UNIVERSITY
OF OSLO

Master's thesis

Contra: Automatically Finding Algebraic Counterexamples to Property-Based Tests

Sophie Adeline Solheim Bosio

Informatics: Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2024

Sophie Adeline Solheim Bosio

Contra: Automatically Finding Algebraic Counterexamples to Property-Based Tests

Supervisors:
Michael Kirkedal Thomsen
Joachim Tilsted Kristensen

Abstract

Property-based testing is a testing methodology based on assertions — called *properties* — that should hold for arbitrary, correctly typed input. Its primary objective is to produce *counterexamples*, if any such values exist, that prove the property wrong. Conventional property-based testing tools employ randomised input generators to find such counterexamples and these are largely efficient. However, should the user wish to test properties which take as input a user-defined algebraic data type, they are required to implement a generator for the type themself. This presents a high barrier to entry.

SMT (Satisfiability Modulo Theories) solvers are powerful software tools that can find satisfying models to logical formulae. Modern SMT solvers can find models to highly complex formulae with incredible efficiency. We propose the idea that SMT solvers might be able to find counterexamples of general user-defined algebraic data types.

In this thesis, we formalise Contra, a small, functional programming language with a native construct for defining properties and an automatic property-checker capable of producing counterexamples of user-defined algebraic data types automatically. We show how we can encode regular program terms as symbolic variables with logical constraints, and how we can utilise an SMT solver to find counterexamples to the corresponding property. We formalise and implement a translation algorithm which accounts for constraints on algebraic data types. We evaluate our approach by synthesising Contra properties with QuickCheck and using Contra's built-in property-checker to validate the result.

Contents

List of Figures

Listings

Preface

During the first semester of my master's degree programme at the University of Oslo, there was an open-invite event where thesis supervisors presented topic suggestions to the prospective thesis authors in the audience. I was very overwhelmed at the time by all the excellent suggestions I had seen previously on the department's homepages and by the time I arrived to see the presentations, I was nervous about selecting a topic. This decision suddenly became easy after I went to the 10th floor to talk to Joachim about his and Michael's presentation on property-based testing. Their genuine passion for functional programming, programming language design, and software quality was outright inspiring. I would soon find out they were also incredibly knowledgeable and generous with their time and energy.

The original thesis proposal entailed using (partial) program inversion to derive well-typed generators for a small functional programming language. I was doubly lucky then to work so closely with Michael, whose experience with reversible and invertible programming languages is outright impressive, and Joachim, who, on top of writing an invertible programming language himself, is also a QuickCheck-aficionado. Eventually, I shifted the focus of the thesis to use SMT solvers and work with constraint solving instead, but the thesis is still full to the brim of things I have learned from them. Looking back, I can say with certainty how crucial it was to have followed their special curriculum in functional programming, semantics, and types.

I would like to convey my deepest gratitude to Michael and Joachim for being my supervisors. I have been extremely lucky to work with them and I will always carry with me the many lessons they have taught me and the memories of fun, frivolous and fruitful discussions at the 10th floor. It has been a genuine pleasure.

From the university, I would also like to thank Lars Tveito for all his advice and guidance throughout the writing of this thesis. I will miss all our nerdy conversations in the hallway and by the kitchenette. His students are extremely lucky to have him. A thanks and the best of wishes go to my fellow students Triera, Sergey and Stian, who I have spent many hours with in the time leading up to this thesis project, and whose future work I am genuinely excited to see. Thank you to all the PhD students on the tenth floor who have graciously accepted me as a half-permanent visitor in their office. They have made me feel so welcome and it has been a joy to spend the majority of my writing days with them. I wish them all the very best of luck on their theses and in their future endeavours.

Finally, I would like to thank my family. My parents and my brother, Victor, have always encouraged me to pursue my academic interests and been available to give advice, comfort, and encouragement. I would like to thank all four of my grandparents, whose achievements and resilience inspire me to this day. I want to thank Sandra, my favourite person in the world, for being unfathomably patient and kind always, and for making me laugh, sleep, and take care of myself throughout the writing of this thesis.

Thank you,
Sophie Bosio

Chapter 1

Introduction

The advantages of thorough and frequent software testing are well-documented and we all seem to agree on its merits. As the use of software increasingly permeates our private and professional lives, we progressively become more dependent on the correctness of the software we interact with, and thus interested in assuring software quality by comprehensive testing. However, testing is costly in the short-term and therefore unappealing. Writing a comprehensive test suite is time-consuming and requires significant of effort and forethought on the part of the programmer.

The simultaneous need for and aversion to writing tests has motivated the creation of property-based testing tools. Property-based testing is a software testing methodology in which the programmer formulates assertions — or *properties* — about a program fragment, that should hold true for arbitrary well-typed input. Conventionally, property-based testing tools come equipped with input generators which can produce well-typed input for built-in types. By substituting these generated inputs into the property, the tool implicitly creates a series of distinct, concrete test cases that can be run in the usual way.

Property-based testing stands in contrast to example-driven testing methodologies, such as unit and integration testing. Example-driven tests also assert the equivalence between the *expected* result and the *actual* result of transforming an example input, but a single desired property is often specified using a series of concrete example pairs. The programmer, then, must invent example inputs and correctly work out the intended outputs. In large test suites, this is laborious and risks introducing errors. Further, the programmer may fail to anticipate unexpected edge cases and consequently neglect testing these. Property-based testing, on the other hand, is powerful in part because its randomised nature makes it ignorant of the programmer's intentions and wishes, much like a user might be.

Though a robust and compelling approach to testing, property-based testing in turn presents new challenges. In particular, property-based testing tools have built-in generators for most primitive and some compound types, but not for user-defined algebraic data types. The programmer is then to write such generators by hand. Writing good generators is sufficiently costly on its own that property-based tests in these cases often are foregone in favour of traditional, example-driven tests.

Property-based testing has simplified the process of testing program fragments by automating the instantiation of concrete test cases. The hard problem then becomes the implementation and design of well-typed generators. In an attempt to automate a greater part of the process, we investigate the use of SMT solvers in the automatic generation of counterexamples.

In order to facilitate property-based testing with algebraic data types, this thesis presents the small functional programming language Contra. Contra sacrifices the expressiveness

and power of conventional property-based testing tools for convenience and accessibility by using an SMT solver to produce counterexamples instead of random generators. We make the language approachable by providing a built-in construct for defining properties and a property-checker that does not require hand-written generators for any type expressible in the language.

1.1 Goals

Where property-based testing is an abstraction of concrete test input to the underlying logic behind a collection of test cases, we wish to abstract away the notion of a generator in our language.

We do not have any ambition to outperform hand-written generators, but rather to remove friction between the formulation of a property and checking it, encouraging use of property-based testing. To facilitate the process, we define the following goals for Contra:

- Ability to generate user-defined algebraic data types from their definition alone.
- Automatic property-checker.
- Sound property-checker.
- Support properties as a first-class language construct.
- Familiar, ML-style syntax.

The primary goal and the reason we are implementing yet another programming language, is for Contra to be able to generate user-defined algebraic data types without any user-input beyond the algebraic data type definition itself. Therefore, the language must implement an algorithm to produce well-typed and well-formed inputs from the definition on an algebraic data type alone.

The user should not need an awareness of the notion of a generator or of the underlying mechanics in the language. Instead, the user should only be required to write a well-typed program containing properties, functions, type signatures and algebraic data type definitions. When they request that the properties be checked, the results should be produced automatically without further input.

By *soundness* of the property-checker, we mean that all the counterexamples generated by the property-checker should be true counterexamples. In other words, we should never get a "counterexample" that does *not* cause the property to fail. On the other hand, having *completeness* would mean that all counterexamples that exist for a given property would be reachable by the property-checker. This would imply that there were no counterexamples to a property that the property-checker could not find, given enough time. In our case, this is a much harder problem and outside the scope of this thesis. We therefore resign ourselves to the possibility that there are counterexamples we will never find, but we do guarantee that all counterexamples are true counterexamples and therefore useful.

We envision a programming language where properties can be defined as straightforward functions that return Boolean values. The programmer identifies functions as properties to be checked with a special =⋆=-operator, but during normal program execution, properties act as regular functions. This means the programmer can use properties to define other properties in turn. "Properties" should be "first-class citizens" on par with functions. In other words, properties should be a native construct available in the core language.

Since functional languages lend themselves well to property-based testing, the functional paradigm is a natural choice for Contra. As users of the language are more likely from the academic environment than from industry, it makes sense to choose a syntax modelled after the ML-family of languages, which are commonplace in academia. Being an otherwise ordinary functional language, we hope that the approach described in this thesis can applied to more expressive languages in the future.

1.2 Contributions

The main contribution from this thesis is a translation algorithm from properties given by conventional terms in a small functional programming language, including algebraic data types, into symbolic formulae with encoded constraints that can be automatically checked by an SMT solver. The translation approach is validated by comparing Contra's property-checker to manual property-checking with hand-written generators in Haskell.

In the process of writing the property-checker, we have:

- Implemented a parser, an interpreter, and type-inference and unification algorithms.
- Implemented a partial evaluator.
- Written a translation algorithm from regular terms into SMT-checkable formulae.
- Created an SMT-compatible enumeration of algebraic data type instantiations.

The property-checker we create cannot handle recursive function definitions, but it *can* generate well-formed, well-typed, and valid counterexamples of user-defined, mutually recursive algebraic data types.

The implementation is written in Haskell[26], a general-purpose, purely functional and statically typed programming language. Haskell is well-suited for writing interpreters, parsers, and partial evaluators, and there exist many Haskell libraries for interacting with SMT solvers. We choose the library SBV [14], which is both powerful and convenient to use.

1.3 Chapter Overview

Chapter 2 lays the theoretical foundation for the thesis by introducing the key concepts and background material that we will rely on throughout the thesis. We give a brief introduction to property-based testing and indicate points of friction when using conventional property-based testing tools. We review the literature on the tools and techniques we will utilise in our design and implementation. Finally, we highlight existing tools and frameworks in the field from which we draw inspiration.

Chapter 3 presents an analysis of the problem space. On the basis of our theoretical understanding and our goals for the project, we begin to narrow the scope of the project to a suitable size. At this point, we identify requirements for the design of the language. We formalise a notion of *encoding* algebraic data types, which will form the basis of our implementation.

Chapter 4 introduces the design of the core language, including its abstract syntax, typing rules, and operational semantics. Equipped with a concrete language and our encoding of algebraic data types, we begin to sketch an approach for our translation algorithm.

Chapter 5 covers the implementation of the programming language. We begin with the most abstract aspects of the implementation and gradually approach the translation algorithm. The implementation includes an abstract syntax, parser, type-inferrer, interpreter, partial evaluator, and a property-checker.

Chapter 6 is an evaluation of the language and its property-checker. We evaluate Contra by synthesising Contra properties using QuickCheck and checking that Contra's built-in property-checker is able to find the intended counterexamples.

Chapter 7 is a discussion of the evaluation. We discuss our findings and judge to which degree the final project accomplishes the goals we set for the thesis. We identify limitations of our approach and compare Contra to existing projects.

Chapter 8 reflects work done thus far. We propose future extensions and amendments to the project and leave the reader with our concluding remarks.

1.4 Source Code

The source code is available on GitHub, at the URL https://github.com/SophieBosio/contra. Instructions for how to build, test, and install the program can be found in the `README.md` file in the GitHub repository of the prototype implementation. It requires the Haskell build tool Stack [10].

Chapter 2

Background

In this chapter, we lay the theoretical groundwork for the rest of the thesis. We present the background for the problem analysis and solution design in Chapters 3 and 4 by covering property-based testing generally, algebraic data types, SMT solvers, partial evaluation, and related work. This chapter should cover the prerequisites necessary to understand the particulars of the problem and the trade-offs we make in the design of Contra.

2.1 Property-Based Testing with QuickCheck

QuickCheck is the pioneer of property-based testing tools, developed for Haskell by Claessen and Hughes [8]. Re-implementations and derivative tools exist for a host of other programming languages [3, 12, 34]. QuickCheck allows the user to formulate properties as Haskell functions, with certain special constructs. For instance, the type of a property is not a simple `Bool`, but rather a `Property`. Testing a `Property` provides additional results to aid debugging. Besides the special type, a simple QuickCheck property may look like a regular equality assertion. In his tutorial *How to Specify it!* [18], Hughes demonstrates how to use QuickCheck to test that reversing a list twice yields the original list:

```
prop_Reverse :: [Int] -> Property
prop_Reverse xs = reverse (reverse xs) === xs
```

QuickCheck comes with built-in generator instances for basic types, including `[Int]`. Thus, given a `reverse` function, the above property `prop_Reverse` can be tested directly with no further action on the part of the programmer. A generator instantiates a series of integer lists and checks the property for each one. If a test case fails, that particular list is called a *counterexample* to the property. QuickCheck will generate random lists of integers until a counterexample is found or a preset number of test cases succeed. Discovered counterexamples can be *shrunk* down to minimal examples for which a given property does not hold.

QuickCheck presents a typeclass `Arbitrary` for types that can be generated this way. It has two functions. A mandatory function, `arbitrary`, which implements a generator, and an optional function, `shrink`, which specifies valid shrinking strategies for the type.

A QuickCheck generator for a type `a`, and therefore the implementation of `arbitrary` for that type, is represented by the monad `Gen a`. QuickCheck includes monadic combinators for the user to tune existing generators or build custom generators. They allow the user to restrict, modify, or combine the output of one or multiple generators.

However, should the user define an algebraic data type they wish to test, they must also write the relevant generator (s). Even with the monadic generator combinators, this remains

a significant barrier to entry and to the wide-spread adoption of property-based testing, which has motivated the effort to design derivative works to facilitate the process [13, 22, 23].

2.1.1 Algebraic Data Types

An algebraic data type (ADT) is a composite type, which consists of user-defined data type constructors, each over a (possibly empty) sequence of other types. These may be recursive, meaning the algebraic data type, in its own definition, occurs as a field in a constructor. Consider the data-type definition of a singly-linked list in Haskell.

```
data List a = Nil | Cons a (List a)
```

A list of type `a` is either the empty list (`Nil`) or the concatenation of an element `a` and another list.

Formally, an algebraic data type is a (possibly recursive) sum type, consisting of product types [33]. Each constructor functions as a tag which identifies a given product type and separates it from the others. Each constructor must therefore be distinct. The fields, meanwhile, form an n-ary tuple. We can abuse formal notation slightly and represent them in the following way.

$$\text{List } a = (\text{Nil}) + (\text{Cons},\ (a,\ (\text{List } a)))$$

If there is only one constructor, the result is a single product type. For instance, a type representing a Cartesian coordinate is a product type,

```
data CartesianCoordinate = Coordinate Float Float
```

which we can represent inthe following way.

$$\text{CartesianCoordinate} = (\text{Coordinate},\ (\text{Float},\ \text{Float}))$$

Since a constructor can have field types, as is the case for `Coordinate` which takes two `Floats`, we can consider each product type to represent a *set*. The set consists of all the possible values of that product type. In the case of a Cartesian coordinate, this is coincidentally the constructor `Coordinate` over the Cartesian product of all possible `Floats`. For a list of type `Int`, the set consists of `Nil` and all possible `ints` concatenated with all possible lists of `Ints`.

As such, each product type forms a set. And since the constructors are all distinct, the sum type of product types is isomorphic to a (tagged) disjoint union in set theory. No constructors (tags) can overlap, but the fields of each constructor can, because the constructor makes the whole set distinct.

2.1.2 Generators for Algebraic Data Types

We now turn to a more practical example. Consider an algebraic data type representing a binary search tree (BST). It has the 0-ary constructor `Leaf` and the 4-ary constructor `Branch`. In the latter, the `BST` constructor itself occurs, making the definition recursive. The code below is in Haskell, taken from *How to Specify it'!*.

```
data BST k v = Leaf | Branch (BST k v) k v (BST k v)
  deriving (Eq, Show , Generic)
```

Writing generators for user-defined algebraic data types is hard for a number of reasons: They must be sound, complete, and interesting. In other words, a generator that only generates valid instances of a given type (sound), that can generate *all* such instances (complete), and that prefers generating instances different from one another (interesting). A list generator that produces the empty list half the time, is neither particularly interesting nor very useful. At the same time, we do want to produce some "trivial" instances, such as the empty tree or the tree with only one node, since these can represent edge cases with respect to the function under test. An interesting generator is therefore one that avoids generating many similar results, instead trying to cover as much of the input domain as possible.

As Lampropoulos et al. point out [23], testing an algebraic data type with an invariant property requires special attention. By invariant property, we mean any conditions besides well-typedness that constrain the values of a type. For instance, a given binary tree is only a valid binary *search* tree if it is sorted so that, for all nodes, all the keys in the left sub-tree are lesser than the key in the node, and that all the keys in the right sub-tree are greater than it. This property is invariant and must be preserved by operations on the tree. To test this with properties, we must therefore write a generator that only produces valid instances a priori *and* there must be a validity check in the conclusion of property. The algebraic data type definition, the properties, and the generator must all be kept in sync. This creates additional cognitive overload and risks introducing bugs.

The naïve approach to generating a binary search tree would be to randomly generate either a leaf or a branch, and in the case of a branch, to randomly generate an integer for the key, generate the value and repeat. It is very unlikely that such an approach would satisfy the preconditions, and would thus lead us to discard a great number of candidates. Let us consider `Arbitrary` instance for the naïve binary (search) tree generator from *How to Specify It!*:

```
instance (Ord k, Arbitrary k, Arbitrary v) => Arbitrary (BST k v) where
  arbitrary = do
    kvs <- arbitrary
    return (foldr (uncurry insert) nil (kvs :: [(k, v)]))
  shrink = genericShrink
```

With its special types, typeclasses, functions, monads and combinators, QuickCheck comprises an embedded domain-specific language [8] that the programmer must learn before they are able to write a generator. Although the semantics of QuickCheck generation is unsurprising, becoming familiar with the DSL is non-trivial. This is true for other property-based testing-oriented languages and frameworks like Luck, as well. Writing generators by hand grows increasingly complex as the algebraic data types we want to generate become more specific.

In summary, we can name at least three challenges to writing generators for algebraic data types by hand:

- The generator must be tuned to produce a sensible distribution of possible inputs.
- The generator is mutually dependent upon the specification of the algebraic data type in question and the two must be coordinated and synchronised.
- Domain-specific languages or tools for specifying generators have special syntax and constructs that the programmer must learn in order to write a generator.

Ideally, we would implement a derivation algorithm for generators for arbitrary user-defined algebraic data types. However, this has proven to be rather difficult [22] and

existing languages with such capabilities tend to also make use of more advanced language constructs, such as refinement types [37]. If we constrain ourselves to a garden-variety functional programming language, we instead propose that we might be able to produce counterexamples to algebraic data types using SMT solvers.

2.2 SMT Solvers

Satisfiability Modulo Theories (SMT) solvers are software tools designed to decide the satisfiability of a logical first-order formula. A formula is satisfiable if and only if there exists an assignment to its free variables such that the formula as a whole evaluates to true. SMT solvers are software tools that implement algorithms which can solve this question a decidable subset of first-order formulae. Since we will use SMT solvers extensively, we will dedicate this section to give an overview of SMT solvers and symbolic execution.

2.2.1 SAT — Boolean Satisfiability

Boolean satisfiability (SAT) is the most well-known variation of the satisfiability question. As the term "Boolean" implies, SAT problems are restricted to the satisfiability of propositional formulae. They consist only of Boolean variables and logical conjunction, disjunction, negation, and implication. Depending on the input format accepted by the solver, we may need to rewrite the formula on a specific form, such as Conjunctive Normal Form.

The propositional formula is satisfiable if and only if there exists an assignment to the Boolean variables, called an *interpretation*, such that the formula evaluates to True. Conversely, it is called *unsatisfiable* if no such assignment exists, and *valid* if all possible assignments satisfy the formula. Finding a satisfying interpretation, if it exists, is a highly non-trivial, NP-complete decision problem. This means there exists no algorithm that can solve SAT in polynomial time, and any problem that can be reduced to SAT is consequently also in NP.

SAT solvers are tools that implement highly efficient backtracking algorithms in conjunction with clever heuristics to reduce the complexity of formulae and therefore reduce the number of possible assignments. Most modern SAT solvers are built on the framework provided by the Davis-Putnam-Logemann-Loveland (DPLL) search-based algorithm. DPLL is based principally on branching, Boolean constraint propagation, and backtracking. Grossly simplified, these solvers compute an interpretation in an iterative manner, by deciding an assignment to a variable, propagating forward the assignment and calculating the consequences of the assignment, checking for conflicts, and either backtrack or continue[20].

It is common for a SAT solver to only accept formulae written on Conjunctive Normal Form (CNF). Formulae on CNF must have a specific structure and do not contain logical implication. Given that a formula on the form $p \rightarrow q$ is logically equivalent to $\neg p \vee q$, however, all Boolean formulae can be rewritten to eliminate implication. A formula on CNF must conform to the following rules:

- A Boolean variable is an *atom*.
- An atom or its negation is a *literal*.
- A disjunction of literals is a *clause*.
- A formula on CNF is a conjunction of clauses.

However, formulae passed to SMT solvers must respect rewriting formulae by hand can cause an explosion in variable names and for this reason, there exist tools to translate Boolean propositional formulae into formulae on CNF [27].

2.2.2 SMT — Satisfiability Modulo Theories

Satisfiability Modulo Theories is a generalisation of SAT to the satisfiability of first-order formulae with a background theory. A first-order formula is formed with variables, logical connectives, quantifiers, and function and predicate symbols. A formal theory T is a set of sentences in a formal language. In terms of SMT solving, this means that our formulae can consist of propositional connectives and a set Σ of axioms of the background theory, a set of additional function and predicate symbols that uniquely define the theory and from which we can derive other sentences.

For instance, solving a formulae with the background theory of linear arithmetic, we can include integer numbers, predicates and relations on integers, and the usual arithmetic operators in our formulae, provided that the result of the formulae is still a Boolean value. When allowed to select a background theory, we can investigate the satisfiability of more complex formulae. Different SMT solvers can decide satisfiability for formulae with background theories such as arithmetic, bit-vectors, uninterpreted functions, and arrays, or even a combination of these.

The underlying logic is no longer propositional logic, but a *many-sorted* first-order logic with equality. A *sort* corresponds to a type in a programming language, and a many-sorted logic allows formulae with multiple sorts. By contrast, unsorted logics, such as propositional logic, can only contain variables of *one* sort.

The goal of the SMT solver is to find an assignment to all the variables, function symbols, and predicate symbols such that the formula evaluates to True. Such an assignment is called a *model* of the formula. Formally, we say that given some first-order formula ϕ, an SMT solver attempts to find a model $\mathscr{M}$ that satisfies the formula, written $\mathscr{M} \vDash \phi$ [28].

To solve problems with these background theories, SMT solvers employ SAT solvers under the hood, backtracking and heuristics to arrive produce a result where possible. Modern SMT solvers, such as Microsoft's Z3, are based on an extended, abstract framework building on DPLL [30].

SMT is typically NP-hard, and at best NP-complete, since it reduces to SAT. It is, in general undecidable, since full first-order logic is only semi-decidable. However, there are some theories within first-order logic that are decidable, and some that are decidable by quantifier-elimination.

It is often more ergonomic to formalise problems in terms of SMT, using variables of different sorts, than it is to do so directly in SAT. However, since the query formula must ultimately be reducible to a SAT formula, SMT solvers also perform translation from the first-order input formulae to propositional formulae. SMT solvers can take two distinct approaches to translation. The *eager* approach translates the entire formula to propositional logic up-front and pass it to the underlying SAT solver. The *lazy* approach, on the other hand, translates sub-equations and query solvers with specific knowledge of a particular theory, solving the formula as a whole in an incremental fashion. This approach has the potential of limiting the problem space early on by determining the satisfiability of constraints that appear in the beginning of the formula.

2.2.3 SMT-Lib & SBV

The most common input format to SMT solvers is called SMT-Lib [35]. SMT-Lib adopts a LISP-like syntax and reads like a declarative programming language. It allows the user to define symbolic variables, add logical constraints to them, and prove the satisfiability of formulae. SMT-Lib aims to be a common input format for querying solvers. It makes explicit what might otherwise be implicit in a more specialised query format. Namely, it distinguishes between the underlying logic of the solver, the set of available background theories, and the class of formulae the solver accepts as inputs. However, writing SMT-Lib code requires some experience. For this reason, several programming languages have bindings from near-native language constructs into SMT-Lib.

SBV [14] (SMT-Based Verification) is one such bindings library, for Haskell. It is a domain-specific language that allows programmer to write regular Haskell programs with symbolic types. SBV is an ergonomic choice for specifying formulating formulae from programs, because it provides functions for generating symbolic variables and adding logical constraints to them while maintaining a Haskell-like syntax. These symbolic variables can either be used concretely or be used to prove, check, or optimise the code using an SMT solver. SBV provides support for many different solvers and the user is free to select their own backend solver.

SBV generates SMT-Lib code corresponding to the symbolic Haskell expressions, but besides ergonomics, SBV is easier to use because it performs additional analysis and optimises the generated SMT-Lib code. It ensures static single assignment of variables, meaning each constraint on or assignment to a variable is effectuated by creating a fresh variable name. This work would have been tedious for a human, but is very efficiently computable by a machine. SBV can also perform static checks to reduce the complexity of the formula to be generated or to give early warnings, e.g., controlling for division-by-zero statically.

2.2.4 Symbolic Execution

Symbolic execution is conceptually similar to property-based testing. During normal program execution, we evaluate all the program statements with the concrete program input and interrupt execution on undefined input. This is akin to running a concrete test case. During symbolic execution, however, we can abstract away particular input and instead we use symbolic input variables, which is akin to property-based testing. A symbolic variable has no concrete value, but we can narrow down its possible values by adding constraints to it as we analyse the program. When executing a program symbolically, we investigate the possible paths the program can take and constrain our variables on a per-path basis. If we arrive at a point where the current execution path branches based on the value of a symbolic input, we choose a path, record the condition as a constraint on that symbolic variable and continue. We can repeat this process for divergent branches until we have either covered all possible paths in the program or we time out. This way, we can get an overview of which related inputs take us down which execution paths.

During analysis, we accrue additional constraints over each symbolic variable. If one is used in an operation, its constraints are used to calculate the constraints of the resulting symbolic variable. If one is assigned a concrete value, then it is decided to be a constant. Eventually, for each path, every symbolic variable will either be a constant, a constrained variable, or a free variable. By looking at a given path and the accumulated constraints along it, we might be able to conclude that there is a conflict in those constraints, meaning that branch is unreachable. In other words, that it is logically impossible to create a model

satisfying all the constraints. In terms of property-based testing, if there is no path that leads to the Boolean `false`, then we know that the property holds. If not, then there *does* exist a model that assigns the symbolic variables in such a way that the property fails and from it, we can present a counterexample to the property.

2.3 SBV In Practice

SBV provides several constructs that we will make use of, both when translating the property into a formula and when presenting counterexamples. This section gives a brief introduction to the aspects of SBV that are relevant to our approach.

2.3.1 Symbolic Variables

Internally, potential symbolic variables are represented in SBV by the typeclass `SymVal`, which declares the functions `mkSymVal`, `literal`, `fromCV` and `isConcretely` for defining new symbolic variables of a given type and to extract the concrete value, if any, from them. Of these, the most interest are the two first functions. `mkSymVal` is used indirectly to create symbolic variables of SBV's built-in symbolic types. In other words, it defines the symbolic representation of a given type. `literal`, on the other hand, takes a concrete value and transforms it into a symbolic variable constrained to that particular value. Concrete types that are instances of this typeclass, can be made symbolic.

To work with symbolic values directly, SBV exports the `SBV a` type constructor, which takes a concrete type, and convenient type synonyms for these. For instance, unbounded symbolic integers have the type `SBV Integer` and the type synonym `SInteger`. This naming convention encapsulates some complexity and makes working with symbolic types feel ergonomic and distinctly Haskell-like.

At time of writing, the current version of SBV (version 10.9) supports a wide variety of symbolic types. A few notable instances include symbolic Boolean values, signed and unsigned words, unbounded signed integers, lists, tuples, sums, products, sets, arrays, and uninterpreted functions over symbolic values.

2.3.2 The `Symbolic` Monad

In order to create and track symbolic variables, SBV defines a monad called `Symbolic`. The `Symbolic` monad is responsible for remembering the names of created symbolic variables and all logical constraints on them.

Creating named or unnamed symbolic variables is as simple as using the corresponding constructor or the `literal` function inside the monad. Most of the constructors use the same name as the symbolic type, except with a lowercase "s", e.g., to construct a named symbolic integer of type `SInteger`, we call the constructor `sInteger`. Unnamed symbolic variables use a constructor of the same name, appended with an underscore, e.g., `sInteger_`.

We can then use modified relational and logical operators to query about or impose constraints on the symbolic variables. In the current version of SBV at time of writing, these follow the same naming convention as Haskell but are prepended with a dot, e.g, `.<`.

Using SBV's built-in functions and operations, we can, for instance, create a named symbolic integer variable called "exampleVar" and constrain its value to be strictly greater than 0.

```
example :: Symbolic SInteger
example =
  do sx <- sInteger "exampleVar"
       constrain (sx .< literal 0)
       return sx
```

2.3.3 Proving Formulae with SBV

SBV distinguishes between the notions of *satisfying* and *proving* a formula, and consequently export the two functions `sat` and `prove`. Both their results, `SatResult` and `ThmResults`, are wrappers around a more generic `SMTResult`.

```
sat   :: Satisfiable a => a -> IO SatResult
prove :: Provable    a => a -> IO ThmResult
```

As the name indicates, `sat` attempts to find a model, given the symbolic variables in the formula, such that the model satisfies the formula. `prove`, on the other hand, attempts to prove the *validity* of the formula. Recall that a formula is valid if and only if *all* possible models satisfy the formula. To prove validity, SBV attempts to satisfy the *negation* of the original formula. If it can find a single unsatisfying model, then the formula is not valid.

If we unwrap the result of calling `prove` from the `IO` monad and the `ThmResult` wrapper, we get a `SMTResult`. Two `SMTResult` constructors are particularly interesting to us: `Unsatisfiable` and `Satisfiable`. If the *negation* of the original formula was unsatisfiable, then the formula is valid — and so we know that the property we translated, always holds true. If the negated formula was satisfiable, then the solver provides a model as witness to this proof — this is the counterexample to our property!

By extracting this model from the `ThmResult`, we can print the counterexample to the user. Thanks to the information captured by the `Symbolic` monad, we have even associated the correct variable names with the specific values that constitute the counterexample.

2.4 Partial Evaluation

As we have discussed, modern SMT solvers are incredibly capable software tools. Nonetheless, the time complexity still grows *exponentially* with the number of variables in the formula. SBV does perform optimisation ahead of time, but we might in some cases be able to reduce the complexity of a property *before* we begin translating it into a formula. Partial evaluation is a promising candidate.

Partial evaluation is a technique to optimise a program by *specialisation*. Besides straightforward program optimisation, partial evaluation can be used to generate compilers. An executable program is a specialisation of the programming language's interpreter over a partial input, namely the executable's source code. In other words, a partially interpreted interpreter with respect to a source code is equivalent to a compiling the static parts of the source code. If the partial evaluator is self-applicable, then the partial evaluation of itself with respect to an interpreter yields a compiler [19]. Further, we can employ partial evaluators to implement REPLs (Read-Eval-Print Loops).

Partial evaluation is similar to regular evaluation by an interpreter, with the exception that we allow some of the input to remain undefined. By contrast, if a variable name is not bound in the environment during interpretation of a term, the interpreter will throw an error. The partial evaluator, meanwhile, will simply leave the variable in place and continue evaluation. By using the available arguments and evaluating every part of the program that

become static when we substitute the variable in the term for the actual input, we are often able to reduce the complexity of the program. The result is called a *residual* program. When no further reductions can be performed, we must wait for more input to be supplied before a term can be fully normalised to a value, but the residual program should execute more efficiently than the original.

Given the following function in Haskell.

```
func :: Int -> Int -> Int
func x y = (x + x) + y
```

If we know the first argument to this function is `5`, then we can derive a new function where `5` has been substituted for `x` in the function definition. Since both arguments to the inner application of `+` are now static, we can evaluate that term as well, resulting in the integer `10`. At this point, there is nothing more we can do to further evaluate the function and we stop. The result of the partial evaluation is the specialised function `func'`, which takes only one argument.

```
func' :: Int -> Int
func' y = 10 + y
```

Importantly, partial evaluation should never alter the semantics of a term. In other words, partially evaluating `func x y` applied to `5` and then interpreting func' 3 should yield the same result as interpreting `func 5 3`.

In functional programming languages with first-class functions, such as Haskell, we call this currying. Languages with "first-class" functions allow for functions to return other functions as output, as opposed to only returning values as output. To see the difference more clearly, let us consider an example from [19].

Imagine we wanted to write a function that adds two `Int`s together. In a language without currying, we must write a function plus that accepts all the arguments to a function at the same time. In Haskell, we can represent this behaviour with a tuple of `Int`s. In a language *with* currying, we can write a function `add` that takes one `Int`, and returns a function that takes the second, which finally returns the result. The parentheses are optional in this case, but included to illustrate this point.

```
plus :: (Int, Int) -> int
plus (x, y) = x + y
```

```
add :: Int -> (Int -> Int)
add x y = x + y
```

In languages that feature currying, a multi-argument function is really a nesting of single-argument functions. Each nested function accepts one of the arguments, and returns a function that accepts the next, and so on until the innermost function, which finally returns the result. This enables partial function application, which amounts to partial evaluation, wherein the programmer only provides the first argument and obtains a specialised function.

We distinguish between *online* and *offline* partial evaluation [11]. An online partial evaluator performs partial evaluation dynamically at runtime, and therefore uses the runtime environment accumulated up until the term that is currently being evaluated. By contrast, an offline partial evaluator performs a static analysis of the program text head of time, and based on its analysis determines which variables will be instantiated to a concrete value and which will remain dynamic.

While offline partial evaluators perform an arguably simpler static analysis, an online partial evaluator is more easily derivable from an interpreter than a static one, precisely because both are based on runtime evaluation. When performing online partial evaluation,

however, we must be mindful of certain pitfalls. In particular, it can be more difficult to ensure termination in an online setting. For both approaches, however, we must mindful of recursive function definitions. If we are not careful, we might get stuck unrolling its definition infinitely.

2.4.1 A Small, Online Partial Evaluator

In their paper, Cook & Lämmel present an online partial evaluator for a simple, pure, functional first-order language [11].

The only values in their language are Boolean and integer values. They define the operations `Equal`, `Add`, `Sub`, `Mul` on these values. The programmer can define their own functions, which consist of a function name, a list of variable names, and an expression. Expressions are constants, variables, function application, primitive operations, and if-statements. A program consists of function definitions and a single expression, namely a call to the `main` function.

Cook & Lämmel transform a program by partial evaluation by recursive calls to a function that partially evaluates each expression in the program, starting with the `main` function. They employ memoisation to avoid specialising the same function to the same argument(s) repeatedly, and they handle recursion gracefully. The partial evaluator uses the `State` monad to collect specialised functions, constituting the residual program.

The entry point to the partial evaluator is a function called `peval`, which runs the `State` monad on the result of partially evaluating the `main` function.

The transformation of a single expression is given by the function `peval' :: Expr -> Env -> State [FDef] Expr`. From the function signature of `peval'`, we can see that the partial evaluator needs access to a runtime environment and that the `State` monad is used to store function definitions. The environment `Env` contains binding variable names to values. The function definitions `[FDef]` are ones that have been specialised in the course of partial evaluation.

Most of the partial evaluations are easy to follow. A constant is partially evaluated to itself and a variable is partially evaluated to a value if it is defined in the environment, and left as a variable if not. An operation on expressions can be evaluated fully if all the expressions partially evaluate to concrete values, otherwise we return the operation expression with the partially evaluated expressions. An if-statement can be partially evaluated if its condition partially evaluates to a concrete value, in which case we partially evaluate the corresponding branch, and otherwise we return the if-statement with partially evaluated branches.

The most interesting case is the partial evaluation of function application. First, the definition of the function is looked up in the program, which is available from the argument to the `peval` function via closure. Each argument to the function is partially evaluated and the static and dynamic arguments are partitioned. If there are no dynamic arguments, the function is evaluated directly. However, if there are dynamic arguments, then we can only specialise the function to the static arguments.

They fabricate a new function name from the original name of the function and a hash of the static arguments. This name is unlikely to have been used by the programmer and it uniquely defines the specialisation of the function to those particular arguments. The specialised name is looked up in the `State` monad. If it has not already been specialised, then partial evaluation commences.

However, in order to handle recursive functions gracefully, an additional step is performed before the function is partially evaluated. The fabricated function name is added to the state with a placeholder value. This way, when the function is partially evaluated and arrives at a function call to itself, it will not keep unrolling its own definition, because

it see that the name is bound in the state and simply return what it was able to partially evaluate. After a placeholder has been added to the state, then, the actual body of the partially evaluated function is obtained. Finally, the placeholder value is replaced with the real function body.

Cook & Lämmel note that their implementation does not guarantee termination. In other words, if the user writes a program that never terminates, then neither will the partial evaluator.

2.5 The Reader, Writer, and State Monads

We have already seen some examples of specific monads, namely the `Gen` monad from QuickCheck and the `Symbolic` monad from SBV. We have also seen Cook & Lämmel use the generic `State` monad to keep track of specialised functions in their online partial evaluator. We will be using monads quite extensively in this thesis, and we therefore consider it worthwhile to give a brief introduction to the particular monads we will see repeatedly. These are the `Reader`, `Writer`, and `State` monads. We do not, however, attempt to give an introduction to monads generally. If unfamiliar with the concept, the reader might find the tutorial by Petricek useful [32].

In computer science, we typically use monads to perform stateful computations in a purely functional manner. Pure functions cannot have side effects, such as reading from or writing to a global variable. They are completely defined by their explicit inputs and outputs. As such, a pure function *must* return the same output every time it is called. A function like this is called *referentially transparent*, because a call to a pure function with a particular value can be replaced with its output losslessly. All this means we can a priori not perform stateful computations in a pure language like we would in a language with mutable, global variables.

Such pure functions are easy to reason about, safe to compose, and simple to test. At the same time, many computations are desirable *and* stateful, such as printing to the terminal or updating a runtime environment.

Generally, we can imagine the state of a program at a given point in time as a combination of program state up until the current computation. Each time we perform a stateful computation, we wish to return the result of the computation but also to alter the state by joining the old state with some new resulting state.

In order to accomplish this in a purely functional language, we can embellish the output of a function with result of the computation *and* some update to the state. These functions are still pure because, even though they involve state, they return the *same* state update every time for a given input. Then we can define a function to extract the result from an embellished output, pass the result on to the next function call until we were finished computing, and finally combine the states. Monads are a way for us to abstract away the threading of results into the next computation and the progressive combination of state.

It turns out that some monadic patterns are so common that there exist libraries of generalised, pre-defined monads. Three of the most common patterns are called the `Reader`, the `Writer`, and the `State` monads. These all handle different kinds of "state" and define general operations to combine such states.

The `Reader` monad defines a "state" or *environment* primarily for read-only operations. It is well-suited to track variable bindings in a runtime setting, however, because it allows the user to *modify* the environment for certain computations only. This is done by using the function `local :: (r -> r) -> m a -> m a`, where `(r -> r)` is the function to modify

the `Reader` environment, the former `m a` is the monad to run in the modified environment, and the latter `m a` is the result of the computation once it returns, which has regained its original environment. Using the `Reader` is popular for writing interpreters in part because using `local` simplifies working with local scopes.

The `Writer` monad provides a primarily write-only environment, which is useful for accumulating state from computations, such as log strings or constraints. We can add output to the accumulation by using the function `tell :: w -> m ()`, where `w` is the type being accumulated and the output `m ()` is just the monad with no computation result. Just as for the `Reader`, however, it is not truly a write-only environment, and `Writer` exports functions besides `tell` that allow us read access to the current environment.

The `State` monad provides both read and write access to the environment. This monad is an ideal choice each computation depends explicitly on the current state and require the state to be *updated* by way of deleting and inserting. For instance, Cook & Lämmel require read access to the list of specialised function definitions to decide whether or not to partially evaluate the function body, and if it has not already been specialised, they need to first write and then update the function definition for a newly specialised function. `State` exports the functions `get`, `put`, and `modify`, for reading, writing, and updating state, respectively.

These three monads will feature heavily in Chapter 5. Especially in the form of the monad `RWS` (Reader Writer State) which combines the three monads into a single one. There exists an implementation of the `RWS` monad for Haskell, from the monad transformer library `mtl` [16].

2.6 Unification & Type-Inference

A unification of two terms is a substitution of the free variables such that the two become equal or equivalent. Unification finds many areas of application, such as constraint solving, pattern-matching, type-inference, and query resolution, as in logic programming languages like Prolog [9]. We will use unification directly for pattern-matching, and we will touch on it while implementing a type-inference algorithm for Contra. We therefore take a moment do discuss some key concepts.

A *substitution* θ is a mapping between variable names and terms. One substitution is θ more *general* than another θ' if there exists some other substitution σ such that $\sigma \circ \theta = \theta'$ [33]. The *most general unifier* is the most general substitution that unifies two terms.

We mentioned that unification can be used for type-inference. The Hindley-Milner type system (HM) is a classic type system for the lambda calculus with parametric polymorphism. HM has been used to implement concrete type systems with automatic type-inference in the programming language ML (Meta Language). It has since been adapted to ML derivatives, such as Haskell [26]. It is capable of inferring the most general type of a program without type-hints or -annotations from the programmer, instead relying solely on the program text and the typing rule of each term.

The type-inference algorithm can be implemented by creating a *constraint* set over the terms in a program according to their typing rules, and solving the constraints by a unification procedure. One approach to generating these constraint sets, is to use de Bruijn indices [33].

2.6.1 Type-Inference by de Bruijn Indices

The approach consists in annotating terms with either a concrete type or with a *unification variable* and adding constraints to those type annotations. A unification variable is denoted by a unique index. When each term is annotated, we attempt to resolve the constraints in order to propagate concrete types such that every term is typed by the end. Any constraint conflicts are signalled as type errors.

Either, a term's type judgement rule allows us to determine the its type directly, because it is a simply typed value or because its type is uniquely determined by its sub-terms, or we generate fresh unification variables to stand for those sub-terms we cannot determine the type of at present, and we add constraints to the unification variables according to the type judgement rules.

A constraint in this case states that two types should be equal. The constraint is on the form $[\tau_1 :=: \tau_2]$, where τ_1 and τ_2 are two types that should be equal. We can for instance generate a constraint on the form $[v_i :=: \textbf{Integer}]$ where v_i denotes a unification variable with the index i and we are asserting that, by our type judgement rules, the value annotated with that unification variable, *must* have type **Integer** — otherwise there is a type conflict in the program.

Values can be type-annotated directly with their concrete type according to their typing rules. Patterns and other terms are annotated indirectly using the typing rules.

For instance, in the application of a term to a function, we know that the argument term should have the same type as the function's input pattern. Even if the argument is a compound term whose type we cannot decide, we can add the constraint nonetheless and solve it later when we have determined the type of the argument.

Each type must be either a concrete type or a unification variable, and each index in the set of unification variables must be unique. If, due to the way the terms were annotated, the indices are not unique, we must perform alpha renaming on structure of each type to ensure that any indices contained in them are unique relative to the indices in the rest of the program.

By structure of the type, we mean that some types can be compound and contain other types. For instance, the type of a function abstraction is an arrow from one type to another, $\tau_1 \to \tau_2$. We must therefore recursively perform alpha renaming on the indices.

The unification procedure becomes relevant during the constraint solving. To determine whether or not two types are equal, we must likewise consider the structure of each type. A constraint can for instance require that $[\tau_1 \to v_i :=: v_j]$, in which case v_j must also be an arrow type and its first component must unify against τ_1 and its second component against v_i. Finally, if unification was successful, we obtain a unifier that we can apply to the types to propagate the constraints and resolve the unification variables to concrete types. If allowed, unconstrained type variables can remain polymorphic at this point.

2.7 Related Work

Research in the field of property-based testing has given rise to several programming languages and frameworks. We can distinguish between at least three major approaches to the problem: Narrowing-based, smart enumeration, and SMT-based approaches. We take a moment to consider a handful of these that will serve as inspiration for the design and implementation of Contra.

2.7.1 Narrowing-Based Approaches

Narrowing is a technique to solve sets of equations where the terms in the equations contain variables. If we have *ground* terms (terms without free variables) and a set of rewriting rules, then we can iteratively transform a term by applying the appropriate rewriting rule, reducing the term. If we encounter an unknown variable and cannot continue rewriting, narrowing is a way to determine the value of the variable. Narrowing uses the structure of a term to guess plausible substitutions for the free variables in it and rewriting the term according to the evaluation rules. It tries to substitute the variables in such a way that it can keep rewriting the term, in order to eventually produce a concrete value. If the guess was ineffective at reducing the term, the algorithm backtracks. Narrowing can be characterised by local choices and backtracking, and the specific narrowing strategy determines when and how variables are instantiated.

Luck [23] is a domain-specific language for writing generators using predicates with lightweight annotations. It presents novel language constructs for controlling the distribution of values and influencing the amount of constraint solving that happens before variable instantiations. It draws from both narrowing strategies [2] and constraint solving approaches. Constraint solving is based on analysing the program and accruing logical constraints over each free variable. Based on the variable's appearance in branch conditions and operations, we can build a set of constraints on each variable that limit the possible values it can have. In Luck, Lamproploulos et al. both generate a constraint set based on the semantic rules of the language *and* they allow the user to explicitly instantiate variables, sample sub-generators, or affect the constraint sets by evaluating expressions solely for their side-effects. Luck, however, presents the user with many of the same problems as QuickCheck: It is still domain-specific and has non-conventional syntax.

Lampropoulos et al. also present a framework for generating generators based on *inductive relations*, which are similar to algebraic data types, but represent relations between objects rather than a structured object in itself, in the proof assistant Coq. They also generate correctness proofs for the soundness and completeness of their generators. Similarly to Luck, they employ narrowing to guide the generation of well-formed terms, but they implement an algorithm that can derive the generator entirely from the definition of the inductive relation. They do this by representing sets of potential values as *ranges*, consisting of undefined variables constrained by a type, a fixed value (provided by the user as argument to the generator but inaccessible to the algorithm at derivation-time), an unknown, or a constructor over a range. They show how they can derive well-typed generators, starting with an instantiation of ranges for the inputs to the generator, the output of the generator, and all the variables in the definition of the inductive relation that need instantiation. By unifying the fixed user inputs against the ranges of the other variables, they are able to generate and compose sub-generators for the inductive relation as a whole.

2.7.2 Smart-Enumeration Approaches

In their paper *Feat : Functional Enumeration of Algebraic Types* [13] Duregård et al. present an efficiently computable enumeration of algebraic data types. An enumeration of a set A is a bijection between the elements in A and the natural numbers. Informally, it a way of assigning a unique index to every element in a set. By robust, mathematical specification, they define an enumeration of the set of possible values for a given algebraic data type. They even demonstrate how to define enumerations for mutually recursive data types with their approach. Equipped with well-defined enumeration, the user is able to query the enumeration for the value at a given index, even up index 10^{100}, and receive the calculated

value promptly. As implemented in the paper, however, their approach does not provide control over the values of field types, such as integers or Boolean values, to the constructors. For large indices, Feat outperforms both QuickCheck and Smallcheck [36], which checks properties for values up to some recursion depth and increases the depth progressively, rather than generating random values from the start as QuickCheck does.

2.7.3 SMT-Based Approaches

SMT solvers have a wide range of applications, and we are not the first to suggest their use in property-based testing. TARGET (Type Targeted Testing) [37] is a Haskell testing tool using SMT solver and *refinement types* by Seidel et al. A refinement type is a term type enriched with logical predicates, which narrow the possible *values* of the refinement type. The combination of types and predicates is clearly well-suited to specify property-based tests, which require well-typed terms, but in some cases also *valid* terms, e.g., integers strictly greater than 5 or lists with a length less than 10 elements. TARGET finds counterexamples through an iterative three-stage process. The arguments are translated into logical queries, which are solved by an SMT solver. Next, concrete values are extracted from the model. Finally, TARGET checks the property with the generated inputs. If this results in a counterexample, the counterexample is reported, and if not, the SMT solver is queried for *different* inputs. This way, validity constraints on the input values are accounted for by being encoded in the refinement type and solved for by the SMT solver. Seidel et al. demonstrate the time-efficiency of this approach at large input sizes, where QuickCheck's performance begins to suffer significantly.

Chapter 3

Analysis

Under the assumption that property-based testing would be more widespread if it were more accessible, we can level some amount of criticism at the existing work we have seen in Section 2.7. While these approaches are more capable of finding interesting counterexamples than we expect to be, we contend that they have one of two problems: Either, they present a steep learning curve to new users and risk alienating users who might otherwise benefit from the use of property-based testing. Or alternatively, they do not support user-defined algebraic data types. In this chapter, we set forth a number of criteria for Contra and we discuss the selection of methods based off of these requirements.

3.1 Small Core & Specialised Constructs

With the aim of making Contra easy to adopt and understand for the average functional programmer, we choose a standard, ML-style syntax for Contra. As Haskell is a well-known member of the ML-family and relatively familiar language to most functional programmers, we consider it sensible to model Contra's syntax closely after that of Haskell. In order to make the learning curve as gentle as possible, Contra should have no exotic syntax or non-standard language constructs.

The notable exception to the standard syntax, is the inclusion of "property" as a base construct in the language. We make property definitions a core program statement, on par with function definitions. Defining a property should be as simple as writing a function that returns a Boolean value. Moreover, properties should behave as regular functions when they are not actively being checked, meaning they can be composed and returned from other functions. In other words, properties should be a first-class citizen of the core language.

To make programmatic reasoning easier and to make the final translation algorithm clearer, we opt for a small core language. While we keep the number of built-in statements to a minimum, Contra should still be expressive enough for the user to define syntactic sugar for most of the constructs they may be missing from other similar languages. We can for instance leave out logical connectives, which can be easily defined by a two-argument function using pattern-matching to return the appropriate result.

A small core language also implies a minimal amount of types. We make do with the foundational types of functional programming languages, but because an explicit goal of the language is to find counterexamples for algebraic data types, these must naturally feature.

We must require that every constructor in the definition of an algebraic data type is unique on the top-level, meaning a single constructor cannot feature twice in a definition. Moreover, for each algebraic data type definition, the set of data type constructors in the definition, must be *disjoint* from all other such sets in the program text, meaning that a data

type constructor can feature in the top-level of *at most* one algebraic data type definition. These requirements follow from the formal definition of an algebraic data type. Moreover, though, they are necessary in order to determine the type of a given constructor and the types of its fields. These requirements mean that the following two algebraic data type definitions in Haskell are invalid. The first because the data type constructor `Ctr1` appears twice on the top-level of the definition. The second because the constructor `Ctr2` appears in the definition of both `ADT1` and `ADT2`.

```
data ADT1 = Ctr1 Int | Ctr1 Bool | Ctr2 Int

data ADT2 = Ctr2 String
```

3.2 Counterexamples by SMT Solving

As we have seen, producing a counterexample to a user-defined algebraic data type is clearly non-trivial. This is especially true for mutually recursive data types. The most intuitive approach for a human to finding a value for a variable of an algebraic data type, effectively boils down to a backtracking approach: Consider the first constraint that appears on the variable, and select one of the data type constructors satisfy the constraint. Move on the next constraint and either discard the candidate or continue, possibly by instantiating some field(s) of the constructor. The absolutely most efficient algorithms we have for solving this class of problems, is SMT solvers.

By using SMT solvers, we are pursuing the goal of finding *counterexamples* explicitly. Traditional property-based testing tools generate random input and can therefore report a wide variety of counterexamples, effectively stress-testing the program. We, on the other hand, simplify the process to target input that makes a property *fail*. This we do by employing an SMT solver to find a model that satisfies the negation of the formula-representation of the property.

This approach affects the user experience, because the SMT solver will attempt to find the simplest possible solution. For the programmer, it is generally easier to identify the bug from a series of counterexamples whose common factors hint at the underlying issue than from a single example. However, randomised property-based testing often reports several successful tests as well, and we may have to run a large number of tests before the counterexample is found. The perfect tool would always find all counterexamples and therefore be *complete*, but as we have seen, this is generally undecidable. If there is an error in the program, counterexamples are the only truly interesting test result. Using an SMT solver, we *will* find the counterexample on the first test run if the tool is able to do so at all, and the only results we report are a *single* counterexample, a *proof* that the property holds, or an unknown result from the solver timing out.

In order to derive a formula that is logically equivalent to the property we wish to check, we must implement some translation algorithm. The translation must preserve all the logical constraints on each input variable represented by the original property, such that they are logically equivalent. Otherwise, our property-checker would be *unsound*. Again, SBV marks itself as a robust choice, because the creation and constraining of symbolic variables of basic types is so ergonomic.

3.3 Inlining and Reduction by Partial Evaluation

Considering the potential complexity of the generated formulae, we can perform some optimisation steps on our end before we even begin translation via SBV. The vast majority

of the properties we are interested in checking will make a call to some function defined elsewhere in the program text. Sometimes, however, the function will be used on some concrete arguments in the property and can therefore be specialised *before* the translation step.

As long as the properties do not call any *named, recursive* functions from the program text, we can specialise and inline functions in the definition of the property. In cases where the user specifies the first arguments to a function, we can specialise the function. We might even be able to eliminate unreachable execution branches statically, or translate the term to a property that is trivially satisfiable or unsatisfiable. If there is only one argument lacking for a given function call, we can inline the definition completely. By implementing a partial evaluator to inline and specialise function calls, we can therefore expect a small performance gain and occasionally simpler formulae.

As online partial evaluation can be derived with relative ease from an existing interpreter, this is the style of partial evaluator we choose for Contra. Having made this choice, we can already outline some of the requirements of the partial evaluator.

In order for the partial evaluator to know about the defined, top-level variables in the program text, it must have access to the program text as a runtime environment. To make function specialisation possible, it must have a separate state to keep track of the specialised functions and the untouched functions, the sum of which yields the residual program. It must handle recursive function calls gracefully. Finally, it must perform a simple static check to eliminate unreachable execution branches.

3.4 Online Translation

When the user asks to have a Contra program checked, all the properties should be checked at once and the results reported back to the user. In order to reap the potential benefits of the partial evaluation, we partially evaluate each property with respect to the program text and collect the residual program for each property checked. Only functions used in a property will be specialised in this approach. We then proceed to translate the partially evaluated property with respect to the residual program text. Finally, we can use SBV to query the backend SMT solver for a solution, and move on to the next property.

When translating a single property, we must first generate an SBV formula corresponding to the term. Because SBV handles static single assignments and performs optimisations on our behalf, we are not preoccupied with the explosion of variable names or duplicate code. We can take a similar approach to translation as we did to partial evaluation, namely what we can characterise as an "online translation". By this, we mean we can translate a property by translating each sub-term recursively in a runtime environment. As long as the partial evaluator performs its function correctly, we should not risk introducing any redundant execution paths this way.

In the case of a partial evaluator for a functional language, the runtime environment must include the program's function definitions for lookup and it must enable the partial evaluator to save specialised functions. The translator, by comparison, must have access to the program text for two purposes. First, to retrieve the definition of any function that was not inlined as part of the partial evaluation, which would happen if the function had more than one unknown argument or if the function was a named recursive function. Second, it must have knowledge of the algebraic data type definitions in the program text.

Further, the runtime environment used by the translator must interface with SBV in order to create fresh symbolic variables. However, we only need to create fresh symbolic variables

for the *inputs* to the property. The rest should exist as function definitions in the program text and are otherwise undefined.

Fortunately, SBV comes equipped with functions for creating fresh symbolic variables for most of the types we are interested in. But, of course, not algebraic data types. Therefore, we require an *encoding* of algebraic data types as symbolic variables of a type supported by SBV. This is arguably the greatest challenge to our endeavour.

3.5 Encoding Algebraic Data Types

With a small core language with few types, the translation of regular types is simple using SMT, since we are allowed to use variables of different types in the same formula. However, an algebraic data type cannot be straight-forwardly embedded in the formula. We need the SMT solver to somehow select a constructor, instantiate its fields, and collect constraints on both the constructor and its fields based on the rest of the formula. Further, it needs to know when to backtrack and discard a candidate.

We need to use (a combination of) the symbolic types that are supported by SBV, and in turn the SMT solver, to represent an arbitrary algebraic data type. The first challenge is the selection of a constructor. Inspired by Duregård et al. [13], we investigate the possibility of enumerating our algebraic data types.

We know that an algebraic data type definition is equivalent to a tagged disjoint union. The set of constructors in an algebraic data type definition is both countable and finite. Since it is both countable and finite, we can define a bijection between the set of possible constructors for a given algebraic data type and the natural numbers. In other words, we can *enumerate* the constructors of an algebraic data type. By including the name of the algebraic data type on either side, we can enumerate each of the type's constructors and label them with an integer number. We can notice that the algebraic data type name is not a necessary component of the bijection, since we do not allow multiple definitions of the same constructor name, but this will aid our intuition later as it improves the readability of our code.

$$(\mathscr{D}, \mathscr{C}) \longleftrightarrow (\mathscr{D}, \mathbb{N})$$

This bijection allows us to represent the choice of a constructor with a symbolic integer. When we want to create a symbolic variable representing of an algebraic data type, we can create a symbolic integer and constrain it to one of the valid enumerators for that algebraic data type, yielding a *selector* for one of the type's constructors.

Let the direct image function of the bijection (the bijection applied in the right direction) be the following function.

$$\text{selector} : (\mathscr{D}, \mathscr{C}) \to (\mathscr{D}, \mathbb{N})$$

Let the pre-image function of the bijection (the bijection applied in the left direction) be the following function.

$$\text{reconstruct} : (\mathscr{D}, \mathbb{N}) \to (\mathscr{C}, \mathscr{D})$$

Let the *cardinality* of an algebraic data type ADT denote the number of top-level constructors it has, written $|ADT|$.

Then, each algebraic data type in the program text can be represented as a tagged disjoint union, such that the set of all the sets of algebraic data types is disjoint.

$$
\begin{aligned}
ADT_1 &:= Ctr_{1,1}\,[\tau_{1,i1}] \mid Ctr_{1,2}\,[\tau_{1,j1}] \mid \ldots \mid Ctr_{1,n1}\,[\tau_{1,m1}] \\
ADT_2 &:= Ctr_{2,1}\,[\tau_{2,i2}] \mid Ctr_{2,2}\,[\tau_{2,j2}] \mid \ldots \mid Ctr_{2,n2}\,[\tau_{2,m2}] \\
&\cdots \\
ADT_k &:= Ctr_{k,1}\,[\tau_{k,ik}] \mid Ctr_{k,2}\,[\tau_{k,jk}] \mid \ldots \mid Ctr_{k,nk}\,[\tau_{l,mk}]
\end{aligned}
$$

$$\cong$$

$$
\begin{aligned}
ADT_1 &= \bigsqcup_{n \in |ADT_1|} \{(Ctr_{1,n}, [\tau_{1,m}])\} \\
ADT_2 &= \bigsqcup_{n \in |ADT_2|} \{(Ctr_{2,n}, [\tau_{2,m}])\} \\
&\cdots \\
ADT_k &= \bigsqcup_{n \in |ADT_k|} \{(Ctr_{k,n}, [\tau_{k,m}])\}
\end{aligned}
$$

such that

$$\bigsqcup_{0 < a \leq k} ADT_a$$

Since each constructor $Ctr_{a,b}$ is unique in the program text, we can losslessly encode it by its index a, b. For an algebraic data type ADT_a, each of its constructors is given by $Ctr_{a,b}$. Because of this, we can exchange a for the name of the algebraic data type ADT_a, which must also be unique on the top-level of the program text.

An instantiation of an algebraic data type is thus given by the name of the algebraic data type, the index of the constructor with respect to this algebraic data type name, and the list of constructor fields it has.

The next step is to encode the instantiation of its field types as symbolic variables. However, at the point in the translation where we create the selector, we do not yet know which constructor will be selected, and therefore we do not know how what fields it should have or which types they have. Therefore, we cannot simply create symbolic variables for the fields and be done. In the Chapter 5, we will see how we can keep translating without yet knowing the fields we should instantiate.

Chapter 4

Design

The design of the core Contra language is based on the FUN programming language[38]. In the first part of this chapter, we present the abstract syntax, typing rules, and operational semantics of the language. In the second part, we present a sketch of the symbolic translation of Contra terms into semantically equivalent constrained symbolic variables.

4.1 Abstract Syntax

The abstract syntax of Contra is given in Figure 4.1. The term $\overline{n}$ denotes an integer literal. The names x, f, and q ranges over an infinite set of immutable *variables*, denoting a variable name, a function name, and a property name, respectively. $\mathscr{C}$ denotes well-formed, defined data constructor names and $\mathscr{T}$ well-formed, defined data type names. Note square brackets denote a possibly empty sequence of entities, whereas a symbol followed by a subscript letter *without* surrounding square brackets denotes a sequence of entities where $i \geq 1$.

4.1.1 Terms

The syntax of Contra deliberately resembles that of a garden-variety functional programming language. A Contra term t is a pattern p, a value v, or a compound term.

Compound terms include standard lambda abstraction (function definition), function application, **let**- and **case**-statements, as well as simple arithmetic, relational, and logical operations. Additionally, a term can be a data constructor applied to the appropriate number (possibly zero) of well-typed terms. A pattern p is a subset of terms that can be pattern-matched by a unification algorithm. A pattern is either a value v or a variable, defined by a well-formed and bound variable name x. A value v is a literal term. In Contra, literal terms are unit, an integer, or a Boolean value. A term is *canonical* if is it a pattern with no free variables or it is a lambda term.

4.1.2 Program Statements

A Contra program consists of function declarations, property declarations, optional function/property type signature declarations, and data type declarations followed by a special end statement, ϵ, marking the end of the program text.

A function declaration binds the function name as a variable and maps it to the function definition, which is a lambda abstraction. Recall that a lambda abstraction is an anonymous function. When a defined function is applied to an argument, the function name variable is looked up in the runtime environment to retrieve the function definition, and if the

$$
\begin{array}{rr}
x \in \textbf{Name} & \text{(Well-formed variable names)} \\
f \in \textbf{Name} & \text{(Well-formed function names)} \\
q \in \textbf{Name} & \text{(Well-formed property names)} \\
\mathscr{C} \in \textbf{Name} & \text{(Well-formed data constructor names)} \\
\mathscr{D} \in \textbf{Name} & \text{(Well-formed data type names)}
\end{array}
$$

$$
\begin{array}{rr}
v ::= \textbf{Unit} & \text{(Unit)} \\
\quad \mid \overline{n} & \text{(Integer numbers)} \\
\quad \mid \textbf{True} \mid \textbf{False} & \text{(Boolean values)} \\
\quad \mid \mathscr{C}\ [v_i] & \text{(Constructor)} \\
p ::= v & \text{(Value)} \\
\quad \mid x & \text{(Variable)} \\
\quad \mid \mathscr{C}\ [p_i] & \text{(Constructor)} \\
t ::= p & \text{(Pattern)} \\
\quad \mid \lambda p.\ t & \text{(Lambda abstraction)} \\
\quad \mid f\ \ t & \text{(Function application)} \\
\quad \mid \textbf{let}\ p = t_1\ \textbf{in}\ t_2 & \text{(Let statement)} \\
\quad \mid \textbf{case}\ t_0\ \textbf{of}\ ;\ p_i \rightarrow t_i & \text{(Case statement)} \\
\quad \mid \mathscr{C}\ [t_i] & \text{(Constructor)} \\
\quad \mid t_0 + t_1 & \text{(Addition)} \\
\quad \mid t_0 - t_1 & \text{(Subtraction)} \\
\quad \mid t_0 < t_1 & \text{(Less than)} \\
\quad \mid t_0 > t_1 & \text{(Greater than)} \\
\quad \mid t_0 = t_1 & \text{(Equal)} \\
\quad \mid \textbf{not}\ t & \text{(Logical negation)}
\end{array}
$$

$$
\begin{array}{rr}
\Delta ::= x\ ::\ \tau.\ \Delta & \text{(Function or property signature)} \\
\quad \mid f\ [x_i] = t.\ \Delta & \text{(Function definition)} \\
\quad \mid q\ [x_i] =\!*\!= t.\ \Delta & \text{(Property definition)} \\
\quad \mid \textbf{adt}\ \mathscr{D} = (\mathscr{C}\ [\tau_i])_j.\ \Delta & \text{(ADT definition)} \\
\quad \mid \epsilon &
\end{array}
$$

Figure 4.1: The abstract syntax of Contra.

argument is well-typed and matches on the input pattern, the argument is substituted for the input pattern in the function definition body and evaluated. Function application can be written without surrounding the argument in parentheses, except when needed for syntactic disambiguation. Function definitions with the same name and type can be desugared into a single function definition with a case-statement matching on the input of each definition, provided the pattern-matches are exhaustive.

A property declaration is a special function function, with the restriction that its return type *must* be a Boolean value. To visually distinguish property declarations from function declarations, the name of the property is followed by the sign `=*=` rather than the usual `=`. It was chosen because it visually suggests multiplication and equality, which in turn hints at the idea that a property represents the truth of a statement over multiple inputs. Should a property name be used in a function during normal program execution, the property is treated as a regular function.

The programmer may provide optional type signatures to their function and property declarations. The name of the signature must match the function or property name. The type will typically be a (possibly nested) arrow type, except in the case of constants. Where there is a discrepancy between the intended type of a function or property (as given by the signature) and the inferred type of the function or property, the type-checker can alert the programmer at the point of the definition rather than at the point of mismatch in application somewhere else in the program.

Contra allows the user to define custom algebraic data types. These types may not, in the current implementation, be parameterised or generalised. An algebraic data type definition consists of a well-formed data type name $\mathscr{T}$ followed by a data type constructor **Ctr**. The constructor, in turn, consists of a well-formed data constructor name $\mathscr{C}$ and a (possibly empty) list of types. These types may include the algebraic data type itself, such that algebraic data type definitions may be recursive.

4.1.3 Disambiguation of Syntactic Sugar

The syntax of Contra is deliberately limited, in order to facilitate reasoning about programs and translation of terms into formulae. In some cases, however, we can use these base terms to represent syntactic constructs from modern functional programming languages. In an attempt to make the language more approachable, we therefore provide syntactic sugar. The disambiguation of such syntactic sugar is given in Figure 4.2.

Though they are not present in the abstract syntax of the language, the user may write `if`-statements in the conventional way. These are recognised by the parser and simply desugared into a `case`-statement, where the condition of the `if`-statement is the selector term, and a pattern-match against a hard-coded `True` pattern selects the `then`-branch and a pattern-match against a hard-coded `False` pattern selects the `else`-branch. The intended semantics of an `if`-statement are clearly preserved and the patterns are exhaustive by definition, since the selector must evaluate to a Boolean value.

All function definitions are desugared into a nested anonymous function. That is, functions defined by a function name followed input arguments on the left-hand side of the `=` operator are desugared into a function definition containing only the function name on the left-hand side of the equation and each input argument is instead represented as the argument to a lambda term.

In the interest of maintaining close syntactic resemblance to modern programming languages like Haskell and others, the user may define multiple versions of a function by pattern-matching on the input. The different function definitions are consolidated by desugaring them into a single function definition with a `case`-statement. New variable

$$[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!] ::= [\![\textbf{case } t_0 \textbf{ of } ; \textbf{ True} \to t_1 \ ; \textbf{ False} \to t_2 \ ]\!]$$
$$[\![\langle function\rangle \ p_1 \ p_2 \ ... \ p_i = t]\!] ::= [\![\langle function\rangle = (\lambda p_1. \ (\lambda p_2. \ ... \ (\lambda p_i. \ t)))]\!]$$

$$\left[\!\!\left[\begin{array}{l} \langle function\rangle \ p_{1,1} \ p_{1,2} \ \ldots = t_1 \\ \langle function\rangle \ p_{2,1} \ p_{2,2} \ \ldots = t_2 \\ \qquad\qquad\qquad\qquad \ldots \\ \langle function\rangle \ p_{n,1} \ p_{n,2} \ \ldots = t_n \end{array}\right]\!\!\right] ::= \left[\!\!\left[\begin{array}{r} \langle function\rangle = (\lambda^*\text{a}. \ (\lambda^*\text{b}. \ \ldots \\ (\textbf{case } [*a, *b, \ldots] \textbf{ of} \\ ; \ [p_{1,1}, p_{1,2}, \ldots] \to t_1 \\ ; \ [p_{2,1}, p_{2,2}, \ldots] \to t_2 \\ \qquad\qquad\qquad \ldots \\ ; \ [p_{n,1}, p_{n,2}, \ldots] \to t_n \\ \qquad\qquad\qquad)\ldots)) \end{array}\right]\!\!\right]$$

Figure 4.2: Disambiguation of syntactic sugar in Contra. Angled brackets denote placeholder text for a given function name.

$$\begin{aligned} \text{free}(v) &= \emptyset \\ \text{free}(x) &= \{x\} \\ \text{free}(\mathscr{C}[p_i]) &= \bigcup_i \text{free}(p_i) \end{aligned}$$

Figure 4.3: Definition of the function free.

names are fabricated with the otherwise illegal prefix $*$ and pattern-match on an internal list pattern. The list representation is not exposed to the user. The original pattern-match inputs are likewise represented as lists and together with their corresponding function body, they represent a distinct case branch. Applying the function binds the input patterns to the fabricated $*$-variables, which can then pattern-match on the list alternatives, selecting the correct branch. This way, the programmer can write function definitions in a more ergonomic style, while we maintain a small core language under the hood.

4.2 Type Grammar & Typing Rules

The type grammar of Contra and its corresponding typing rules are given in Figure 4.4.

Each type judgement rule $\Delta\Gamma \vdash t : \tau$ is a proof that from the program text Δ and the typing environment Γ, the term t has the type τ. The typing environment Γ is a mapping from variable names to types. Because Contra does not feature parameterised algebraic data types, we know statically from the program text Δ what types the patterns, and thus any free variables, in a data type constructor must have.

The function free returns the free variable names in a pattern and is straight-forwardly defined by induction on the syntax of the pattern. Its definition is given in Figure 4.3.

4.2.1 Type Grammar

The types in Contra are either simple or compound.

The simple types are **()**, **Integer**, **Boolean**, and algebraic data types. The **()** type is a special type with the standard interpretation: It is inhabited by only one term, namely **unit**,

which holds no further information. **Integer** and **Boolean** are interpreted in the usual way. Algebraic data types are denoted simply by a well-formed data type name and refer to an algebraic data type declaration in the program text.

The only compound types in Contra are the arrow types. Arrow types denote the types of lambda abstractions (functions) and describe the input type(s) and output type of these.

4.2.2 Conceding Polymorphism

The algebraic data types in Contra can be recursive, but in the interest of simplicity, they do not support polymorphic type variables. This is a trade-off, as the user must manually define different instantiations of their algebraic data types, if they wish to test several variations using the same type constructors. On the other hand, it greatly simplifies the implementation of the translation algorithm. For instance, the following parametric data type definition in legal in Haskell.

```
data List a = Nil | Cons a (List a)
```

In Contra, the user must decide on a concrete type instead of the type variable `a`. For instance, they can create an `IntList`.

```
adt IntList = Nil | Cons Integer IntList .
```

Because it is not the focus of this thesis, we simplify matters further by disallowing general polymorphism. With this restriction, non-polymorphic algebraic data-types allows us a completely deterministic type-checking procedure. Contra can then feature a simple type-inference algorithm that can determine the type of any non-ambiguously typed term and throw errors on type-mismatches, even for algebraic data types.

4.3 Operational Semantics

The operational semantics of Contra are given in Figure 4.5. We provide the big-step, call-by-value semantics of Contra and omit the small-step semantics, on the basis that these are completely conventional.

In order to evaluate a term t, it must be *closed*. A term is closed if and only if it contains no free variables. In particular, we note that a lambda term $\lambda p.t$ is closed if and only if, at time of evaluation, the free variables in p *only* occur as free variables in t. Likewise, a let-bound term $\textbf{let}\ p = t_1\ \textbf{in}\ t_2$ is closed if and only if, at time of evaluation, the free variables in p do *not* occur as free variables in t_1 and that they *only* occur as free variables in t_2. This requirement is necessary to avoid shadowing of variable names. Other variables that occur in the body of a lambda or a **let**-statement are only free if they do *not* occur in the scope of the body. In particular, variables are not considered free if they exist as top-level function or property definitions in the program text.

The function unify takes a pattern p and a term t, and returns a unifier. The unifier is a mapping θ from the free variables in the pattern p to sub-terms in t such that applying the mapping θ to the pattern p makes it equal to t. We write the application of θ to p as $\theta(p)$. The function unify' is equivalent to unify, except that it accepts two patterns p_0 and p_1. The implementation of the unification algorithm is described in detail in Section 5.4.1.

Each semantic judgement rule $\Delta \vdash t \downarrow s$ is a proof that in the program text Δ, the closed term t evaluates to the closed term s.

$$\tau ::= \ () \mid \textbf{Integer} \mid \textbf{Boolean} \mid (\tau_1 \to \tau_2) \mid \mathscr{D}$$

$$\tau\text{-Unit} : \frac{}{\Delta\Gamma \vdash \textbf{Unit} : ()} \qquad \tau\text{-Number} : \frac{}{\Delta\Gamma \vdash \overline{n} : \textbf{Integer}}$$

$$\tau\text{-True} : \frac{}{\Delta\Gamma \vdash \textbf{True} : \textbf{Boolean}} \qquad \tau\text{-False} : \frac{}{\Delta\Gamma \vdash \textbf{False} : \textbf{Boolean}}$$

$$\tau\text{-Variable} : \frac{}{\Delta\Gamma \vdash x : \tau}(\Gamma(x) = \tau)$$

$$\tau\text{-Constructor} : \frac{\Delta\Gamma \vdash t_i : \tau_i}{\Delta[\textbf{adt}\ \mathscr{D} = (\mathscr{C}\ [\tau_i])_j]\Gamma \vdash \mathscr{C}\ [t_i]\ : \mathscr{D}}$$

$$\tau\text{-Lambda} : \frac{\Delta\Gamma[x_i \mapsto \tau_i] \vdash p : \tau_1 \quad \Delta\Gamma[x_i \mapsto \tau_i] \vdash t : \tau_2}{\Delta\Gamma \vdash \lambda p.t : (\tau_1 \mapsto \tau_2)}(x_i \in \text{free}(p))$$

$$\tau\text{-Application} : \frac{\Delta\Gamma \vdash t_1 : (\tau_1 \to \tau_2) \quad \Delta\Gamma \vdash t_2 : \tau_1}{\Delta\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\tau\text{-Let} : \frac{\Delta\Gamma \vdash t_1 : \tau_1 \quad \Delta\Gamma[x_i \mapsto \tau_i] \vdash p : \tau_1 \quad \Delta\Gamma[x_i \mapsto \tau_i] \vdash t_1 : \tau_2}{\Delta\Gamma \vdash \textbf{let}\ p = t_1\ \textbf{in}\ t_2 : \tau_2}(x_i \in \text{free}(p))$$

$$\tau\text{-Case} : \frac{\Delta\Gamma \vdash t_0 : \tau_1 \quad \Delta\Gamma[x_i \mapsto \tau_i] \vdash p_i : \tau_1 \quad \Delta\Gamma[x_i \mapsto \tau_i] \vdash t_i : \tau_2}{\Delta\Gamma \vdash \textbf{case}\ t_0\ \textbf{of}\ \ ;\ p_i \to t_i\ \ : \tau_2}(x_i \in \text{free}(p))$$

$$\tau\text{-Plus} : \frac{\Delta\Gamma \vdash t_0 : \textbf{Integer} \quad \Delta\Gamma \vdash t_1 : \textbf{Integer}}{\Delta\Gamma \vdash t_0 + t_1 : \textbf{Integer}}$$

$$\tau\text{-Minus} : \frac{\Delta\Gamma \vdash t_0 : \textbf{Integer} \quad \Delta\Gamma \vdash t_1 : \textbf{Integer}}{\Delta\Gamma \vdash t_0 - t_1 : \textbf{Integer}}$$

$$\tau\text{-Lt} : \frac{\Delta\Gamma \vdash t_0 : \textbf{Integer} \quad \Delta\Gamma \vdash t_1 : \textbf{Integer}}{\Delta\Gamma \vdash t_0 < t_1 : \textbf{Boolean}}$$

$$\tau\text{-Gt} : \frac{\Delta\Gamma \vdash t_0 : \textbf{Integer} \quad \Delta\Gamma \vdash t_1 : \textbf{Integer}}{\Delta\Gamma \vdash t_0 > t_1 : \textbf{Boolean}}$$

$$\tau\text{-Equal} : \frac{\Delta\Gamma \vdash t_0 : \tau \quad \Delta\Gamma \vdash t_1 : \tau}{\Delta\Gamma \vdash t_0 = t_1 : \textbf{Boolean}} \qquad \tau\text{-Not} : \frac{\Delta\Gamma \vdash t_0 : \textbf{Boolean}}{\Delta\Gamma \vdash \textbf{not}\ t_0 : \textbf{Boolean}}$$

Figure 4.4: Type grammar and type judgement rules of Contra.

While all Contra terms have the usual semantics, we will briefly cover the semantics of the compound terms.

A lambda abstraction is an anonymous function defined by an input pattern p and a function body t_0. The application of a term t_1 to a well-typed argument term t_2 is evaluated by first evaluating t_1 to a lambda abstraction $\lambda p.t_0$ and evaluating the argument term t_2 to s_2. We proceed by unifying the evaluated argument term s_2 against the input pattern p of the lambda and finally, we evaluate the result of applying the unifier to the function body t_0. In the case where the input pattern is a simple variable x, this amounts to substituting all occurrences of x with s_2 in the function body t_0.

A **let**-statement binds a a pattern p to a term t_1 in the body of a term t_2. To evaluate a **let**-statement, we first evaluate the term t_1 to s_1. Then we unify the pattern p against s_2, resulting in a unifier. Finally, we evaluate the result of applying the unifier to the term t_2. In the case where the input pattern is a simple variable x, this similarly amounts to substituting all occurrences of x with s_1 in the term t_1.

The **case**-statement consists of a term t_0 and a list of pattern-term pairs (p_i, t_i) representing branches. First, we evaluate the term t_0 to a pattern p_0. Next, we attempt to unify the selector pattern p_0 against the patterns p_i. If a match is found, this results in a unifier, and we evaluate the statement as a whole by evaluating the result of applying the unifier to the corresponding term t_i. If no match is found, an error is raised. We allow for the selector to be a term, provided that it evaluates to a pattern. Thus, the term `2 + 4` is a valid selector term, since it evaluates to the pattern `5` and therefore can be unified against other patterns.

4.3.1 Equality of Terms & Types

In order to evaluate ↓ —Equal-True/False, we must define an equality relation on terms, which in turn requires an equality relation on types. We define the equality of terms by induction of the syntax of the term.

The equality of types is straight-forward and is given by the following rules.

- The types **()**, **Integer**, and **Boolean** are only equal to themselves.
- A data constructor $\mathscr{D}$ is only equal to itself.
- An arrow type $(\tau_1 \to \tau_2)$ is equal another arrow type $(\tau_3 \to \tau_4)$ if an only of τ_1 is equal to τ_3 and τ_2 is equal to τ_4.
- No other types are equal.

The equality of values are similarly simple and is given by the following rules.

- **unit** is equal to itself.
- An integer is equal to another integer if they have the same value.
- A Boolean value is equal to another Boolean value if they have the same value.
- A constructor value $\mathscr{C}$ with values $[v_i]$ is equal to another constructor value precisely if they have the same name and the same fields v_i.
- No other values are equal.

The equality of patterns is given by the following rules.

- A pattern is equal to another pattern if they are both values and the values are equal.
- A variable is equal to another variable if they have the same name.

- A constructor pattern $\mathscr{C}$ with patterns $[p_i]$ is equal to another constructor pattern precisely if they have the same name and the same fields p_i.
- No other patterns are equal.

The equality of terms is given by the following rules.

- A term is equal to another term if they are both patterns and the patterns are equal.
- An arithmetic term is equal to another arithmetic term if their operands are pair-wise equal.
- A relational term is equal to another relational term when their operands are pair-wise equal.
- A negation term is equal to another negation term if the terms they negate are equal.
- A lambda term $\lambda p_1.\ t_1$ is equal to another lambda term $\lambda p_2.\ t_2$ if p_1 is equal to p_2 and t_1 is equal to t_2.
- A function application $t_1 t_2$ is equal to another function application $t_3 t_4$ if t_1 is equal to t_3 and t_2 is equal to t_4.
- A **let**-statement **let** $p_1 = t_1$ **in** t_2 is equal to another let-statement **let** $p_2 = t_3$ **in** t_4 if the patterns p_1 and p_2 are equal, t_1 is equal to t_3 and t_2 is equal to t_4.
- A **case**-statement **case** t_1 **of** ; $p_i \rightarrow t_i$ is equal to another case-statement **case** t_2 **of** ; $p_j \rightarrow t_j$ if t_1 is equal to t_2, all the patterns p_i are equal to all the corresponding patterns p_j, and all the terms t_i are equal to to the corresponding terms t_j.
- A constructor term $\mathscr{C}$ with term $[t_i]$ is equal to another constructor term precisely if they have the same name and the same fields t_i.
- No other terms are equal.

4.4 Translation Sketch

To give a feeling of what steps are involved in the process of generating counterexamples, we sketch the steps from the algorithm takes from start to finish. We propose the following main functions.

- `checkProperty :: Program Type -> PropertyDef -> IO ()`
- `translate :: Term Type -> Formula SValue`
- `createSymbolic :: Pattern Type -> Formula SValue`

`Program Type` is the type-inferred program, received by the property-checker after the program text has been parsed and type-checked. Because the program is already typed at this point, we can be certain that the terms and patterns will be typed as well. `SValue` is the symbolic representation of one of Contra's values.

First, the property checker receives a request to prove the properties in a program. An entry point function `check` receives a list of property definitions and the program text. For each property, `check` makes a call to the function `checkProperty`. `checkProperty` partially evaluates the property with respect to the program and collects the partially evaluated property definition and the residual program. It then calls a function to generate an SBV formula from the property. Next, it calls a function which proves the formula and prints the

$$\downarrow\text{-Unit} : \frac{}{\Delta \vdash \textbf{Unit} \downarrow \textbf{Unit}} \qquad \downarrow\text{-Number} : \frac{}{\Delta \vdash \overline{n} \downarrow \overline{n}}$$

$$\downarrow\text{-True} : \frac{}{\Delta \vdash \textbf{True} \downarrow \textbf{True}} \qquad \downarrow\text{-False} : \frac{}{\Delta \vdash \textbf{False} \downarrow \textbf{False}}$$

$$\downarrow\text{-Variable} : \frac{}{\Delta[x = t] \vdash x \downarrow t} \qquad \downarrow\text{-Constructor} : \frac{\Delta \vdash t_i \downarrow s_i}{\Delta \vdash \mathscr{C}\ [t_i] \downarrow \mathscr{C}\ [s_i]}$$

$$\downarrow\text{-Lambda} : \frac{}{\Delta \vdash \lambda p.t \downarrow \lambda p.t}$$

$$\downarrow\text{-Application} : \frac{\Delta \vdash t_1 \downarrow \lambda p.t_0 \quad \Delta \vdash t_2 \downarrow s_2 \quad \theta = \text{unify}(p, s_2) \quad \Delta \vdash \theta(t_0) \downarrow s}{\Delta \vdash t_1\ t_2 \downarrow s}$$

$$\downarrow\text{-Let} : \frac{\Delta \vdash t_1 \downarrow s_1 \quad \Delta \vdash t_2 \downarrow s_2 \quad \theta = \text{unify}(p, s_1) \quad \Delta \vdash \theta(s_2) \downarrow s}{\Delta \vdash \textbf{let}\ p = t_1\ \textbf{in}\ t_2 \downarrow s}$$

$$\downarrow\text{-Case} : \frac{\begin{array}{c}\Delta \vdash t_0 \downarrow p_0 \quad \Delta \vdash p_i \downarrow p_j \quad \Delta \vdash t_i \downarrow t_j \\ k = min\ \{j \mid \text{unify'}(p_0, p_j) \neq \emptyset\} \quad \theta = \text{unify'}(p_0, p_k) \quad \Delta \vdash \theta(t_k) \downarrow s\end{array}}{\Delta \vdash \textbf{case}\ t_0\ \textbf{of}\ \ ;\ p_i \rightarrow t_i\ \ \downarrow s}$$

$$\downarrow\text{-Plus} : \frac{\Delta \vdash t_0 \downarrow \overline{n_0} \quad \Delta \vdash t_1 \downarrow \overline{n_1}}{\Delta \vdash t_0 + t_1 \downarrow \overline{n_0 + n_1}} \qquad \downarrow\text{-Minus} : \frac{\Delta \vdash t_0 \downarrow \overline{n_0} \quad \Delta \vdash t_1 \downarrow \overline{n_1}}{\Delta \vdash t_0 - t_1 \downarrow \overline{n_0 - n_1}}$$

$$\downarrow\text{-Lt-True} : \frac{\Delta \vdash t_0 \downarrow \overline{n_0} \quad \Delta \vdash t_1 \downarrow \overline{n_1}}{\Delta \vdash t_0 < t_1 \downarrow \textbf{true}} (n_0 < n_1)$$

$$\downarrow\text{-Lt-False} : \frac{\Delta \vdash t_0 \downarrow \overline{n_0} \quad \Delta \vdash t_1 \downarrow \overline{n_1}}{\Delta \vdash t_0 < t_1 \downarrow \textbf{false}} (n_0 \geq n_1)$$

$$\downarrow\text{-Gt-True} : \frac{\Delta \vdash t_0 \downarrow \overline{n_0} \quad \Delta \vdash t_1 \downarrow \overline{n_1}}{\Delta \vdash t_0 > t_1 \downarrow \textbf{true}} (n_0 > n_1)$$

$$\downarrow\text{-Gt-False} : \frac{\Delta \vdash t_0 \downarrow \overline{n_0} \quad \Delta \vdash t_1 \downarrow \overline{n_1}}{\Delta \vdash t_0 > t_1 \downarrow \textbf{false}} (n_0 \leq n_1)$$

$$\downarrow\text{-Equal-True} : \frac{\Delta \vdash t_0 \downarrow s_1 \quad \Delta \vdash t_1 \downarrow s_2}{\Delta \vdash t_0 = t_1 \downarrow \textbf{True}} (s_1 = s_2)$$

$$\downarrow\text{-Equal-False} : \frac{\Delta \vdash t_0 \downarrow s_1 \quad \Delta \vdash t_1 \downarrow s_2}{\Delta \vdash t_0 = t_1 \downarrow \textbf{False}} (s_1 \neq s_2)$$

$$\downarrow\text{-Not-True} : \frac{\Delta \vdash t_0 \downarrow \textbf{True}}{\Delta \vdash \textbf{not}\ t_0 \downarrow \textbf{False}} \qquad \downarrow\text{-Not-False} : \frac{\Delta \vdash t_0 \downarrow \textbf{False}}{\Delta \vdash \textbf{not}\ t_0 \downarrow \textbf{True}}$$

Figure 4.5: Big-step operational semantics of Contra.

result. Finally, it returns the residual program to check so specialised functions can be used further.

The property checker is an orchestrator which assembles the pieces, but the most interesting part of the validation procedure happens in the translator. The translator receives the partially evaluated property and an initial program environment, representing the state of the program text after specialisation at the time a given property is translated.

Let us imagine we want to check property on the following form.

```
propertyName x y ... z =*= <body> .
```

By syntactic desugaring, we get the following property definition, consisting of the property name and its definition as a lambda term. This is what we input to the translator.

```
("propertyName", (\\x -> (\\y -> (... (\\z -> <body>)) ...)))
```

The goal of the property-checking is to find concrete values for the input variables $x, y, \ldots, z$ such that the property fails, if possible. In order to translate the property into a formula that can be checked by SBV, we need to *lift* the input variables $x, y, \ldots, z$. In other words, we must create fresh symbolic variables for $x, y, \ldots, z$ of the correct types in the `Symbolic` monad. At this point, the program has already been type-inferred and each term, including the input variables, are annotated by a concrete type. Therefore, we are able to determine the symbolic type that each variable should have.

The function `translateProperty` is the entry point to the translation algorithm. It begins by lifting all the inputs to the property and returning the remaining body of the property. For each of the variables $x, y, \ldots, z$, we make a call to `createSymbolic`, whose job it is to produce a well-typed symbolic variable with the given name.

However, when we are translating the body of the formula, we do not want to create new variables, but instead to constrain the ones we created in the beginning. Once created, therefore, we must also *bind* the name to the symbolic variable so that we are able to retrieve those symbolic variables by their name later. For this purpose, we require an environment, and again the `Reader` environment is a good fit for our purposes. By looking up the variables in the `Reader` environment, we gradually accumulate constraints on our initial variables. The only exception is stand-alone lambda terms, which calls for the creation and binding of fresh symbolic variables.

Finally, in order to encode user-defined algebraic data types with symbolic integers, we require access to the program text in order to apply the bijection to convert between a constructor and its index.

Once all the input variables to the property have been lifted and bound, `translateProperty` calls the function `translate` to recursively translate the body. The purpose of `translate` is to encode all the logical constraints on each symbolic variable in a way that the semantics of the original property is preserved.

After lifting the input variables and imposing the required constraints on them, we will have a formula that SBV can use to query an SMT solver. `translateFormula` returns the final result back to `checkProperty`, which makes a function call to prove the formula and print the result.

Chapter 5

Implementation

The implementation of Contra is divided into five modules, namely **Core**, **Environment**, **Analysis**, **Semantics**, and **Validation**. The driver file is called `Main`. We cover each module in turn. The order they are presented in corresponds roughly to a the transition from abstract design, through implementation, and finally to validation.

Core: The abstract `Syntax` of and `Parser` for Contra.

Environment: The `Environment`, and the monads `ERWS` and `ERSymbolic`. The data type `Environment` with convenient access to parts of the program text during analysis. The `Environment` is used to define two monads, an `ERWS` (Environment Reader Writer State) monad and a `ERSymbolic` (Environment Reader Symbolic) monad.

Analysis: The `TypeInferrer` and `Unification` algorithms. The `TypeInferrer` uses the `ERWS` monad.

Semantics: The `Interpreter`, `PartialEvaluator` and a rudimentary `REPL`.

Validation: This is the main module of interest and contains the driver, `PropertyChecker`, and the files `Translator`, `Formula`, and `SymbolicUnification`. They all use the `ERSymbolic` monad.

If we concern ourselves only with regular program execution, we can create a diagram that represents the most important dependency relations of the **Semantics** modules. This amounts to all the files outside the **Validation** module. The diagram is given in Figure 5.1.

Similarly, a diagram concerned principally with the **Validation** module is given in Figure 5.2.

5.1 Implementation Language

Contra is implemented in Haskell, a purely functional and statically typed programming language in the ML-family. It is frequently used in the implementation of programming language interpreters and compilers, and it has a mature ecosystem of libraries for writing parsers. As we will touch on later, we have made use of the monads from the `mtl` (Monad Transformer Library) package [16] and the monadic parser combinator library `Parsec` [25]. Managing stateful computations by use of monads make programmatic reasoning easier and state change more explicit than it normally is in imperative languages.

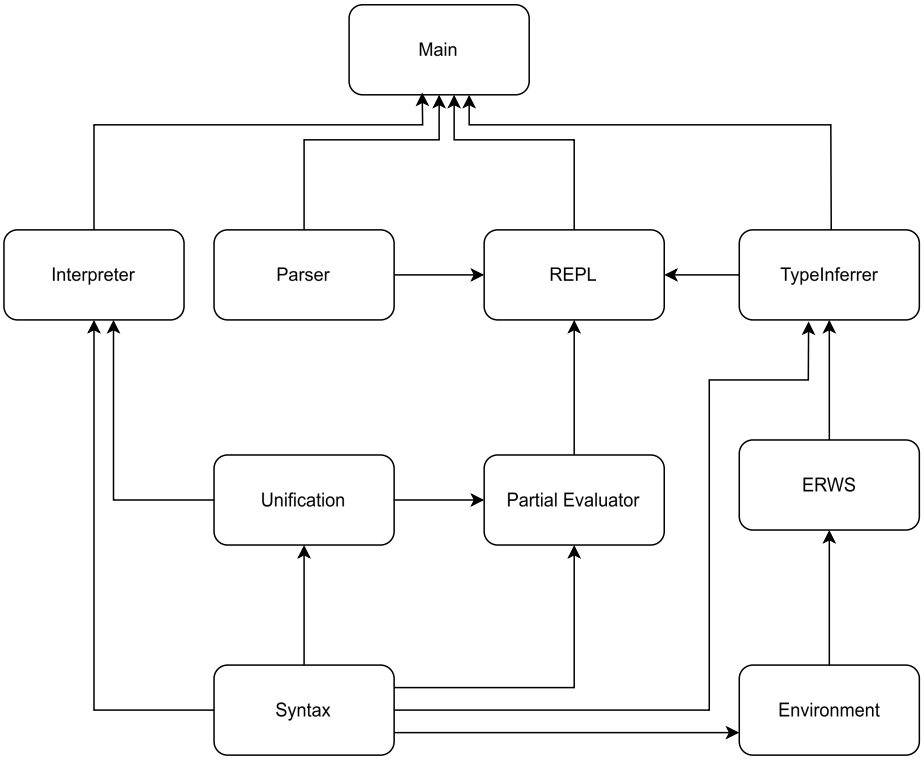

Figure 5.1: Principal dependencies for regular program execution.

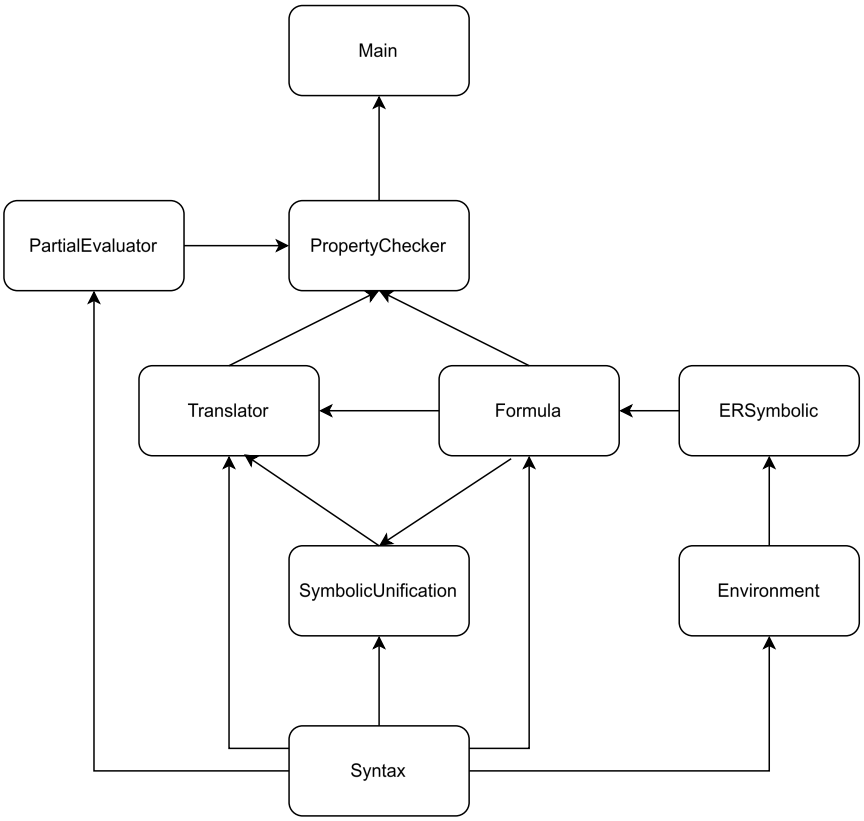

Figure 5.2: Principal dependencies for **Validation** module.

Further, Haskell is based on the lambda calculus, it has algebraic data types, and it is type-inferred. These are features we would also like to have in Contra, and can be implemented in Haskell relatively straight-forwardly. As Haskell is frequently used in the academic environment and often taught in courses on functional programming generally, it has influenced the design of many other programming languages[5, 6, 15, 17, 24, 31]. Due to this background, we may reasonably think that the average functional programmer will be somewhat familiar with its syntax, and modelling the syntax of Contra closely after that of Haskell therefore becomes a natural choice.

Finally, Haskell has several libraries providing bindings into SMT solvers. The current implementation uses the library `SBV`, but a similar translation approach should be possible using a different library of bindings, as well.

5.2 Core

The **Core** module contains the files `Syntax` and `Parser`. The `Syntax` is imported by nearly every other file and defines the abstract syntax of Contra. The `Parser` is used by the driver, `Main`, and the `REPL`.

5.2.1 Syntax

The abstract syntax closely follows the design from Section 4.1, except that we introduce an under-the-hood term and type for lists, which are only used internally when flattening pattern-matching multi-line function definitions.

In order to work with annotation variables, we define the abstract syntax of Contra over a polymorphic type variable `a` such that every term can be *annotated* during program analysis. Because terms occur in the program statements, namely in function and property definitions, this extends to the program as a whole, which also takes the type variable `a`. In type signatures in the implementation, we will therefore see types such as `(Term a)` and `(Program a)`.

The `Syntax` also exports functions for working with annotation variables, for retrieving fragments of the program text, and for conversion between `TConstructor`, `PConstructor`, and `VConstructor`.

We use the functions defined in `Syntax` to retrieve information about the program text during analysis via functions in the `Environment` data type.

Finally, the instance declarations for `Show` and `Eq` for terms, types, and programs are in `Syntax`.

5.2.2 Parser

The `Parser` is written using the monadic parser combinator library Parsec [25]. As the word "combinator" in the name suggests, `Parsec` allows us to write small, atomic parsers and combine them into bigger parsers. Writing parsers this way is ergonomic and referentially transparent, reducing the complex problem of writing a language parser to the moderate problem of writing many small parsers and combining them sanely.

Noteworthy aspects in the parser implementation are the special `=*=`-syntax for properties and the desugaring of **if**-statements and pattern-matching multi-line function definitions into **case**-statements. These are implemented straight-forwardly as discussed in Section 4.1.3, using the internal list type.

The parser records position information on the annotation variable of each term a such that errors from program analysis can be indicated with the line and column number of the relevant term(s). This information we receive directly from Parsec.

5.3 Environment

By "environment" in this section, we mean the program environment during analysis. In other words, the program text that the user has written, with function, property, and algebraic data type definitions.

The environment is used to define the monads `ERWS` and `ERSymbolic`, which are used in the `TypeInferrer` and the **Validation** module, respectively. The idea for the `Environment` and the implementation of the `ERWS` monad are from the implementation of the invertible language Jeopardy [21].

5.3.1 `Environment`

The file `Environment` provides convenient functions for access to the program text during analysis. We omit the implementation of each function, as they are straightforwardly defined. The program environment is used the definition of both the `ERWS` monad and the `ERSymbolic` monad. In addition to the environment itself, we define the type synonyms `Mapping` and `MapsTo`, which both aforementioned monads use. The code is given in Listing 5.1.

```
type Mapping a b = a -> b
type MapsTo  a b = Mapping a b -> Mapping a b

data Environment m a =
  Environment
    { envFunctions  :: [(F, Term a)]
    , envProperties :: [(P, Term a)]
    , datatype      :: C -> m D
    , fieldTypes    :: C -> m [Type]
    , constructors  :: D -> m [Constructor]
    , selector      :: (D, C)       -> m (D, Integer)
    , reconstruct   :: (D, Integer) -> m (D, C)
    , cardinality   :: D -> m Integer
    }

programEnvironment :: Monad m => Program a -> Environment m a
programEnvironment p =
  Environment { ... }
```

Listing 5.1: The `Environment`.

5.3.2 The ERWS Monad

The `ERWS` (Environment Reader Writer State) monad is an extension of the `RWS` (Reader Writer State) monad with an `Environment` over the new `ERWS` monad and an annotation variable type.

Since the program `Environment` is a read-only environment just like the `Reader`, we can implement it by embellishing `RWS`'s `Reader` environment. We define the monad `ERWS e r w`

s a by combining the program environment defined by the annotation variable e and the reader value r as a tuple, and inserting this tuple into the RWS monad as its Reader value. w and s are passed directly to RWS as we normally would. We then lift the operations of the RWS monad to the ERWS monad and give them their usual names so we can use them as we normally would. The ERWS monad implementation, borrowed from Jeopardy, is given in Listing 5.2. We omit the lifting of the RWS monad operations for brevity.

```
-- * Declaration
newtype ERWS e r w s a =
  ERWS { coERWS :: RWS.RWS (Environment (ERWS e r w s) e, r) w s a
     }

-- * Typeclass declarations
instance Monoid w => Monad (ERWS e r w s) where
  return  = pure
  m >>= f = ERWS $ coERWS m >>= coERWS . f

instance Monoid w => Applicative (ERWS e r w s) where
  pure = ERWS . RWS.return

  e0 <*> e1 = e0 >>= \f -> f <$> e1

instance Monoid w => Functor (ERWS e r w s) where
  fmap f = ERWS . fmap f . coERWS

-- * Running the monad
runERWS :: Monoid w => ERWS e r w s a -> Program e -> r -> s ->
    (a, s, w)
runERWS erws p r = RWS.runRWS (coERWS erws) (programEnvironment p,
    r)

-- * Retrieve environment
environment :: Monoid w => ERWS e r w s (Environment (ERWS e r w
    s) e)
environment = ERWS $ fst <$> RWS.ask
```

Listing 5.2: The ERWS monad (from Jeopardy).

5.3.3 The ERSymbolic Monad

We use the same technique to create the ERSymbolic monad. In this case, we start by transforming the Symbolic monad using the ReaderT monad transformer. It adds a Reader environment to the inner monad. Now that we have a Reader environment, we can proceed to add the Environment program environment as before. Because of the sheer number of operations in the Symbolic monad, we do not lift them all. Instead, we only lift its operations one step, into the ReaderT, but not into the ERSymbolic monad. Then we only need to prefix Symbolic operations with lift once to lift them into ERSymbolic. This has the advantage of making operations in the Symbolic monad explicit in the code. The implementation of the ERSymbolic monad is given in Listing 5.3.

```
-- * Environment, Reader, Symbolic
newtype ERSymbolic e r a =
```

```
  ERSymbolic { coERSymbolic :: Reader.ReaderT (Environment
     (ERSymbolic e r) e, r) Symbolic a }

instance Monad (ERSymbolic e r) where
  return  = pure
  m >>= f = ERSymbolic $ coERSymbolic m >>= coERSymbolic . f

instance Applicative (ERSymbolic e r) where
  pure      = ERSymbolic . Reader.return
  m1 <*> m2 = m1 >>= \f -> f <$> m2

instance Functor (ERSymbolic e r) where
  fmap f = ERSymbolic . fmap f . coERSymbolic


-- * Run the monad
runFormula :: ERSymbolic e r a -> Program e -> r -> Symbolic a
runFormula formula p r = Reader.runReaderT (coERSymbolic formula)
   (programEnvironment p, r)


-- * Environment
environment :: ERSymbolic e r (Environment (ERSymbolic e r) e)
environment = ERSymbolic $ Reader.asks fst


-- * Reader
local :: (r -> r) -> (ERSymbolic e r b -> ERSymbolic e r b)
local f = ERSymbolic . Reader.local (second f) . coERSymbolic

ask :: ERSymbolic e r r
ask = ERSymbolic $ Reader.asks snd


-- * Symbolic
lift :: Symbolic a -> ERSymbolic e r a
lift = ERSymbolic . Reader.lift
```

Listing 5.3: The `ERSymbolic` monad.

5.4 Analysis

The **Analysis** module contains the `TypeInferrer` and the `Unification`. These perform program analysis crucial to other parts of the program. The `TypeInferrer` is used by the driver, `Main`, and by the `REPL`. The `Unification` algorithm is used by the `Interpreter` and the `PartialEvaluator`.

5.4.1 Unification

The `Unification` algorithm is a modification of the most general unifier for Contra terms in such a way that we can employ unification in our interpreter and our partial evaluator to determine if two patterns match. We implement the unification algorithm in the usual way, but we take care to unify terms lazily.

For our implementation of Contra, we only need to consider the unification of patterns, as we will only be using unification to determine pattern-matches. A pattern match is either unsuccessful or it is given by a transformation from pattern to pattern. A transformation is a substitution on the form $[p \mapsto p']$.

We find the transformation, if any, by unifying two patterns. The result of unification is a substitution, given by a unifier. Either, it is impossible to find a unifier for the two patterns and the unifier is empty, or the unifier is a map from pattern to pattern on the form $[p \mapsto p']$ such that applying the unifier makes the two patterns equal.

In Contra, we must adjust the most general unifier to use it on our fabricated **case**-statements. Recall that multiple definitions of the same function can be flattened into a single definition with a **case**-statement if all of the patterns given in each definition are disjoint. In that case, we fabricate variable names for the input to the function. These appear as a list in the selector term of the **case**-statement, and must be able to pattern-match on the particular cases given in the original multi-line definition of the function.

In other words, we transform a function on the form it appears on the left to a function on the form appearing on the right in the following code.

```
func p11 p12 ... p1n = def_1 .
func p21 p22 ... p2n = def_2 .
.
.
.
func pn1 pn2 ... pnn = def_n .
```

```
func = (\ *a -> (\ *b -> ... (\ *z ->
case [*a, *b, ..., *z] of
; [p11, p12, ..., p1n] -> def_1
; [p21, p22, ..., p2n] -> def_2
; ...
; [pn1, pn2, ..., pnn] -> def_n.
```

As such, we must take care that our unification algorithm is not too *eager*. We are likely to encounter a situation where a partially applied function containing a **case**-statement allows us to eliminate unreachable execution paths. However, we must take care not to be too eager in our approach. Otherwise, we risk eliminating *reachable* paths. Consider for instance the following function,

```
func :: Integer -> Integer -> Integer .
func 3 y = 6 .
func x 5 = 10 .
func x y = x + y .
```

which is flattened to the following definition.

```
func :: Integer -> Integer -> Integer
func = (\ *a -> (\ *b ->
  case [*a, *b] of
    ; [3, y] -> 6
    ; [x, 5] -> 10
    ; [x, y] -> x + y))
```

If we attempted an aggressive unification approach, then the partial evaluation of `func 3` would lead us to discard the branch for the alternative `[x, 5]`, even though it might still be selected. We therefore implement our unification algorithm in a *lazy* manner, requiring the whole term — in this case, the entire list of fabricated arguments — to be concrete before we rule out any substitutions.

We also allow variables to substitute variables. This allows to check if unification is *possible* between two patterns. If not, we can safely eliminate their corresponding execution branches. This becomes relevant for the partial evaluator.

```
unifyPattern :: Pattern a -> Pattern a -> Substitution Pattern a
unifyPattern v@(Variable x _) w@(Variable y _)
  | x == y    = mempty
  | otherwise = v `replaces` w
unifyPattern v@(Variable x _) p | not $ p `contains` x = p
   `replaces` v
unifyPattern p v@(Variable x _) | not $ p `contains` x = p
   `replaces` v
```

5.4.2 Type Inference

Type-inference is the process by which the type of a term is determined automatically by an algorithm based on how it appears in the program text and the type judgement rules of the language. We implementation a type-inference algorithm for a non-polymorphic first-order functional language using de Bruijn indices, in the style of a Hindley-Miler type-inferrer.

Because we do not have type polymorphism or parameterised algebraic data types, the type-inference algorithm can be implemented using de Bruijn indices and constraint solving.

When type-inferring a whole program, we add constraints demanding type accord between the signature and the definition of a function or property. We also demand at this stage that all properties return Boolean values.

Implementation Requirements

We have defined the abstract syntax of Contra in such a way that we can associate the unification variable or concrete type with a given term. Let each term in the abstract syntax of Contra contain a polymorphic annotation variable a that can be populated with information as we analyse the program. Then we need to define a new, internal type: The type of a unification variable, denoted by an index. Now we can annotate a term by using a ContraType as the annotation variable type. The annotation is either a concrete type or a unification variable.

Since the program statements for function and property definitions also contain terms, programs must also be annotated by the same polymorphic annotation variable a.

Type-annotating a program consists of taking a general program `Program a`, throwing away whatever a was previously, and annotating each program statement recursively, eventually replacing a with a type to produce a `Program Type`, which is either concrete or a unification variable. Then we add the signature-definition-accord constraints and the property-returns-Boolean constraints to the constraints generated by type-annotation. Finally, we resolve the constraints by comparing the constraints and propagating equalities, possibly yielding a valid type substitution, which is a mapping from unification indices to concrete types. At the end, all terms have a concrete type or are ambiguously typed.

In order to implement the algorithm as described above, the type-inferrer needs to have an overview of three states. First, the next fresh index, so we can create fresh unification variables. Second, the current bindings of variable names to types. And third, the list of unification constraints.

However, we must make one additional consideration, namely the typing of algebraic data types. In order for the type-inference algorithm to determine the type of a constructor term, it is not sufficient to consult the typing rules alone. The type-inferrer also needs to access the program text in order to look up a given constructor and find the corresponding algebraic data type name, as well as to determine what types its fields should have.

In summary, the type-inferrer must keep track of the current unification index, the variable bindings (to types, rather than to terms), the type equality constraints, and the program environment.

The `Annotation` Monad

We have defined the abstract syntax of Contra in such a way that every program statement, term, pattern, and value contains a polymorphic annotation variable `a`. This enables us to implement a type-inference algorithm with de Bruijn indices as described.

Recall that the type-inference algorithm must keep track of the current index, which we can use to create a fresh unification variable, as well as the current mapping from variable names to types, and the list of constraints accumulated thus far. In addition, we need access to the program environment in order to determine the type of algebraic data types and their fields.

The `RWS` monad is a natural choice for tracking the first three states in the type-inferrer. We extend the standard implementation with an `Environment` in order to access the program environment and get the `ERWS` monad. The idea and implementation of the new, extended monad is taken from the implementation of Jeopardy [21].

We instantiate this monad with concrete types and call it `Annotate`. The implementation is given in Listing 5.4.

```
type ConstraintError  = String
data Constraint       =
  Constraint
    { type1 :: Type
    , type2 :: Type
    , info :: ConstraintError
    }

type Bindings     = Mapping Name Type
type Annotation a = ERWS a Bindings [Constraint] Index
```

Listing 5.4: Abbreviations used in the `TypeInferrer`

We leave the annotation variable as the annotation variable `a` because while we do need access to the program text, which we get through the `Environment`, we can safely discard annotation info when we annotate the terms with types. Therefore, we accept programs and terms with any annotation variable. However, we know that in our program, the type-inferrer is only ever called *after* the parser, in which case the annotation variable holds source position information. We therefore include this information in the type constraint, so that we are able to provide the user with the source position of type conflicts when resolving constraints. The program does not break even if `a` does not hold useful information, however, because we require that it is an instance of the `Show` typeclass, meaning we are always able to print it.

We use the type synonym `Mapping` from `Environment` to define our bindings, which is a mapping between variable names and types. Then we define a data type to represent type constraints, which consists of two types that should be equal and information from the annotation variable that was there before we discarded it in favour of the type annotation. The index is represented by an integer.

The `Reader` monad is optimal for keeping track of the mapping between variable names and types, because we can use `local` to handle the scope of different variables. If we had

used a global environment, we would not be able to annotate variables with the same name but in different scopes with different types, and every variable name would have had to be unique. With the `Reader` monad, on the other hand, we can add bindings that are valid for the local scope only. We define a function `bind` to add a new binding to the environment as we annotate terms recursively and `local` to apply the binding to a computation.

The `Writer` monad is the natural choice for accumulating type equality constraints. It provides a write-only environment for collecting computation outputs. Because we only ever want to add constraints — and never modify or remove them — the ideal use case of the `Writer` monad maps neatly onto ours. We define helper functions to add constraints using `Writer`'s `tell` function..

Finally, the `State` monad provides both read and write access to the environment. When we are creating a fresh unification variable, we need to read the environment to access the index we should use, but we also need to increment the index for next time. The `State` monad provides the functions `get` and `put` for getting the current state and for replacing the state in the monad, respectively.

We extend this monad with the `Environment` data type, which provides functions for accessing the program text. This way, our type-inferrer should be capable of typing every non-ambiguous term in Contra. This gives us all the benefits of the `RWS` monad, with added read-only access to the program text via the `Environment`. We can query the environment for the algebraic data type name of a constructor, as well as the types of all its fields. Armed with this information, we proceed to add constraints in accordance with the typing rules.

Implementation

Because the implementation of the annotation procedure follows directly from Contra's typing rules, we omit the details for brevity. Let us instead consider a few illustrative examples.

For convenience, we here reproduce the typing judgement for τ-Plus and for τ-Application.

$$\frac{\Delta\Gamma \vdash t_0 : \textbf{Integer} \quad \Delta\Gamma \vdash t_1 : \textbf{Integer}}{\Delta\Gamma \vdash t_0 + t_1 : \textbf{Integer}} \; \tau\text{-Plus}$$

$$\frac{\Delta\Gamma \vdash t_1 : (\tau_1 \to \tau_2) \quad \Delta\Gamma \vdash t_2 : \tau_1}{\Delta\Gamma \vdash t_1\ t_2 : \tau_2} \; \tau\text{-Application}$$

The following two helper functions are relevant. `fresh` gets a fresh unification index from the state, creates a new unification variable — the type `(Variable' i)` where `i` is the unification index — and increments the state with `put`. `addConstraint` takes an annotated term and a type, and constrains the term's annotation to be equal to the type. It also record the source position information from the term. The constraint is created using `tell`.

`Plus` is type-annotated by annotating each operand, and requiring that they both be integers. Finally, the type of the term is determined to be integer, also. We can see how this is precisely equivalent to the term's typing rule.

```
annotate :: Show a => Term a -> Annotation a (Term Type)
annotate (Plus t0 t1 _) =
  do t0' <- annotate t0
     t1' <- annotate t1
     addConstraint t0' Integer' (show $ annotation t0)
     addConstraint t1' Integer' (show $ annotation t1)
     return $ Plus t0' t1' Integer'
```

`Application` is type-annotated by creating two fresh unification variables. This corresponds to the types τ_1 and τ_2 in the typing rule for function application. We have no way of knowing at present time what the types of the function or the argument may be, but we do know that the argument type must correspond to the input type of the function. At this time, we also demand that the term t_1 has an arrow type. We annotate the (eventual) function and the argument, and add the corresponding constraints. Finally, we determine that the type of the application must be equal to the function's output type.

```
annotate :: Show a => Term a -> Annotation a (Term Type)
annotate (Application t1 t2 _) =
  do tau1 <- fresh
     tau2 <- fresh
     t1'  <- annotate t1
     t2'  <- annotate t2
     addConstraint t1' (tau1 :->: tau2) (show $ annotation t1)
     addConstraint t2' tau1             (show $ annotation t2)
     return $ Application t1' t2' tau2
```

Let us finally consider a trickier case. We here reproduce the typing rule for (term) constructors.

$$\tau\text{-Constructor}: \frac{\Delta\Gamma \vdash t_i : \tau_i}{\Delta[\mathbf{adt}\ \mathscr{D} = (\mathscr{C}\ [\tau_i])_j]\Gamma \vdash \mathscr{C}\ [t_i] : \mathscr{D}}$$

A data type constructor and its fields is type-annotated using the `Environment`. Let us consider the case for term constructors, whose fields are other terms. We begin type-annotation by annotating each of the fields. Next, we retrieve the program environment and looking up the constructor in the algebraic data type definitions. Then, we look up the types of the fields of the constructor, also in the program environment. We constrain the type of each field to be equal to the corresponding field type from the program text. At this point, we attempt to strengthen the `TConstructor` in case it can be reduces to a `PConstructor` or `VConstructor`. Finally, we determine the type of the term to be the algebraic data type name we looked up.

```
annotate :: Show a => Term a -> Annotation a (Term Type)
annotate (TConstructor c ts _) =
  do ts' <- mapM annotate ts
     env <- environment
     adt <- datatype env c
     cs  <- fieldTypes env c
     addConstraints ts' cs (map (show . annotation) ts)
     return $ strengthenIfPossible c ts' (ADT adt)
```

5.5 Semantics

The **Semantics** module contains the `Interpreter`, `PartialEvaluator` and the `REPL`. These are all the files directly involved with the evaluation of terms and execution of program statements. The `Interpreter` implements the operational semantics of the language directly, while the `PartialEvaluator` makes a few adjustments. The `REPL` utilises the `PartialEvaluator` to partially evaluate terms interactively.

5.5.1 Interpreter

The interpreter is implemented in the conventional way and follow directly from the operational semantics of the language. We use a `Reader` monad with the `Program a` as its environment in order to access the top-level definitions of functions and properties. A function `evaluate :. Term a -> Runtime a (Term a)` evaluates a term according to the big-step operational semantics.

A term is either a canonical term that evaluates to itself, or it can be evaluated by the evaluation of its components. Arithmetic, relational, and logical operations are evaluated the usual way. Constructors are evaluated by evaluating all its fields and returning the resulting constructor.

A variable is either bound in an input pattern and substituted in-place by unification or it is a standalone term that is looked up in the program text (the `Reader` environment). If the variable is *not* substituted by unification or bound to a function or property definition in the program text, then it is unbound. We also signal an error if it is bound in multiple locations in the program text.

Function application, **let**-statements, and **case**-statements all require unification.

A function application is on the form $t_1 t_2$ and is evaluated by first evaluating t_1 and verifying that the result is a function on the form $\lambda p.t_0$. Otherwise, we cannot apply it and we signal an error. We then evaluate t_2 to t_2'. Next, we unify p and the evaluated argument t_2' and apply the corresponding substitution to the body of the function t_0. Finally, we evaluate t_o where p is substituted by t_2'.

A **let**-statement is on the form **let** $p = t_1$ **in** t_2 and is evaluated by evaluating t_1 to t_1'. We unify p against t_1' and, if successful, apply the substitution to the body of the statement, t_2. Finally, we evaluate t_2 where p is substituted by t_2'.

A **case**-statement is on the form **case** t_0 **of** ; $p_i \rightarrow t_i$. It is evaluated by first evaluating t_0 to a pattern p. If t_0 does not evaluate to a pattern, it cannot be used in a pattern-match and we signal an error. Next, we unify p against the patterns p_i in order and select the first successful unification. If none unify, we signal an error. Else, we apply the substitution we get to the corresponding term t_i and evaluate the result.

For each pattern, we verify that the free variables in it do not occur on the top-level of the program, thus guarding against name collisions.

The user can evaluate programs using the interpreter. By default, the interpreter looks for and executes a function called `main` and throws an error if this function does not exist. From the `main` function, the interpreter evaluates sub-terms recursively.

The user can install the Contra executable by building and installing the source code with the Haskell build tool Stack. Instructions can be found in the `README.md` file in the GitHub repository of the prototype implementation.

With Contra installed, the user can evaluate — or *execute* — a program by invoking the Contra command in their terminal followed by the path to a Contra program.,

```
> contra <program-name>.con
```

5.5.2 Partial Evaluation

We implement an online partial evaluator in the style of Cook & Lämmel [11]. The full implementation can be found in the GitHub repository.

Recall that Cook & Lämmel design and implement an online partial evaluator for a small, first-order functional programming language. However, their language is different from Contra in several regards, which means we will have to make adjustments to the original partial evaluator.

In their approach, only functions that are used by the `main` function are specialised and thus included in the final residual program. In Contra, however, we want to specialise the functions used in properties incrementally, and pass on the specialised program as the program to be used for the next property. We therefore cannot throw away the parts of the program that were not used in the current pass of the partial evaluator. We only want to *add* function definitions, containing the specialised functions, to the rest of the program. That way, the next pass of the partial evaluator has access to the specialised functions *and* the original program. For clarity, this extended program is what we mean by "residual program" in the context of Contra.

To this end, we only use a `State` monad, which has the program text as its environment, that we can read from and add to. This monad, the `State` monad specialised to programs, the `PartialState` monad.

Let `partiallyEvaluate :: Program a -> Term a -> (Term a, Program a)` be a function that acts as an entry point to the partial evaluator. Its sole function is to call the function to partially evaluate a term — the property definition — in the context of a program and run the `State` monad on the result to extract the final term and the residual program.

The function `partial` is the function that partially evaluates terms, which it does recursively. The naïve implementation has the type signature `partial :: Term a -> PartialState a (Term a)`.

The partial evaluation of simple terms is follows naturally from the interpreter and the operational semantics. Values partially evaluate to themselves, variables are looked up in the function and property definitions of the program text, and constructor values and patterns are partially evaluated by evaluating all its fields.

Arithmetic and relational terms, as well as negation, are partially evaluated by partially evaluating its sub-terms, and if these are canonical, so is the result the whole term, and otherwise we reconstruct the term but with partially evaluated sub-terms. Recall that a term is canonical if it is a pattern with no free variables or it is a lambda term.

Compared to Cook & Lämmel's approach, the cases that we need to modify significantly or create entirely from scratch are the following:

- **let**-statements.
- Lambda terms.
- Function application.
- **case**-statements.

let-statements are relatively straight-forward to partially evaluate. Recall that a **let**-statement has the form **let** $p = t_1$ **in** t_2. We check that the pattern p that we are binding, does not shadow a top-level binding. Then, we partially evaluate the pattern p and the term t_1 we want to bind to it. If the term t_1 partially evaluates to a canonical term, then we check that it is a pattern and not a lambda, and unify the two. If unification was successful, we apply the substitution to the term t_2 and partially evaluate it. If t_1 is not canonical, we reconstruct the **let**-statement with the partially evaluated components.

Lambda terms require some additional consideration. As for **let**-statements, we still check that the input pattern does not shadow a top-level binding. However, because lambdas may become nested as a result of the partial evaluation, we must avoid *variable capture*, i.e.,

conflicting names within the term. Therefore, the partial evaluator keeps track of names that have been used in the current scope, and we control this list of names when we partially evaluate a lambda term.

We add a list of names as a parameter to `partial`. If any variable names are already in use within the current scope, we generate a fresh name by hashing and propagate this name throughout the lambda. Renaming variables is known as *alpha renaming*, or alpha *conversion*, a term borrowed from the underlying lambda calculus. To this end, we define a function `alpha`, which takes a list of free variables, a list of used names, the input pattern to the lambda, and the body of the lambda. It returns an updated list of used names and the (possibly renamed) input pattern and term. Equipped with these, we partially evaluate the body of the lambda with the new names and reconstruct the lambda. Since we cannot evaluate it to a value, we simply return it.

For our purposes, it is crucial that the partial evaluator performs alpha renaming from the outside and in. This way, the original names of the input variables to a property are preserved, while inner variables that represent intermediate states of values are renamed to avoid conflict. Then we can still retrieve the original names and use them to print counterexamples.

```
partial :: [Name] -> Term a -> PartialState a (Term a)
partial ns (Lambda p t a) =
  do notAtTopLevel p
     let fvs = freeVariables' p
     let (ns', alphaP, alphaT) = alpha fvs ns p t
     t'  <- partial ns' alphaT
     return $ (Lambda alphaP t' a)
```

To perform function application, we need to adjust the approach of Cook & Lämmel slightly. Their language accepted a list of function arguments all at once, whereas Contra has curried functions and which only accept a single argument. We also distinguish between named and anonymous functions. Let us cover named functions first.

The application of a named function is partially evaluated by retrieving the `PartialState` and looking up the name in the functions and properties of the current version of the program. If the name is not a lambda term, then we cannot apply it to an argument and we signal an error. If it is a lambda term, then we partially evaluate the argument.

Since our function can take only a single argument, we are relieved from partitioning dynamic and static arguments. If the argument is a canonical term, then we fabricate a new function name using the original function name and a hash of the argument. We look up the name in the state to see if has already been specialised. If it has, then we return it directly. If not, we use the same trick as before. We bind the fabricated name to an undefined function, before we partially evaluate the function body with the argument term substituted for the input pattern and obtain the specialised function body. Then we replace the undefined definition with the actual function body. We record the specialised function definition in the state using the function `bind`. `bind` is a function that takes a name and a term, creates or updates a function definition and adds it to the program text. Finally, return the result. If the input argument is *not* canonical, then we reconstruct the application with the unevaluated function term and the partially evaluated argument.

```
partial :: [Name] -> Term a -> PartialState a (Term a)
partial ns (Application t1@(Pattern (Variable x _)) t2 a) =
  do env <- get
     case lookup x (functions env ++ properties env) of
```

```
      Just (Lambda p t0 _) ->
        do t2' <- partial ns t2
           if canonical t2'
             then let x' = show x ++ show (hash (show t2')) in
                    case lookup x' (functions env ++ properties
                       env) of
                      Just specialised -> return specialised
                      Nothing -> do bind x' undefined
                                    result <- partial ns $
                                       substitute p t0 t2'
                                    bind x' result
                                    return result
             else return $ Application t1 t2' a
      Just _  -> error $ "Variable '" ++ x ++ "' is not a
         function"
      Nothing -> error $ "Unbound function name '" ++ x ++ "'"
```

The partial evaluation of all other applications, which are on the form $t_1\ t_2$, is much simpler. We begin by partially evaluating t_1 and then t_2. The term t_1 that we are applying should either be an anonymous function or a nested function application which eventually evaluates to a function, since we have already handled the case for named functions. If it is not a function, we must signal an error. However, it is possible that it *is* a function but that the inner argument is a variable. In that case, t_1' may be a non-canonical lambda, where variables *outside* its scope appear in its body. In that case, we reconstruct the application. If both the terms t_1' *and* t_2' are concrete, then we can evaluate the function application directly. If not, we reconstruct the function application with the partially evaluated components.

```
partial :: [Name] -> Term a -> PartialState a (Term a)
partial ns (Application t1 t2 a) =
  do t1' <- partial ns t1
     t2' <- partial ns t2
     if canonical t2' && canonical t1'
       then do f <- function t1'
               partial ns (f t2')
       else return $ Application t1' t2' a
```

Finally, we come to the partial evaluation of **case**-statements, which are on the form **case** t_0 **of** ; $p_i \rightarrow t_i$. We first partially evaluate the selector term t_0. If the result is a canonical term, then we check that it is a pattern and not a lambda, unify it against the patterns p_i and select the first match. We apply the substitution to the corresponding term t_i and partially evaluate the result. If the selector term was *not* a canonical term, then we partially evaluate all the p_i and t_i and we use the partially evaluated selector term to eliminate unreachable cases in the pairs. This is the part of the code that might reduce the complexity of our properties. Finally, we reconstruct the **case**-statement with the partially evaluated selector term and branches.

```
partial :: [Name] -> Term a -> PartialState a (Term a)
partial ns (Case t0 ts a) =
  do v <- partial ns t0
     if canonical v
       then do (u, t) <- firstMatch (strengthenToPattern v) ts
               partial ns $ applyTransformation u t
       else do alts   <- mapM (partial ns . weakenToTerm . fst) ts
```

```
bodies <- mapM (partial ns . snd) ts
let alts' = map strengthenToPattern alts
let cases = zip alts' bodies
let ts'   = eliminateUnreachable v cases
return $ Case v ts' a
```

Just as Cook & Lämmel's implementation, we should note that this partial evaluator does not perform any termination check, but does handle recursive function definitions.

5.5.3 REPL - Read-Eval-Print Loop

Using the parser, type-inferrer, and partial evaluator, we can implement a simple prototype REPL (Read-Eval-Print Loop). This enables the user to (partially) evaluate terms interactively. The REPL runs in a loop, reading user input, (partially) evaluating it, and printing the result to the screen. The Contra REPL is rudimentary and prototypical in comparison with other programming languages, but access to a REPL is nonetheless useful to mock up functions progressively and to validate the output of functions on-the-fly.

Just as for the interpreter, the user must build and install the Contra executable on their machine according to the installation instructions. Once installed, the user can start a REPL session with the keyword "`contra`" and the REPL will wait for a term or command at the prompt.

The user can load a program using "`:l <program-name>.con`", where "`<program-name.con>`" is a path to a Contra program. Note that only one program can be loaded at a time. The user then has access to all the function and property definitions in the file.

The user can also define their own functions and constants (nullary functions) using the special syntax "`def <var-name> = <term>`". Under the hood, this is appended to the current program text as a function definition. The blank REPL session is in fact a REPL session with the program `End` loaded.

The user can quit the REPL with "`:q`".

```
> contra
* Started the Contra REPL! *

contra> :l example-program.con
Loaded file "example-program.con"
contra> def inc = (\x -> 1 + x)
(\"x" -> 1 + "x")
contra> inc 5
6
contra> :q
>
```

Because the REPL uses the partial evaluator, the user can partially apply functions, and even name the specialised function by using "`def`".

```
> contra
* Started the Contra REPL! *

contra> def plus = (\x -> (\y -> x + y))
(\"x" -> (\"y" -> "x" + "y"))
contra> def plusFive = plus 5
(\"y" -> 5 + "y")
contra> plusFive 3
```

```
8
contra> :q
>
```

Notably, the prototype REPL does not allow the user to check properties. Moreover, there is no check for type-correspondence between a function and its argument when only partially applied. This means that the REPL does *not* throw type errors until the function is called on its last argument and evaluated completely. This is a significant limitation, as it leads to false confidence when specialising functions. However, we still believe the REPL to be useful to the development experience and therefore choose to include it in spite of this.

5.6 Validation

The **Validation** module contains the principal files used to check properties. Every file in the module imports the `Syntax` and the `PropertyChecker` imports the `PartialEvaluator`. The main monad used in the translation algorithm relies uses the `ERSymbolic` monad and so relies on the `Environment`, as well. The files contained in the **Validation** module are: The `PropertyChecker`, the `Formula`, the `Translator`, and the `SymbolicUnification` algorithm.

The `PropertyChecker` is the orchestrator, which partially evaluates properties, calls the `Translator` to translate each property into a formula, proves the formula and prints the result, before repeating the process for the next property using the specialised program.
The `Formula` contains the definition of the middle layer between Contra and SBV, including the `Formula` monad, the `SValue` custom symbolic variable type, and the custom equality relation on `SValues`. The implementation of `Formula` is given in Appendix A.
The `Translator` contains the main translation algorithm. Its entry point function is called from the `PropertyChecker`. It relies on the monad and definitions in the `Formula` file and on the pseudo-unification in the `SymbolicUnification` file.
The `SymbolicUnification` algorithm is responsible for pseudo-unification, between concrete Contra patterns and `SValues`.

Our implementation uses the Haskell library SBV and Microsoft's Z3 [29] as the backend solver, chosen because of its performance and free availability.

5.6.1 Overview

Recall the translation sketch from Section 4.4. We want to check a program's properties by, for each property, executing the following steps:

1. Partially evaluate the property.
2. Translate the property to a symbolic formula.
    - Lift input patterns to the property.
        - Create symbolic variable(s) in `Symbolic` monad.
        - Bind symbolic variables in local bindings (`Reader` environment).
3. Translate property body recursively.
4. Realise symbolic Contra formula as an SBV formula.

5. Prove formula.
6. Print result.
7. Collect residual program from partial evaluation and check next property with the specialised program.

Most of the high-level steps are handled by the `PropertyChecker`. The `Translator` is the main file in this module, however, and handles steps 2 and 3. Before we can begin translating Contra terms to SBV formulae, however, we need to define some custom constructs to act as a middle layer between Contra and SBV.

5.6.2 Custom Symbolic Types

Considering that our translation algorithm is an "online" translator and we will be calling the translation function recursively for each sub-term, we must be able to return a unified type for different symbolic variables. For instance, we need to translate values, which means we must be able to return both symbolic integers *and* symbolic Boolean values from the same function. Because of Haskell's type system, however, we are only allowed to return output of *one* type from our translation function. We therefore introduce a type wrapper called `SValue`.

Moreover, we need a way to represent Contra values that do not have a corresponding symbolic type in SBV, namely **unit**, the internal `List` used in **case**-statements, and algebraic data type instances.

We will define the symbolic data type for algebraic data types with the constructor `SCtr` for now, and return to this point later. This kind of symbolic algebraic value is given by `D`, `C` and `[SValue]`, where `D` is the name of the algebraic data type, `C` is the name of the data type constructor, and `[SValue]` is the list of fields.

The definition of `SValue` (at this point) can be found in Listing 5.5.

```
data SValue =
    SUnit
  | SBoolean SBool
  | SNumber  SInteger
  | SArgs    [SValue]
  | SCtr     D C [SValue]
  deriving Show
```

Listing 5.5: The initial definition of `SValue`.

5.6.3 The `Formula` Monad

As discussed in the translation sketch, the translation algorithm requires access to the program environment in order to retrieve function definitions and algebraic data type definitions from the program text. We can use the same `Environment` as we used for the type-inference algorithm to provide access to the program text. However, we do not need the full power of the `RWS` monad. On the other hand, we *do* need the `Symbolic` monad and a `Reader` environment.

We need the `Symbolic` monad in order to create fresh symbolic variables and to impose logical constraints on these variables. We only need to create *fresh* symbolic variables for the *input* variables to the property, however. As we progress in our translation and meet

those same variable names, we can simply refer to the already created symbolic variables. In fact, we must, for the translation to be sound. For this purpose, we use the `ReaderT` monad transformer, also from `mtl`, to add a Reader environment to the Symbolic monad.

This is why we implemented the `ERSymbolic e r a` (Environment Reader Symbolic) monad. It accepts a program `Environment` annotation variable of type `e`, a `Reader` environment of type `r` and wrap computation result of type `a`.

We let `Formula a` be the monad instantiated by `ERSymbolic Type Bindings a`, where `Type` is the annotation variable in the program environment and `Bindings` are a mapping from variable names to symbolic variables, given by the wrapper type `SValue`. `a` is the type of the result of a computation in the `Formula` monad.

We also define the helper function `bind`, which updates the bindings such that the name `x` becomes bound to a symbolic variable `sv`. We include the definition of a `Mapping`, defined in `Environment`, as a comment for convenience.

```
-- type Mapping      a b = a -> b
-- type MapsTo       a b = Mapping a b -> Mapping a b

type Bindings   = Mapping X SValue
type Formula  a = ERSymbolic Type Bindings a

bind :: X -> SValue -> X `MapsTo` SValue
bind x sv look y =
  if y == x      -- If bindings applied to a 'y' == 'x'
     then sv     -- then we should now return 'sv'
     else look y -- If called with some other 'y',
                 -- then return the old binding for 'y'
```

Listing 5.6: Definition of the `Formula` monad and the helper function `bind`.

5.6.4 Lifting Patterns & Creating Symbolic Variables

The entry point to the translation algorithm is a function called `translateToFormula`. It makes a call to the function `liftPropertyInputPatterns`, which lifts the input variables in the `Formula` monad and returns the bindings and the property body. Then, it translates the body of the property with the generated bindings in the local environment.

`liftPropertyInputPatterns` calls the function `liftPattern` for each input variable and collects the bindings. `liftPattern` makes a call to the function `createSymbolic` to create a new, named symbolic variable in the `Symbolic` monad. `liftPattern` binds the variable name to the created symbolic variable and returns the bindings. Finally, when there are no more input variables to lift and bind, `liftPropertyInputPatterns` returns the accumulated bindings and the remaining body of the property.

`createSymbolic` is one of the principal functions of the translation algorithm. The implementation for every type of symbolic variable except for symbolic algebraic data types is given in Listing 5.7. We will cover algebraic data types on their own in Section 5.6.10.

```
createSymbolic :: Pattern Type -> Formula SValue
createSymbolic (Variable _ Unit')    = return SUnit
createSymbolic (Variable x Integer') =
  do sx <- lift $ sInteger x
     return $ SNumber sx
createSymbolic (Variable x Boolean') =
```

```
    do sx <- lift $ sBool x
       return $ SBoolean sx
  createSymbolic (Variable x (Variable' _)) =
    do sx <- lift $ free x
       return $ SNumber sx
  createSymbolic (Variable _ (TypeList [])) =
    do return $ SArgs []
  createSymbolic (Variable x (TypeList ts)) =
    do let names = zipWith (\tau i -> show (hash (x ++ show tau)) ++
       show i)
                    ts
                    ([0..] :: [Integer])
       let ps    = zipWith Variable names ts
       sxs <- mapM createSymbolic ps
       return $ SArgs sxs
  createSymbolic (Variable x (ADT adt)) = undefined
  createSymbolic p = error $
       "Unexpected request to create symbolic sub-pattern '"
    ++ show p ++ "' of type '" ++ show (annotation p) ++ "'"
    ++ "\nPlease note that generating arbitrary functions is not
       supported."
```

Listing 5.7: The incomplete definition of `createSymbolic`.

The intuition for `createSymbolic` is relatively simple. We receive an input variable named "x" of a given type, and we want to create a symbolic variable with that name. By giving the variable the same name in the `Symbolic` monad, SBV will create a model where that name is associated with the final value. When printing a counterexample, the name of the input variable will be used to display the witness value.

`Unit'` does not have a corresponding symbolic type, but it does not need one either. A value of type `Unit'` is always the same value, namely unit, so it is only ever equal to itself and no other operations can be performed on it.

`Integer'` and `Boolean'` have direct symbolic counterparts, so we make a call to their corresponding SBV constructors and name them. We wrap and return the result.

We do not, in the current implementation, support polymorphic input variables. Therefore, we should never be asked to generate a symbolic variable for an input variable of type `(Variable i)`, because `(Variable i)` represents an unresolved unification variable. However, we can define the case for polymorphic inputs as if we did, simply by declaring it a free variable. SBV is then free to instantiate it to any value to satisfy constraints in the formula.

Similarly, we should never be asked to create a symbolic input variable for `TypeLists`, since this is an internal type. It represents a list of pattern-matchable arguments in a flattened function definition. Still, we are able to do so, and for completeness, we cover this case, as well. We fabricate new names for each variable by hashing the string ""x-<type-name>"" and appending the index of the type in the list. This ensures each variable name is unique and correlated to the original variable name. We create symbolic variables for each of the types in the list and return the reconstructed list as the symbolic type `SArgs`.

Finally, the current implementation does not support generation of arbitrary functions, so a request to generate this type results in an error.

5.6.5 Simple Translation

Some Contra terms are straight-forward to translate, due to SBV's functions. We will cover these first.

`translateToFormula` receives the bindings and the property body back from the helper function `liftPropertyInputPatterns`, and proceeds to call `translate` with the bindings in the local environment of the `Formula` monad. It is `translate`'s task to encode all the constraints from the property as constraints on the symbolic variables.

`translate` is one of the main functions of the translation algorithm. It is a function from `Term Type` to `Formula SValue`. At this stage, however, we note that SMT solvers cannot a priori handle *recursive* functions. As a safety measure, we therefore include a `RecursionDepth` as an argument to `translate`, an integer representing the maximum number of recursive function calls before timeout. Let `translate` be the following function.

```
translate :: RecursionDepth -> Term Type -> Formula SValue
```

`translate` makes calls to the helper functions `translatePattern` and `translateValue`.

```
translatePattern :: RecursionDepth -> Pattern Type -> Formula SValue

translateValue   :: Value Type -> Formula SValue
```

Translating values is as simple as making calls to the `literal` function with the concrete value, and wrapping the result with the appropriate `SValue` constructor. Since no recursion can occur, it does not take a `RecursionDepth` argument.

When translating patterns, variables are the only special case we need to consider. A variable name needs to be looked up. The variable name is either bound to a symbolic input variable that we have created and need to retrieve from the `Reader` environment, or it is a function external to the property that was not inlined during partial evaluation that we need to retrieve from the `Environment` program environment.

SBV has native constructs for symbolic arithmetic operations and for logically constraining the values of its built-in symbolic types. Adding constraints to integer and Boolean variables is therefore just a matter of unwrapping the `SValue`, applying SBV's constructs, and wrapping the result.

The notable exception is symbolic equality. In Contra, arithmetic and logical operations are only defined on integer and Boolean values. However, equality is defined on all terms. Because we have implemented our own type, `SValue`, we also need to define an equality relation on this type, corresponding to the equality relation on terms defined in the operational semantics, in Section 4.3.1.

5.6.6 Symbolic Equality

We define a relation `sEqual` on `SValues`. The implementation is given in Listing 5.8.

```
sEqual :: SValue -> SValue -> Formula SValue
sEqual  SUnit           SUnit           = return $ SBoolean sTrue
sEqual (SBoolean   b) (SBoolean    c) = return $ SBoolean (b .== c)
sEqual (SNumber    n) (SNumber     m) = return $ SBoolean (n .== m)
sEqual (SArgs     xs) (SArgs     ys) =
  do eqs <- zipWithM sEqual xs ys
     return $ SBoolean $ sAnd $ map truthy eqs
sEqual _             _              = return $ SBoolean sFalse
```

```
sEqual (SCtr adt x xs) (SCtr adt' y ys) =
  do eqs <- zipWithM sEqual xs ys
     return $ SBoolean $ sAnd $
         fromBool (adt == adt')
       : fromBool (x   == y   )
       : map truthy eqs
```

Listing 5.8: Initial definition of symbolic equality on `SValues`.

The implementation is mostly conventional. For symbolic integer and Boolean values, we unwrap the values and compare them with SBV's comparison operator. The symbolic unit value is only equal to itself. A symbolic algebraic value created with `SCtr` is equal to another if they have the same data type name, the same constructor name, and all their (symbolic) fields are equal.

Currently, however, our implementation of `createSymbolic` and consequently of `SValue` and `sEqual` are not complete. We return to the implementation of `sEquals` for symbolic algebraic data types in Section 5.6.10.

5.6.7 Translation With Symbolic Unification

In the operational semantics of Contra, we see that the there are three terms that depend on unification and pattern-matching, namely function application, **let**-statements, and **case**-statements. Each requires that an already translated `SValue` is unified against a regular Contra pattern. This leads us to implement a *symbolic* unification algorithm. Symbolic unification will enable us to translate the two former of the statements, but not the **case**-statement directly.

The function of the symbolic unification is two-fold. Firstly, should throw an error if it is impossible to unify the pattern and the symbolic value, e.g., if they are of different types. Secondly, it should produce and return bindings, e.g., symbolically unifying a Contra variable against a symbolic integer should bind the variable name to the symbolic integer.

Using the unification algorithm, a **let**-statement on the form **let** $p = t_1$ **in** t_2 can be translated by translating the bound term t_1 to a symbolic variable, then symbolically unifying it against the pattern p. The unification returns the updated bindings, which are then used to translate the body of the **let**-statement, t_2, resulting in a translation where the pattern is bound in the *local* scope only, which agrees with the operational semantics of the language.

In order to handle function application, we take inspiration from the approach used in the partial evaluator. A function application consists of a term t_1 being applied to another term t_2. The term t_1 is one of three possible terms. It is a lambda term that can be applied directly, it is a variable that is *bound* to a lambda term and can be applied directly, or it is a nested application which will eventually yield a lambda term. We begin in any case by evaluating the argument term t_2 to a symbolic variable. Next, we attempt to unify the term with the input pattern to the (eventual) function t_1 and return the bindings and function body.

If t_1 is a lambda term, then it is a anonymous function that was either defined as such by the user or was inlined by partial evaluation. Since only recursion free functions were inlined this way, we can safely apply the function. We symbolically unify the input pattern of the function with the argument, now a symbolic variable, and we return the resulting bindings and the untranslated function body.

If t_1 is an application of an application $t_1' t_2'$ to the symbolic argument sv, then as before, we begin by translating the argument term t_2'. Next, we unify and bind the now symbolic t_2' to the (eventual) function t_1'. However, to avoid infinite recursion, we decrement the

recursion depth variable at this point. Once this function call returns the necessary bindings and the innermost function, we unify and bind the original symbolic argument *sv* to the inner function. Finally, we collect all the bindings generated and return the inner body of that application.

If t_1 is a *variable* however, we know that this must be a recursive function that was not handled in the partial evaluation stage. The current implementation of Contra does not handle recursive functions, but we implement the translation as if it did: First, look up the variable name in the program text via the environment. If it is not a function or not bound, we throw an error. If it is a lambda term, then we unify and bind the input to the argument as in the case for a plain lambda. Note that this will trigger the the case where the recursion depth is zero and we throw an error.

5.6.8 Branching & Pseudo-Unification

Since the symbolic unification algorithm cannot account for the concrete values of symbolic variables at time of unification and therefore not unify two patterns in the traditional sense, we can characterise the algorithm as a "pseudo-unification" algorithm. We might be tempted to implement symbolic unification in such a way that we *constrain* the symbolic variable(s) to be equal to their concrete counterparts during unification. However, this would break the semantics of **case**-statements. We will illustrate this problem after we have suggested an implementation, at which point it will be easier to identify the issue with this approach.

The **case**-statement demands particular consideration. We cannot simply unify the selector pattern against the branch alternatives, because it would throw an error if unification failed. This would imply that we artificially interrupt translation because preceding branch alternatives were impossible to unify with the symbolic selector variable.

Instead, we implement a function `unifiable`, which does *not* throw any errors or return the bindings necessary to evaluate a **case**-branch. It simply determines whether it is possible for a given symbolic variable to pattern-match on a given concrete pattern. For instance, the pattern `C {x}`, where `"C"` is a constructor and `"x"` is a variable representing a field, cannot be unified against the symbolic variable `(SCtr "ExampleType" "D" [])`, since it has a different data constructor, `"D"`.

The last step required to translate **case**-statements, is symbolic conditional statements. Thankfully, SBV does have a built-in for `if-then-else`-statements, but we will need to adjust this somewhat. SBV's conditional statement `ite`, short for "if-then-else", is defined in the following way.

```
ite :: Mergeable a => SBool -> a -> a -> a
ite t a b
  | Just r <- unliteral t = if r then a else b
  | True                  = symbolicMerge True t a b
```

In the above code, `Mergeable` is a typeclass defining types that can be combined by symbolic conditional statements. The function `ite` is implemented by way of the function `symbolicMerge`, which merges two values. The parameter `True` indicates in particular that both branches *must* have the same type.

In order to implement **case**-statements, we must therefore first make `SValue` an instance of the `Mergeable` typeclass and implement the function `symbolicMerge`. The implementation can be found in Listing 5.9. Note once more that it is incomplete, as it is missing the case for the symbolic algebraic data type variables.

```
instance Mergeable SValue where
  symbolicMerge = const merge

merge :: SBool -> SValue -> SValue -> SValue
merge _  SUnit         SUnit       = SUnit
merge b (SNumber  x) (SNumber  y) = SNumber  $ ite b x y
merge b (SBoolean x) (SBoolean y) = SBoolean $ ite b x y
merge b (SCtr adt x xs) (SCtr adt' y ys)
  | adt == adt' = SCtr adt (ite b x y) (mergeList b xs ys)
  | otherwise   = error $
    "Type mismatch between data type constructors '"
    ++ x ++ "' and '" ++ y ++ "'\n\
    \Of types '" ++ adt ++ "' and '" ++ adt' ++ ", respectively."
merge b (SArgs   xs) (SArgs   ys) = SArgs $ mergeList b xs ys
merge _ x y = error $ "Type mismatch between symbolic values '"
              ++ show x ++ "' and '" ++ show y ++ "'"

mergeList :: SBool -> [SValue] -> [SValue] -> [SValue]
mergeList sb xs ys
  | Just b <- unliteral sb = if b then xs else ys
  | otherwise              = error $ "Unable to merge arguments '"
                             ++ show xs ++ "' with '" ++ show ys
                                ++ "'\n\
                             \Impossible to determine Boolean
                                condition."
```

Listing 5.9: The initial `SValue` instance of the `Mergeable` typeclass.

Now that we can get the bindings for a successful unification and we can merge `SValues`, we are ready to implement the translation of **case**-statements. The implementation is given in Listing 5.10.

```
translate :: RecursionDepth -> Term Type -> Formula SValue
translate depth (Case t0 ts _) =
  do sp <- translate depth t0
     translateBranches depth sp ts

translateBranches :: RecursionDepth
                  -> SValue -> [(Pattern Type, Term Type)]
                  -> Formula SValue
translateBranches _  _ [] = error "Non-exhaustive patterns in case
   statement."
translateBranches depth sv [(alt, body)] =
  if unifiable alt sv
    then do bs <- symbolicallyUnify alt sv
            local bs $ translate depth body
    else translateBranches depth sv []
translateBranches depth sv ((alt, body) : rest) =
  if unifiable alt sv
    then do bs    <- symbolicallyUnify alt sv
            alt'  <- local bs $ translatePattern alt
            cond  <- alt' `sEqual` sv
            body' <- local bs $ translate depth body
            next  <- translateBranches depth sv rest
            return $ merge (truthy cond) body' next
```

```
    else translateBranches depth sv rest
```

Listing 5.10: Translation of **case**-statements.

We note that `truthy` is a helper function that returns the SBV `SBool` wrapped inside an `SValue SBoolean`.

We begin by translating the selector term t_0 to a `SValue`. Next, we translate each branch with the function `translateBranches`. This we do by checking if the symbolic selector is *unifiable* with the case alternative pattern. In particular, the function `unifiable` returns a Boolean value, but does *not* constrain any variables or create bindings.

If the symbolic selector is not unifiable with the pattern, then we continue. If it is, then we symbolically unify the two and apply the bindings from the unification to the alternative pattern. This should mean all variables in the pattern are substituted by the appropriate symbolic variables. Next, we translate the *condition*, which returns `SBoolean (SBV True)` if the symbolic selector is equal to the symbolic alternative, where the unifier has now been applied. We translate the body of that alternative using the relevant bindings. Finally, we make a recursive call to `translateBranches` to translate the rest of the **case**-statement. If the condition is true, then we should select the body we evaluated with the current bindings. If not, then we should return another, nested conditional. Lastly, is there is only one branch left, we should unify the symbolic variables, apply the corresponding bindings to the body and translate it. If unification fails, then the **case**-statement contained non-exhaustive patterns and we signal an error.

Now, consider what would have happened if we had implemented symbolic unification in such a way that the symbolic values were *constrained* to be equal to their concrete counterparts by unification. Then, if we had had two branches that could have been selected, the symbolic unification of the first alternative with the symbolic variable would have constrained the symbolic variable in such a way that we would *always* select the first branch. Consider the following program.

```
(\*a -> (\*b ->
  case [*a, *b] of
    ; [3, y] -> 6
    ; [x, 5] -> 10
    ; [x, y] -> x + y))
```

`unifiable` would have determined that the symbolic integer variable `*a` could match on the first element in the first alternative, `3`, and the constraining implementation of `symbolicallyUnify` would have *constrained* `*a` to be equal to `3`. Then, when translating the second branch, this constraint would persist! This is clearly unsound, since the variable `x` in the second alternative is free. This is why `symbolicallyUnify` must refrain from imposing constraints on the symbolic variables and instead generate only bindings.

5.6.9 Handling Recursion

In the current implementation, Contra as a language can handle general recursion, but the property-checker cannot. This is because the translation algorithm has no heuristic to select branches that terminate. Instead, attempts to investigate branches with recursive function calls and therefore simply keeps unrolling the function definition until it reaches the maximum recursion depth. However, the current implementation would support an extension of the property-checker that utilised something like uninterpreted functions. In that case, it would likely also need a special symbolic variable type for uninterpreted

functions, in such a way that it could leave a function definition unrolled, but represented as a symbolic variable.

5.6.10 Translating Algebraic Data Types

Finally, we come to the principal part of the algorithm. We require a strategy for imposing constraints on symbolic algebraic data types, and to this end, we suggest an encoding of algebraic data types to enable constraint generation. In particular, we must be able to compare symbolic algebraic data types against concrete patterns.

When translating a *concrete* algebraic term from Contra, we know for certain which constructor has been used and which fields it has, and we can simply translate it by creating the corresponding `SCtr` with concrete strings and literal fields.

However, when we wish to *create* a symbolic variable of a user-defined algebraic data type, we do not know which constructor we should choose nor which fields it should have. Therefore, we require a different `SValue` to represent these symbolic, uninstantiated algebraic data type variables. In order to represent an arbitrary instance of a particular algebraic data type to the SMT solver, we must encode it with a combination of the symbolic variables that are available to us through SBV.

At this stage, we recall the bijection between the constructors of an algebraic data type and its index. Since we can enumerate the constructors and apply the bijection in either direction to find the *selector* of a data constructor or to *reconstruct* the data constructor from its selector, we can represent the particular instantiation of an algebraic data type using a symbolic integer.

SBV can choose the value of the symbolic integer to select a constructor, and we can compare concrete and symbolic constructors using the bijection.

The Naïve Implementation

Let us consider a naïve implementation of the `createSymbolic` case for algebraic data types and see why using this idea directly fails to produce algebraic data types.

Our desired outcome, is for SBV to symbolically select one of the constructors using a symbolic integer, in such a way that it can backtrack if a candidate must be discarded. Further, we need to be able to compare the created symbolic variable to concrete constructor terms, so we must be able to compare their fields.

Based on the number of constructors, we create a symbolic integer and constrain in to be within the range of possible constructor indices. In order to select a constructor, we use a nested conditional statement, where each branch matches the selector to one of the valid indices, effectively selecting the corresponding constructor. For each constructor we might select, we look up its fields and instantiate the corresponding symbolic variables. To avoid infinite recursion, we introduce a maximum recursion depth and throw an error if the recursion depth reaches zero.

```
createSymbolic :: RecursionDepth -> Pattern Type -> Formula SValue
createSymbolic 0     (Variable x (ADT adt)) = error $
  "Reached max. recursion depth while trying to generate symbolic
     ADT "
  ++ adt ++ "' for variable '" ++ x ++ "'"
createSymbolic depth (Variable x (ADT adt)) =
  do env  <- environment
     ctrs <- constructors env adt
     si   <- createSelector ctrs
```

```
      sCtr <- selectConstructor (depth - 1) adt si ctrs
      desc <- lift $ sString x
      lift $ constrain $ desc .== literal (show sCtr)
      return sCtr

createSelector :: [Constructor] -> Formula SInteger
createSelector ctrs =
  do si <- lift sInteger_
     let cardinality = literal $ toInteger $ length ctrs
     lift $ constrain $
       (si .>= 0) .&& (si .< cardinality)
     return si

selectConstructor :: RecursionDepth -> D -> SInteger ->
   [Constructor]
                  -> Formula SValue
selectConstructor 0     d _  _  = error $
  "Reached max. recursion depth while trying to \
  \create symbolic variable for ADT '" ++ d ++ "'"
selectConstructor _     d _  [] = error $
  "Fatal: Failed to create symbolic variable for ADT '" ++ d ++ "'"
selectConstructor depth d _  [Constructor c types] =
  do let names = zipWith (\tau i -> show (hash (d ++ show tau)) ++
     show i)
                 types
                 ([0..] :: [Integer])
     let fields = zipWith Variable names types
     sFields <- mapM (createSymbolic depth) fields
     return $ SCtr d c sFields
selectConstructor depth d si ((Constructor c types) : ctrs) =
  do env   <- environment
     sel   <- selector env d c
     let names = zipWith (\tau i -> show (hash (d ++ show tau)) ++
        show i)
                 types
                 ([0..] :: [Integer])
     let fields = zipWith Variable names types
     sFields <- mapM (createSymbolic depth) fields
     next  <- selectConstructor depth d si ctrs
     return $ merge (si .== literal sel) (SCtr d c sFields) next
```

Listing 5.11: The naïve encoding of algebraic data types

However, as it stands, this implementation will always reach maximum recursion depth for recursive algebraic data types. This is because it has no way of knowing when to choose a constructor with recursive fields and not. If we wanted to ensure termination, we would have to disable the selection of constructors with recursive fields at a given threshold. However, this cutoff would be completely arbitrary, since we do not know what kind of algebraic data type the user might like to define.

Instead, we pivot and choose a different strategy to selecting constructors.

The Final Implementation of Symbolic Algebraic Data Types

We still make use of the bijection between constructors and indices, but in this strategy, we leave the list of fields *empty*. This may seem bizarre. Consider, however, that we never need

to know the values of the *fields* of a symbolic algebraic data type except when it is *compared* to another variable. This means that we can *delay* the selection of a constructor and the instantiation its fields until we have a concrete value to compare it to.

We therefore select the constructor using a symbolic integer, as before, but we define an `SValue` that leaves more of the information uninterpreted. We call this new symbolic variable `SADT`, to distinguish it from `SCtr`, which represents a literal value. The final definition of `SValue` can be found in Listing 5.12.

```
data SValue =
    SUnit
  | SBoolean SBool
  | SNumber  SInteger
  | SArgs    [SValue]
  | SCtr     D C [SValue]
  | SADT     X D SInteger [SValue]
  deriving Show
```

Listing 5.12: The complete definition of `SValue`.

We include the original variable name of the term as an identifier (`X`) in order to fabricate consistent names for its fields. We are now ready to implement `createSymbolic` for arbitrary algebraic data types. The implementation is given in Listing 5.13.

```
createSymbolic (Variable x (ADT adt)) =
  do let ident = x ++ "$" ++ adt
     env   <- environment
     si    <- lift $ sInteger ident
     upper <- cardinality env adt
     if upper == 1
       then do lift $ constrain $ si .== 0
               (_, c) <- reconstruct env (adt, 0)
               types  <- fieldTypes env c
               svs    <- ensureInstantiated ident [] types
               return $ SADT ident adt si svs
       else do lift $ constrain $ (si .>= 0) .&& (si .< literal
          upper)
               return $ SADT ident adt si []
```

Listing 5.13: The `createSymbolic` case for algebraic data types.

We retrieve the environment to collect information from the program text. First, we find the upper bound of the constructor indices using `cardinality` from the environment. Next we create a selector and name it "<x>$<ADT>", where "<x>" is the original variable name and "<ADT>" is the name of the algebraic data type. We use the dollar sign because it is an otherwise illegal character in variable names and therefore cannot shadow an existing name In the general case, we constrain the selector to be within the range of possible constructor indices. Finally, we return the `SADT`.

The fields of the algebraic data type are instantiated on a on-demand basis when the symbolic variable is compared to an `SCtr` using `sEquals` or symbolically unified against one. We define the cases for comparing two `SADT`s and for comparing an `SADT` and a `SCtr`. The implementation is given in Listing 5.14. Likewise, we define the case for symbolically unifying a concrete constructor pattern against a symbolic `SADT`. The implementation is given in Listing 5.15.

```
sEqual :: SValue -> SValue -> Formula SValue
sEqual (SADT x adt si xs) (SADT y adt' sj ys) =
  do eqs <- zipWithM sEqual xs ys
     return $ SBoolean $ sAnd $
         fromBool (adt == adt')
       : (si .== sj)
       : map truthy eqs
sEqual (SCtr adt c  xs) (SADT ident adt' sj ys)
  | adt == adt' = coerce ident adt (c, sj) (xs, ys)
  | otherwise   = return $ SBoolean sFalse
sEqual (SADT ident adt si xs) (SCtr adt' c  ys)
  | adt == adt' = coerce ident adt (c, si) (ys, xs)
  | otherwise   = return $ SBoolean sFalse
```

Listing 5.14: The cases for comparing SADTs to each other and to SCtrs.

```
sUnify (PConstructor c ps (ADT adt)) (SADT ident adt' si svs)
  | adt == adt' =
      do env    <- environment
         (_, i) <- selector env (adt, c)
         lift $ constrain $ si .== literal i
         types  <- fieldTypes env c
         svs'   <- ensureInstantiated ident svs types
         foldrM (\(p, sv) u -> do u' <- sUnify p sv
                                  return $ u <> u'
                ) mempty $ zip ps svs'
```

Listing 5.15: The symbolic unification of a concrete constructor pattern against an SADT.

In the particular case where we create a symbolic variable for an algebraic data type with exactly one constructor, we instantiate the fields right away. This is a necessary step, as otherwise, it would be impossible to find a counterexample to such trivially untrue properties as the following.

```
adt Example = C Integer .

falsifiable :: Example -> Example -> Boolean .
falsifiable x y =*= x == y .
```

Clearly, this property does *not* hold for the algebraic data type `Example`, since the integer fields can be different. However, this is never checked unless they are instantiated upfront. The only way this could cause infinite recursion, were if the type itself were infinitely recursive. On the assumption that users would not be interested in writing infinitely self-recursive algebraic data types, we do not guard against this kind of recursion.

The helper functions `coerce`, `ensureInstantiated`, and `instantiate` are given in Listing 5.16. The function `ensureTypeAccord` is omitted for brevity, as it simply checks type accord between `SValues` and concrete `Types` in the expected way.

`coerce` coerces a symbolic variable `SADT` to be equal to a concrete constructor value `SCtr`. It does this by constraining the symbolic selector to be equal to the corresponding, literal selector of the concrete value. Next, it ensures that all symbolic fields have been instantiated. If they have, it ensures type accord between the symbolic and the concrete fields and constrains them pair-wise. If not, it instantiates them.

`ensureInstantiated` is also used by the first call to `createSymbolic` in the case where there is exactly one constructor.

`instantiate` fabricates variable names by joining the name of the algebraic data type and the original identifier of the algebraic data type variable with the keyword "$field" and finally the index of the field, starting from 0. This way, the assignment to variables, which is given by integers indicating the constructor selectors for algebraic data types and regular symbolic variables otherwise, is human-readble and can be parsed into a pretty-printed result.

```
coerce :: X -> D -> (C, SInteger) -> ([SValue], [SValue]) ->
   Formula SValue
coerce ident adt (c, si) (xs, ys) =
  do env    <- environment
     (_, i) <- selector env (adt, c)
     lift $ constrain $ si .== literal i
     types  <- fieldTypes env c
     ys'    <- ensureInstantiated ident ys types
     eqs    <- zipWithM sEqual xs ys'
     return $ SBoolean $ sAnd $
         (si .== literal i)
       : map truthy eqs


ensureInstantiated :: X -> [SValue] -> [Type] -> Formula [SValue]
ensureInstantiated _     [ ] [   ] = return []
ensureInstantiated ident [ ] types = instantiate ident types
ensureInstantiated _     svs types = ensureTypeAccord svs types >>
   return svs

instantiate :: X -> [Type] -> Formula [SValue]
instantiate ident types =
  do let names = map (((ident ++ "$field") ++) . show) ([0..] ::
     [Int])
     let vars  = zipWith Variable names types
     mapM createSymbolic vars
```

Listing 5.16: Helper functions to create symbolic algebraic data types.

An informal, diagrammatic representation of the steps to create a symbolic algebraic data type variable and to instantiate its fields at constraint generation time is given in Figure 5.3.

The complete implementation, including the cases for merging `SADTs` and `SCtrs`, can be found in Appendix A.

5.7 Test Suite

The test suite can be found in the GitHub repository for the prototype implementation. It requires the Haskell build tool Stack and instructions for running the test suite can be found in the file `README.md` available in the repository. The automatic test suite can be found in the repository at `contra/test`. It includes basic tests of Contra's base modules, such as the parser, unification algorithm, and partial evaluator. It also includes two test files covering the property-checker, namely `PropertyCheckerTests` and `Benchmark`.

In order to isolate the property-checker from the parser, type-inferrer, and partial evaluator, `PropertyCheckerTests` perform a series of tests with concrete examples of already-inlined and typed properties. The test suite verifies that the property-checker can find counterexamples where expected, and that theorems (properties true for all inputs) are reported as such.

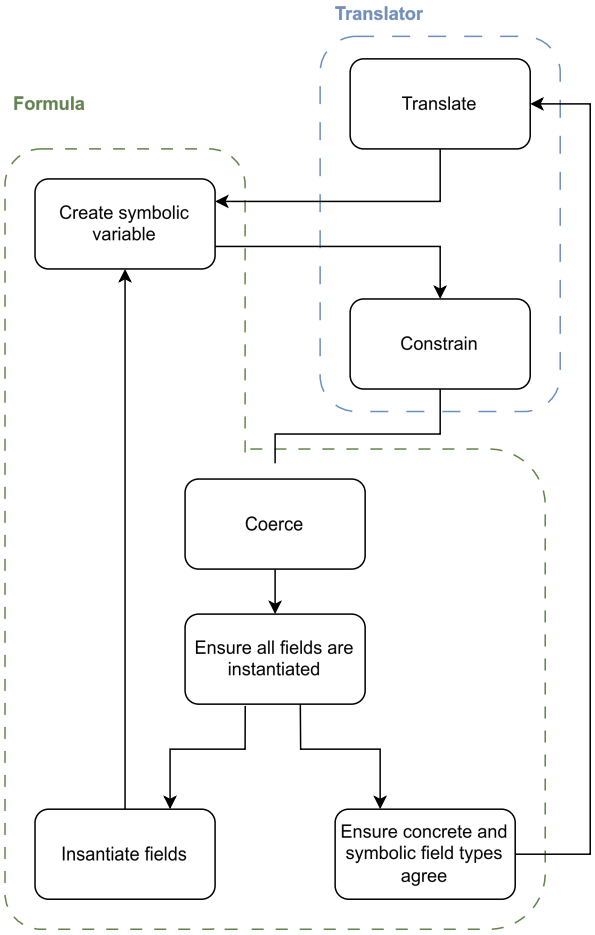

Figure 5.3: Abstract steps to create a symbolic algebraic data type variable and instantiate its fields.

However, by writing Contra programs by hand and checking these concrete examples, we cannot justify a great sense of program correctness. After all, we may also fall victim to biases that lead us to test only well-functioning examples. Moreover, the manually written tests can only cover a fixed amount of cases. This is why we have also implemented the `Benchmark`. It uses QuickCheck to generate arbitrary values and insert them into Contra properties as counterexamples, then validates the response from Contra's property-checker to confirm that it returned the expected counterexample. We discuss this further in Chapter 6.

Besides automatic test, a number of example programs are provided in the directory `contra/examples`. The parent directory contains files that are suitable for the user to experiment with. The sub-directory `contra/examples/recursive` contains files with recursive function calls that can be executed and partially interpreted, but whose properties cannot be checked in the current implementation. These are nonetheless interesting. The sub-directory `contra/examples/test` that contain small, concrete examples designed to test general features. Finally, the sub-directory `contra/examples/errors` contains invalid files for testing purposes.

With the Contra executable built and installed (via Stack), the user can check the properties of a program with the following command.

```
> contra --check <program-name>.con
```

Then, all the properties defined in the program text will be checked, and the result for each one will be printed. The printout states the result of the check, which is "OK", "FAIL" or "Unknown". In case of a "FAIL", the counterexample is printed. A counterexample consists of a list of each variable generated by the property-checker, given by its name, its value, and its type.

Chapter 6

Evaluation

In this chapter, we evaluate Contra to determine to what degree it accomplishes the goals we set for it in the introductory chapter. To recapitulate, Contra aims to be a conventional functional programming language with support for properties as first-class language construct and a property-checker capable of producing counterexamples of user-defined algebraic data types. The property-checker must be sound — i.e., all reported counterexamples must make the property fail — and automatic.

We evaluate Contra in two stages. In the first stage, we verify that the built-in property-checker can produce valid counterexamples to hand-written properties, using simple built-in types, algebraic data types, and mutually recursive algebraic data types. In the last stage, we demonstrating its ability to find counterexamples to properties with QuickCheck-generated counterexamples. We have QuickCheck generate integer, Boolean, algebraic data types, and mutually recursive algebraic data type values. In the process, we compare the complexity of the Contra programs and their corresponding Haskell programs.

6.1 Hand-Written Non-Algebraic Input Types

Writing simple arithmetic properties in Contra is trivial. We define the following syntactic sugar.

```
and :: Boolean -> Boolean -> Boolean .
and True True = True .
and x    y    = False .

or :: Boolean -> Boolean -> Boolean .
or True y     = True  .
or x    True  = True  .
or x    y     = False .
```

We can for instance test the properties commutativity, associativity, and closure under equality of the addition operator.

```
addCommutative x y     =*= (x + y) == (y + x) .
addAssociative x y z   =*= ((x + y) + z) == (x + (y + z)) .
addClosed x y xx yy =*=
  if and (x == xx) (y == yy)
     then (x + y) == (xx + yy)
     else True .
```

```
Checking 'addCommutative' >>>  OK
Checking 'addAssociative' >>>  OK
Checking 'addCongruent'   >>>  OK
```

Similarly, we can define the the same properties for the subtraction operator. Unsurprisingly, this returns counterexamples for commutativity and associativity, but not for closure under equality.

```
subCommutative x y   =*= (x - y) == (y - x) .
subAssociative x y z =*= ((x - y) - z) == (x - (y - z)) .
subClosed x y xx yy =*=
  if and (x == xx) (y == yy)
     then (x - y) == (xx - yy)
     else True .
```

```
Checking 'subCommutative' >>>  FAIL
Counterexample:
  a = 0 :: Integer
  b = 1 :: Integer

Checking 'subAssociative' >>>  FAIL
Counterexample:
  a = 0 :: Integer
  b = 0 :: Integer
  c = 1 :: Integer

Checking 'subCongruent' >>>  OK
```

We can easily check simple logical theorems, such as the variations of De Morgan's laws and the law of the excluded middle. These are all proven true by Contra.

```
-- De Morgan's Laws
negatedDisjunction :: Boolean -> Boolean -> Boolean .
negatedDisjunction p q =*= (not (or p q)) == (and (not p) (not q)) .

negatedConjunction :: Boolean -> Boolean -> Boolean .
negatedConjunction p q =*= (not (and p q)) == (or (not p) (not q)) .

-- Substitution forms
substitutionConjunction :: Boolean -> Boolean -> Boolean .
substitutionConjunction p q =*= (and p q) == (not (or (not p) (not q))) .

substitutionDisjunction :: Boolean -> Boolean -> Boolean .
substitutionDisjunction p q =*= (or p q) == (not (and (not p) (not q))) .

-- Law of the excluded middle
exMiddle :: Boolean -> Boolean .
exMiddle p =*= (or p (not p)) .
```

6.2 Hand-Written Algebraic Input Types

Because it does not support recursive function calls, the cases for using Contra over QuickCheck and similar tools, are primarily those properties for which we wish to test the application of a non-recursive function on an arbitrary instance of an algebraic data type.

In Contra, we can define an algebraic data type for Cartesian products and some properties of this type in the following way.

```
adt Cartesian = Product Integer Integer .

-- Always true
identity :: Cartesian -> Boolean .
identity x =*= x == x .

commutativity (Product {x, y}) =*= (x + y) == (y + x) .

associativity (Product {x, y}) z =*= ((x + y) + z) == (x + (y + z)) .

-- Falsifiable
different :: Cartesian -> Cartesian -> Boolean .
different x y =*= x == y .

subCommutativity (Product {x, y}) =*= (x - y) == (y - x) .

subAssociativity (Product {x, y}) z =*= ((x - y) - z) == (x - (y - z)) .

-- Closure Under Equality
cartesianClosed :: Cartesian -> Cartesian -> Boolean .
cartesianClosed x y =*=
  case (x == y) of
    ; True  -> let (Product {x1, y1}) = x in
               let (Product {x2, y2}) = y in
                   (and (x1 == x2) (y1 == y2))
    ; False -> True.
```

```
Checking 'identity' >>>  OK
Checking 'commutativity' >>>  OK
Checking 'associativity' >>>  OK
Checking 'different' >>>  FAIL
Counterexample:
  x = Product {1, 0}  :: Cartesian
  y = Product {0, 0}  :: Cartesian

Checking 'subCommutativity' >>>  FAIL
Counterexample:
  x = 0 :: Integer
  y = 1 :: Integer

Checking 'subAssociativity' >>>  FAIL
Counterexample:
  x = 0 :: Integer
  y = 0 :: Integer
  z = 1 :: Integer

Checking 'cartesianClosed' >>>  OK
Checking 'sameProjections' >>>  FAIL
Counterexample:
  x = 0 :: Integer
  y = 1 :: Integer
```

6.3 Mutually Recursive Algebraic Data Types

Contra is capable of finding counterexamples of mutually recursive algebraic data types.

```
adt A = Zero | One B .
adt B = Null | Two A .

falsifiableA :: A -> Boolean .
falsifiableA a =*=
  case a of
    ; One {Two {One {Two {Zero}}}} -> False
    ; x -> True .

falsifiableB :: B -> Boolean .
falsifiableB b =*=
  case b of
    ; Two {One {Two {One {Null}}}} -> False
    ; x -> True .

Checking 'falsifiableA' >>>  FAIL
Counterexample:
  a = One {Two {One {Two {Zero {}}}}}  :: A

Checking 'falsifiableB' >>>  FAIL
Counterexample:
  b = Two {One {Two {One {Null {}}}}}  :: B
```

This is possible even for complex, mutually recursive algebraic data types.

```
adt X = Stop | XY Y | XZ Z | XW W .

adt Y = YY Boolean Boolean X .
adt Z = ZZ Boolean         X .
adt W = WW Integer         X .

falsifiableX :: X -> Boolean .
falsifiableX x =*=
  case x of
    ; XY {YY {True, False, XZ {ZZ {True, XW {WW {5, Stop}}}}}} -> False
    ; y -> True .

Checking 'falsifiableX' >>>  FAIL
Counterexample:
  x = XY {YY {True, False, XZ {ZZ {True, XW {WW {5, Stop {}}}}}}}  :: X
```

6.4 QuickCheck-Generated Algebraic Data Types

While the above results are promising, it is difficult to justify high confidence in the correctness of our implementation without further inspection. The `Benchmark` test file generates instances of algebraic data types and inserts them in the Contra program instead of the concrete examples we have seen up until now. In other words, use the same structure of a **case**-statement, but substitute the concrete term that makes the program fail, with one QuickCheck has generated for us. All the programs we will see in this section, both the Contra and the Haskell variants, are available as standalone files in the directory `contra/benchmark` for more comfortable reading.

The first part of the `Benchmark` generates arbitrary integer and Boolean values. The second part generates mutually recursive `A`'s and `B`'s. The third and final part generates the

complex mutually recursive algebraic data types `X`, `Y`, `Z`, and `W`. We will omit the first section. While it is crucial to validate our approach, it not particularly interesting to demonstrate.

We note that the version of QuickCheck used in the following examples is from the package `Test.Tasty.QuickCheck`, version 0.10 [7].

Let us consider the QuickCheck implementations of the two programs above, starting with the program generating `A`'s and `B`'s. The implementation of the generators is given in Listing 6.1.

```
import Test.Tasty.QuickCheck

data A = Zero | One B deriving (Show, Eq)
data B = Null | Two A deriving (Show, Eq)

instance Arbitrary A where
    arbitrary = sized genA

genA :: Int -> Gen A
genA 0 = elements [Zero]
genA n = oneof [return Zero, One <$> genB (n `div` 2)]

instance Arbitrary B where
    arbitrary = sized genB

genB :: Int -> Gen B
genB 0 = elements [Null]
genB n = oneof [return Null, Two <$> genA (n `div` 2)]
```

Listing 6.1: QuickCheck generators for mutually recursive algebraic data types `A` and `B`.

We can use these generator to fabricate Contra programs, and use Contra's built-in property-checker to validate the result. Indeed, QuickCheck confirms that Contra was able to identify all of the 100 generated `A`'s and `B`'s.

```
Contra can find QuickCheck-generated counterexamples of mutually recursive ADTs:
  QuickCheck-generated A's. :                                    OK (1.75s)
    +++ OK, passed 100 tests.
  QuickCheck-generated B's. :                                    OK (1.72s)
    +++ OK, passed 100 tests.
```

Next, we consider the generators for `X`, `Y`, `Z`, and `W` instances. The implementation is given in Listing 6.2.

```
import Test.Tasty.QuickCheck

data X = Stop | XY Y | XZ Z | XW W deriving (Show, Eq)
data Y = YY Bool Bool X            deriving (Show, Eq)
data Z = ZZ Bool      X            deriving (Show, Eq)
data W = WW Integer   X            deriving (Show, Eq)

instance Arbitrary X where
    arbitrary = sized genX

genX :: Int -> Gen X
genX 0 = elements [Stop]
```

```
genX n = oneof
  [ return Stop
  , XY <$> genY (n `div` 3)
  , XZ <$> genZ (n `div` 3)
  , XW <$> genW (n `div` 3)
  ]

instance Arbitrary Y where
    arbitrary = genY 0

genY :: Int -> Gen Y
genY _ = YY <$> arbitrary <*> arbitrary <*> arbitrary

instance Arbitrary Z where
    arbitrary = genZ 0

genZ :: Int -> Gen Z
genZ _ = ZZ <$> arbitrary <*> arbitrary

instance Arbitrary W where
    arbitrary = genW 0

genW :: Int -> Gen W
genW _ = WW <$> arbitrary <*> arbitrary
```

Listing 6.2: QuickCheck generators for mutually recursive algebraic data types `X`, `Y`, `Z`, and `W`.

Again, QuickCheck confirms that Contra was able to produce the correct counterexample in 100 cases.

```
Contra can find QuickCheck-generated counterexamples of mutually recursive ADTs:
  ...
  QuickCheck-generated X's. :                                OK (2.15s)
    +++ OK, passed 100 tests.
```

By comparing the Contra programs to their counterparts in Haskell, using QuickCheck, it is immediately obvious that Contra is much simpler. The QuickCheck generators are not very complicated, but they do introduce non-native syntax and writing one is non-trivial.

Chapter 7

Discussion

In this chapter, we reflect on the results of the evaluation and the degree to which we believe we have achieved the goals we defined for Contra. We compare our approach and our results to existing tools, and we re-evaluate our design choices.

7.1 Comparing Results to Initial Goals

In the introductory chapter, we set the following goals for Contra:

- Ability to generate user-defined algebraic data types from their definition alone.
- Automatic property-checker.
- Sound property-checker.
- Support properties as a first-class language construct.
- Familiar, ML-style syntax.

In terms of the programming language itself, we initially wanted to design and implement a small, functional programming language with conventional syntax and mostly conventional language constructs, because we believed this would facilitate its adoption by existing (functional) programmers. At the same time, we sought to abstract away the notion of generators completely, in such a way that the end user would not need to be aware of the concept at all. Finally, we wanted properties to be a native language construct — what we could characterise as "first-class" properties.

We consider this goal to be accomplished. The average user does not need to know any implementation details or make any special considerations when writing a Contra program, and in particular, properties. Properties are defines as Boolean functions, and are simply marked for automatic property-checking using the syntactic identifier `=*=`. Besides this special construct, Contra programs are concise and conventional. Contra does not require the user to familiarise themself with any exotic language features, relying instead on plain Boolean functions.

As we saw in the last section of the evaluation, Section 6.4, Contra programs are significantly less complex than their counterparts in Haskell, even using a conventional and ergonomic tool such as QuickCheck. We can especially notice how long the Haskell-QuickCheck program generating `X`, `Y`, `Z`, and `W` types is in comparison to the same Contra program. Because the property-checker can look up algebraic data type definitions in the program text, we can remove all the lines of generators and `Arbitrary` instances from the

Haskell program, and what remains is (modulo syntactic differences) sufficient to produce the equivalent Contra program.

Meanwhile, property-checker we have implemented is both automatic and capable of producing counterexamples to properties containing user-defined algebraic data types, which it can do based on the built-in language constructs and the program text alone. Besides the definition of the algebraic data type, no further input is required from the user. The property-checker preserves the semantics of the concrete terms it translates, and we have in our testing only found valid counterexamples, even for mutually recursive algebraic data types.

Using an SMT solver is akin to model-checking and attempts to prove a given formula for *all* possible inputs. Where the problem is decidable, we therefore consider the property-checker a superior alternative to hand-written unit tests, since it relieves the programmer of the burden of inventing examples and can in some cases prove definitively that a property holds generally.

7.2 Limitations

Compared to other tools and frameworks, Contra's greatest advantage is its ease of use. On the other hand, we can clearly identify limitations to our approach. In this section, we identify potential for improvement, comparing Contra to existing programming languages and property-based testing tools.

7.2.1 Termination & Recursion

Firstly, none of the interpreter, partial evaluator, or property-checker have termination checks. Therefore, it is up to the user to manually interrupt non-terminating program executions. We propose the idea of implementing a termination checker in the style of Agda, which builds on the terminal checker in Foetus [1]. The termination checker defined a *dependency relation* on the arguments to functions and checking for each call that the dependency order decreases. In other words, it statically checks if the arguments to each subsequent function call becomes *smaller*, and therefore monotonically tends towards termination.

A more ambitious — and more interesting — approach might be to utilise a supercompiler rather than a partial evaluator [4]. Supercompilation is similar to partial evaluation, but is significantly more advanced. A supercompiler performs several passes and can simplify or specialise the program in broader ways than is possible by partial evaluation. It can handle loop fusion, inlining, redundancy elimination, and recursion. We imagine a tool using supercompilation would be more efficient than Contra.

From a user's point of view, perhaps the greatest limitation in Contra is the lack of support for recursive function calls in properties. Modern-day SMT solvers have support for uninterpreted functions, and we see potential in registering recursive function definitions as uninterpreted SMT functions.

Although they are more complex than Contra, tools such as TARGET and QuickCheck account for termination when generating the *input*, not while handling the user's *function definitions*. As long as the functions terminate in practice, these tools can generate inputs for properties which call recursive functions. Contra on the other hand, does not impose a bound on the recursion depth of the *input*, but instead it cannot handle recursion in the users' *function definitions*.

7.2.2 Language-Dependent Implementation

We may also question the "tagging" approach we used in our implementation to determine the selector for a given data type constructor, and thus to encode algebraic data types. We relied directly on the ordered nature of lists in Haskell, and we can criticise this choice as opaque and possibly unsafe. Instead, we might imagine, since each constructor is required to be unique in the program text, that we utilise something like annotation variables to make a pass through the program text and tag each constructor with a unique index. This way, we would not have to rely on the (string) equality of the names of the algebraic data types or the particular implementation language, and we could have used the index directly, with the guarantee that each index be unique.

7.2.3 Type System & Polymorphism

Contra lacks both general type polymorphism for functions and parameterised algebraic data types. We imagine neither of these would be particularly difficult to implement, even using the current implementation as a starting point. As a first step, we propose implementing template polymorphism, also known as "ad-hoc polymorphism", by which the user writes a single algebraic data type definition or function signature using a polymorphic type variable, and program analysis fabricates variations of both types and functions specialised to concrete types. In other words, several definitions are created under the hood based on the "template" definition, which contains a type variable indicating where concrete types should be instantiated and placed in the body of the definition.

Moreover, the prototype REPL does not respect type conformity during partial evaluation, only at the point of complete evaluation. This means the user can produce the specialised, and *ill-typed*, function. For instance one that adds a Boolean value to an integer variable. Only when applying this function to another term, do we produce a type error. This has not been the focus of the thesis and has therefore been neglected, but we recognise that the implementation is lacklustre and its utility is lessened by this fact. However, we still believe a REPL can aid understanding and enable ad-lib experimentation. For this reason, it is included in the implementation.

Other languages and tools we have seen, such as QuickCheck and TARGET, are embedded in Haskell and have support for polymorphic function definitions in the general program text. However, each property must still be defined using concrete types. We can argue that writing generators is more difficult than writing different, non-polymorphic versions of the same function and that Contra therefore remains easier to use. At the same time, writing several concrete functions for each new type leads to an explosion in code size and creates several program fragments that must thereafter be kept in sync.

7.2.4 Partial Evaluator

The partial evaluator may not have proven as useful as we had hoped. Since Contra is built on the lambda calculus and only specialises functions one argument at a time, unlike Cook & Lämmel's language, for instance, a function cannot be specialised and therefore not inlined if its first argument is a non-canonical term. I.e., given a function called `plus` which is a simple wrapper around the built-in addition operator, we *can* specialise and inline the first function call to `plus` but *not* on the second.

```
specialisable :: Integer -> (Integer -> Integer) .
specialisable x = plus 1 x .
```

```
unspecialisable :: Integer -> (Integer -> Integer) .
unspecialisable x = plus x 1.
```

Without support for recursive functions, it is tricky to evaluate the utility of partial evaluation on larger program texts and the benefit of execution path elimination, however, where it might prove more useful.

Given that the branches in a **case**-statement forms a set and we can construct the set of all possible pattern-matches on a selector term, it is possible to statically determine whether or not a **case**-statement contains exhaustive patterns. Such static analysis would alert the user early about incomplete functions and possibly avoid infinite loops in the property-checker. We believe this would be a beneficial and worthwhile addition to the language.

7.2.5 Restricted Counterexamples

As we could see in the evaluation, the SMT solver under the hood does indeed find the simplest possible counterexample. The benefit of this is that the example is already minimal and does not require further shrinking. The disadvantage is that the property-checker will produce this example every time. If the programmer cannot identify the issue from the given counterexample, Contra cannot at present produce any other counterexamples than these.

Tools like TARGET prove that it is possible to *force* the SMT solver to produce different counterexamples. This is an interesting idea to pursue in the further development of Contra. Other tools, such as Luck, allow the user to specify what inputs are valid and which subsets we might like to generate, whereas Contra attempts to generate *failing* input under the specified constraints imposed on a value. All test results are therefore interesting, but the quality of the counterexamples suffers.

Other domain-specific languages, such as Luck, provide fine-grained control over distribution, allowing the user to tune the generators to produce interesting *and* correct inputs. Moreover, Luck in particular allows the user influence over the internal processes of the generator, such as explicit instantiation and sub-generator sampling, in a way that is more ergonomic than QuickCheck and more expressive than Contra. It does, on the other hand, require the user to learn this programming language with its domain-specific syntax and semantics.

Finally, Contra is incapable of generating arbitrary functions as input to properties. Incorporating the generation of arbitrary functions does introduce several questions to contend with, relating to the "interestingness" and relevance of various terms. At the same time, the generation of input functions enables property-based testing of a whole new class of functions, namely higher-order functions which take other functions as arguments, such as functions which traverse structures and apply a function to each element.

Chapter 8

Conclusion

In this concluding chapter, we reflect on our work. From the discussion of limitations in Section 7.2, we suggest some future work to extend or amend aspects of Contra. Finally, we leave the reader with some concluding remarks.

8.1 Automatically Generated Algebraic Counterexamples

In this thesis, we have designed and implemented a small functional programming language with first-class properties, capable of producing simple counterexamples to property-based tests with user-defined algebraic data types without the need for hand-written generators.

To achieve this, we have defined an encoding of algebraic data type variables as combinations of symbolic variables, and demonstrated that this encoding preserves the semantics of the algebraic data type in question. We formalised this notion in Section 3.5. In Section 5.6.10, we showed how it could be implemented in such a way that we were able to generate user-defined algebraic data types of unbounded recursion depth — including mutually recursive definitions. In order to preserve the semantics of algebraic data types encoded as symbolic variables, we implemented a custom equality relation which both enforced and validated constraints on our custom symbolic types.

In Section 6.4, we were able to demonstrate definitively that Contra is as capable as QuickCheck of generating both built-in and algebraic data types, but that Contra programs are significantly shorter and less complex than their counterparts. This is because properties are a first-class construct in Contra and all definable generators can be derived automatically from the algebraic data type definitions in the program text, removing the need for the user to consider generators at all.

Throughout, we have maintained a familiar syntax and semantics for regular program execution and introduces what we believe to be minimal extensions to the core language in order to define and check properties automatically. The resulting language is completely conventional in all but its property-checking features.

8.2 Future Work

From the limitations we have discussed, we consider the following changes to the implementation to be the most crucial.

Firstly, to implement support for uninterpreted functions. This we believe could enable Contra to find counterexamples to a restricted class of properties containing recursive function calls, which would greatly improve its utility to the end user.

Secondly, to extend Contra's type system with a form of polymorphism. The most significant benefit would be that the user can define parametric algebraic data types with polymorphic type variables. Such a type system could be implemented using template polymorphism, and a majority of the current implementation could stay as-is.

Finally, we would like to check for termination and exhaustion. A termination checker in the style of Agda and Foetus would be possible to implement in Contra without alteration to the language design. Similarly, a static check for exhaustive patterns in **case**-statements could be implemented using the existing unification machinery. Both of these extensions to the language would alert the user at an earlier stage of a faulty specification of a function or property. In the prototype implementation, recursive functions simply lead to infinite loops and non-exhaustive pattern-matches are not reported until translation time.

8.3 Concluding Remarks

Although we can identify room for improvement in terms of the expressiveness of the language and the quality of the counterexamples produced by our approach, we hope the accessibility of the language inspires use. In the future, we see many opportunities to amend these short-comings, for instance by extending Contra with polymorphic type variables and parameterised algebraic data types, or by bettering support for recursive functions.

Through Contra, however, we have been able to demonstrate how SMT solvers can be used to automatically generate well-typed, well-formed, and valid counterexamples of user-defined algebraic data types, including (mutually) recursive ones. We were able to achieve this without introducing any non-conventional language features that the programmer must familiarise themself with. We see this as a step towards fully automated property-based testing, and we anticipate further work in this field.

As both the complexity and the prevalence of software grows, software validation must becomes increasingly attractive also. We strongly believe the effort to facilitate testing is a worthwhile one, and that property-based testing is a particularly promising candidate. While Contra does not have the most expressive core language or the most potent property-checker, we believe it is difficult to surpass it in terms of user-friendliness, conciseness and accessibility.

Appendices

Appendix A

Appendix A - The Formula

```
{-# LANGUAGE FlexibleContexts, ScopedTypeVariables, TypeOperators
    #-}

module Validation.Formula where

import Core.Syntax
import Environment.Environment
import Environment.ERSymbolic

import Data.SBV
import Data.Hashable (hash)
import Control.Monad (zipWithM)


-- * Maximum recursion depth for function calls
type RecursionDepth = Int

defaultRecDepth :: RecursionDepth
defaultRecDepth = 20


-- * Custom symbolic variables
data SValue =
    SUnit
  | SBoolean SBool
  | SNumber  SInteger
  | SCtr     D C [SValue]
  | SADT     X D SInteger [SValue]
  | SArgs    [SValue]
  -- SArgs represents the fabricated argument list we create when
  -- flattening function definitions into a Case-statement
  deriving Show


-- * The Formula monad
type Bindings   = Mapping X SValue
type Formula  a = ERSymbolic Type Bindings a

bind :: X -> SValue -> X `MapsTo` SValue
bind x tau look y =
  if x == y      -- Applying the bindings to some 'y' equal to 'x'
```

```
      then tau    -- you should now get back 'tau'
      else look y -- If you call it with some other 'y',
                  -- then return the old binding for 'y'


-- * Create symbolic variables for SBV to instantiate during
    solving
createSymbolic :: Pattern Type -> Formula SValue
createSymbolic (Variable _ Unit')    = return SUnit
createSymbolic (Variable x Integer') =
  do sx <- lift $ sInteger x
     return $ SNumber sx
createSymbolic (Variable x Boolean') =
  do sx <- lift $ sBool x
     return $ SBoolean sx
createSymbolic (Variable x (Variable' _)) =
  do sx <- lift $ free x
     return $ SNumber sx
createSymbolic (Variable _ (TypeList [])) =
  do return $ SArgs []
createSymbolic (Variable x (TypeList ts)) =
     -- We should never be asked to create input for this type
        (internal only)
  do let names = zipWith (\tau i -> show (hash (x ++ show tau)) ++
     show i)
                  ts
                  ([0..] :: [Integer])
     let ps    = zipWith Variable names ts
     sxs <- mapM createSymbolic ps
     return $ SArgs sxs
createSymbolic (Variable x (ADT adt)) =
  do let ident = x ++ "$" ++ adt
     env   <- environment
     si    <- lift $ sInteger ident
     upper <- cardinality env adt
     if upper == 1
       then do lift $ constrain $ si .== 0
               (_, c) <- reconstruct env (adt, 0)
               types  <- fieldTypes env c
               svs    <- ensureInstantiated ident [] types
               return $ SADT ident adt si svs
       else do lift $ constrain $ (si .>= 0) .&& (si .< literal
          upper)
               return $ SADT ident adt si []
createSymbolic p = error $
     "Unexpected request to create symbolic sub-pattern '"
  ++ show p ++ "' of type '" ++ show (annotation p) ++ "'"
  ++ "\nPlease note that generating arbitrary functions is not
     supported."


-- * SValue (symbolic) equality
sEqual :: SValue -> SValue -> Formula SValue
sEqual  SUnit           SUnit           = return $ SBoolean sTrue
sEqual (SBoolean    b) (SBoolean     c) = return $ SBoolean (b .==
   c)
```

```
sEqual (SNumber     n) (SNumber      m) = return $ SBoolean (n .==
   m)
sEqual (SCtr adt x xs) (SCtr adt' y ys) =
  do eqs <- zipWithM sEqual xs ys
     return $ SBoolean $ sAnd $
         fromBool (adt == adt')
       : fromBool (x   == y   )
       : map truthy eqs
sEqual (SADT _ adt si xs) (SADT _ adt' sj ys) =
  do eqs <- zipWithM sEqual xs ys
     return $ SBoolean $ sAnd $
         fromBool (adt == adt')
       : (si .== sj)
       : map truthy eqs
sEqual (SCtr adt c  xs) (SADT ident adt' sj ys)
  | adt == adt' = coerce ident adt (c, sj) (xs, ys)
  | otherwise   = return $ SBoolean sFalse
sEqual (SADT ident adt si xs) (SCtr adt' c  ys)
  | adt == adt' = coerce ident adt (c, si) (ys, xs)
  | otherwise   = return $ SBoolean sFalse
sEqual (SArgs     xs) (SArgs     ys) =
  do eqs <- zipWithM sEqual xs ys
     return $ SBoolean $ sAnd $ map truthy eqs
sEqual _             _              = return $ SBoolean sFalse

truthy :: SValue -> SBool
truthy (SBoolean b) = b
truthy  SUnit       = sTrue
truthy v            = error $
  "Expected a symbolic boolean value, but got " ++ show v


-- * Coerce a symbolically created algebraic data type to be equal
   to a concrete one
coerce :: X -> D -> (C, SInteger) -> ([SValue], [SValue]) ->
   Formula SValue
coerce ident adt (c, si) (xs, ys) =
  do env    <- environment
     (_, i) <- selector env (adt, c)
     lift $ constrain $ si .== literal i
     types  <- fieldTypes env c
     ys'    <- ensureInstantiated ident ys types
     eqs    <- zipWithM sEqual xs ys'
     return $ SBoolean $ sAnd $
         (si .== literal i)
       : map truthy eqs


ensureInstantiated :: X -> [SValue] -> [Type] -> Formula [SValue]
ensureInstantiated _     [ ] [   ] = return []
ensureInstantiated ident [ ] types = instantiate ident types
ensureInstantiated _     svs types = ensureTypeAccord svs types >>
   return svs

instantiate :: X -> [Type] -> Formula [SValue]
instantiate ident types =
```

```
  do let names = map (((ident ++ "$field") ++) . show) ([0..] ::
     [Int])
     let vars  = zipWith Variable names types
     mapM createSymbolic vars

ensureTypeAccord :: [SValue] -> [Type] -> Formula ()
ensureTypeAccord [      ] [        ] = return ()
ensureTypeAccord [      ] _          = error
  "Fatal: Symbolic algebraic data type was missing fields."
ensureTypeAccord _        [        ] = error
  "Fatal: Symbolic algebraic data type was missing fields."
ensureTypeAccord (sv:svs) (tau:taus) =
  (if tau `correspondsTo` sv
    then return ()
    else error $
         "Type mismatch occurred in equality check of constructor
            fields.\n\
         \Unsatisfiable constraint: '" ++ show sv ++ "' not of
            expected type '"
         ++ show tau ++ "'")
   >> ensureTypeAccord svs taus


-- * Correspondence between concrete and symbolic types
correspondsTo :: Type -> SValue -> Bool
Unit'     `correspondsTo` SUnit             = True
Integer'  `correspondsTo` (SNumber       _) = True
Boolean'  `correspondsTo` (SBoolean      _) = True
(TypeList taus) `correspondsTo` (SArgs svs) =
  and $ zipWith correspondsTo taus svs
(ADT adt) `correspondsTo` (SCtr   adt' _ _) = adt == adt'
(ADT adt) `correspondsTo` (SADT _ adt' _ _) = adt == adt'
_         `correspondsTo` _                 = False


-- * SValues are 'Mergeable', meaning we can use SBV's
   if-then-else, called 'ite'.
instance Mergeable SValue where
  symbolicMerge = const merge

merge :: SBool -> SValue -> SValue -> SValue
merge _  SUnit        SUnit       = SUnit
merge b (SNumber  x) (SNumber  y) = SNumber  $ ite b x y
merge b (SBoolean x) (SBoolean y) = SBoolean $ ite b x y
merge b (SCtr adt x xs) (SCtr adt' y ys)
  | adt == adt' = SCtr adt (ite b x y) (mergeList b xs ys)
  | otherwise   = error $
    "Type mismatch between data type constructors '"
    ++ x ++ "' and '" ++ y ++ "'\n\
    \Of types '" ++ adt ++ "' and '" ++ adt' ++ ", respectively."
merge b (SADT x adt si xs) (SADT y adt' sj ys)
  | adt == adt' = SADT (ite b x y) adt (ite b si sj) (mergeList b
     xs ys)
  | otherwise   = error $
    "Type mismatch between symbolic data types '"
    ++ adt ++ "' and ' " ++ adt' ++ "'"
```

```
merge b (SCtr adt c xs) (SADT ident adt' si ys)
  | adt == adt' = ite b (SCtr adt c xs) (SADT ident adt' si ys)
  | otherwise   = error $
    "Type mismatch between concrete data type '" ++ adt ++
    "' and symbolic data type variable '" ++ ident ++
    "' with type '" ++ adt' ++ "'"
merge b (SADT ident adt si xs) (SCtr adt' c ys)
  | adt == adt' = ite b (SADT ident adt si xs) (SCtr adt' c ys)
  | otherwise   = error $
    "Type mismatch between concrete data type '" ++ adt' ++
    "' and symbolic data type variable '" ++ ident ++
    "' with type '" ++ adt ++ "'"
merge b (SArgs   xs) (SArgs   ys) = SArgs $ mergeList b xs ys
merge _ x y = error $ "Type mismatch between symbolic values '"
              ++ show x ++ "' and '" ++ show y ++ "'"

mergeList :: SBool -> [SValue] -> [SValue] -> [SValue]
mergeList sb xs ys
  | Just b <- unliteral sb = if b then xs else ys
  | otherwise              = error $ "Unable to merge arguments '"
                             ++ show xs ++ "' with '" ++ show ys
                                ++ "'\n\
                             \Impossible to determine Boolean
                                condition."
```

Listing A.1: The full implementation of `Formula`, the middle layer between Contra and SBV.

Bibliography

[1] Andreas Abel. 'foetus–termination checker for simple functional programs'. In: *Programming Lab Report* 474 (1998).

[2] Sergio Antoy. 'Programming with narrowing: A tutorial'. In: *Journal of Symbolic Computation* 45.5 (2010), pp. 501–522. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2010.01.006. URL: https://www.sciencedirect.com/science/article/pii/S0747717110000143.

[3] Thomas Arts et al. 'Testing Telecoms Software with Quviq QuickCheck'. In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. ERLANG '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 2–10. DOI: 10.1145/1159789.1159792. URL: https://doi.org/10.1145/1159789.1159792.

[4] Maximilian Bolingbroke and Simon Peyton Jones. 'Supercompilation by evaluation'. In: *Proceedings of the third ACM Haskell symposium on Haskell*. 2010, pp. 135–146.

[5] Ana Bove, Peter Dybjer and Ulf Norell. 'A brief overview of Agda–a functional language with dependent types'. In: Springer. 2009, pp. 73–78.

[6] Edwin Brady. 'Idris, a general-purpose dependently typed programming language: Design and implementation'. In: *Journal of functional programming* 23.5 (2013), pp. 552–593.

[7] Roman Cheplyaka. *tasty-quickcheck: QuickCheck support for the Tasty test framework.* Available on Hackage. 2024. URL: https://hackage.haskell.org/package/tasty-quickcheck.

[8] Koen Claessen and John Hughes. 'QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs'. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. ACM, 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[9] Alain Colmerauer. 'An introduction to Prolog III'. In: *Commun. ACM* 33.7 (July 1990), pp. 69–90. ISSN: 0001-0782. DOI: 10.1145/79204.79210. URL: https://doi.org/10.1145/79204.79210.

[10] Various Contributors. *The Haskell Tool Stack*. URL. 2024. URL: https://docs.haskellstack.org.

[11] William R Cook and Ralf Lämmel. 'Tutorial on online partial evaluation'. In: *arXiv preprint arXiv:1109.0781* (2011).

[12] Maxime Dénès et al. 'QuickChick: Property-based testing for Coq'. In: *The Coq Workshop*. Vol. 125. 2014, p. 126.

[13] Jonas Duregård, Patrik Jansson and Meng Wang. 'Feat: functional enumeration of algebraic types'. In: *ACM SIGPLAN Notices* 47.12 (2012), pp. 61–72.

[14] Levent Erkok. *SBV: SMT-Based Verification: Symbolic Haskell theorem prover using SMT solving.* Available on Hackage. 2024. URL: https://hackage.haskell.org/package/sbv.

[15] Richard Feldman. *Elm in action*. Manning Publications, 2020.

[16] Andy Gill. *mtl: Monad classes for transformers, using functional dependencies*. Available on Hackage. 2024. URL: https://hackage.haskell.org/package/mtl.

[17] Rich Hickey. 'The Clojure programming language'. In: *Proceedings of the 2008 symposium on Dynamic languages*. 2008, pp. 1–1.

[18] John Hughes. 'How to Specify It! A Guide to Writing Properties of Pure Functions'. In: *Trends in Functional Programming: 20th International Symposium, TFP 2019*. Vancouver, BC, Canada: Springer-Verlag, 2019, pp. 58–83. DOI: 10.1007/978-3-030-47147-7_4.

[19] Neil D Jones, Carsten K Gomard and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[20] Hadi Katebi, Karem A. Sakallah and João P. Marques-Silva. 'Empirical Study of the Anatomy of Modern Sat Solvers'. In: *Theory and Applications of Satisfiability Testing - SAT 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 343–356. ISBN: 978-3-642-21581-0.

[21] Joachim Tilsted Kristensen, Robin Kaarsgaard and Michael Kirkedal Thomsen. *Jeopardy: An Invertible Functional Programming Language*. 2022. arXiv: 2209.02422 [cs.PL].

[22] Leonidas Lampropoulos, Zoe Paraskevopoulou and Benjamin C Pierce. 'Generating good generators for inductive relations'. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–30.

[23] Leonidas Lampropoulos et al. 'Beginner's Luck: A Language for Property-Based Generators'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. ACM, 2017, pp. 114–129. DOI: 10.1145/3009837.3009868.

[24] The Rust Programming Language. *The Rust Reference Influences*. 2024. URL: https://doc.rust-lang.org/reference/influences.html (visited on 06/03/2024).

[25] Dan Leijen, Paolo Martini and Antoine Latter. *parsec: Monadic parser combinators*. Available on Hackage. 2024. URL: https://hackage.haskell.org/package/parsec.

[26] Simon Marlow et al. 'Haskell 2010 language report'. In: (2010).

[27] Gabriele Masina, Giuseppe Spallitta and Roberto Sebastiani. 'On CNF Conversion for Disjoint SAT Enumeration'. In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 15:1–15:16. DOI: 10.4230/LIPIcs.SAT.2023.15. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2023.15.

[28] Leonardo de Moura and Nikolaj Bjørner. 'Satisfiability Modulo Theories: An Appetizer'. In: *Formal Methods: Foundations and Applications*. 2009, pp. 23–36. ISBN: 978-3-642-10452-7.

[29] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[30] Robert Nieuwenhuis, Albert Oliveras and Cesare Tinelli. 'Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)'. In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977. ISSN: 0004-5411. DOI: 10.1145/1217856.1217859. URL: https://doi.org/10.1145/1217856.1217859.

[31] Martin Odersky et al. 'An overview of the Scala programming language'. In: (2004).

[32] Tomas Petricek. 'What we talk about when we talk about monads'. In: *The Art, Science, and Engineering of Programming* 2.3 (Mar. 2018). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/12. URL: http://dx.doi.org/10.22152/programming-journal.org/2018/2/12.

[33] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[34] *QCheck: QuickCheck inspired property-based testing for OCaml*. https://github.com/c-cube/qcheck. commit: 15a247771e97d0ed4d0474c5c25a15f6814eab4d. 2023.

[35] Silvio Ranise and Cesare Tinelli. 'The SMT-LIB format: An initial proposal'. In: *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*. Citeseer. 2003.

[36] Colin Runciman, Matthew Naylor and Fredrik Lindblad. 'Smallcheck and lazy smallcheck: automatic exhaustive testing for small values'. In: (2008), pp. 37–48. ISSN: 0362-1340. DOI: 10.1145/1543134.1411292. URL: https://doi.org/10.1145/1543134.1411292.

[37] Eric L Seidel, Niki Vazou and Ranjit Jhala. 'Type targeted testing'. In: *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer. 2015, pp. 812–836.

[38] Glynn Winskel. *The formal semantics of programming languages: an introduction*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0262231697.