it would be the type system that assigns meaning to programs rather than the semantics.

the remaining parts.

We will call such a typing discipline for verifiable programming languages "Consensual typing" as the interaction between user and computer resembles a controversal debate that goes on until reaching a typing that is agreeable to both parties – a consensus.

Furthermore, note that consensual typing generalizes both soft- and gradual typing disciplines: It generalizes soft typing as it also applies a type inference algorithm as far as possible, but instead of resorting to runtime type checks, it additionally offers the possibility to proof type safety manually. Like gradual typing, removing the **typesafe**-tags from the assertions in parts of the program permits mixing type-safe and type-unsafe program parts. However, unlike gradual typing it is possible to safeguard entire dynamically typed programs without rewriting them into statically typed ones.

9. Related Work

We will first discuss prior work that is related to our program logic (see Section 4.5) or its proof theory (Chapter 5) and then summarize work that is related to our Layer of Abstraction (Chapter 7) or Interactive Type Inference (Chapter 8).

Semantics of Dynamically Typed Languages

Recently, with the advent of JavaScript establishing itself as 'the assembly language of the web', the interest in the verification of dynamically typed languages spiked. As a first step, several operational semantics have been published for dynamically typed languages, many of which have been validated by testing [67, 76, 29, 34]:

- Python [67, 76],
- PHP [29],
- JavaScript [55, 34].

However, in order to enable program verification, a second step of providing a program logic is neccessary. Unfortunately, so far only JavaScript seems to have been blessed with such logics [68, 31]. Also, these works focus on soundness as well as direct applicability to real-world programming languages while the focus of our work was on completeness (for closed programs) and on studying the proof-theoretic implications of dynamic typing.

Tagged Hoare Logic

While the entire literature on Hoare logic is clearly related, we do not known of any other formalism or notation with the aim of consolidating the multitude of separate proof systems into one. However, Huisman and Jacobs [43] published work on dealing with abnormal (or abrupt) program termination in the context of Hoare logic. Their approach focuses on reasoning about programs with recoverable abnormal terminations like exceptions or break statements in a style similar to our extension for non-fatal type errors (see Section 12.1). However, its aims seem to only partially overlap with Tagged Hoare logic as it requires the introduction of a multitude of additional proof systems (two for each kind of abnormal termination) instead of consolidating them into one. Also, while Tagged Hoare Logic aims at treating all forms of abnormal program behavior (divergence, type errors, failures) alike, divergence clearly plays a special role in the work of Huisman and Jacobs, since it is not regarded as a form of "abrupt termination". Instead, every kind of abrupt termination (in our case: typeerrors and failures) is given an additional notion of correctness called "total" for

excluding divergence. When adopting the notation from [43], a triple for "total type-safe correctness"

$$[p]S[typeerror(q)]$$

is equivalent to our tagged triple

$$\{p\}S\{\neg\mathbf{typesafe} \wedge \mathbf{terminates} \wedge q\}.$$

However, the same triple for their "partial type-safe correctness"

$$\{p\}S\{typeerror(q)\}$$

states that when S is executed from a state satisfying p and yields a typeerror, then it does so in a state satisfying q. Note that this does not state anything about what happens when S terminates normally and hence violates

$$\{p\}S\{typeerror(q)\} \Rightarrow \{p\}S\{q\}.$$

Thus, while it is theoretically possible to define selectors like this also for Tagged Hoare Logic, this would break soundness of the CONS rule. Hence, while tagged Hoare Logic was carefully designed to allow a uniform treatment of all notions of correctness, this apparently precludes defining exotic notions like these. Since we did not yet find any tangible benefits such notions might have over well-behaved ones, further investigation is necessary to clarify the relationship of these two formalisms.

Predicates as Recursion Bounds and Loop Variants

While this technique is novel in the context of Hoare Logic, note that David Harel [38, Rule C^* on Page 32] also used predicates as loop variants to give a complete proof system for his First-Order Dynamic Logic (which also allows reasoning about total correctness of programs). Since our assertion language **AL** includes arithmetic, the (relative) completeness proof for our logic can be regarded as related to Harel's proof of "arithmetical completeness".

Type Safety

The relationship to Soft- and Gradual Typing has already been discussed in Section 8.11. There has also been work on extending such abstraction-based verifiers to handle many ideoms common in dynamically typed languages [35, 52].

Correctness

To the best of our knowlegdge, [31] currently is the only[1] axiomatic semantics for a type-safe notion of correctness of a dynamically typed language. As discussed in

[1] [68] only treat partial correctness. Also, they restrict the programming language to allow a form of (type-unsafe) pure expressions

Chapter 4 it uses type safety preconditions, considers all variables to be of object type and does not use pure expressions and would thus benefit from our approach.

Nguyên et al. [60] proposed an automatic contract verifier for untyped higher-order functional languages based on symbolic execution inserting run-time checks for contracts it cannot statically guarantee. Since they use a mechanism similar to widening to enforce termination, their approach also combines abstraction-based and symbolic reasoning.

Drawing on their work on the verification of untyped higher-order functional programs [21], Chugh et al. [20, 19] provide a dependent-type system for an untyped functional "core calculus" λ_{JS} into which JavaScript programs can be translated. Neither soundness nor completeness is demonstrated for their system.

Swamy et al. [77] semi-automatically reason about a wide range of JavaScript idioms by translating them into the dependently-typed functional language F^* and using its SMT-based reasoning engine. They also noticed that the type information generated by an abstraction-based type safety verifier (GateKeeper in their case) are useful to improve the effectiveness of automatic reasoning engines. However, they did not feed the symbolically derived proof results back into GateKeeper and did not use the type information to ease the annotation burden for their users. Since their main focus lies on a novel encoding of Dijkstra's predicate transformer semantics, using F^*'s dependent type inference to effectively reason about imperative programs in a style similar to Hoare logic, we consider the approaches to be largely complementary.

In general, all fully automatic approaches [18, 60, 35, 21] are necessarily incomplete. They can however be used as automatic type safety verifiers. Furthermore, all purely symbolic approaches [21, 20, 77, 31, 68] require the type information to be manually specified in method contracts and loop invariants.

Both the idea and the term "Layer of abstraction" are inspired by the work of Gardner, Maffeis and Smith [31] on reasoning about JavaScript. However, their work abstracts from the peculiarities of the JavaScript variable store, while ours abstracts from the complexity of dynamic typing and is applicable to virtually any dynamically typed language. The same holds for the JuS tool [59], which is based on their logic and developed by the same group.

The decomposition rule used to establish the layer of abstraction is inspired by similar constructions in [5].

Some tools for verification of statically typed imperative programs [24] allow using a "pure" subset of the programming language (that is side-effect-free and guaranteed to terminate) within assertions. The ability of our layer of abstraction to allow the use of well-typed "pure" program expressions in assertions can be seen as an extension of this idea to dynamically typed programs.

Combining Static Analysis with Program Logics:

There has been a considerable amount of work on integrating algorithmic decision procedures (mostly model checking) and deductive methods for program verification (see [81] for pointers). Due to the deep connection between data flow analysis and model checking [72], many of these techniques can be considered as related.

Note that our conjunctive refinement differs from abstraction refinement since it is not the abstraction that is refined, but the analysis result.

Also, translations from typings (f.i. from type systems for information-flow properties) to program logics are commonly used in the Proof-Carrying Code (PCC) Community [37] to avoid the need for property-specific proof-checkers. Although PCC is a completely different application area, their aim was also to integrate results derived by different inference systems into one common representation – and incidently they also chose a program logic as their "lingua franca".

A closely related proposal also integrating symbolic with abstraction-based reasoning is MIXY [45], a framework for mixing symbolic execution with type checking. In their system, the user partitiones his/her program into s-blocks and t-blocks. While s-blocks are analysed using symbolic execution, type analysis is applied to t-blocks. The results of both analyses are bidirectionally exchanged using so-called MIX-rules: Type analysis results are translated into a matching start environment for symbolic execution and types ensured by (exhaustive!) symbolic execution can be used for type analysis. Also, the aim is related: What Phang et. al called "balancing precision vs. efficiency" can also be interpreted as "combining automation with completeness", although Phang et al. do not proof their system complete. Our approach could most likely be integrated into their framework as "Hoare-Logic blocks" (in their notation probably $\{_h\ e\ _h\}$) with typing $\Gamma \vdash \{_h\ e\ _h\} : \tau$ for which a Hoare-triple $\{p_e\}e\{q_e\}$ must be derived where $p_e \triangleq \Xi(\Gamma)$ and $pr_{\mathbf{r}}(q_e) = \tau$.

Part IV.

Applications

"In theory, there is no difference between theory and practice.
But, in practice, there is."
– *Jan L. A. van de Snepscheut*

In this part of the thesis we will demonstrate several applications of the theories previously developed through the discussion of a number of case studies. In order to reduce both manual effort and the probability of human error, a prerequisite of conducting case studies of a certain size and complexity is automation. Thus several artifacts have been implemented in the course of this PhD thesis that together enable verifying **dyn**-programs in the way previously described.

Unfortunately, due to time constraints we were neither able to implement the tags from our Tagged Hoare Logic (described in Section 3.2) nor the Layer of Abstraction (described in Chapter 7). Thus the implementation does not yet allow a conclusive evaluation of our approach.

However, in the following, we will demonstrate its merits on a selection of case studies to convince our readers that

- it enables verifying **dyn** programs despite any adverse conditions (see Chapter 4),
- it supports deriving sound and precise type information for typesafe **dyn** programs, and
- the Layer of Abstraction makes the verification of **dyn** programs very similar to that of **stat** programs.

In order to demonstrate the last point, we will verify the case study in Section 11.3 once without the Layer of Abstraction and once while manually emulating its effect.

The remainder of this part is organized as follows: Chapter 10 describes the artifacts and their implementation. Chapter 11 demonstrates our approach by verifying a number of case studies, and Chapter 12 discusses a number of extensions to the developed formalisms to prepare them for features and phenomena that might be encountered in more realistic programming languages.

10. Implementation

An Interpreter, a Type Inference Algorithm and an auto-active Verification Tool based on a Weakest Precondition Calculus for **dyn** were implemented during my time as a PhD candidate. The Interpreter and type inference were written by myself in Ruby [56] and the Verification Tool was implemented by Dennis Kregel as his Master Thesis Project [47] in C# [57].

While the Interpreter was mainly used for experimentation with the language, the Verification Tool supports reasoning about the correctness of annotated **dyn** programs by generating a set of verification conditions and checking their validity using the SMT-Solver Z3 [10]. Since the implementation of the type inference algorithm supports trusted assumptions, in combination those tools are able to simulate the entire process of interactive type inference as described in Chapter 8, even though the translation between assertions and typing assertions has to be performed manually.

The remainder of this chapter is organized as follows: Section 10.1 introduces annotated **dyn**-programs. Section 10.2 gives the Weakest Precondition Calculus used in the Verification Tool. Section 10.3 discusses how to solve verification conditions with recursive predicates in an off-the-shelf SMT-solver like Z3. Section 10.4 discusses several practical issues that revolve around the use of the Rule of Adaptation to handle Method Calls in our Weakest Precondition Calculus.

10.1. Annotations

Introducing annotations requires modifying the syntax of **dyn** in the following way:

$$
\begin{array}{ll}
meth ::= \textbf{method}\ \ m(\overrightarrow{u})\ \ \textbf{requires}\ \ p\ \ \textbf{ensures}\ \ p\ \ \{S\} \mid \textbf{invariant}\ \ p & \\
S ::= \textbf{assert}\ \ p & \\
e ::= \textbf{while}\ \ e\ \ \textbf{inv}\ \ p\ \ \textbf{do}\ \ S\ \ \textbf{done} & \text{with } p \in Asrt
\end{array}
$$

The modified syntax of method declarations serves to allow the usual method contracts with pre- (**requires**) and postcondition (**ensures**), the **invariant** keyword allows declaring class invariants that are automatically conjoined to the pre- and postconditions of all methods belonging to the class they are declared in (Note that this includes inherited methods). The additional **inv** keyword in while loops allows specifying a loop invariant and the **assert** statement allows the introduction of intermediate assertions into the verification procedure. This is useful when the calculated preconditions get too large or too complicated for the solver to handle. Introducing an intermediate assertion effectively splits one big verification problem into two smaller ones. As will

be discussed in the Sections 10.4 and 10.4.1, especially method calls are prone to cause verification conditions to grow out of proportion.

In **dyn**, however, such method calls may occur not only as statements, but also as part of expressions. Note that even innocent-looking loop conditions such as $l.size() < 5$ or $i + 1 < 2 * j$ contain several method calls. Practice has shown that when verifying dynamically typed programs, it is quite often necessary to split verification conditions not only on statement boundaries, but also within expressions.

However, we refrain from introducing additional syntax for adding assertion annotations also within expressions because it is always possible to extract the subexpression containing the method call in question to a prepended assignment statement. It is then possible to augment this assignment with **assert** statements before and/or after it. Note that this is possible even in the case of the expression being a loop condition, although the control flow in this case dictates the duplication of the assignment like in

$$b := e; \textbf{while } b \textbf{ do } S; b := e \textbf{ done}$$

Although this duplication is inconvenient, it still results in code that is more intuitively comprehensible than any syntax for assertions in expression we could come up with.

10.2. Weakest Precondition Calculus

The Verification Tool utilizes the following Tagged Weakest Precondition Calculus for **dyn** programs directly derived from our Hoare logic (see Section 4.5).
The (Tagged) Weakest Precondition Calculus

$$WPC : 2^{\mathcal{T}ags} \mapsto Prog_d \mapsto Asrt \mapsto Asrt \times 2^{Asrt}$$

$$WPC_{\textbf{tags}}(S, q) \triangleq (WP_{\textbf{tags}}(S, q), VCG_{\textbf{tags}}(S, q))$$

maps a set of tags, a **dyn**-statement S and a postcondition q to the weakest precondition of S with respect to the postcondition q and a set of verification conditions, both in the sense of the notion of correctness indicated by the tags.

$WP_{\textbf{tags}}$ and $VCG_{\textbf{tags}}$ are defined inductively over the structure of S:

- $WP_{\textbf{tags}}(\text{null}, q) \triangleq q[\mathbf{r} := \textbf{null}]$
- $WP_{\textbf{tags}}(\text{u}, q) \triangleq q[\mathbf{r} := \text{u}]$
- $WP_{\textbf{tags}}(@\text{v}, q) \triangleq q[\mathbf{r} := \textbf{self}.@\text{v}]$
- $WP_{\textbf{tags}}(e_1 == e_2, q) \triangleq WP(v_1 := e_1; v_2 := e_2, q[\mathbf{r} := v_1 = v_2])$ where v_1, v_2 are fresh variables.
- $WP_{\textbf{tags}}(e \text{ is\_a? } C, q) \triangleq WP(e, q[\mathbf{r} := [\![\mathbf{r}]\!] \in \{C\}])$.

- $WPC_{\mathbf{tags}}(\mathrm{u} := e, q) \triangleq WPC_{\mathbf{tags}}(e, q[\mathrm{u} := \mathbf{r}])$
- $WPC_{\mathbf{tags}}(@\mathrm{v} := e, q) \triangleq WPC_{\mathbf{tags}}(e, q[\mathbf{self}.@\mathrm{v} := \mathbf{r}])$
- $WP_{\mathbf{tags}}(S_1; S_2, q) \triangleq WP_{\mathbf{tags}}(S_1, q')$, $VCG_{\mathbf{tags}}(S_1; S_2, q) \triangleq VCG_{\mathbf{tags}}(S_1, q') \cup VCG_{\mathbf{tags}}(S_2, q)$ with $q' = WP_{\mathbf{tags}}(S_2, q)$
- $WPC_{\mathbf{tags}}(\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}, q) \triangleq (p, VC \cup VC_1 \cup VC_2)$ with
  $WPC_{\mathbf{tags}}(e, \exists b : \mathbb{B} \bullet fs(\mathbf{tags}) \wedge ts(\mathbf{tags}) \wedge \mathbb{B}(\mathbf{r}, b) \wedge p') = (p, VC)$,
  $p' \equiv (b \wedge p_1 \vee \neg b \wedge p_2)$,
  $WPC_{\mathbf{tags}}(S_1, q) = (p_1, VC_1)$,
  $WPC_{\mathbf{tags}}(S_2, q) = (p_2, VC_2)$,
  $fs(\mathbf{tags}) \equiv \begin{cases} \mathbf{r} \neq null & \text{if } \mathbf{failsafe} \in \mathbf{tags} \\ true & \text{otherwise} \end{cases}$, and
  $ts(\mathbf{tags}) \equiv \begin{cases} \mathbf{r} \neq null \rightarrow \mathbb{B}(\mathbf{r}) & \text{if } \mathbf{typesafe} \in \mathbf{tags} \\ true & \text{otherwise} \end{cases}$
- $WPC_{\mathbf{tags}}(\mathbf{while}\ e\ \mathbf{inv}\ p\ \mathbf{var}\ \upsilon\ \mathbf{do}\ S\ \mathbf{done}, q) \triangleq (p', VC \cup VC' \cup VC'')$ with
  $(p', VC) = WPC_{\mathbf{tags}}(e, p \wedge fs(\mathbf{tags}) \wedge ts(\mathbf{tags}))$, $fs(\mathbf{tags})$ and $ts(\mathbf{tags})$ as above,
  $VC' = \{p \rightarrow p'', (\exists b : \mathbb{O} \bullet p[\mathbf{r} := b] \wedge \mathbb{B}(b, false) \wedge \mathbf{r} = null) \rightarrow q, p \rightarrow \exists z' : \mathbb{N} \bullet \upsilon(z'), term(\mathbf{tags})\}$,
  $(p'', VC'') = WPC_{\mathbf{tags}}(S, p')$,
  $term(\mathbf{tags}) \equiv \begin{cases} p \wedge \mathbb{B}(\mathbf{r}, true) \wedge \upsilon(z) \rightarrow \hat{p} & \text{if } \mathbf{terminates} \in \mathbf{tags} \\ true & \text{otherwise} \end{cases}$
  where $\hat{p} \equiv WP_{\mathbf{tags}}(S; e, \forall z' : \mathbb{N} \bullet \upsilon(z') \rightarrow z' < z)$, z is a fresh logical variable of type $\mathbb{N}$ only used to determine termination of this loop, $\upsilon(z)$ is a predicate containing a free occurrence of z and $\upsilon(z')$ denotes the result of substituting z' for all free occurrences of z in the predicate $\upsilon(z)$.
- $WPC_{\mathbf{tags}}(\mathbf{begin\ local}\ \overline{u} := \overline{t}; S\ \mathbf{end}, q) \triangleq WPC_{\mathbf{tags}}(\overline{u} := \overline{t}; S, q[\overline{u} := \overline{v}])[\overline{v} := \overline{u}]$ where $\overline{v}$ is a list of fresh variables in 1 to 1 correspondence with the local variables $\overline{u}$.
- $WP_{\mathbf{tags}}(e_0.\mathrm{m}(e_1, ..., e_n), q) \triangleq WP_{\mathbf{tags}}(v_0 := e_0; ...; v_n := e_n; v_0.\mathrm{m}(v_1, ..., v_n), q)$ where $v_0, ..., v_n$ are fresh variables.
- $WP_{\mathbf{tags}}(v_0.\mathrm{m}(v_1, ..., v_n), q) \triangleq \bigvee_{C \in T_m^n} [\![v_0]\!] \in \{C\} \wedge \exists \overline{c}_C \bullet (p'_C \wedge \forall \overline{v} : \mathbb{O}^{n+1} \bullet q'_C \rightarrow q)$ where $\overline{v} = v_0, ..., v_n$ is a list of all variables free in $S \equiv v_0.\mathrm{m}(v_1, ..., v_n)$ (the formal parameters of the call), $\overline{c}_C$ is a list of all variables free in p'_C or q'_C, but not in S or $\{z\}$, $T_m^n = \{C' \mid C' \text{ supports a method m of arity } n\}$ and

$\{p'_C\}v_0.\mathrm{m}(v_1,...,v_n)$ **var** $r_{m_C}\{\mathbf{tags}_C \wedge q'_C\}$ is the method contract of method m of arity n declared in class C, $r_{m_C}(z)$ is the variant of the method to be called (method m in class C), which is a predicate with a free occurrence of z and z is a fresh logical variable of type $\mathbb{N}$ which is only used to establish termination of recursive method calls.

- $VCG_{\mathbf{tags}}(\{p\}\mathbf{method}\ \ m(u_1,...,u_n)\ \ \mathbf{var}\ \ r_m\ \ \{S\}\{q\}) = \{vc_1, vc_2\} \cup VC_1$ with

$$vc_1 \equiv \begin{cases} p \wedge r_m(z) \rightarrow WP_{\mathbf{tags}}(S', q), & \text{if } \mathbf{terminates} \in \mathbf{tags} \\ p \rightarrow WP_{\mathbf{tags}}(S', q) & \text{otherwise} \end{cases}$$

$$vc_2 \equiv \begin{cases} p \rightarrow \exists z' : \mathbb{N} \bullet r_m(z'), & \text{if } \mathbf{terminates} \in \mathbf{tags} \\ true & \text{otherwise} \end{cases}$$

$$VC_1 = VCG_{\mathbf{tags}}(S, q).$$

  where $S' \equiv$ **begin local self**$, \overline{u} := \overline{v}; S$ **end**, $\overline{v} \equiv v_0, ...v_n$ are the same fresh variables as used in the rule for method calls of the form $e_0.\mathrm{m}(e_1,...,e_n)$ and $u_1,...,u_n$ are the formal arguments of method m.

- $WPC_{\mathbf{tags}}(\mathbf{new}\ \mathrm{C}(e_1,...,e_n), q) \triangleq WPC_{\mathbf{tags}}(\mathbf{new}\ C.\mathrm{init}(e_1,...,e_n), q)$.

- $WP_{\mathbf{tags}}(\mathbf{new}\ C, q) \triangleq q[\mathbf{r} := \mathbf{new}_C]$.

with $VCG_{\mathbf{tags}}(S, q) \triangleq \{\}$ for all S and q where above definitions do not imply otherwise.

10.3. Solving Verification Conditions with Recursive Predicates Using Off-the-Shelf SMT Solvers

Translating our Verification Conditions into SMTLIB2 is a rather straightforward process. Since type information in SMTLIB2 is explicit, we of course need a simple type inference algorithm for our Assertion Language, but afterwards it is a simple recursive translation – apart from recursive predicates.

The attentive reader might have noticed that several of our mapping predicates as well as several other predicates used in our case studies are recursive. SMTLIB2 syntactically prohibits the definition of recursive functions or predicates, so we have to find a way to suitably encode recursion.

Linear Recursion

Let us start with the simplest possible case – linear primitive recursion. The general form of a linear primitive recursive predicate definition is

$$P(x, a_1, ..., a_n) \equiv (x = 0 \wedge p_0) \vee (x > 0 \wedge p_1 \wedge P(x - 1, e_1, ..., e_n))$$

The attribute "linear" indicates that the predicate has exactly on recursive call (to itself) and "primitive" means that it is syntactically restricted to ensure termination (which is the reason for the variable x in the general form). For all predicate definitions of this form, we can give an equivalent one using quantification over sequences instead of recursion:

$$\begin{aligned} P(x, a_1, ..., a_n) \equiv \exists s_1,..., s_n : \mathbb{L}\bullet \\ & p_0[x, a_1, ..., a_n := 0, s_1[0], ..., s_n[0]] \wedge \\ & s_1[x] = a_1 \wedge ... \wedge s_n[x] = a_n \wedge \\ & \forall i : \mathbb{N} \bullet 0 < i \leq x \Rightarrow p_1\omega \wedge \\ & \qquad s_1[i-1] = e_1\omega \wedge ... \wedge s_n[i-1] = e_n\omega \end{aligned}$$

where $\omega = [x, a_1, ..., a_n := i, s_1[i], ..., s_n[i]]$.

Since quantification over sequences can be expressed as quantification over arrays, which is supported by Z3 [17], this gives us a way to automatically reason about linear primitive recursive predicates using Z3 and similar SMT-Solvers. Note that while this technique is sufficient for all predicates used in this thesis, it is possible to extend the technique to tree-recursion as follows.

Tree Recursion

The general form of a primitive tree recursive predicate of degree m is

$$P(x, a_1, ..., a_n) \equiv (x = 0 \wedge p_0) \vee (x > 0 \wedge p_1 \wedge P(x-1, e_1^1, ..., e_n^1) \wedge ... \wedge P(x-1, e_1^m, ..., e_n^m))$$

Note that m denotes the number of recursive calls of the predicate to itself, not the number of its formal arguments. To encode it into SMTLIB2, we will be using the following auxiliary predicates

$$\begin{aligned} & sizelevel_m(l) \equiv m^l \\ & slevel_m(l) \equiv m^l - 1 \\ & indexof_m(l, i) \equiv slevel_m(l) + i \\ & child_m^k(l_p, i_p, l_c, i_c) \equiv l_c = l_p + 1 \wedge i_c = m * i_p + (k-1) \end{aligned}$$

Now for all predicates of above form, we can again formulate an equivalent one using quantification over arrays

$$
\begin{aligned}
P(x, a_1, ..., a_n) \equiv \exists s_1, ..., s_n : \mathbb{L}\bullet \\
&(\forall i : \mathbb{N} \bullet 0 \leq i < sizelevel(x) \Rightarrow \\
&\quad p_0[x, a_1, ..., a_n := 0, s_1[indexof(x, i)], ..., s_n[indexof(x, i)]]) \wedge \\
&s_1[indexof(0,0)] = a_1 \wedge ... \wedge s_n[indexof(0,0)] = a_n \wedge \\
&\forall l, i : \mathbb{N}\bullet \\
&\quad 0 \leq l < x \wedge 0 \leq i < sizelevel(l) \Rightarrow \\
&\quad\quad \exists l_1, ..., l_m, i_1, ..., i_m : \mathbb{N}\bullet \\
&\quad\quad\quad child_m^1(l, i, l_1, i_1) \wedge ... \wedge child_m^m(l, i, l_m, i_m) \wedge \\
&\quad\quad\quad p_1\omega \wedge \\
&\quad\quad\quad s_1[indexof(l_1, i_1)] = e_1^1\omega \wedge ... \wedge s_n[indexof(l_1, i_1)] = e_n^1\omega \wedge \\
&\quad\quad\quad \vdots \\
&\quad\quad\quad s_1[indexof(l_m, i_m)] = e_1^m\omega \wedge ... \wedge s_n[indexof(l_m, i_m)] = e_n^m\omega
\end{aligned}
$$

with $\omega = [x, a_1, ..., a_n := l, s_1[indexof(l, i)], ..., s_n[indexof(l, i)]]$.

Mutual Recursion

Finally, primitive mutual recursion allows for multiple predicates to have arbitrary cyclic dependencies. The general form for such primitive mutually recursive predicates is

$$P_1(x, a_1, ..., a_n) \equiv \psi_1, \ldots, P_k(x, a_1, ..., a_n) \equiv \psi_k$$

where

$$\psi_i \equiv (x = 0 \wedge p_0^i) \vee (x > 0 \wedge p_1^i \wedge P_{j_1}(x-1, e_1^{i,1}, ..., e_n^{i,1}) \wedge ... \wedge P_{j_{m_i}}(x-1, e_1^{i,m_i}, ..., e_n^{i,m_i}))$$

and $j_i \in \mathbb{N}_k^1$ for all $i \in \mathbb{N}_k^1$.

Note that it is always possible to construct a single, primitive tree-recursive predicate P of degree $\sum_i^k m_i$ with an additional parameter y such that $P(y, x, a_1, ..., a_n)$ is equivalent to $P_y(x, a_1, ..., a_n)$ for all $y \in \mathbb{N}_k^1$ as

$$P(y, x, a_1, ..., a_n) \equiv y = 1 \Rightarrow \psi_1' \wedge \ldots \wedge y = k \Rightarrow \psi_k'$$

where ψ_i' is equivalent to ψ_i with all calls to $P_j(x, a_1, ..., a_n)$ replaced by $P(j, x, a_1, ..., a_n)$.

Since the tree-recursive predicate P() can be processed further as outlined in the previous section, our translation can thus be applied to arbitrary primitive recursive predicates.

Discussion

Although the above translation works for primitive recursive predicates only and is hence limited to modelling primitive recursive functions, quantification easily allows for also encoding the μ-operator since

$$\mu x.P(x, \overrightarrow{a}) \Leftrightarrow \exists x : \mathbb{N} \bullet P(x, \overrightarrow{a}) \wedge \forall x' : \mathbb{N} \bullet 0 \leq x' < x \Rightarrow \neg P(x', \overrightarrow{a})$$

The combination of quantification and primitive recursive predicates hence allows encoding arbitrary μ-recursive functions. However, μ-recursive functions are known to allow for simulating Turing-Machines and hence also allow for expressing the Halting-Problem, which is the very reason why satisfiability of our Assertion Language **AL** is undecidable. In practice, of course, we are not trying to simulate Turing Machines. However, above observation is still relevant since the ability to do so significantly reduces the probability of our assertions falling into a decidable subset of our Assertion Language and hence places a heavy burden on the theorem prover. In our experience, Z3 was able to solve many simple problems despite their use of recursive predicates. However, several larger verification conditions using them resulted in the answer 'unknown'. We will show some of these cases when presenting the respective case studies.

10.4. Practical Issues with the Rule of Adaptation

Testing the Verification Tool on sample **dyn** programs revealed a number of practical issues regarding the so-called Rule of Adaptation used for handling Method Calls in our Weakest Precondition Calculus. This rule was originally introduced by Hoare [42] and later enhanced by Olderog [63].

Before its introduction, Hoare logics used a number of so-called "Adaptation Rules" to adapt a method's contract $\{p'\}a_0.m(a_1, ..., a_n)\{q'\}$ to derive the statements necessary to reason about any call $\{p\}e_0.m(e_1, ..., e_n)\{q\}$ to this method. As was shown by Gorelick [33] in his seminal completeness proof, it is possible to use the most general contract $\{WP(a_0.m(a_1, ..., a_n), \overline{x} = \overline{v})\}a_0.m(a_1, ..., a_n)\{\overline{x} = \overline{v}\}$ and then derive any true statement about the method from it using these Adaptation Rules. Unfortunately, the proof is non-constructive and uses the Rule of Consequence which makes it hard to automate the process of deriving a weakest precondition for a method call with respect to a given postcondition as required for a Weakest Precondition Calculus. For this reason, Hoare suggested the Rule of Adaptation which basically fuses all previous Adaptation Rules into one. Its basic idea is to treat the method as the most general finitely-based state transformer satisfying it's contract and deriving a weakest precondition ensuring that even this generalized state transformer satisfies the postcondition (which implies the same property for every less general one) (see [63]).

10.4.1. Simplifying Intermediate Results

Dynamic dispatch in object-oriented programs lets the program's control flow depend on type information. In the absense of such type information due to dynamic typing,

we hence have to take all options into consideration when reasoning about method calls. The weakest precondition of a method call is thus a large disjunction over all (n) classes supporting a method matching the call's name and arity.

When the postcondition of such a method call is large, this can become problematic, since its size is multiplied by the factor n in the processing of the WPC. Note that, especially for constructor calls (init) n can be quite large. In order to cope with the explosion in the size of verification conditions, postconditions of method calls are hence ideal targets for automatic simplification procedures.

Also, all those cases are prefixed by a typecheck $[\![v_0]\!] \in \{C\}$, which in the common case of a constructor call is replaced by either $true$ or $false$ in the substitution for object creation (see Appendix A) since v_0 refers to just the object created by $\mathbf{new}_C$ as shown in the following:

$$WP\left(\begin{array}{r} [\![v_0]\!] \in \{C_1\} \wedge p_1 \vee \\ \vdots \\ v_0 := \mathbf{new}_{C_i}, [\![v_0]\!] \in \{C_i\} \wedge p_i \vee \\ \vdots \\ [\![v_0]\!] \in \{C_n\} \wedge p_n \end{array}\right) = \begin{array}{l} false \wedge p_1 \vee \\ \vdots \\ true \wedge p_i \vee \\ \vdots \\ false \wedge p_n \end{array} \Leftrightarrow p_i$$

All disjuncts starting with $false$ herein are obviously unsatisfiable and can hence be pruned. We hence designed the simplification procedure to also be effective in pruning these irrelevant disjuncts from the weakest preconditions of constructor calls.

The Simplifier implemented in our Verification Tool recursively applies several simple rewrite rules listed in Appendix B.

10.4.2. Detecting Ill-Suited Method Specifications

The Rule of Adaptation allows for the fully automatic adaptation of a method's specification to the circumstances of each of its calls. However, like for the Adaptation Rules used in Hoare logic prior to its introduction, its completeness is only established for the case of most general method specifications, i.e., the case where all method specifications are of the form

$$\{WP(S, \overline{\mathrm{x}} = \overline{v})\} S \{\overline{\mathrm{x}} = \overline{v}\}.$$

This is due to the fact that such a most general correctness formula captures the entire graph of the method in question and hence is guaranteed to contain enough information for all adaptations that might be required.

In practice, users of a Verification Tool like ours cannot be expected to be aware of subtle proof-theoretic properties of the proof system used underneath. While most theoreticians think of completeness as a guarantee that they never get stuck while proving a valid property with a certain proof system, the following example demonstrates that in this case, the assumption of most general method contracts not only has the character of a "best practice", but that deviating from it can cause rather subtle problems.

For this example, we will leave our dynamically typed setting, as dynamic typing is not relevant for this issue and the mapping predicates for **dyn** distract unnecessarily. We will hence suppose all variables to be of type $\mathbb{N}$.

Consider a user annotating a method for the addition of two natural numbers with the following contract

$$\textbf{method } add(a, b) \textbf{ requires } true \textbf{ ensures } \mathbf{r} = a + b$$

This contract is valid and hence accepted by the Verification Tool. However, when calculating the weakest precondition of a call $add(x, y)$ with respect to the postcondition $\mathbf{r} < 5$ using the Rule of Adaptation, we get

$$true \wedge \forall a, b, \mathbf{r} \bullet \mathbf{r} = a + b \rightarrow \mathbf{r} < 5,$$

which is equivalent to $false$. This is astonishing as it clearly is possible to satisfy the postcondition by choosing values for x and y that satisfy $\mathrm{x} + \mathrm{y} < 5$.

While the Rule of Adaptation must clearly be allowed to return "false", and the Hoare triple $\{false\}add(x, y)\{\mathbf{r} < 5\}$ is clearly valid, in this case $false$ is NOT the weakest precondition as $\mathrm{x} + \mathrm{y} < 5$ is also a valid answer. We are hence dealing with an issue of incompleteness rather than unsoundness.

At the heart of the matter lies the fact that the Rule of Adaptation abstracts the actual method $add()$ to the most general finitely-based state-transformer satisfying its specification. Note that in this case, the specification

$$\textbf{method } add(a, b) \textbf{ requires } true \textbf{ ensures } \mathbf{r} = a + b$$

is also satisfied by the method

```
method add(a,b) {
  a := 5;
  b := 5;
  10
}
```

as it fails to require that a and b remain invariant. Note that the latter method does NOT offer any way to satisfy the postcondition $\mathbf{r} < 5$ and hence leaves $false$ as the only valid option for the weakest precondition calculus.

Of course such subtle pitfalls should be safeguarded against. We hence would like our tool to check method contracts not only for validity, but also for generality.

The easiest way to do so would be to enforce the postconditions to always be $\overline{\mathrm{x}} = \overline{v}$. However, note that $\overline{\mathrm{x}}$ abbreviates a sequence containing all local variables, which can be quite lengthy. Also, since methods are evaluated within begin-local blocks, the only local variable that can actually be modified is **r** while all other local variables are anyway guaranteed to be invariant. This strategy would hence force our users to add a quite lengthy conjunction of $\mathrm{x} = v$ terms to the precondition in order to make it imply the weakest precondition.

From the standpoint of usability it would hence be advisable to automatically add the terms $\bar{x} = \bar{v}$ to both pre- and postcondition of each method contract where $\bar{x}$ contains all local variables not occurring free in both. Note, however that this does not solve the problem since the variables a and b occur free in the postcondition of above example.

Our investigations culminated in the observation that the formula

$$\exists \overrightarrow{v'}, \overrightarrow{v''} \bullet \forall \overrightarrow{x} \bullet q[\overrightarrow{\forall}/\overrightarrow{v'}] \wedge \neg q[\overrightarrow{\forall}/\overrightarrow{v''}]$$

where $\{\overrightarrow{\forall}\} = free(p) \cap free(q) \cap \mathfrak{V}_L$, $|\overrightarrow{v'}| = |\overrightarrow{v''}| = |\overrightarrow{\forall}|$ and $\{\overrightarrow{x}\} = (free(q) \cap \mathfrak{V}_L) \setminus \{\overrightarrow{\forall}\}$ seems to be a good indicator for the generality of a method contract $\{p\}S\{q\}$. Unfortunately, due to time constraints we were neither able to implement it nor to properly evaluate its merits.

10.4.3. Invariance

Inspecting verification conditions for which the SMT-Solver returned "unknown" led to the observation that every application of the Rule of Adaptation introduces several quantifiers and hence makes solving the verification condition significantly harder. However, often the method call itself was completely irrelevant to the remaining condition. This problem is aggravated by the abundance of method calls introduced by desugaring operations to method calls.

In order to reduce the burden on the solver, it seems like a promising strategy to negate the default assumption that every method call is relevant for the property being verified. In this mindset it makes sense to test method calls for relevance before allowing them to "clutter" the weakest precondition with their Rule of Adaptation-induced quantifiers.

Testing a method call $v_0.\mathrm{m}(v_1, ..., v_n)$ for relevance with respect to a postcondition q takes the form of a solver-query for

$$test \equiv q \rightarrow WP(v_0.m(v_1, ..., v_n), q)$$

If valid, then clearly, the postcondition q is invariant over the method call and can hence also serve as its precondition. The Alternative Weakest Precondition Calculus WPC' applying this strategy would hence replace the clause for method calls with

- $WPC'(v_0.m(v_1, ..., v_n), q) = \begin{cases} q & \text{if } test \\ WPC(v_0.m(v_1, ..., v_n), q) & \text{otherwise} \end{cases}$

Again, although manually applying this strategy to verification conditions for which Z3 returned "unknown" sometimes had positive effects, due to time constraints we were not able to implement and properly evaluate this strategy.

11. Case Studies

In this chapter, the concepts developed previously will be demonstrated on a number of case studies. First, in order to demonstrate that the Hoare logic described in Chapter 4.5 and in particular the verification tool implementing it enable the verification of real dynamically typed programs, Section 11.1 will apply it to a small example that was chosen to violate static typing rules and that could hence not be realized in a statically typed language without rewriting it significantly. Also, we will show that the concept of Interactive Type Inference allows not only to verify the type safety of typical dynamically typed programs (Section 11.2), but also sufficies for rather involved typing problems like in our Evaluator Example (Section 11.4). Functional Correctness proofs will be given for a small example chosen to demonstrate the merits of our Layer of Abstraction in Section 11.3 as well as the Evaluator Example (Section 11.4).

11.1. Dynamic Typing

In order to show how our logic and tool are able to verify the type-safety of programs that would not be accepted by a static type checker, we chose the simple example depicted in Figure 11.1. While its type safety problem boils down to path sensitivity and could thus be solved also by advanced type inference algorithms, most statically typed languages do not use a path sensitive algorithm for reasons of scalability. It can hence be considered a typical dynamically typed program. Also, since the type inference algorithm described in Section 2.1.3 is not able to establish type safety on its own, it is well-suited to demonstrate the interaction between our Program Logic and the Interactive Type Inference.

Although it does not look like it, there are plenty of method calls hidden in this example. Remember that the operation + in line 11 is desugared to a call to a method m_+ on their first operand (x in this case) and that constants like 5 are desugared to quite a few constructor- and method calls (see Figure 2.1). To make type safety non-obvious, we give the following assumptions about these methods:

$$\{\mathbb{N}(\mathrm{v}_0, n_0) \wedge \mathbb{N}(\mathrm{v}_1, n_1)\}\mathrm{v}_0.m_+(\mathrm{v}_1)\{\textbf{typesafe} \wedge \mathbb{N}(\mathbf{r}, n_0 + n_1)\}$$
$$\{\mathbb{S}(\mathrm{v}_0, s_0) \wedge \mathbb{S}(\mathrm{v}_1, s_1)\}\mathrm{v}_0.m_+(\mathrm{v}_1)\{\textbf{typesafe} \wedge \mathbb{S}(\mathbf{r}, s_0 s_1)\}$$

So + is a typesafe operation when applied to numerics (where is denotes addition) and strings (where is denotes concatenation), but we do not know anything about applications to mixed operands (adding a string to a numeric or the other way around).

Since a path-insensitive type inference (like the one described in Section 2.1.3 when not using trusted assumptions) would merge the abstract states resulting from both

```
1   method num_or_string(b)
2     requires [[b]] ∈ {boolean}
3     ensures typesafe
4   {
5     if b then
6       x = y = 5
7     else
8       x = "foo"; y = "bar"
9     end;
10    assert ([[x]] ∈ {num} ∧ [[y]] ∈ {num}) ∨ ([[x]] ∈ {string} ∧ [[y]] ∈ {string});
11    x + y
12  }
```

Figure 11.1.: A simple dynamically typed program

branches at the end of the conditional, it would consider both variables x and y as of type $\{num, string\}$ and thus has to raise a type error as it could not exlude the possibility of a mixed-type application of the operation + in line 11.

However, adding the assertion in line 10 resolves this problem, as it contains exactly the information that the type inference algorithm needs to establish type safety of the operation in line 11. At the same time, the assertion can be automatically established by our verification tool that is based on the Tagged Weakest Precondition Calculus given in Section 10.2. We have hence shown above program to be type-safe. The source code of this case study as well as all files neccessary to reproduce the verification can be found in the folder `case␣studies/dynamic-typing` in the supplementary material.

11.2. Coerce Protocol

The coerce protocol is a mechanism used throughout the Ruby standard library. It is fundamentally linked to dynamic typing as it significantly complicates the application of static typing even to the most trivial expressions of the programming language such as $a + b$. We hence consider it an essential benchmark for our typing technique since failing in this respect would entail failing to be applicable to dynamically typed languages like Ruby.

The mechanism serves to allow the extension of infix operators $(+, *, -,$ etc) to user-defined data-types. Like many other pure object-oriented languages, Ruby desugars infix operators to method calls. This allows for user-defined data types to also support them by implementing these methods. However, note that this is only the case for expressions of the form $a + b$ where a is an instance of the user-defined data-type, but does not work for expressions of the form $b + a$ whenever b is an instance of a type defined elsewhere (for instance in the standard library). Since especially for commutative operations like addition, these two expressions should be equivalent, this

limitation is often perceived as inconvenient and unintuitive.

The coerce protocol as used in the Ruby standard library offers a solution to this problem. Instead of implementing the operation + as

```
method op+(other) {
  B+ /* the actual addition */
}
```

it is implemented as

```
class num {
  method op+(other)
  {
    if other is_a? num then
      B+ /* the actual addition */
    else
      t = other.coerce(self);
      t.first() + t.second();
    end
  }
}
```

hence bringing a second method `coerce()` into play. This method is meant to convert the operands into suitable data-types before applying the actual operation. It allows for converting both operands as well as for reversing their order. Consider the following user-defined data-type implementing elements of a Galois field. Such a *Galois field* or *Finite field* GF_p for some prime number p, contains p elements $0, ..., p-1$. With the operations addition, subtraction, multiplication and division (all modulo p), GF_p satisfies all field axioms.

```
class Galois {

  method init(element, prime) {
    @element = element.mod(prime);
    @prime = prime;
    self
  }

  method add(other) {
    if other is_a? numeric then
      assert [[other]] ∈ {numeric};
      res = @element + other;
      assert [[res]] ∈ {numeric};
      new Galois(res, @prime)
    else
      if other is_a? Galois then
        assert [[other]] ∈ {Galois};
```

```
        res = @element + other.element();
        assert [[res]] ∈ {numeric};
        new Galois(res, @prime)
      else
        t = other.coerce(self,new tuple(null, null));
        t.first().add(t.second())
      end
    end
  }

  method element() {
    @element
  }

  method multiply(other) {
    if other is_a? numeric then
      assert [[other]] ∈ {numeric};
      res = @element * other;
      assert [[res]] ∈ {numeric};
      new Galois(res, @prime)
    else
      if other is_a? Galois then
        assert [[other]] ∈ {numeric};
        res = @element * other.element();
        assert [[res]] ∈ {numeric};
        new Galois(res, @prime)
      else
        l = other.coerce(self, new tuple(null, null));
        l.get(0).multiply(l.get(1))
      end
    end

  }

  method coerce(other, tuple) {
    tuple.set_first(null);
    if other is_a? Galois then
      tuple.set_first(other)
    else
      if other is_a? numeric then
        assert [[other]] ∈ {numeric};
        tuple.set_first(new Galois(other, @prime))
      else
        typeerror
      end
```

```
    end;
    tuple.set_second(self);
    tuple
  }

  method toString() {
    @element.toString() + " mod " + @prime.toString()
  }

}
```

with a standard library implementing the coerce protocol, this implementation allows not only for **new** `Galois(2,3) + 5` and **new** `Galois(2,3) + new Galois(1,3)`, but also for `5 + new Galois(2,3)`.

While this flexibility and extensibility is clearly desirable, the downside is that a static type system usually fails to determine the return type for operations implemented this way. The reason for this is twofold: For one, most type systems fail to handle the type-distinctions introduced by dynamic type checks (`is_a?`). However, there are specialized type systems for dynamically typed languages like occurrence typing [79] that are able to handle this. The second reason is that operations like + are typically used quite often within a program and when extending their applicability to multiple data-types like in the example, it is quite likely that these occurrences differ in the data-types involved. Since in the presence of dynamic type checks it is important to distinguish these calls in order to calculate precise results for each of them, typing larger programs using the coerce protocol also requires context sensitivity. Additionally, since all operations call the same coerce-method, the coerce protocol requires a context-depth of at least two, which is a requirement hardly met by type inference algorithms in common use.

Verification

The blue assertions in above listing are sufficient to establish type safety by means of the type inference algorithm outlined in Section 2.1.3 using them as trusted assumptions. Note that these assertions can be trivially established using the following information:

- The instance variable @element of class *Galois* is of type *numeric*,
- $\{[\![\mathrm{v}_0]\!] \in \{numeric\} \wedge [\![\mathrm{v}_1]\!] \in \{numeric\}\} v_0.op_+(v_1) \{[\![\mathbf{r}]\!] \in \{numeric\}\}$, and
- $\{[\![\mathrm{v}_0]\!] \in \{numeric\} \wedge [\![\mathrm{v}_1]\!] \in \{numeric\}\} v_0.op_*(v_1) \{[\![\mathbf{r}]\!] \in \{numeric\}\}$

While the first can be established by the type inference algorithm itself, the latter two are derivable from the method contracts of op_+ and op_*. The source code and all files neccessary to reproduce this verification can be found in the folder `case␣studies/coerce-protocol` of the supplementary material.

11.3. In-Place Value Switching

This case study was taken from the Master Thesis of Reinhard Kluge [46]. A Translation to **dyn** yields the following program:

```
assert x = x' ∧ y = y';
x = x + y;
y = x - y;
x = x - y;
assert x = y' ∧ y = x';
```

The rationale behind this short program is to swap the values of the variables x and y without using a third (temporary) variable. Kluge [46] could show that this is correct whenever the values referenced by x and y are of a data-type that adheres to the axioms for abelian groups with respect to the operations $+$ and $-$.

Axioms for abelian groups:

$$\forall a, b, c : G$$

$\bullet a + b = b + a$	(Commutativity)
$\bullet (a + b) + c = a + (b + c)$	(Associativity)
$\bullet a + 0 = 0 + a = a$	(Neutral Element)
$\bullet \exists (-a) : G \bullet a + (-a) = 0$	(Inverse Elements)

The operation $-$ is then defined as $a - b \triangleq a + (-b)$.

In the context of **dyn**, this means that above program should be correct as long as the objects referenced by x and y support methods m_+ and m_- that satisfy above axioms.

Verification (Using the Layer of Abstraction)

Under the assumption that x and y are of type integer (which implements elements of $\mathbb{Z}$, which form an abelian group), we can verify above program using the Layer of Abstraction. Every program step can be handled by the rule PURE ASGN and its weakest precondition can hence be calculated using backward substitution as shown below:

```
assert x̂ = x' ∧ ŷ = y';
assert (x̂ + ŷ) − ((x̂ + ŷ) − ŷ) = y' ∧ ((x̂ + ŷ) − ŷ) = x';
x = x + y
assert x̂ − (x̂ − ŷ) = y' ∧ (x̂ − ŷ) = x';
y = x - y
assert x̂ − ŷ = y' ∧ ŷ = x';
x = x - y
assert x̂ = y' ∧ ŷ = x';
```

The only interesting case in this proof is the rule of consequence at the very beginning, just like in the statically typed variant considered by Kluge. Since the Layer of Abstraction reduces this verification condition to a statement about integers, it can easily be established using Z3, thus proving the program correct. Another interesting case in this respect is the type-independent generalization of this verification as proposed by Kluge: When replacing the operations + and − by uninterpreted functions and axiomatizing them with above axioms for abelian groups, Z3 is still able to proof the verification condition. However, in this case the correctness proof is much more general: It not only applies for the case of x and y being of type integer, but for all data-types supporting the operations + and − such that their instances form an abelian group.

We feel that this generalization lends itself well to the polymorphic nature of dynamically typed programs. Some ideas as to how it could contribute to a more modular verification of such programs can be found in Section 13.2.

Verification (Without the Layer of Abstraction)

When attempting to verify this program without the Layer of Abstraction, not only do we have to use the mapping predicate $\mathbb{Z}$ to map the variables x and y to integers before attempting any arithmetic on them, but also must we refrain from using the convenient rule PURE ASGN, thus handling + and − as ordinary method calls. Since method calls are handled using the Rule of Adaptation, as a very first step we need to establish

$$\{x - y = y' \wedge y = x'\}\ \texttt{x = x - y}\ \{x = y' \wedge y = x'\}$$

using the Rule of Adaptation. However, as it turns out, Z3 already returns "unknown" for the resulting verification condition of this very first step (whose translation to SMT-LIB2 can be found in Appendix D). We conclude that verifying the above program without the Layer of Abstraction is not possible due to incompleteness of Z3. Hence using type information in the Layer of Abstraction seems to be a good strategy to significantly reduce the burden our verification places on the SMT solver when verifying **dyn** programs. The **dyn** source code as well as all other files necessary for reproducing both types of verification can be found in the folder `case␣studies/in-place␣value␣switching` of the supplementary material.

11.4. Nested Lists as Abstract Syntax Trees

To demonstrate how the techniques developed enable the convenient verification of dynamically typed programs despite hard typing problems, we will proof the evaluator example from Section 8.1 both type-safe and correct. Figure 11.3 shows all annotations[1] necessary to prove that `calc()` derives a given term's value. The data structure used to model parse-trees is depicted in Figure 11.2.

[1] Again, all recursive predicates can instead be expressed using quantification over arrays, at the expense of readability (see Section 10.3)

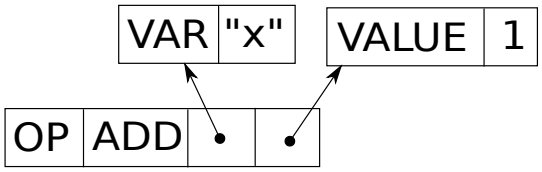


Figure 11.2.: Nested lists are used to model parse trees of arithmetic expressions. Exemplary structure for the expression $x + 1$.

Verification (using the Layer of Abstraction)

Type safety: One can focus on the verification of type safety by replacing the postcondition of method `calc()` by $[\![\mathbf{r}]\!] \in \{numeric\}$. The given invariant then enables deriving the assertions in lines 23, 26, 29, and hence a proper typing of the remaining program. The invariant can be established directly from the definition of $parsetree2()$. With the types of `env` ($\mathbb{M}$) and `r` ($\mathbb{N}$) known, their mapping can be automated. The complex ad-hoc data structure `tree` is given the (imprecise) type $\mathbb{L}$ ($\sim \mathbb{N} \mapsto \mathbb{O}$) and its elements hence need manual mapping. These mappings are encapsulated in predicates ($valuetree2, vartree3, optree5$) and can furthermore be ignored.

The verification conditions for establishing the type safety assertions are then as follows:

(TS1) line 23: $Inv \wedge parsetree2(\widehat{\text{tree}}, \widehat{\text{env}}) \wedge \widehat{tree}[0] = \widehat{VALUE} \rightarrow valuetree1(\widehat{\text{tree}})$,

(TS2) line 27: $Inv \wedge parsetree2(\widehat{\text{tree}}, \widehat{\text{env}}) \wedge \widehat{tree}[0] = \widehat{VAR} \rightarrow vartree2(\widehat{\text{tree}}, \widehat{\text{env}})$,

(TS3) line 31: $Inv \wedge parsetree2(\widehat{\text{tree}}, \widehat{\text{env}}) \wedge \widehat{tree}[0] = \widehat{OP} \rightarrow optree2(\widehat{\text{tree}}, \widehat{\text{env}})$.

Establishing the postcondition generates the following verification conditions:

(TS4) line 23: $valuetree1(\widehat{\text{tree}}) \rightarrow [\![\widehat{tree}[1]]\!] \in \{numeric\}$,

(TS5) line 27: $vartree2(\widehat{\text{tree}}, \widehat{\text{env}}) \rightarrow [\![\widehat{env}[\widehat{tree}[1]]]\!] \in \{numeric\}$,

(TS6) line 31: $optree2(\widehat{\text{tree}}, \widehat{\text{env}}) \rightarrow parsetree2(\widehat{tree}[2], \widehat{\text{env}})$,

(TS7) line 31: $optree2(\widehat{\text{tree}}, \widehat{\text{env}}) \rightarrow parsetree2(\widehat{tree}[3], \widehat{\text{env}})$.

Note that since $[\![null]\!] \in \{numeric\}$ holds, we do not have to take care about the cases where `calc()` returns *null* and can hence remove the assertions in lines 35 and 39. Given the postcondition and the assertions in lines 23, 26, 29 as trusted assumptions, type safety can then be established using the type inference algorithm described in Section 2.1.3.

The invariant Inv and all verification conditions could be verified using Z3[2]. We hence conclude that our Evaluator Example is type-safe.

[2]In order to verify (TS3) with Z3, we had to break the recursion by replacing the recursive calls to $parsetree2()$ in the definition of $optree5()$ by *true*. However, since the very same call to $optree5$

```
1  def valuetree2(t : 𝕃, n : ℕ) ≜ ℕ(t[0], VALUE^) ∧ ℕ(t[1], n);
2  def valuetree1(t : 𝕃) ≜ ∃n : ℕ • valuetree2(t, n);
3  def vartree3(t : 𝕃, e : 𝕄, x : 𝕊) ≜ ℕ(t[0], VAR^) ∧ 𝕊(t[1], x) ∧ e[x] ≠ null
4  def vartree2(t : 𝕃, e : 𝕄) ≜ ∃x : 𝕆 • vartree3(t, e, x);
5  def optree5(t : 𝕃, e : 𝕄, op : ℕ, l : 𝕃, r : 𝕃) ≜ ℕ(t[0], OP^) ∧ ℕ(t[1], op) ∧ 𝕃(t[2], l) ∧
6      parsetree2(l, e) ∧ 𝕃(t[3], r) ∧ parsetree2(r, e);
7  def optree2(t : 𝕃, e : 𝕄) ≜ ∃op : ℕ, l : 𝕃, r : 𝕃 • optree5(t, e, op, l, r);
8  def parsetree2(t : 𝕃, e : 𝕄) ≜ valuetree1(t) ∨ vartree2(t, e) ∨ optree2(t, e);
9  def treeval3(tree : 𝕃, env : 𝕄, n : ℕ) ≜
10       valuetree2(tree, n)  ∨
11       ∃x : 𝕊 • vartree3(tree, env, x) ∧ ℕ(env[x], n)  ∨
12       ∃op, n_l, n_r : ℕ, l, r : 𝕃 • optree5(tree, env, op, l, r) ∧ treeval3(l, env, n_l) ∧ treeval3(r, env, n_r)  ∧
13         op = ADD^ → n = n_l + n_r  ∧ ...;
14
15 invariant Inv ≡ ∀t : 𝕃, e : 𝕄 • parsetree2(t, e) →
16         t[0] = VALUE^ → valuetree1(t) ∧ t[0] = VAR^ → vartree2(t, e) ∧ t[0] = OP^ → optree2(t, e);
17
18 method calc(env, tree)
19   requires parsetree2(tree^, env^)
20   ensures treeval3(tree^, env^, r^)
21   {
22     if tree[0] == VALUE then
23       assert valuetree1(tree^); tree[1]
24     else
25       if tree[0] == VAR then
26         assert vartree2(tree^, env^); env[tree[1]]
27       else
28         if tree[0] == OP then
29           assert optree2(tree^, env^);
30           if tree[1] == ADD then
31             calc(env, tree[2]) + calc(env, tree[3])
32           else
33             if ...
34             else
35               assert false; null
36             end
37           end
38         else
39           assert false; null
40         end
41       end
42     end
43   }
```


Figure 11.3.: Correctness proof for the evaluator example using the Layer of Abstraction

Functional Correctness: The type information from our type safety proof allows for identifying numerous pure expressions (`tree[1]`, `env[tree[1]]`, `tree[1] == ADD`, etc.). Establishing the specified property for the first two branches then only requires applying PURE EXPR. The conditional in line 5 can be handled by PURE COND. The verification conditions for functional correctness are as follows:

- (VC1 – 3) ≡ (TS1 – 3) from above.
- (VC4) line 23: $valuetree1(\widehat{\text{tree}}) \rightarrow treeval3(\widehat{\text{tree}}, \widehat{\text{env}}, \widehat{\widehat{\text{tree}[1]}})$,
- (VC5) line 27: $vartree2(\widehat{\text{tree}}, \widehat{\text{env}}) \rightarrow treeval3(\widehat{\text{tree}}, \widehat{\text{env}}, \widehat{\text{env}}[\widehat{\widehat{\text{tree}[1]}}])$,
- (VC6 – 7) ≡ (TS6 – 7) from above.
- (VC8) $treeval3(\widehat{\widehat{\text{tree}[2]}}, \widehat{\text{env}}, n_l) \wedge treeval3(\widehat{\widehat{\text{tree}[3]}}, \widehat{\text{env}}, n_r) \wedge \widehat{\widehat{\text{tree}[0]}} = \widehat{OP} \wedge \widehat{\widehat{\text{tree}[1]}} = \widehat{ADD} \rightarrow treeval3(\widehat{\text{tree}}, \widehat{\text{env}}, n_l + n_r)$,
- (VC9) $parsetree2(\widehat{\text{tree}}, \widehat{\text{env}}) \wedge \widehat{\text{tree}[0]} \neq \widehat{VALUE} \wedge \widehat{\text{tree}[0]} \neq \widehat{VAR} \wedge \widehat{\text{tree}[0]} \neq \widehat{OP} \rightarrow false$,
- (VC10) $parsetree2(\widehat{\text{tree}}, \widehat{\text{env}}) \wedge \widehat{\text{tree}[0]} = \widehat{OP} \wedge \widehat{\text{tree}[1]} \neq \widehat{ADD} \rightarrow false$.

Unfortunately, it turns out that Z3 is not able to handle the encoded version of the recursive predicate $treeval3()$ as it returns "unknown" even for the most trivial formulas containing it (the SMT-LIB translation for $treeval3$ can be found in Appendix D). However, manually unfolding its recursive version and generalizing the verification conditions by replacing all further occurrences by an uninterpreted function enabled Z3 to solve them. Only (VC8) (of the form $A \rightarrow C$) needed to be split into two implications $A \rightarrow B$ and $B \rightarrow C$ for some assertion B. The original (VC8) then follows by transitivity of implication. We conclude that the method `calc()` satisfies its specification.

The source code of this case study as well as all files neccessary to reproduce the verification of type safety and functional correctness can be found in the folder `case␣studies/evaluator-example` in the supplementary material.

is used both by $parsetree2()$ on the left of the implication and by $optree2()$ on the right of it, reflexivity of implication ensures that this manipulation does not alter the result. To verify (TS6) and (TS7), we had to replace the recursive definition of $parsetree2$ by an uninterpreted function. However, as this proof shows that the implication holds regardless of the concrete definition of $parsetree2$, it also holds for this particular definition.

12. Extensions

As stated initially, **dyn** and **stat** are simplified model languages tailored to provide insight into the issue of dynamic typing in program verification. In order to bridge the gap to real-world dynamically typed languages, we will in this chapter consider a number of extensions to the developed formalisms that brings them closer to their real-world counterparts.

12.1. Non-Fatal Typeerrors

Unlike **dyn**, real-world dynamically typed languages often have non-fatal type errors, meaning they are implemented as exceptions and hence recoverable. To model this kind of behaviour, we need to introduce a means of catching and handling such errors into the programming language **dyn**. Consider the following syntax extension for expressions:

$$e ::= \textbf{try}\ S\ \textbf{catch typeerror}\ S\ \textbf{end} \mid \textbf{try}\ S\ \textbf{catch fail}\ S\ \textbf{end}$$

with the following operational semantics for these expressions (so-called try-catch-blocks):

- $$\frac{\langle S_1, \sigma\rangle \ \xrightarrow{*}\ \textbf{final}\langle\tau\rangle}{\langle \textbf{try}\ S_1\ \textbf{catch typeerror}\ S_2\ \textbf{end}, \sigma\rangle \ \rightarrow\ \textbf{final}\langle\tau\rangle}$$ (OP-TRY-TS-SAFE)
- $$\frac{\langle S_1, \sigma\rangle \ \xrightarrow{*}\ \textbf{fail}\langle\tau\rangle}{\langle S', \sigma\rangle \ \rightarrow\ \textbf{fail}\langle S', \tau\rangle}$$ where $S' \equiv \textbf{try}\ S_1\ \textbf{catch typeerror}\ S_2\ \textbf{end}$ (OP-TRY-TS-FAIL)
- $$\frac{\langle S_1, \sigma\rangle \ \xrightarrow{*}\ \textbf{typeerror}\langle\sigma'\rangle \qquad \langle S_2, \sigma'\rangle \ \xrightarrow{*}\ \textbf{final}\langle\tau\rangle}{\langle \textbf{try}\ S_1\ \textbf{catch typeerror}\ S_2\ \textbf{end}, \sigma\rangle \ \rightarrow\ \textbf{final}\langle\tau\rangle}$$ (OP-TRY-TS-CATCH)
- $$\frac{\langle S_1, \sigma\rangle \ \xrightarrow{*}\ \textbf{final}\langle\tau\rangle}{\langle \textbf{try}\ S_1\ \textbf{catch fail}\ S_2\ \textbf{end}, \sigma\rangle \ \rightarrow\ \textbf{final}\langle\tau\rangle}$$ (OP-TRY-FS-SAFE)
- $$\frac{\langle S_1, \sigma\rangle \ \xrightarrow{*}\ \textbf{typeerror}\langle\tau\rangle}{\langle \textbf{try}\ S_1\ \textbf{catch fail}\ S_2\ \textbf{end}, \sigma\rangle \ \rightarrow\ \textbf{typeerror}\langle \textbf{try}\ S_1\ \textbf{catch fail}\ S_2\ \textbf{end}, \tau\rangle}$$ (OP-TRY-FS-TYPEERROR)
- $$\frac{\langle S_1, \sigma\rangle \ \xrightarrow{*}\ \textbf{fail}\langle\sigma'\rangle \qquad \langle S_2, \sigma'\rangle \ \xrightarrow{*}\ \textbf{final}\langle\tau\rangle}{\langle \textbf{try}\ S_1\ \textbf{catch fail}\ S_2\ \textbf{end}, \sigma\rangle \ \rightarrow\ \textbf{final}\langle\tau\rangle}$$ (OP-TRY-FS-CATCH)

Hence, try-catch blocks allow recovery from **typeerror**s or **fail**ures raised by the statement S_1, in which case they execute S_2. They thus enrich **dyn** by providing the ability to construct typesafe/failsafe programs from non-typesafe/non-failsafe subprograms.

It turns out that while our Tagged Hoare Logic provides a good framework to reason about such constructs that cross the traditional border given by the notions of correctness, in its current form it is not able to express the neccessary rules as it misses one crucial ingredient: tag-negation.

Negated tags allow expressing error-properties as opposed to safety-properties:

$$\{p\}S\{\neg\textbf{typesafe}\}$$

Denotes the fact that the program S will surely yield a **typeerror** whenever executed in a state satisfying p.

To allow this, we extend our set of tags by also adding negations:

$$\mathcal{B}ase\mathcal{T}ags = \{\textbf{terminates}, \textbf{typesafe}, \textbf{failsafe}\}$$
$$\mathcal{T}ags = \mathcal{B}ase\mathcal{T}ags \cup \{\neg\textbf{tag} \mid \textbf{tag} \in \mathcal{B}ase\mathcal{T}ags\}$$

and define the corresponding selectors as

$$\mathcal{S}_{\neg\textbf{terminates}}(\rho) = \begin{cases} \{\neg\bot\} & \text{if } \rho \text{ is finite} \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\neg\textbf{tag}}(\rho) = \begin{cases} \{\neg\Uparrow(\textbf{tag})\} & \text{if } \rho = C_0, ..., C_n \wedge C_n \neq \Uparrow(\textbf{tag})\langle\sigma\rangle \\ \{\sigma\} & \text{if } \rho = C_0, ..., C_n \wedge C_n = \Uparrow(\textbf{tag})\langle\sigma\rangle \\ \{\} & \text{otherwise} \end{cases}$$

for all other tags. From these definitions, it immediately follows that the tagged Hoare triple

$$\{true\}S\{\textbf{tag} \wedge \neg\textbf{tag}\}$$

is false for any program S and tag **tag**.

Within a Tagged Hoare Logic with negated tags, we are then able to give the following rules for reasoning about try-catch blocks:

RULE: TRY-TS-SAFE

$$\frac{\{p\}S_1\{\textbf{typesafe} \wedge \textbf{tags} \wedge q\}}{\{p\}\textbf{try } S_1 \textbf{ catch typeerror } S_2 \textbf{ end}\{\textbf{typesafe} \wedge \textbf{tags} \wedge q\}}$$

RULE: TRY-TS-CATCH

$$\frac{\{p\}S_1\{\neg\textbf{typesafe} \wedge \textbf{tags} \wedge r\} \qquad \{r\}S_2\{\textbf{typesafe} \wedge \textbf{tags} \wedge q\}}{\{p\}\textbf{try } S_1 \textbf{ catch typeerror } S_2 \textbf{ end}\{\textbf{typesafe} \wedge \textbf{tags} \wedge q\}}$$

RULE: TRY-FS-SAFE

$$\frac{\{p\}S_1\{\textbf{failsafe} \wedge \textbf{tags} \wedge q\}}{\{p\}\textbf{try } S_1 \textbf{ catch fail } S_2 \textbf{ end}\{\textbf{failsafe} \wedge \textbf{tags} \wedge q\}}$$

RULE: TRY-FS-CATCH

$$\frac{\{p\}S_1\{\neg\textbf{failsafe} \wedge \textbf{tags} \wedge r\} \qquad \{r\}S_2\{\textbf{failsafe} \wedge \textbf{tags} \wedge q\}}{\{p\}\textbf{try } S_1 \textbf{ catch fail } S_2 \textbf{ end}\{\textbf{failsafe} \wedge \textbf{tags} \wedge q\}}$$

12.1.1. Soundness

We extend the inductive argument from Section 5.1 with the following cases in the induction step:

- **TRY-TS-SAFE:**

  Partial correctness: Assuming $\vdash \{p\}S_1\{\textbf{typesafe} \wedge q\}$ for some statement S_1 and assertions p and q, then by the induction hypothesis $\models \{p\}S_1\{\textbf{typesafe} \wedge q\}$ also holds. Hence executing S_1 from a state $\sigma \models p$ will not yield a **typeerror** and if it terminates, it will do so in a final state σ' with $\sigma' \models q$. Hence, the above operational rule OP-TRY-TS-SAFE is the only one applicable and the statement $S \equiv \textbf{try } S_1 \textbf{ catch typeerror } S_2 \textbf{ end}$ thus satisfies the same specification.

  Termination: According to OP-TRY-TS-SAFE, when S_1 terminates, so will S, which satisfies the conclusion of TRY-TS-SAFE.

  Failsafety: In case S_1 fails, the operational rule OP-TRY-TS-FAIL is applicable and S will also fail, which satisfies the conclusion of TRY-TS-SAFE.

- **TRY-TS-CATCH:**

  Partial correctness: Assuming $\vdash \{p\}S_1\{\neg\textbf{typesafe} \wedge r\}$ and $\vdash \{r\}S_2\{\textbf{typesafe} \wedge q\}$ for some statements S_1 and S_2 and some assertions p,r and q. Then by the induction hypothesis, $\models \{p\}S_1\{\neg\textbf{typesafe} \wedge r\}$ and $\models \{r\}S_2\{\textbf{typesafe} \wedge q\}$ also hold. Hence, when executing S_1 from a state $\sigma \models p$, it will surely yield a **typeerror** in a state $\sigma' \models r$. Thus the above operational rule OP-TRY-TS-CATCH is the only one applicable and S_2 will be executed from σ'. By $\models \{r\}S_2\{\textbf{typesafe} \wedge \textbf{tags} \wedge q\}$ we know that when it terminates, it will do so without raising typeerrors and in a state $\sigma'' \models q$. Thus OP-TRY-TS-CATCH is indeed applicable and – if S_1 and S_2 terminate – then executing the statement $S \equiv \textbf{try } S_1 \textbf{ catch typeerror } S_2 \textbf{ end}$ from σ will also terminate in the state $\sigma'' \models q$, which was the desired conclusion.

  Termination: According to OP-TRY-TS-CATCH, if S_1 typeerrors and S_2 terminates, S will also terminate, which satisfies the conclusion of TRY-TS-CATCH.

  Failsafety: Same argumentation as for Termination.

- **TRY-FS-SAFE:** Same argument as for TRY-TS-SAFE, only with the roles of **fail** and **typeerror** reversed.

- **TRY-FS-CATCH:** Same argument as for TRY-TS-CATCH, only with the roles of **fail** and **typeerror** reversed.

We conclude that this extension preserves soundness of our Hoare logic.

12.1.2. Completeness

We extend the inductive argument from Section 5.2.6 by adding the following cases to the induction step:

- $S \equiv$ **try** S_1 **catch typeerror** S_2 **end**:

  Partial correctness: Assume $\models \{p\}S\{\textbf{typesafe} \wedge q\}$. Then by the expressivity of the assertion language, there is an assertion p' such that $\{p'\}S_1\{\textbf{typesafe}\}$ and $\{\neg p'\}S_1\{\neg\textbf{typesafe}\}$ hold. Now, according to the operational semantics, for $\{p\}S\{q\}$ to be true, $\{p \wedge p'\}S_1\{\textbf{typesafe} \wedge q\}$ must hold. Also, by the expressivity of the assertion language, there must be an intermediate assertion r such that $\{p \wedge \neg p'\}S_1\{\neg\textbf{typesafe} \wedge r\}$ and $\{r\}S_2\{q\}$ both hold. Now the last two are just the premises of the TRY-TS-CATCH rule. An application of said rule hence derives $\{p \wedge \neg p'\}S\{\textbf{typesafe} \wedge q\}$. Also, $\{p \wedge p'\}S_1\{\textbf{typesafe} \wedge q\}$ is the premise of the rule TRY-TS-SAFE. An application of said rule hence derives $\{p \wedge p'\}S\{\textbf{typesafe} \wedge q\}$. An application of the DISJ rule then derives $\{(p \wedge p') \vee (p \wedge \neg p')\}S\{\textbf{typesafe} \wedge q\}$ and an application of the CONS rule derives the desired result.

  Typesafety: Try-catch blocks are always typesafe. However, the **typesafe**-tag could be omitted using an application of the CONS rule when desired.

  Termination & Failsafety: TRY-TS-SAFE and TRY-TS-CATCH allow deriving other tags as long as they are derivable for S_1 and S_2 (which they are when true according to the induction hypothesis).

- $S \equiv$ **try** S_1 **catch fail** S_2 **end**: Just like the last case, only with **failsafe** instead of **typesafe**.

We conclude that this extension preserves the proof-theoretic properties of our Hoare logic.

12.2. Optional Variables

Since type information in dynamically typed languages is attached to values rather than variables, there is no need to declare the latter. Instead, in these languages, variables are created upon their first assignment and accessing a non-existing variable yields a runtime type error.

In order to model this behaviour in **dyn**, we introduce the special value $\boxtimes$ of type $\mathbb{O}$ for marking variables as "not yet created" and modify the operational semantics of variable access to cause **typeerror** when accessing them:

- $\langle \mathrm{v}, \sigma \rangle \rightarrow \textbf{final}\langle \sigma[\mathbf{r} := \sigma(\mathrm{v})] \rangle$ where $\mathrm{v} \in \mathfrak{V}$ and $\sigma(\mathrm{v}) \neq \boxtimes$
- $\langle \mathrm{v}, \sigma \rangle \rightarrow \textbf{typeerror}\langle \mathrm{v}, \sigma \rangle$ where $\mathrm{v} \in \mathfrak{V}$ and $\sigma(\mathrm{v}) = \boxtimes$

Additionally, the operational semantics of **begin local**-blocks and object creation must be modified to instantiate all local- and instance variables to ⊠ instead of *null*.

Now that **dyn** in this respect behaves just like its real-world counterparts, we also need to modify our axiomatic semantics to reason about it. First, we need to add the special value ⊠ as a constant to the logical expressions of our assertion language **AL**.

$$l ::= \boxtimes$$

Then, we replace the rules for variable access with the following:

AXIOM: VAR

$$\{p[\mathbf{r} := v]\}v\{p\}$$

VAR-TAG

$$\{\mathbf{typesafe} \rightarrow v \neq \boxtimes\}v\{\mathbf{tags}\}$$

Note: includes the case of $v \equiv \mathbf{self}$.

AXIOM: IVAR

$$\{p[\mathbf{r} := \mathbf{self}.@\mathrm{v}]\}@v\{p\}$$

IVAR-TAG

$$\{\mathbf{typesafe} \rightarrow \mathbf{self}.@\mathrm{v} \neq \boxtimes\}@v\{\mathbf{tags}\}$$

and the rules for **begin local**-blocks by

RULE: BLCK

$$\frac{\{p\}\overrightarrow{u}\,\overrightarrow{u} := \overrightarrow{t}\,\overrightarrow{\boxtimes}; S\{\mathbf{tags} \wedge q\}}{\{p\}\ \mathbf{begin\ local}\ \overrightarrow{u} := \overrightarrow{t}; S\ \mathbf{end}\{\mathbf{tags} \wedge q\}}$$

where $\mathfrak{V}_L \cap \mathit{free}(q) = \emptyset$, $\{\overrightarrow{u}\} \subset \mathfrak{V}_L$ and $\{\overrightarrow{t}\} \subseteq \mathfrak{V}_L \cup \{\mathrm{null}\}$, $\{\overrightarrow{u}\} = ((\mathit{var}(S) \cup \mathit{change}(S)) \cap \mathfrak{V}_L) \setminus \{\overrightarrow{u}\}$ and $\overrightarrow{\boxtimes}$ is a sequence of ⊠ constants of fitting length.

Finally, we redefine the initial value $\mathit{init}_C.@v$ for all instance variables $@v \in \mathfrak{V}_I \setminus \{@\mathbf{c}\}$ of objects of all classes $C \in \mathfrak{C}$ to be ⊠ instead of *null*, which is used in the substitution for object creation.

Only minor modifications are required to apply our soundness and completeness arguments to this modified Hoare logic and modified operational semantics (see [27]). In order to also adapt the Layer of Abstraction and the Interactive Type Inference to this modified setting, we introduce a new class $C_{\boxtimes} \in \mathfrak{C}$ and stipulate $\boxtimes.@\mathbf{c} = \theta_{C_{\boxtimes}}$.

Tracking uninitialzed variables is possible by adopting the following rule for typing method declarations:

- $$\frac{FG(S) = \underset{1}{\circ} \overset{G_S}{\dashrightarrow} \underset{2}{\bullet}}{FG(\text{method C.m}(p_1, ..., p_n)\{S\}) = \underset{M^n_{C.m}}{\circ} \overset{f}{\rightarrow} \underset{1}{\circ} \overset{G_S}{\dashrightarrow} \underset{2}{\circ} \rightarrow \underset{R^n_{C.m}}{\circ}}$$

  with $f \triangleq \lambda\hat{\sigma}.\bigsqcup\{f'_C(C^n_m) \mid C \in C^n_m(a_0)\}$ where $f'_C(\hat{\sigma}) = \hat{\sigma}[\mathit{self}, p_1, ..., p_n, \overrightarrow{v} := \{C\}, \hat{\sigma}(a_1), ..., \hat{\sigma}(a_n), \overrightarrow{C_{\boxtimes}}]$, $\{\overrightarrow{v}\} = ((\mathit{var}(S) \cup \mathit{change}(S)) \cap \mathfrak{V}_L) \setminus \{\overrightarrow{p}\}$ and $\overrightarrow{C_{\boxtimes}}$ is a sequence of $C_{\boxtimes}$-values of fitting length.

And instead of $\bot$ using the value $\hat{\boxtimes}$ (mapping all variables to $\{\boxtimes\}$) as extremal value when instantiating the monotone framework. Basically, this is the same modification as in the operational semantics: when the modified BLCK rule and $init_C$ initialize all local- and instance variables to $\boxtimes$, the type inference algorithm has to consider them as instances of $C_{\boxtimes}$. In order to ensure the additional constraint that variables must be assigned before access, we furthermore need to redefine the least precise typesafe typing $ty^\dagger$.

Definition 30. *For a program π, the* least precise type-safe typing $ty^\dagger_\pi$ *is a typing where*

- *for every method call* $e_0.m(e_1, ..., e_n)$

$$ty^\dagger_\pi(e_{0\bullet})(\mathbf{r}) = \{C \in \mathfrak{C} \mid C \textit{ supports method } m \textit{ of arity } n\},$$

- *in the case of* ***stat****, for every operation* $e_1 \oplus e_2$ *of type* $\mathbb{T}_1 \times \mathbb{T}_2 \mapsto \mathbb{T}$,

$$ty^\dagger_\pi(e_{1\bullet})(\mathbf{r}) = \mathbb{T}_1, ty^\dagger_\pi(e_{2\bullet})(\mathbf{r}) = \mathbb{T}_2$$

- *for every conditional or while loop with condition* e

$$ty^\dagger_\pi(e_\bullet)(\mathbf{r}) = \{bool\},$$

- *for all accesses to local variables* u

$$ty^\dagger_\pi(_\circ u)(u) = \mathfrak{C} \setminus \{C_{\boxtimes}\},$$

- *for all accesses to instance variables* $@v$

$$ty^\dagger_\pi(_\circ @v)(\mathbf{self}.@v) = \mathfrak{C} \setminus \{C_{\boxtimes}\}, \textit{ and}$$

- *for all other program locations* $L \in \mathbf{Loc}_\pi$

$$ty^\dagger_\pi(L) = \top.$$

By definition, a program π with optional variables is type-safe iff it has a sound[1] typing ty that is precise enough to establish type safety ($ty \sqsubseteq ty^\dagger_\pi$).

Using this definition, we have a type inference algorithm that statically prevents this additional kind of type error. Like its predecessor, it is sound but incomplete. We hence extend the typing assertions from Chapter 8 with literals of the form u = $\boxtimes$ and **self**.@v = $\boxtimes$ and observe that the Galois Connection between typings and typing assertions can also be trivially adapted to accomodate these additional values.

[1] if a method call, conditional or while loop is unreachable, sound typings may assign the type $\bot$ to its receiver / condition.

In summary, the entire formal development in this thesis can be trivially adapted to languages featuring optional variables and all results carry over to this case as well.

A related feature that is commonly found in dynamically typed languages[2] are *introspective tests for variable existence*. Since these are equivalent to

$$\textbf{try } x; true \textbf{ catch typeerror } false \textbf{ end}$$

they can be emulated in **dyn** by combining the extensions for Non-Fatal Type errors and Optional Variables.

12.3. Method Update

Languages combining class-based object orientation with dynamic typing (like Ruby and Python) often support a feature we call "method update" allowing for programs to override methods at runtime – often it is not even required for the arity of the new method to match the old version.

Transformation: As there can only be finitely many methods and only finitely many method update statements in a finite program, it is possible to match them against each other and regard each method update as an additional "version" of the method to be updated. Then, for each method $C.m$ in the original program having multiple versions, the corresponding **dyn** program must have a global state $g_{C.m}$ for storing the information which version is the current one. Since **dyn** programs usually do not have any global state, we accomplish this by introducing a class *Global* encapsulating this state information and passing a reference g to its only instance into each and every method in the program. Second, let there be versions $C.m_1, ..., C.m_k$ of a method $C.m$ with arities $n_1, ..., n_k$ within the original program, then the function $a_{C.m}(n) = \{C.m_i \mid n_i = n\}$ groups all versions having the same arity. For each arity n, such that $|a_{C.m}(n)| > 0$, the corresponding **dyn** program contains a method $C.m_n$ with arity n whose body is structured as follows

```
if g.version_C_m() == 1 then
 # body of C.m_1 here
else
  if g.version_C_m() == 2 then
   # body of C.m_2 here
  else
    ...
     else
       typeerror
     end
  end
end
```

[2] Ruby: `defined?(x)`, Python: `x in locals()`, JavaScript: `typeof x !== 'undefined'`

where $a_{C.m}(n) = \{C.m_1, C.m_2, ..., C.m_l\}$.

Then, we only have to treat the updating of a method $C.m$ as setting its global state to a new value v by `g.set_version_C_m(v)`.

Furthermore, when translating every method call to a method $C.m$ of arity n in the original program as a call to $C.m_n$ in the corresponding **dyn** program, the result is behaviourally equivalent[3].

12.4. The Connection to Higher-Order Programs

Another feature of functional languages that is often found in dynamically typed languages are *closures* (e.g. JavaScript functions or ruby blocks). Closures are characterized by two properties: Firstly, they allow passing around (a reference to) code as data and secondly, they capture the values of all free variables within their body upon creation.

Transformation: In **dyn**, we can emulate both properties by introducing a new class C_c for each closure c. This class defines a method $do()$ of the same arity as the closure and whose method body is just c's body with all free variables replaced by corresponding instance variables as well as a constructor *init* taking all variables as arguments that occur free in c's body and storing their value in corresponding instance variables. Now, the closure definition $c = \lambda p_1, ..., p_n.S$ can be replaced by

$$c = \textbf{new } C_c(v_1, ..., v_k),$$

where $v_1, ..., v_k$ are all variables occurring free in S and each call $c(e_1, ..., e_n)$ can be replaced by the method call

$$c.do(e_1, ..., e_n).$$

The resulting program is behaviorally equivalent[4]. Note that a finite program can only contain a finite number of closures and this replacement hence will only introduce a finite number of additional classes C_c.

12.4.1. Discussion

Note that above transformation for introducing closures into **dyn** is mere syntactic sugaring. Since closures are a feature typically found in functional programming languages and intimately linked to so-called Higher-Order Programming, this observation suggests that dynamically typed object-oriented languages like **dyn** are capable of emulating higher-order behavior. In fact, passing references to code is quite common in dynamically typed programs as the following example illustrates

```
class list {
  method sum(zero) {
```

[3] verifying this would of course require formalizing a version of **dyn** with method update

[4] verifying this would of course require formalizing a version of **dyn** with closures

```
    if self.empty? then
      zero
    else
      self.head() + self.tail().sum(zero)
    end
  }
}
```

the method `sum(zero)` uses the operator + to sum up all elements of a list. In functional languages one would say that the operator + is folded over the list. Just like in functional languages, + does not have a predetermined meaning, but can be thought of as an "argument". Of course it is not an actual argument by itself, but rather "attached" to the list elements, but the effect is the same: sum() is a higher-order method that is useful both to sum up lists of integers and to concatenate lists of strings.

Note that our program logic currently does not allow for higher-order reasoning. For closed programs where we know all the closures and all possible meanings of the operator +, we can reason about programs using closures or methods like $sum(zero)$ using a case distinction when calling the $do()$-method of a closure or the +-operator. However, adopting an open-world assumption will require higher-order reasoning. We will discuss this issue further in Chapter 13.

12.5. Reflection and Introspection

While in statically typed languages, *reflection* (similar to typecasts) provides a way to circumvent the type system and to work around the restrictions it imposes, in a dynamically typed language, calling any method on any receiver is the default modus operandi. Reflection hence boils down to nothing more than normal dynamically typed method calls.

Under the closed-world assumption, *introspection* also does not provide any additional benefits as any run-time test of the form "does the object x support a method m of arity n?" is equivalent to $x\ is\_a?\ I$ for some pre-computable set of class names $I \subseteq \mathfrak{C}$ and questions of the form "does the object x have an instance variable $@v$?" can already be emulated in **dyn** as discussed at the end of Section 12.2.

In summary, the only way how the combination of reflection and introspection can provide additional expressiveness for dynamically typed languages is in an open-world setting where it allows for calling methods whose names were not known at the time of verification (enabling for instance a test framework to call all methods whose names begin with "test" in a certain class). Unfortunately, as will be discussed in detail in Chapter 13, our current formal development is not yet applicable to open-world settings as the issue of Modularity needs to be addressed before such questions may be properly studied.

Part V.

Further Avenues of Research

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."

– *Alan Turing*

13. Modularity

It was stated already that the program logic as developed in Section 4.5 is only applicable to closed programs – that is, programs whose classes and methods are all known at the time of verification. It is hence not possible to prove a program in multiple parts – the canonical use case for this being libraries – where one usually wishes to verify each library only once and independent of any program using it and then use the properties guaranteed by this proof in the verification of any program using the library.

Being non-modular, our program logic unfortunately does not support this and hence requires programs to be verified together with their libraries. Of course, it would still be possible to provide method contracts, invariants and assertions in the source code of a library for use in these proofs, however, in order to achieve completeness, these would occasionally need to be modified to proof particular programs correct.

In the following sections, we will discuss two different kinds of modularity, one that is common to all program logics (Section 13.1) and one that is particular to dynamically typed programming languages (Section 13.2). In both cases, we will give examples as to why verifying programs in multiple parts induces incompleteness and suggest some strategies that might help in mitigating the problem.

13.1. Reasoning about Unknown Types

In this section, we will discuss a form of modularity that is independent of typing. The canonical example are libraries which exist in both statically and dynamically typed languages. The problem our program logic has in this respect is due to the way it handles method calls. In order to calculate the weakest precondition of a method call, it is necessary to know the set of all methods matching name and arity of the call in order to construct a big disjunction over all these possibilities (see the rule for method calls in Section 10.2). When using this methodology to verify a method contract $\{p\}v_0.m(v_1,...,v_n)\{q\}$, the set of known classes $\mathfrak{C}$ and their methods $\mathfrak{M}$ (the *environment* of the call) are implicitly assumed to be identical to the ones present in every call to the method. Should the method be called in a different environment (which is the case when a program uses the library), it is possible that the set of methods used in the proof is not appropriate any more, which can break the logic's completeness.

Consider the following example: Under the assumption that the class C_1 is the only one implementing a method $ex()$ of arity 0, and this implementation satisfies the specification

$$\{[\![v_0]\!] \in \{C_1\}\}v_0.ex()\{\textbf{terminates}\},$$

our program logic is able to derive the following contract:

```
method example(x)
  requires ⟦x⟧ ∈ {C1}
  ensures terminates
{
  x.ex()
}
```

Given this contract, it is possible to derive

$$\{true\}\textbf{self}.\text{example}(\textbf{new } C_1())\{\textbf{terminates}\}$$

However, in an environment with an additional class C_2 also implementing the method $ex()$ of arity 0, but instead satisfying the specification

$$\{[\![v_0]\!] \in \{C_2\}\}v_0.ex()\{\textbf{typesafe}\},$$

it is not possible to derive

$$\{true\}\textbf{self}.\text{example}(\textbf{new } C_2())\{\textbf{typesafe}\}$$

without modifying the proof of the method `example`. For this reason, proving this program in parts, where the method `example` is contained in one part and the call `self.example(new` C_2`())` in another, would induce incompleteness.

The classical way to mitigate this problem is to introduce some way of stating assumptions about classes and their methods that are unknown at the time of verification and when composing multiple such parts, checking that the actual implementations meet the stated assumptions. In statically typed, object-oriented languages, it is customary (see f.i. [50]) to use the inheritance hierarchy for this purpose as type systems for OO languages usually feature subtyping and hence allow only child classes to take their parent's place. Phrased differently, this means that whenever a variable / parameter is of a type C, then the type system ensures that all objects that the variable will ever reference to will be instances of a class C' such that C' is a subclass of C. Hence, in such a setting, ensuring that our correctness proof remains valid boils down to a discipline called behavioral subtyping.

Definition 31 (Behavioral Subtyping)**.** *Whenever a class C' is a subclass of another class C, then*

- *C' must support all methods of C in the same arity.*
- *for every method $m(p_1, ..., p_n)$ with method contract $\{p\}C.m(p_1, ..., p_n)\{q\}$ in C and method contract $\{p'\}C'.m(p_1, ..., p_n)\{q'\}$ in C', $p' \to p$ and $q \to q'$ must hold.*

While the first requirement is called *subclassing* and enforces that no type errors can be caused by substituting an instance of C' for an instance of C, the second is much stronger and enforces that the method implementations in the child class C'

must satisfy the same contracts that were given in the parent class C. To see this, observe that from every method contract in the child class $\{p'\}\cdot\{q'\}$ one can derive the respective one in the the parent class $\{p\}\cdot\{q\}$ by an application of the CONS rule. Behavioral subtyping thus enforces the Liskov Substitution Principle [54], stating that a child class can strengthen the contracts of its parent, but may not weaken them.

Hence, in such a setting, it would be sufficient to give a class C whose method contracts state the minimal requirements necessary for our code to be correct and verify our code in terms of this type C using its method contracts. Due to subtyping, we can then be sure that regardless of any unknown classes, whenever our method is called, the parameter passed will be of type C or a subtype thereof and will hence satisfy the requirements we based our verification on.

13.1.1. Interfaces

Unfortunately, in a dynamically typed language, such a construction would not be of much use since

- There is no type system enforcing subtyping.
- Inheritance is usually used as mere code reuse and implies neither subclassing nor behavioural subtyping – and hence the Liskov Substitution Principle does not in general hold for such languages.

However, the Java type system offers a more flexible mechanism called *Interfaces*, that can be repurposed to provide modularity also for dynamically typed languages. In Java, an interface is a class whose methods do not have method bodies. Just like a class, an interface constitutes a type. The inheritance relation for interfaces is called "implements" and is established nominally by adding the names of all implemented interfaces to a class's declaration. For a class C to implement an interface I, subclassing is required. However, contrary to inheritance between classes, a class may implement multiple interfaces.

To adapt this idea for **dyn**, we also introduce classes that do not have method bodies and call them interfaces. Contrary to Java, however, our interfaces contain method contracts in addition to the method declaration (name and arity of the method). Also, we do not extend the class declaration to list all implemented interfaces, but instead introduce an additional top-level keyword

$$\pi ::= \texttt{C implements I}$$

stating the fact that class C implements interface I. Like in Java, a class may implement any number of interfaces. However, in addition to subclassing, we require behavioral subtyping.

Then, we are able to use the very same line of modular reasoning like in statically typed languages, but without requiring inheritance along the way. Any program that interacts with values of a type that is not known at verification time may introduce an interface representing this type and capturing all assumptions about the behavior

of its method calls. After verifying the program in terms of these assumptions (the method contracts in the interface), one can compose it with an actual implementation for this type and only has to verify behavioral subtyping between the implementation and the interface.

Alternatively to the `implements` statements introduced above, it would be conceivable to infer the mapping `C implements I` automatically by checking the result of an (interactive) type inference for instances of a class C that were passed into methods verified in terms of an interface I.

13.2. Algebraic Properties

Since contrary to statically typed ones, dynamically typed programming languages do not restrict the types of method arguments by default, dynamically typed programs often exhibit additional reuse patterns to those known under static typing. For example, most methods in **dyn** could be called polymorphic independent of whether this was intended by their author or not. For example, as the operation $\leq$ is desugared to a method call, any sorting algorithm written in **dyn** for lists of numerics would automatically also be applicable to lists of any type that supports this operation. Note, however, that this would only be correct if the operation is indeed an order relation (transitivity is exploited by most sorting algorithms).

However, when verifying a sorting algorithm operating on lists of numerics in our program logic, we need to use the mapping predicate $\mathbb{N}$ in order to express that a list element e is smaller or equal than another one $e \leq e'$, which is necessary for instance to express sortedness of the list. Now, recall that the definition of $\mathbb{N}()$ entails that the argument of type object is an instance of class `numeric`, hence making our proof only applicable when the algorithm is in fact used to sort lists of numerics. Should a program use the algorithm polymorphically, we would be required to derive multiple proofs for it – one for each type of list elements, even though the underlying argument stays the same. For this reason it would not only be esthetically pleasing, but also of practical significance to make our logic more modular in respect of allowing for its proofs to be more type-independent.

One problem certainly encountered, however, will be that Hoare logic is not well-suited to express properties like transitivity of an order relation. While the statement

$$\forall a, b, c : \mathbb{N} \bullet a \leq b \land b \leq c \rightarrow a \leq c$$

is a first-order sentence and hence expressible in our assertion language, it denotes the transitivity of the order relation $\leq$ of the natural numbers. When trying to apply it to objects of some type X, i.e.

$$\forall a, b, c : \mathbb{O} \bullet [\![a]\!] \in \{X\} \land [\![b]\!] \in \{X\} \land [\![c]\!] \in \{X\} \rightarrow (a \leq b \land b \leq c \rightarrow a \leq c)$$

the sentence is not well-typed any more since the type system of **AL** does not allow applying $\leq$ to operands of type $\mathbb{O}$. Also, we don't want $\leq$ to have some fixed interpretation, but to denote a call to the method $op_{\leq}$. Method calls, however only exist

in **dyn** programs and not in **AL** assertions. The only appropriate place for them is hence in the middle of Hoare Triple. Unfortunately, properties like Transitivity[1] are not properties of a single method call, but relate multiple (in this case 3) different calls to the same method (each having different arguments) to each other.

While due to time constraints this problem will have to remain open in this thesis, there are several ideas for approaching it that we document in this section as they might be useful for future researchers in the area.

13.2.1. Algebraic Specifications

The problem outlined above has been noted before and – like often in academia – there are other approaches for specifying program behavior that are better suited for this kind of properties. Transitivity is just one example of a class of properties known as *algebraic* properties, meaning they can be expressed as boolean combinations of equations over terms using the functions to be specified. "Functions" here is intentionally plural, as so-called algebraic specifications often specify one function is terms of many others – a property that makes them ideally suited for data structures, which is why they are foremost used for the specification of so-called "algebraic data-types". However, as the terminus "function" already indicates, algebraic specifications were developed for functional programming languages and are hence not ideally suited for dealing with side-effects. There is, however, work on applying them to imperative, object-oriented languages [74].

Integrating Algebraic Specifications into our Hoare logic hence poses multiple challenges: Assuming that we can specify an Interface as an Algebraic Data-type, then

1. How do we verify a method against this Interface?
2. How do we verify that a class conforms to the Interface?
3. How are we dealing with side-effects?

The first challenge is the simplest one: First we introduce a new data-type in our SMT-Solver for this Interface. For each method on this data-type, we declare a corresponding uninterpreted function. Then we can translate the algebraic specification of the data-type directly into assumptions about these uninterpreted functions. Furthermore, we add a mapping-predicate $\mathbb{M}$ for our Interface mapping it's instances to values of the new data-type. We can also let this mapping uninterpreted (the only guarantee being that the same object is always mapped to the same value). Now every method is given a specification of the following form

$$\{\mathbb{M}_0(v_0, v_0') \wedge ...\mathbb{M}_n(v_n, v_n') \wedge \ \wedge r' = f_m(v_0', ..., v_n')\}v_0.m(v_1, ..., v_n)\{\mathbb{M}_{n+1}(\mathbf{r}, r')\}$$

where $\mathbb{M}_i$ are appropriate mapping predicates (for the argument and return types) and f_m is the uninterpreted function corresponding to the method m. Now when a method

[1] This is the case for most algebraic properties (for instance Symmetry, Commutativity and Associativity also relate multiple calls to the same method with each other).

to be verified calls methods on instances of the Interface type, then these method specifications will ensure that the verification conditions of the method will contain the corresponding uninterpreted functions. If the theorem prover is able to validate the conditions using only the assumptions derived from the algebraic specification, then the method is correct for all data types satisfying this specification – we hence have a notion of correctness that is independent of concrete data types. Note that the case study in Section 11.3 was verified in a similar way.

So the next question (corresponding to the second challenge) is: When does a concrete class satisfy an algebraic specification (and hence "implements" the Interface)? Since algebraic specifications are sets of First-Order sentences containing equations, the question boils down to "How can we verify an equational specification using Hoare Logic?".

This is best explained using an example: Let us suppose our Algebraic Specification requires commutativity:

$$\forall a, b : \mathbb{O} \bullet [\![a]\!] \in \{X\} \wedge [\![b]\!] \in \{X\} \rightarrow a + b = b + a \qquad \text{(Commutativity)}$$

Since in **dyn**, the operation + is desugared to a method call, we have

$$\forall a, b : \mathbb{O} \bullet [\![a]\!] \in \{X\} \wedge [\![b]\!] \in \{X\} \rightarrow a.op_+(b) = b.op_+(a) \qquad \text{(Commutativity)}$$

13.2.2. Fusing Hoare Triples

First, we need to decide on an interpretation for the equality-sign (=). For simplicity, let us suppose that the method op_+ returns objects encoding numeric values. Then the equation above can be expressed using two Hoare Triples

$$\{p_1\}a.op_+(b)\{\mathbb{N}(\mathbf{r}, x)\} \qquad \text{and} \qquad \{p_2\}b.op_+(a)\{\mathbb{N}(\mathbf{r}, x)\}$$

where x is a logical variable of type $\mathbb{N}$ supposed to denote the same numeric value in both postconditions and p_1 and p_2 are appropriate preconditions. While it is not possible in standard Hoare Logic to link the value of x between 2 Hoare Triples, the Weakest Precondition Calculus gives us a way to turn Hoare Triples into First-Order statements:

$$WP(a.op_+(b), \mathbb{N}(\mathbf{r}, x)) \wedge WP(b.ap_+(a), \mathbb{N}(\mathbf{r}, x))$$

Note that the two weakest preconditions were conjoined. This way, they become part of the same statement and the occurrences of the variable x on both sides refer to the same variable – and hence the same value. Hence the resulting statement denotes the set of states from which both calls to op_+ yield the same numeric value. In the case of addition of natural numbers the WP would usually yield something like this:

$$\mathbb{N}(a, a') \wedge \mathbb{N}(b, b') \wedge a' + b' = x \wedge \mathbb{N}(b, b'') \wedge \mathbb{N}(a, a'') \wedge b'' + a'' = x$$

which can be simplified to

$$\mathbb{N}(a, a') \wedge \mathbb{N}(b, b') \wedge a' + b' = b' + a'$$

and hence obviously holds for all a and b encoding natural numbers since the addition of natural numbers is commutative. When substituting it for the equation $a.op_+(b) = b.op_+(a)$ in above First-Order statement we get

$$\forall a, b : \mathbb{O} \bullet [\![a]\!] \in \{X\} \wedge [\![b]\!] \in \{X\} \rightarrow \mathbb{N}(a, a') \wedge \mathbb{N}(b, b') \wedge a' + b' = b' + a'$$

which is a valid first-order sentence. In general, interpreting equations as two Hoare Triples with matching postconditions, using the Weakest Precondition Calculus to fuse them like this and substituting the result for the respective equation yields a statement that is valid iff the equations holds in every relevant start state (the set of start states to consider may be restricted using an implication like in the example).

13.2.3. Quantification over computable functions

A second technique that may be used to encode algebraic properties into Hoare Triples is quantification over computable functions. Note that quantification over functions is only possible in second-order logic as the set of natural numbers $\mathbb{N}$ is countably infinite and the set of functions from $\mathbb{N}$ to $\mathbb{N}$ is hence uncountably infinite. However, the set of computable functions is smaller than the set of (arbitrary) functions. A well-known fact from recursion theory is that there is a μ-recursive function (called the "universal function" and denoted by u) such that for every μ-recursive function f,

$$\exists n : \mathbb{N} \bullet \forall x : \mathbb{N} \bullet u(n, x) = f(x)$$

holds. There, n is the Gödel-Number for f and the mapping from μ-recursive functions to their Gödel-Numbers in effect gives a bijection between natural numbers and computable functions, which are hence countable. Also, this can be extended to arbitrary arities using Gödelization (see Section 1.5.5). Since u is a μ-recursive function and we know from Theorem 2 that we can express all μ-recursive functions in **AL**, we can assume u to be part of **AL** without loss of generality. Furthermore, we introduce the following abbreviation for quantifying over computable functions using u:

$$Qf : \mathbb{F} \bullet a \equiv Qn_f : \mathbb{N} \bullet a[u(n_f, x)/f(x)]$$

for $Q \in \{\forall, \exists\}$ where $\mathbb{F}$ should denote the set of computable functions. This way it is possible to quantify over computable functions in Weak Second-Order Logic with Arithmetic and hence in **AL**.

We can then equate μ-recursive functions and methods:

$$EQ \equiv \exists f_\leq : \mathbb{F} \bullet \{\mathbb{M}_0(v_0, v_0') \wedge \mathbb{M}_1(v_1, v_1') \wedge r' = f_\leq(v_0', v_1')\} v_0.op_\leq(v_1)\{\mathbb{M}_r(\mathbf{r}, r')\}$$

where $\mathbb{M}_i$ are appropriate mapping predicates for argument types and result type of $op_\leq$ and the Hoare Triple should again be understood as the Weak Second-Order sentence WP maps it into. Now expressing transitivity is trivial:

$$EQ \wedge \forall a, b, c : \mathbb{N} \bullet f_\leq(a, b) \wedge f_\leq(b, c) \rightarrow f_\leq(a, c)$$

While in theory this is an extremely flexible technique and allows both to specify methods in terms of other methods and verifying methods against algebraic specifications, in practice one can expect it to place a heavy burden on the theorem prover as the ability to simulate μ-recursive function in logic is the main reason Weak Second-Order Logic with Arithmetic is undecidable.

13.3. Higher-Order Behaviour

As outlined in Section 12.4, dynamically typed object-oriented programs are able to emulate higher-order behavior. It might hence be necessary to extend our program logic for higher-order reasoning in order to provide modularity also for these cases (see, for instance, [73]). However, in all cases we considered so far, algebraic specifications turned out to be sufficient. Consider again the example from Section 12.4:

```
class list {
  method sum(zero) {
    if self.empty? then
      zero
    else
      self.head() + self.tail().sum(zero)
    end
  }
}
```

An algebraic specification for the method `sum` would be

$\forall$ `x,a` $:\mathbb{O}$, `l` $:\mathbb{L}$
- `[].sum(x) = x`
- `cons(a,l).sum(x) = a + l.sum(x)`

Note that algebraic specifications can a) be recursive and b) allow specifying a method in terms of other methods (which includes those implicitly passed along with its arguments – like the method + on the list elements), thus allowing higher-order specifications like the above. Note also that this specification closely follows the structure of the recursive implementation and is hence trivial to verify. Furthermore, given this specification and using the techniques discussed in the last section, establishing both

$$\{true\}\ sum([1,2,3])\ \{\mathbb{N}(\mathbf{r},6)\},\ \text{and}$$
$$\{true\}\ sum([\text{"}foo\text{"},\text{"}bar\text{"}])\ \{\mathbb{S}(\mathbf{r},\text{"}foobar\text{"})\}$$

is straightforward, hence realizing both type-independence and verification of higher-order behavior.

14. Conclusion

"The concept of progress acts as a protective mechanism
to shield us from the terrors of the future."
– *From "Collected sayings of Muad'Dib" by the Princess Irulan - Dune*

In this thesis we studied the problem of verifying dynamically typed programs. Apart from the fact that these programming languages do not use a static type checker to ensure type safety and hence require the verification to solve it, we could not find any reason why this interesting type of programming language has been largely ignored by the verification community during the last several decades. While the traditional Hoare logic is not applicable to these programs, the reason for this lies solely in implicit assumptions used to enable the optimizations discussed in Chapter 4. As could be shown in Section 4.5, these assumptions can be readily removed at the price of losing the respective optimizations and leading to a generalization of traditional (statically typed) Hoare logic that we call dynamically typed Hoare logic, a proof system for functional program correctness that is applicable to both statically and dynamically typed programs. As Chapter 5 demonstrates, our dynamically typed Hoare logic offers the same proof-theoretic strength as traditional (statically typed) Hoare logic although proving its soundness and (relative) completeness requires some adaptations.

In Part III of this thesis, we studied the differences between statically and dynamically typed Hoare logic from a more practical perspective to discover that the optimizations were obviously added to the former as a means to offer a certain amount of convenience as well as to allow for optimizations on the side of the automated theorem prover checking the verification conditions. As both aspects are important for practical applications of program verification, we developed an approach for mitigating the drawback dynamically typed languages incur in this respect by incorporating a type inference algorithm into our program logic and using the derived type information to also provide optimized proof rules for dynamically typed programs. As could be shown in Chapters 7-8, not only does the combination of a type inference with our program logic provide a way to significantly reduce the effort of proving type safety, but also does the derived type information enable verifying dynamically typed programs with all the convenience known from traditional Hoare logic (see Chapter 7).

The techniques developed were demonstrated on a number of case studies (see Chapter 11).

We feel that our goal of closing the gap between statically typed and dynamically typed programming languages with respect to verification has been fully met at least from a theoretical perspective since sound and (relative) complete Hoare logics can now be constructed for such languages by using the transformational approach (see

below). Whether or not this goal has also been met from a practical perspective is currently hard to say, since due to time constraints, the preliminary implementation of our verification tool based on the concepts mentioned does not yet allow for a conclusive evaluation. However, given

1. (relative) completeness of interactive type inference,

2. the ease in which type safety of the case study coerce protocol could be established, and

3. the fact that the optimized proof rules in the Layer of Abstraction are basically identical to their statically typed counterparts,

we would be very surprised to find a dynamically typed program causing verification problems that do not arise when verifying a statically typed equivalent.

Unfortunately, the fact that they were largely ignored for decades has created further impediments for applying verification techniques to dynamically typed programming languages: First and foremost the lack of program logics for real-world examples of such languages. To this end, the Transformational Approach was introduced by Apt, De Boer, Olderog and De Gouw [6] as a way to ease the creation of sound and complete program logics, which usually is quite an involved endeavor[1]. The basic idea is to reuse the work invested in proving soundness and completeness of a similar program logic by adapting these results by means of inter-language transformations. This works very well between program logics for similar programming languages and in some cases even between quite different languages. For instance, [6] describes such a transformation from an object-oriented language to mere recursive procedures with arrays.

Unfortunately, prior to this thesis, the transformational approach was not applicable to dynamically typed programming languages, as

1. no sound and complete program logic had been published for any dynamically typed language, and

2. it is not possible to automatically transform dynamically typed programs (i.e. **dyn**) into statically typed programs (i.e. **stat**) due to the additional requirement of statically verifiable type-safety in the latter case.

Fortunately, the Tagged Hoare logic presented in Section 4.5 improves upon this situation and makes the transformational approach henceforth also applicable to dynamically typed languages. Hopefully, this will contribute to the availability of program logics also for real-world dynamically typed languages.

[1] In the case of **dyn**, it took several month to study all the relevant literature and devise the proof in Section 5.2.

14.1. Future Work

Several avenues for further research have already been described in Part V of this thesis. However, there are also several immediate issues not related to Modularity and Algebraic Datatypes:

- *Implementation*: Providing a full-fledged implementation for the verification of **dyn** programs would allow for using **dyn** as an intermediate verification language for dynamically typed languages that do not yet have their own sound and complete axiomatic semantics.
- *Evaluation*: Such an implementation could also be used to verify a larger number of programs and properties and hence provice insights into the extent to which the practical issues arising in the verification of dynamically typed programs have already been adequately addressed by introducing Interactive Type Inference and the Layer of Abstraction.
- *Improvements*: Several small improvements mentioned in this thesis like the Detection of Ill-Defined Method Specifications or the alternative strategy for handling method calls in the WPC could then also be evaluated for assessing their merrit.
- *Generalization*: The basic idea of Interactive Type Inference can quite likely be generalized to arbitrary program analyses. Such an integration of program analysis and program verification might be useful to provide a smooth transition from the former push-button technique to the latter (relative) complete formal method.
- *Performance*: It would be interesting to study to what extent the type information provided by consensual typing can be used to fuel performance optimizations in modern just-in-time-compilers/interpreters and whether it is sufficient to allow full compilation.
- *Alternative*: The case studies (Chapter 11) revealed several incompleteness issues in current state-of-the-art automated theorem proving technology (see Appendix D) that were astounding both in amount and extent. It might hence be worthwhile to invest some time to explore the alternative of interactive theorem proving further, be it only to reach a fair comparison of the two paradigms.
- *Extension*: It might be interesting to allow for the tags of Tagged Hoare Logic to contain additional information. In this way it might be possible to incorporate other Hoare logic extensions like footprints [13] as tags.
- *Comparison*: As mentioned in Chapter 9, the relationship between tagged Hoare Logic and the approach from Huisman and Jacobs [43] is currently unclear. An investigation in this direction might be worthwhile as it might not only provide insights into the compositionality of correctness notions, but might also lead to a consolidation of the two approaches.

Bibliography

[1] Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic Typing in a Statically typed Language. In: Proc. POPL. pp. 213–227. ACM, New York, NY, USA (1989),

[2] Abano, C., Chu, G., Eiseman, G., Fu, J., Yu, T.: Formal Verification with Dafny (2014), Project report of the Governor's School of New Jersey,

[3] ACM Special Interest Group on Programming Languages: Programming Languages Software Award, http://www.sigplan.org/Awards/Software/

[4] Apt, K.R.: Ten Years of Hoare's Logic: A Survey – Part I. ACM Trans. Program. Lang. Syst. 3(4), 431–483 (Oct 1981),

[5] Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of Sequential and Concurrent Programs, 3rd Edition. Texts in Computer Science, Springer-Verlag (2009), 502 pp.

[6] Apt, K.R., de Boer, F.S., Olderog, E.R., de Gouw, S.: Verification of Object-Oriented Programs: A Transformational Approach. J. Comp. Sys. Sci. 78(3), 823–852 (2012),

[7] Apt, K.R., Francez, N., de Roever, W.P.: A proof system for communicating sequential processes. ACM Trans. Program. Lang. Syst. 2(3), 359–385 (1980),

[8] Barnett, M., Lahiri, S., Lal, A., Leino, R., McMillan, K., Moskal, M., Qadeer, S., Schulte, W.: Boogie intermediate verification language, http://research.microsoft.com/en-us/projects/boogie/

[9] Bierman, G.M., Abadi, M., Torgersen, M.: Understanding typescript. In: ECOOP. pp. 257–281 (2014),

[10] Bjørner, N.: The Z3 Theorem Prover, https://github.com/Z3Prover/z3

[11] Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT Solvers. Journal of Automated Reasoning 51(1), 109–128 (2013),

[12] Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proc. pp. 131–146 (2010),

[13] de Boer, F.S., de Gouw, S.: Being and Change: Reasoning About Invariance. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design. LNCS, vol. 9360, pp. 191–204. Springer (2015)

[14] de Boer, F.S., Pierik, C.: How to Cook a Complete Hoare Logic for Your Pet OO Language. In: Formal Methods for Components and Objects. LNCS, vol. 3188, pp. 111–133. Springer (2003),

[15] Boyer, R.S., Moore, J.S.: Proving Theorems about LISP Functions. In: Nilsson, N.J. (ed.) IJCAI. pp. 486–493. William Kaufmann (1973),

[16] Bracha, G.: Pluggable Type Systems. In: OOPSLA Workshop on Revival of Dynamic Languages (2004),

[17] Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI, Proc. LNCS, vol. 3855, pp. 427–442. Springer, Charleston, SC, USA (January 2006),

[18] Cartwright, R., Fagan, M.: Soft Typing. In: Wise, D.S. (ed.) PLDI. pp. 278–292. ACM (1991),

[19] Chugh, R.: Nested Refinement Types for JavaScript. Ph.D. thesis, University of California, San Diego (September 2013),

[20] Chugh, R., Herman, D., Jhala, R.: Dependent Types for JavaScript. In: Proc. OOPSLA 2012. pp. 587–606. ACM, New York, NY, USA (2012),

[21] Chugh, R., Rondon, P.M., Jhala, R.: Nested Refinements: A Logic for Duck Typing. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 231–244. POPL '12, ACM, New York, NY, USA (2012),

[22] Cook, S.A.: Soundness and Completeness of an Axiom System for Program Verification. SIAM Journal of Computing 7(1), 70–90 (1978),

[23] Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL. pp. 238–252. ACM (1977),

[24] Darvas, Á., Leino, K.R.M.: Practical Reasoning About Invocations and Implementations of Pure Methods. In: Dwyer, M.B., Lopes, A. (eds.) FASE. LNCS, vol. 4422, pp. 336–351. Springer (2007),

[25] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. Commun. ACM 5(7), 394–397 (Jul 1962),

[26] Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM 18(8), 453–457 (Aug 1975),

[27] Engelmann, B., Olderog, E.: A Sound and Complete Hoare Logic for Dynamically-Typed, Object-Oriented Programs. In: Ábrahám, E., Bonsangue, M.M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 173–193. Springer (2016),

[28] Engelmann, B., Olderog, E.R., Flick, N.E.: Closing the Gap – Formally Verifying Dynamically Typed Programs like Statically Typed Ones Using Hoare Logic – Extended Version. CoRR abs/1501.02699 (January 2015),

[29] Filaretti, D., Maffeis, S.: An executable formal semantics of PHP. In: Proc. ECOOP 2014. LNCS, vol. 8586, pp. 567–592. Springer (2014),

[30] Furr, M., An, J.h.D., Foster, J.S., Hicks, M.: Static Type Inference for Ruby. In: Proc. of the 2009 ACM Symposium on Applied Computing. pp. 1859–1866. SAC '09, ACM, New York, NY, USA (2009),

[31] Gardner, P., Maffeis, S., Smith, G.D.: Towards a Program Logic for JavaScript. In: Field, J., Hicks, M. (eds.) POPL. pp. 31–44. ACM (2012),

[32] Gödel, K.: On formally undecidable propositions of Principia Mathematica and related systems I. Monatshefte für Mathematik (Nov 1931), translated into english by Martin Hirzel,

[33] Gorelick, G.A.: A Complete Axiomatic System for Proving Assertions about Recursive and Non-Recursive Programs. Tech. Rep. 75, Department of Computer Science, University of Toronto, Canada (1975)

[34] Guha, A., Saftoiu, C., Krishnamurthi, S.: The Essence of Javascript. In: Proc. of the 24th European Conference on Object-oriented Programming. pp. 126–150. ECOOP'10, Springer-Verlag, Berlin, Heidelberg (2010),

[35] Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing Local Control and State Using Flow Analysis. In: Proc. of the 20th European Conference on Programming Languages and Systems. pp. 256–275. ESOP'11/ETAPS'11, Springer (2011),

[36] Guttag, J.V., Horowitz, E., Musser, D.R.: Some Extensions to Algebraic Specifications. SIGSOFT Softw. Eng. Notes 2(2), 63–67 (Mar 1977)

[37] Hamid, N., Shao, Z.: Interfacing Hoare Logic and Type Systems for Foundational Proof-Carrying Code. In: TPHOL 2004, LNCS, vol. 3223, pp. 118–135. Springer (2004),

[38] Harel, D.: First-Order Dynamic Logic. Springer, Secaucus, NJ, USA (1979)

[39] Harper, R.: Dynamic Languages are Static Languages,

[40] Hennessy, M., Plotkin, G.D.: Full Abstraction for a Simple Parallel Programming Language. In: Mathematical Foundations of Computer Science, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7. pp. 108–120 (1979),

[41] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. CACM 12, 576–580, 583 (1969),

[42] Hoare, C.A.R.: Procedures and Parameters: An Axiomatic Approach. In: Engeler, E. (ed.) Symposium on Semantics of Algorithmic Lang., LNM, vol. 188, pp. 102–116. Springer (1971),

[43] Huisman, M., Jacobs, B.: Java Program Verification via a Hoare Logic with Abrupt Termination. In: Maibaum, T. (ed.) Proc. FASE. pp. 284–303. Springer, Berlin, Heidelberg (2000),

[44] Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Proc. 16th International Static Analysis Symposium (SAS). LNCS, vol. 5673. Springer (Aug 2009),

[45] Khoo, Y.P., Chang, B.E., Foster, J.S.: Mixing Type Checking and Symbolic Execution. In: Proc. of PLDI 2010. pp. 436–447 (2010),

[46] Kluge, R.: Automatischer Beweis von Verifikationsbedingungen. Master's thesis, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany (April 2013)

[47] Kregel, D.: Entwicklung einer Verifikationsumgebung für eine dynamisch getypte Programmiersprache. Master's thesis, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany (December 2015)

[48] Lauer, P.: Consistent Formal Theories of the Semantics of Programming Languages (1971), Technical Report 25. 121, IBM Laboratory Vienna

[49] Lazar, D., Arusoaie, A., Serbanuta, T.F., Ellison, C., Mereuta, R., Lucanu, D., Roşu, G.: Executing Formal Semantics with the K Tool. In: Proc. of the 18th International Symposium on Formal Methods (FM'12). LNCS, vol. 7436, pp. 267–271. Springer (August 2012),

[50] Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures, pp. 262–284. Springer (2003),

[51] Leino, K.R.M., Moskal, M.: Usable Auto-Active Verification. In: Usable Verification Workshop. Microsoft Research, Redmond, Washington, USA (2010),

[52] Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: TeJaS: Retrofitting Type Systems for JavaScript. In: Proc. of the 9th Symposium on Dynamic Languages. pp. 1–16. DLS '13, ACM, New York, NY, USA (2013),

[53] Liskov, B., Zilles, S.: Specification Techniques for Data Abstractions. SIGPLAN Not. 10(6), 72–87 (Apr 1975),

[54] Liskov, B.H., Wing, J.M.: A Behavioral Notion of Subtyping. ACM Trans. Program. Lang. Syst. 16(6), 1811–1841 (Nov 1994),

[55] Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: Proc. of the 6th Asian Symposium on Programming Languages and Systems. pp. 307–325. APLAS '08, Springer-Verlag, Berlin, Heidelberg (2008),

[56] Matsumoto, Y.: Ruby, a programmer's best friend, https://www.ruby-lang.org

[57] Microsoft: C#, https://msdn.microsoft.com/de-de/library/kx37x362.aspx

[58] de Moura, L., Bjørner, N.: Z3 – a tutorial,

[59] Naudziuniene, D., Gardner, P., Smith, G.: JuS: Squeezing the Sense out of JavaScript Programs. JSTools (2013),

[60] Nguyên, P.C., Tobin-Hochstadt, S., Van Horn, D.: Soft Contract Verification. In: Proc. ICFP 2014. pp. 139–152. ACM, New York, NY, USA (2013),

[61] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Secaucus, NJ, USA (1999)

[62] Nipkow, T.: Programming and Proving in Isabelle/HOL, https://isabelle.in.tum.de/,

[63] Olderog, E.R.: On the Notion of Expressiveness and the Rule of Adaptation. Theoretical Computer Science 24(3), 337–347 (1983),

[64] Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica 6(4), 319–340 (1976),

[65] Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: Paepcke, A. (ed.) OOPSLA. pp. 146–161. ACM (1991),

[66] Plotkin, G.D.: A Structural Approach to Operational Semantics. J. of Logic and Algebraic Programming 60–61, 17–139 (2004),

[67] Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: The Full Monty: A Tested Semantics for the Python Programming Language. In: OOPSLA 2013 (2013),

[68] Qin, S., Chawdhary, A., Xiong, W., Munro, M., Qiu, Z., Zhu, H.: Towards an Axiomatic Verification System for JavaScript. In: Proc. TASE. pp. 133–141. TASE '11, IEEE, Washington, DC, USA (2011)

[69] Rastogi, A., Chaudhuri, A., Hosmer, B.: The Ins and Outs of Gradual Type Inference. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 481–494. POPL '12, ACM, New York, NY, USA (2012),

[70] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02, IEEE Computer Society, Washington, DC, USA (2002),

[71] Rice, H.G.: Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematical Society 74(2), 358–366 (1953),

[72] Schmidt, D.A., Steffen, B.: Program Analysis *as* Model Checking of Abstract Interpretations. In: Levi, G. (ed.) SAS. LNCS, vol. 1503, pp. 351–380. Springer (1998),

[73] Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested Hoare Triples and Frame Rules for Higher-order Store. Logical Methods in Computer Science 7(3) (2011),

[74] Shu, Q., Wang, S., Liu, Y.: Verifying OO Programs by Linking Algebraic and Abstract Specifications. In: Margaria, T., Qiu, Z., Yang, H. (eds.) Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE. pp. 219–222. IEEE Computer Society (2012)

[75] Siek, J., Taha, W.: Gradual Typing for Objects. In: Ernst, E. (ed.) ECOOP 2007 - Object-Oriented Programming, LNCS, vol. 4609, pp. 2–27. Springer Berlin Heidelberg (2007),

[76] Smeding, G.J.: An executable operational semantics for Python. Master's thesis, Utrecht University, The Netherlands (2009),

[77] Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying Higher-Order Programs with the Dijkstra Monad. In: Proc. PLDI 2013 (June 2013)

[78] Tarski, A.: A Lattice-theoretical Fixpoint Theorem and its Applications. Pacific Journal of Mathematics 5(2), 285–309 (1955),

[79] Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: Hudak, P., Weirich, S. (eds.) Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 117–128. ACM (2010),

[80] Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS, LNCS, vol. 9035, pp. 566–580. Springer (2015),

[81] Uribe, T.E.: Combinations of Model Checking and Theorem Proving. In: Kirchner, H., Ringeissen, C. (eds.) FroCoS. LNCS, vol. 1794, pp. 151–170. Springer (2000),

Symbol Index

α_1, **100**
α_{TI}, 100
$\wedge$, 24, 25, 28, 30, **30**, 33, 34, 39, 41, 44, 50, 51, 56–62, 64, 66–70, 72, 74–79, 83, 84, 87–93, 99–101, 103–106, 108, 112, 121–130, 133–138, 140–143, 154–157, 177–179, 181, 183, 184
$\cdot ::= \cdot$, 39, **39**, 42, 45, 47, 63, 121, 126
$\cdot ::= \cdot$, 13

$\leftrightarrow$, 30, **30**, 41, 44, 56, 68, 104, 108, 184
begin local $\cdot\,;\cdot$ **end**, 42, 121, 122
$\mathbb{B}$, 20, 39–41, 44, 46, 47, 51, 55, **55**, 56, 67, 68, 88, 91, 121, 183, 184
$bool(o)$, **41**
$bool(o, b)$, **41**
$\bot$, 24, **55**

$\mathfrak{C}$, 15, **15**, 18–20, 24, 25, 38–40, 46, 47, 55, 57, 77, 88, 90, 92, 93, 99, 100, 143, 144, 147, 151
C_{null}, 18, **19**, 24, 40, 51, 56, 88
$Conf$, 17, **17**, 58
  $Conf_{error}$, **17**
  $Conf_{final}$, **17**
  $Conf_{proper}$, **17**
new C$(\cdots)$, **39**, 44, 47, 90, 122, 152

$\mathcal{D}$, 14, **14**, 15, 18, 29–31

$\cdot = \cdot$, 57, **57**
$\equiv$, 15, 17, 28, 30, **30**, 32–34, 39, 41, 42, 44, 49, 56, 57, 66, 68, 70, 72–79, 89, 91, 103, 104, 108, 121–124, 128, 137–139, 141–143, 157, 177–179, 181, 183, 184
$\exists$, 13, 28, 30, **30**, 33, 34, 44, 57, 62, 67–70, 72, 73, 76, 83, 84, 89, 121–125, 128, 134, 137, 157, 177–179, 181, 183

fail, 17, 42, 44, 58, 139–142, 169
fail$\langle \cdot\,, \cdot \rangle$, **17**
final, 17, 18, 24, 40–42, 44, 45, 50, 58, 69, 70, 139, 142, 169
final$\langle \cdot \rangle$, **17**
$\forall$, 13, 24, 25, 28, 30, **30**, 33, 34, 50, 57, 58, 67, 70, 72, 74, 76, 77, 79, 91, 99, 100, 102, 121, 123–125, 127, 128, 134, 137, 154, 156–158, 177–179, 181
$\mathfrak{F}$, **49**, 50, 51
$free(\cdot)$, **30**
$\mathbb{F}$, 157, **157**

γ_1, **100**

$\mathcal{H}_d$, 66, **66**, 69, 70, 75, 77–80, 84, 91, 92
$\mathcal{H}_s$, 59, **59**, 66, 68, 76, 77, 84, 91, 92

if $\cdot$ **then** $\cdot$ **else** $\cdot$ **end**, 14, **39**, 42, 45, 48, 103, 104, 121, 179
if $\cdot$ **then** $\cdot$ **else** $\cdot$ **fi**, **57**, 90, 179
if $\cdot$ **then** $\cdot$ **end**, 14
$\rightarrow$, 13–15, 17, 19–21, 24, 25, 27–30, **30**, 32–34, 39–42, 44–46, 48–50, 55–58, 60–62, 64, 67, 70, 72–79, 88, 90–93, 98–108, 119–122, 125, 127, 128, 136–139, 142–144, 152, 154, 156, 157, 181, 183, 184
$\mathfrak{I}$, **30**

$\sqcup$, 24

$\mathfrak{L}_{\mathcal{T}}$, **24**
$\cdot[\cdot]$, 57
$\mathbb{L}$, 92, **92**, 123, 124, 137, 158
u, 15, 19, 24, 25, 38, 39, **39**, 40, 42, 45–47, 50, 63, 100, 101, 120, 126–128, 131–133, 142, 144, 170

$\mathbb{M}$, 92, **92**, 137
$\sqcap$, 24
$\mathfrak{M}$, **15**
$\mathfrak{M}_C$, **15**
$\mathfrak{M}$, 15, 38, 39, 42, 44, 45, 47, 61, 62, 64, 67, 90, 92, 151, 170, 184
$\cdot$.m$(\cdots)$, 14, 15, 20, **39**, 44, 45, 47, 90, 93, 121, 122, 128, 144, 152
$\models$, **30**

$\neg$, 30, **30**, 31, 39, 42, 44, 57, 60, 78, 88, 91, 99, 101, 112, 121, 125, 128, 140–142, 177–179, 181, 184
null, 14, 37–39, 42, 46, 51, 57, 90, 120
null, 14, 19, 38, 41–45, 55, 56, 58, 60, 61, 66, 67, 69, 70, 72, 75, 76, 90, 91, 121, 136, 137, 143
$\mathbb{N}$, 33, 34, 39, 40, 46, 49, **55**, 56, 61, 62, 67, 70, 72–74, 76, 77, 79, 83, 84, 88, 90–92, 121–125, 127, 136, 137, 154, 156, 157

$\mathbb{O}$, 15, 19, 20, 25, 39–41, 46, 48, 50, 51, 55, **55**, 56, 67, 69, 70, 72, 73, 76, 84, 88, 89, 92, 121, 136, 137, 142, 154, 156–158, 183
$\vee$, 30, **30**, 39, 44, 55, 56, 62, 68, 88, 99–101, 103, 104, 121–124, 126, 130, 137, 142, 177–179, 181

pr, 101, **101**, 102, 103, 105, 108, 114
pr_u, 101, **101**

R, **24**
$\mathcal{R}_\tau$, 103, **103**, 104, 105

$\mathcal{S}$, 58, 59, 140
self, 14, 19, 24, 25, 37–40, 42, 43, 45–47, 50, 55, **55**, 57, 61, 66, 67, 89, 99–101, 120–122, 143, 144, 152, 177, 178, 183
$\mathcal{M}$, 17, 18, 26, 58, 59, 74, 78
$[\![\cdot]\!]$, **26**
*, **55**
$\mathring{\Sigma}$, **19**, 24, 46, 49, 50, 100, 102
$\mathring{\sigma}$, 24, **24**, 25, 46–51, 99, 100
$\mathfrak{S}$, **29**, 30
$\sigma[\mathrm{u} := v]$, **15**
$\sigma[o.@\mathrm{x} := v]$, **15**
Σ, 15
Stmt, 17, 51
$\mathbb{S}$, **92**, 137

$\mathcal{T}$, 70, **70**, 75, 76, 78, 79, 83
TAsrt, 99, **99**, 100–103, 105
TLit, **99**, 101
$\top$, 24, **55**
$\mathbb{T}$, 55, **55**, 57, 72, 88, **92**, 183, 184
$[\![\cdot]\!] \in \{\cdots\}$, **57**, 154, 156, 157
typeerror, 17, 42–44, 58, 108, 139–142, 145, 170
typeerror$\langle\cdot,\cdot\rangle$, **17**
$ty^\dagger_\pi$, **19**, 20, **45**, 51, 102, 107–109, 144
$\mathcal{I}$, 99, **99**, 100, 102
$\mathcal{I}(ty)$, 99

$\models$, **26**
$\mathfrak{V}$, **15**, 30, 38, 41, 42, 45, 48, 50, 62, 66, 91, 99, 142, 143
$\mathfrak{V}_I$, 15, **15**, 19, 24, 25, 39, 46, 50, 72, 100, 101
$\mathfrak{V}_L$, 14, **14**, 15, **15**, 19, 24, 25, 39, 42, 45, 46, 49, 60, 72, 100, 101, 105, 128, 143
$\mathfrak{V}_S$, 101

while $\cdot$ **do** $\cdot$ **done**, **39**, 42, 48, 69, 120
while $\cdot$ **inv** $\cdot$ **do** $\cdot$ **done**, 119
while $\cdot$ **inv** $\cdot$ **var** $\cdot$ **do** $\cdot$ **done**, 121

Concept Index

μ-Recursive Functions, **32**

Abstract Domain, **24**
Algebraic Data Type, **35**
Algebraic Properties, **155**
Algebraic Specification, **35**
Assertion, **27**
    Typing-, **99**
        most precise-, **108**
Assertion Language, **26**
Automated Theorem Prover, **35**

Closure, **146**
Consensual Typing, 12, **23**
Correctness
    type-safe partial-, 87
    typesafe partial-, **83**

Denotation, **16**
Dynamic Typing, **20**

Environment, **151**
Error State, **58**

Finite field, **131**
Flow Graph
    directed, labeled, **46**
Formal Methods, **26**
Formula
    closed-, **30**

Galois Connection, **25**
Galois field, **131**
Galois Insertion, **25**
Gradual Typing, **23**

Hoare Logic, **26**
    dynamically typed-, **63**
    statically typed-, 59, **63**
    Tagged-, **57**
Hoare Triple, **26**
    Tagged-, **59**
    Validity, **26**

Interactive Type Inference, 12, **107**
Interface, **153**
Introspection, **147**
    Test for variable existence, **145**
Invariant
    Global Typing-, **99**

Lattice
    complete-, **24**
    join, **24**
    meet, **24**
    pre-order, **24**
Liskov substitution principle, **15**
Logic
    Closure
        Existential-, **30**
        Universal-, **30**
    Formula
        Satisfiability, **30**
        Validity, **31**
    Individual, **29**
    Interpretation, **30**
    Model, **30**
    Satisfaction Relation, **30**
    Signature, **29**

Monotone Framework, **49**
Most General Correctness Formulas, **71**

Object-Orientation
    Behavioral subtyping, **15**

Class, **14**
Classes, **15**
Constructor, **15**
Dynamic Dispatch, **15**
Encapsulation, **14**
Inheritance, **15**
Instance, **14**
Instance Variables, **15**
Method, **14**
Methods, **15**
Receiver, **14**
Runtime Type, **15**
Subclassing, **15**
Subtyping, **15**
objects
inactive, **41**

Precondition
Type-Safety-, **83**, 87
Program
Configurations, **17**
error-, **17**
final-, **17**
proper-, **17**
States, **15**
type-safe-, **18**
well-typed-, **51**
Program State
Encoding, **72**
Program Verification, **26**
Programming Language, **13**
dynamically typed-, **20**
imperative-, **14**
object-oriented-, **14**
class-based-, **15**
purely-, **14**
Semantics, **16**
Axiomatic-, **16**
Denotational-, **16**
Input-Output-, **26**
Operational-, **16**
Structural Operational-, **16**
State Update, **15**
statically typed-, **20**
Variables, **15**
local-, **15**
Projection, **101**
for variables, **101**
Proof
Two-Layered-, **105**
Type-Safety-, **99**
Typing-, **102**
Proof Checking, **34**
Proof Refinement
Step
conjunctive, **106**
Proof Search, **34, 35**

Refinement
Monotonicity, **105**
Step
conjunctive, **103**
Reflection, **147**
Relation
Correctness-, **24**

Sentence, **30**
Shortcut Rules, **65**
SMT-Solver, **35**
Soft Typing, **22**
States
abstract-, **19**
concrete-, **19**
Static Typing, **20**
Subclassing, **152**
Syntactic Sugar, **14**

Theorem Proving, **34**
Automatic-, **34**
Interactive-, **34**
Transfer Function, **49**
Transition, **17**
Type, **19**
Type Inference, **24**
Soundness, **51**
Type-Safety, **18**
Type-Safety Verifier
automatic-, **97**
Typing, **19**
least precise type-safe-, **19**, **144**

Precision, **19**
Soundness, **19**
Typing Assertions, **99**
Typing Literals, **99**

Union Types, 24

Value Domain, **14**
Variable
Occurrence
bound-, **30**
free-, **30**
Verifier
Soundness, **20**

Part VI.

Appendices

A. Substitutions

Analogous to the state update operation $[u := e]$, the program logic uses 3 different kinds of substitutions on assertions.

1. Substitution of local variables $p[x := e]$

The substitution for local variables (or multiple local variables in parallel) is straightforward.

It is defined by induction on the structure of p:

- $y[x := e] \equiv \begin{cases} e \text{ if } x = y \\ y \text{ otherwise} \end{cases}$ (includes $y \equiv \textbf{self}$)
- $v[x := e] \equiv v$
- $true$, $false$, $1, 2, ...$ (constants - unaffected)
- $l.@v[x := e] \equiv l[x := e].@v$
- **if** l **then** l_1 **else** l_2 **end**$[x := e] \equiv$ **if** $l[x := e]$ **then** $l_1[x := e]$ **else** $l_2[x := e]$ **end**
- $l_1 = l_2 \equiv l_1[x := e] = l_2[x := e]$
- $l_1 < l_2 \equiv l_1[x := e] < l_2[x := e]$
- $[\![l]\!] \in \{C_1, ..., C_n\} \equiv [\![l[x := e]]\!] \in \{C_1, ..., C_n\}$
- $l_1 \wedge l_2 \equiv l_1[x := e] \wedge l_2[x := e]$
- $l_1 \vee l_2 \equiv l_1[x := e] \vee l_2[x := e]$
- $(\neg l_1)[x := e] \equiv \neg(l_1[x := e])$
- $(\exists y : T.l_1)[x := e] \equiv \begin{cases} \exists y : T.l_1 \text{ if } x = y \\ \exists y : T.l_1[x := e] \text{ otherwise} \end{cases}$
- $(\forall y : T.l_1)[x := e] \equiv \begin{cases} \forall y : T.l_1 \text{ if } x = y \\ \forall y : T.l_1[x := e] \text{ otherwise} \end{cases}$

2. Substitution of instance variables $p[l.@v := e]$

The substitution for instance variables needs to take aliasing into account. For this, it is handy to have conditionals in the assertion language.
It is defined by induction on the structure of p:

- $l[l.@v := e] \equiv l$ for $l \equiv x, v, \mathbf{self}, n, true, false$ (constants - unaffected)
- $l'.@v'[l.@v := e] \equiv \begin{cases} \mathbf{if}\ l' = l\ \mathbf{then}\ e\ \mathbf{else}\ l'.@v'\ \mathbf{end} \text{ if } @v = @v' \\ l'.@v' \text{ otherwise} \end{cases}$
- $\mathbf{if}\ l'\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2\ \mathbf{end}[l.@v := e] \equiv$

  $\mathbf{if}\ l'[l.@v := e]\ \mathbf{then}\ l_1[l.@v := e]\ \mathbf{else}\ l_2[l.@v := e]\ \mathbf{end}$
- $l_1 = l_2 \equiv l_1[l.@v := e] = l_2[l.@v := e]$
- $l_1 < l_2 \equiv l_1[l.@v := e] < l_2[l.@v := e]$
- $[\![l']\!] \in \{C_1, ..., C_n\} \equiv [\![l'[l.@v := e]]\!] \in \{C_1, ..., C_n\}$
- $(l_1 \wedge l_2)[l.@v := e] \equiv l_1[l.@v := e] \wedge l_2[l.@v := e]$
- $(l_1 \vee l_2)[l.@v := e] \equiv l_1[l.@v := e] \vee l_2[l.@v := e]$
- $(\neg l_1)[l.@v := e] \equiv \neg l_1[l.@v := e]$
- $(\exists y : T.l_1)[l.@v := e] \equiv \exists y : T.l_1[l.@v := e]$
- $(\forall y : T.l_1)[l.@v := e] \equiv \forall y : T.l_1[l.@v := e]$

Lemma 11 (Substitution of instance variables)**.** *For all logical expressions s and t, all assertions p, all instance variables $@u$ and all proper states σ*

$$\sigma(s[@u := t]) = \sigma[@u := t](s) \tag{A.1}$$

$$\sigma \models p[@u := t] \text{ iff } \sigma[@u := \sigma(t)] \models p. \tag{A.2}$$

Proof. By induction on the structure of s and p. □ □

3. Substitution for object creation $p[x := \mathbf{new}_C]$

The substitution for object creation calculates the weakest precondition of an object creation statement. For a slightly simpler case without classes, [5, page 221] defines a substitution $[x := \mathbf{new}]$. This substitution, however, is only applicable to so-called "pure" assertions. Fortunately, except for conditionals, our logical expressions satisfy all requirements and [5, page 223] gives a Lemma that allows eliminating conditionals like ours by substituting them for logically equivalent expressions. We can thus use the substitution and only need to modify it slightly to reflect the addition of classes.

The substitution is then defined by induction on the structure of p:

- $l[x := \mathbf{new}_C] = l$ for $l \equiv \mathbf{self}, null, v, x, n, true, false$
- $l.@v[x := \mathbf{new}_C] \equiv \begin{cases} init_C.@v \text{ if } l \equiv x \\ l.@v \text{ otherwise} \end{cases}$
- $([\![x]\!] \in \{C_1, ..., C_n\})[x := \mathbf{new}_C] \equiv C \in \{C_1, ..., C_n\}$

- $(l_1 = l_2)[x := \mathbf{new}_C] \equiv l_1[x := \mathbf{new}_C] = l_2[x := \mathbf{new}_C]$ if $l_1 \not\equiv x$ and $l_1 \not\equiv if...end$ and $l_2 \neq x$ and $l_2 \not\equiv if...end$
- $(x = l_2)[x := \mathbf{new}_C] \equiv false$ if $l_2 \not\equiv x$ and $l_2 \not\equiv if...end$ (also the symmetric case)
- $(x = x)[x := \mathbf{new}_C] \equiv true$
- $(\mathbf{if}\ l_0\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2\ \mathbf{fi} = l')[x := \mathbf{new}_C] \equiv \mathbf{if}\ l_0[x := \mathbf{new}_C]\ \mathbf{then}\ (l_1 = l')[x := \mathbf{new}_C]\ \mathbf{else}\ (l_2 = l')[x := \mathbf{new}_C]\ \mathbf{end}$
- $\mathbf{if}\ l'\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2\ \mathbf{fi}[x := \mathbf{new}_C] \equiv \mathbf{if}\ l'[x := \mathbf{new}_C]\ \mathbf{then}\ l_1[x := \mathbf{new}_C]\ \mathbf{else}\ l_2[x := \mathbf{new}_C]\ \mathbf{fi}$

  Note: conditionals can always be moved outwards to be the outermost operation in an assertion.
- $l_1 < l_2 \equiv l_1[x := \mathbf{new}_C] < l_2[x := \mathbf{new}_C]$
- $(l_1 \wedge l_2)[x := \mathbf{new}_C] \equiv l_1[x := \mathbf{new}_C] \wedge l_2[x := \mathbf{new}_C]$
- $(l_1 \vee l_2)[x := \mathbf{new}_C] \equiv l_1[x := \mathbf{new}_C] \vee l_2[x := \mathbf{new}_C]$
- $(\neg l_1)[x := \mathbf{new}_C] \equiv \neg l_1[x := \mathbf{new}_C]$
- $(\exists y : T.l_1)[x := \mathbf{new}_C] \equiv \exists y : T.l_1[x := \mathbf{new}_C] \vee l_1[x/y][x := \mathbf{new}_C]$
- $(\forall y : T.l_1)[x := \mathbf{new}_C] \equiv \forall y : T.l_1[x := \mathbf{new}_C] \wedge l_1[x/y][x := \mathbf{new}_C]$

B. Simplification Rules

Definition 32. *The Simplifier* $simp : Asrt \mapsto Asrt$ *recursively applies the following rewrite rules*

- $simp(p \wedge false) \equiv false$
- $simp(false \wedge p) \equiv false$
- $simp(true \wedge p) \equiv p$
- $simp(p \wedge true) \equiv p$
- $simp(p \vee true) \equiv true$
- $simp(true \vee p) \equiv true$
- $simp(false \vee p) \equiv p$
- $simp(p \vee false) \equiv p$
- $simp(false \rightarrow p) \equiv true$
- $simp(p \rightarrow true) \equiv true$
- $simp(true \rightarrow p) \equiv p$
- $simp(p \rightarrow false) \equiv \neg p$
- $simp(\forall x : \tau \bullet p) \equiv p$ *if* $x \notin free(p)$
- $simp(\exists x : \tau \bullet p) \equiv p$ *if* $x \notin free(p)$

C. Pure Expressions

Identifying pure expressions $(pure : \mathcal{T}_s \times Expr_d \mapsto \mathbb{B})$

$pure(e) \triangleq \tau(e)$ defined, $\qquad \tau(e) = inf(\{\mathbb{T} \mid pure(\mathbb{T}, e)\})$

$pure(\mathbb{O}, \text{null}) \triangleq true$,

$pure(\mathbb{T}, \text{u}) \triangleq [\![\text{u}]\!] \in T \wedge \Psi(T) = \mathbb{T} \wedge safe_{\mathbb{T}}(\text{u})$ (includes the case of **self**)

$pure(\mathbb{T}, @\text{x}) \triangleq [\![\textbf{self}.@\text{x}]\!] \in T \wedge \Psi(T) = \mathbb{T} \wedge safe_{\mathbb{T}}(@\text{x})$

$pure(\mathbb{B}, e \text{ is\_a? C}) \triangleq pure(\mathbb{O}, e)$

$pure(\mathbb{B}, e_1 == e_2) \triangleq \exists \mathbb{T} \in \mathcal{T}_s \bullet\ pure(\mathbb{T}, e_1) \wedge pure(\mathbb{T}, e_2)$

$pure(\mathbb{T}, e_0.\text{m}(e_1, ..., e_n)) \triangleq$
$\quad \exists \mathbb{T}_0, ..., \mathbb{T}_n \bullet\ pure(\mathbb{T}_i, e_i)$ for $i \in \mathbb{N}_n \wedge \Psi(\mathbb{T}_0.\text{m}(\mathbb{T}_1, ..., \mathbb{T}_n) \to \mathbb{T})$ defined

$pure(\mathbb{T}, \text{new C}(e_1, ..., e_n)) \triangleq$
$\quad \exists \mathbb{T}_1, ..., \mathbb{T}_n \bullet\ pure(\mathbb{T}_i, e_i)$ for $i \in \mathbb{N}^1_n \wedge \Psi(\Psi(\{C\}).\text{init}(\mathbb{T}_1, ..., \mathbb{T}_n) \to \mathbb{T})$ defined

$pure(\mathbb{T}, \text{u} := e) \triangleq false$

$pure(\mathbb{T}, @\text{v} := e) \triangleq false$

$pure(\mathbb{T}, \text{while } e \text{ do } S \text{ od}) \triangleq false$

$pure(\mathbb{T}, \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi}) \triangleq pure(\mathbb{B}, e) \wedge S_i \equiv e_i \wedge pure(\mathbb{T}, e_i)$ for $i \in \{1, 2\}$

Translation of pure expressions into logical expressions $(\Psi : \mathcal{T}_s \times Expr_d \mapsto LExp)$

$\Psi(e) \triangleq \Psi_{\tau(e)}(e)$, $\Psi_{\mathbb{T}}(\text{x}) \triangleq \hat{\text{x}}$

$\Psi_{\mathbb{O}}(\text{null}) \triangleq \text{null}$,

$\Psi_{\mathbb{T}}(e_0.\text{m}(e_1, ..., e_n)) \triangleq l[v_0, ..., v_n := \Psi_{\mathbb{T}_0}(e_0), ..., \Psi_{\mathbb{T}_n}(e_n)]$
where $pure(\mathbb{T}_i, e_i)$ for $i \in \mathbb{N}_n$ and $\Psi(\mathbb{T}_0.\text{m}(\mathbb{T}_1, ..., \mathbb{T}_n) \to \mathbb{T}) = l$.

$\Psi_{\mathbb{T}}(\text{new C}(e_1, ..., e_n)) \triangleq l[v_1, ..., v_n := \Psi_{\mathbb{T}_1}(e_1), ..., \Psi_{\mathbb{T}_n}(e_n)]$
where $pure(\mathbb{T}_i, e_i)$ for $i \in \mathbb{N}^1_n$ and $\Psi(\Psi(\{C\}).\text{init}(\mathbb{T}_1, ..., \mathbb{T}_n) \to \mathbb{T}) = l$.

$\Psi_{\mathbb{B}}(e_1 == e_2) \triangleq \Psi_{\mathbb{T}}(e_1) = \Psi_{\mathbb{T}}(e_2)$ where $pure(\mathbb{T}, e_i)$ for $i \in \{1, 2\}$.

$\Psi_{\mathbb{B}}(e \text{ is\_a? C}) \triangleq [\![\Psi_{\mathbb{O}}(e)]\!] \in \{\text{C}\}$

$\Psi_{\mathbb{T}}(\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ fi}) \triangleq \text{if } \Psi_{\mathbb{B}}(e) \text{ then } \Psi_{\mathbb{T}}(e_1) \text{ else } \Psi_{\mathbb{T}}(e_2) \text{ fi}$

Soundness

Proof. The Axiom PURE EXPR can be established by induction over the structure of the pure expression e, using the guarantees provided by $pure(e)$. In the cases for

variables, $\mathit{pure}(\mathrm{x})$ implies $\mathit{safe}_{\mathbb{T}}(\mathrm{x})$ for some type $\mathbb{T}$. In the case for method calls, we assume

$$\{p[\hat{\mathbf{r}} := \Psi(\mathbb{T}_0.\mathrm{m}(\mathbb{T}_1, ..., \mathbb{T}_n) \to \mathbb{T}]\}\mathrm{u}_0.\mathrm{m}(\mathrm{u}_1, ..., \mathrm{u}_n)\{p\}$$

with $\mathbb{T}_i = \tau(\mathrm{u}_i)$ for all $i \in \mathbb{N}_n$ to be established for all methods in $\mathfrak{M}_\varepsilon$ (which is precisely the meaning of "correspondence between methods and operations with respect to the mapping Ψ" in Chapter 7).

The rules PURE ASGN, PURE COND, PURE LOOP and PURE METH can then be derived by combining the axiom PURE EXPR with the **dyn** rules for ASGN, COND, LOOP and METH respectively. For instance

AXIOM: PURE ASGN

$$\dfrac{\dfrac{}{\{p'[\hat{\mathbf{r}} := \Psi(e)]\}e\{\mathbf{tags} \wedge p'\} \text{ with } p' \equiv p[x := \mathbf{r}]}\,\text{PURE EXPR}}{\{p[\hat{\mathrm{x}}, \hat{\mathbf{r}} := \Psi(e), \Psi(e)]\}\mathrm{x} := e\{\mathbf{tags} \wedge p\}}\,\text{ASGN}$$

where $\mathit{pure}(e)$, $\tau(e) \sqsubseteq \tau(\mathrm{x})$.

RULE: PURE COND

$$\dfrac{\dfrac{\dfrac{\{p \wedge \Psi(e)\}S_1\{\mathbf{tags} \wedge q\}}{\{p \wedge \hat{\mathbf{r}} \leftrightarrow \Psi(e) \wedge \hat{\mathbf{r}}\}S_1\{\mathbf{tags} \wedge q\}}\,\text{CONS}}{\{r \wedge \mathbb{B}(\mathbf{r}, \mathit{true})\}S_1\{\mathbf{tags} \wedge q\}}\,\text{CONS} \qquad \dfrac{\dfrac{\{p \wedge \neg\Psi(e)\}S_2\{\mathbf{tags} \wedge q\}}{\{p \wedge \hat{\mathbf{r}} \leftrightarrow \Psi(e) \wedge \neg\hat{\mathbf{r}}\}S_2\{\mathbf{tags} \wedge q\}}\,\text{CONS}}{\{r \wedge \mathbb{B}(\mathbf{r}, \mathit{false})\}S_2\{\mathbf{tags} \wedge q\}}\,\text{CONS}}{\begin{array}{c}\{p \wedge \Psi(e) \leftrightarrow \Psi(e)\}e\{\mathbf{tags} \wedge r\} \\ \{r \wedge \mathbb{B}(\mathbf{r}, \mathit{true})\}S_1\{\mathbf{tags} \wedge q\} \\ \{r \wedge \mathbb{B}(\mathbf{r}, \mathit{false})\}S_2\{\mathbf{tags} \wedge q\}\end{array}}\,\text{PURE EXPR}$$

$$\dfrac{\begin{array}{c}\{p\}e\{\mathbf{tags} \wedge r\} \\ \{r \wedge \mathbb{B}(\mathbf{r}, \mathit{true})\}S_1\{\mathbf{tags} \wedge q\} \\ \{r \wedge \mathbb{B}(\mathbf{r}, \mathit{false})\}S_2\{\mathbf{tags} \wedge q\}\end{array}}{\{p\} \text{ if } e \text{ then } S_1 \text{ else } S_2 \text{ fi } \{\mathbf{tags} \wedge q\}}\,\text{COND (simplified)}$$

where $r \equiv p \wedge \hat{\mathbf{r}} \leftrightarrow \Psi(e)$, $\mathit{pure}(e)$, $\mathbf{r} \notin \mathit{free}(p)$ and $\tau(e) = \mathbb{B}$. □

D. Unsovable Verification Conditions

We give here the SMTLIB2 code of several verification conditions that came up in our experiments, but could not be solved using the SMT-Solver Z3. They are listed here in the hope that they might be useful for the automatic theorem proving community. With each verification condition we will point out the precise meaning of "could not be solved".

1. SMTLIB2 - translation of the predicate *treeval*3 from Section 11.4.

Even when simplifying the mapping predicates map_N and map_L using uninterpreted functions, even trivial formulas containing the predicate *treeval*3() cause Z3 to return "unknown".

```
; tree recursion

(declare-const MAX_LEVEL Int)
(assert (= MAX_LEVEL 4))

(define-fun pow2 ((n Int)) Int
                    (ite (= n 0) 1
                    (ite (= n 1) 2
                    (ite (= n 2) 4
                    (ite (= n 3) 8
                    (ite (= n 4) 16
                    32
                    ))))))

(define-fun s-level ((l Int)) Int
                    (- (pow2 l) 1)
                    )

(define-fun size-level ((l Int)) Int
                    (pow2 l)
                    )

(define-fun index-of ((level Int) (index Int)) Int
                         (+ (s-level level) index)
                         )

(define-fun child-a ((level Int) (index Int) (level-child Int)
                     (index-child Int)) Bool
```

```
                    (and (= level-child (+ level 1))
                         (= index-child (* index 2))
                         ))

(define-fun child-b ((level Int) (index Int) (level-child Int)
                     (index-child Int)) Bool
                    (and (= level-child (+ level 1))
                         (= index-child (+ (* index 2) 1))
                         ))

(define-fun valuetree2 ((t MyList) (n Int)) Bool
                   (and
                     (exists ((o2 Obj))
                       (and
                         (select_list t 0 const_value)
                         (select_list t 1 o2)
                         (map_N o2 n)
                       )
                     )
                   )
)

(define-fun vartree3 ((t MyList) (e MyMap) (x Obj)) Bool
                   (and
                     (select_list t 0 const_var)
                     (select_list t 1 x)
                     (not (maps_to e x Null))
                   )
)

(define-fun optree ((t MyList) (l Obj) (r Obj) (op Obj)) Bool
                                  (and
                                    (select_list t 0 const_op)
                                    (select_list t 1 op)
                                    (select_list t 2 l)
                                    (select_list t 3 r)
                                  )
)


(define-fun treeval3 ((t MyList) (e MyMap) (n Int)) Bool
    (exists ((a (Array Int MyList)) (res (Array Int Int)))
      (and
        (= (select a (index-of 0 0)) t)
        (= (select res (index-of 0 0)) n)
```

```
(forall ((i Int) (li Int))
  (=>
    (and (<= 0 li) (< li MAX_LEVEL) (<= 0 i)
         (< i (size-level li)))
    (exists ((lj Int) (ll Int) (rj Int) (rl Int))
      (and
        (child-a li i ll lj)
        (child-b li i rl rj)
        (or
          (and
            (= (select res (index-of li i)) -1)
            (= (select res (index-of ll lj)) -1)
            (= (select res (index-of rl rj)) -1)
          )
          (and
            (valuetree2 (select a (index-of li i))
                        (select res (index-of li i)))
            (= (select res (index-of ll lj)) -1)
            (= (select res (index-of rl rj)) -1)
          )
          (and
            (exists ((x Obj) (v Obj))
              (and
                (vartree3 (select a (index-of li i)) e x)
                (maps_to e x v)
                (map_N v (select res (index-of li i)))
              )
            )
            (= (select res (index-of ll lj)) -1)
            (= (select res (index-of rl rj)) -1)
          )
          (exists ((l Obj) (r Obj))
            (and
              (optree (select a (index-of li i)) l r const_add)
              (map_L l (select a (index-of ll lj)))
              (map_L r (select a (index-of rl rj)))
              (= (select res (index-of li i))
                (+ (select res (index-of ll lj))
                   (select res (index-of rl rj)))
              )
            )
          )
        )
      )
    )
```

```
          )
        )
      )
    )
)
```

an example for such a trivial formula is

```
(declare-const tree MyList)
(declare-const env MyMap)
(declare-const o Obj)

(assert (select_list tree 0 const_value))
(assert (select_list tree 1 o))
(assert (map_N o 2))

(assert (treeval3 tree env 2))
```

The full SMTLIB2 example can be found in the file `unsolvable/treeval3.smt` in the supplementary material.

2. First Verification Condition of Case Study "In-Place Value Switching" (Section 11.3) when verified without the Layer of Abstraction.

The SMTLIB2 translation of the verification condition is as follows:

```
(declare-const pv_other Obj)
(declare-const pv_res Obj)
(declare-const pv_x Obj)
(declare-const lv_x Int)
(declare-const pv_y Obj)
(declare-const lv_y Int)

(assert (not
(=>
  (exists ((lv_x2 Int) (lv_y2 Int))
    (and
      (map_N pv_x lv_x2)
      (map_N pv_y lv_y2)
      (= lv_y (- lv_x2 lv_y2))
      (= lv_x lv_y2)
    )
  )
  ; Rule of Adaptation for x = x - y
  (exists ((lv_s Int) (lv_o Int))
    (and
      (map_N pv_x lv_s)
      (map_N pv_y lv_o)
```

```
      (> lv_s lv_o)
      (forall ((pv_res Obj) (pv_s Obj) (pv_o Obj))
        (=>
          (map_N pv_res (- lv_s lv_o))
          (and
            (map_N pv_res lv_y)
            (map_N pv_y lv_x)
          )
        )
      )
    )
  )
)))
```

when passed to Z3, it returns “unknown”.

The full SMTLIB2 example with all predicate definitions can be found in the file `unsolvable/swap-noLoA.smt` in the supplementary material.