SAT Compilation for Constraints over Structured Finite Domains

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur
(Dr.-Ing.)

vorgelegt an der

Technischen Universität Dresden

Fakultät Informatik

eingereicht von

Dipl.-Inf. (FH) Alexander Bau

geboren am 20.06.1985 in Leipzig

Gutachter

Prof. Dr.-Ing. habil. Heiko Vogler
Technische Universität Dresden

Dr. rer. nat. Carsten Fuhs
Birkbeck, University of London

Tag der Verteidigung: 7. Februar 2017

Dresden im Februar 2017

Abstract

A constraint is a formula in first-order logic expressing a relation between values of various domains. In order to solve a constraint, constructing a propositional encoding is a successfully applied technique that benefits from substantial progress made in the development of modern SAT solvers. However, propositional encodings are generally created by developing a problem-specific generator program or by crafting them manually, which often is a time-consuming and error-prone process especially for constraints over complex domains. Therefore, the present thesis introduces the constraint solver CO^4 that automatically generates propositional encodings for constraints over structured finite domains written in a syntactical subset of the functional programming language Haskell. This subset of Haskell enables the specification of expressive and concise constraints by supporting user-defined algebraic data types, pattern matching, and polymorphic types, as well as higher-order and recursive functions. The constraint solver CO^4 transforms a constraint written in this high-level language into a propositional formula. After an external SAT solver determined a satisfying assignment for the variables in the generated formula, a solution in the domain of discourse is derived. This approach is even applicable for finite restrictions of recursively defined algebraic data types. The present thesis describes all aspects of CO^4 in detail: the language used for specifying constraints, the solving process and its correctness, as well as exemplary applications of CO^4.

Acknowledgments

Foremost, I want to express my gratitude towards my supervisors Johannes Waldmann (Hochschule für Technik, Wirtschaft und Kultur Leipzig) and Heiko Vogler (Technische Universität Dresden) for their extensive support throughout the process of developing CO^4 and writing this thesis. Without their encouragement and the valuable criticism they provided, I would not have been able to actually accomplish any of the goals that were set for this project.

I want to thank Carsten Fuhs (Birkbeck, University of London) for his helpful remarks on several iterations of this thesis. Furthermore, I am thankful to Eric Noeth, Markus Lohrey, Jörg Endrullis, Sebastian Will, and René Thiemann for giving me the opportunity to contribute to exciting projects and publications.

Finally, I want to thank the European Social Fund for granting me the financial support (ESF grant 100088525) which enabled me to begin with my research in the first place.

Contents

Chapter 1

Introduction

Constraint programming is a declarative programming paradigm where the solution of a problem is specified by a logical formula written in a formal language. In the context of constraint programming, such a formula is called a constraint as it constrains the properties of the problem's solution. While programs written in imperative languages specify an algorithm for solving a particular problem, a constraint often does not hint how the problem's solution can be computed. Here, the solving algorithm is implemented in a designated program called constraint solver.

Separating the specification of a problem's solution from its computation has at least two immediate advantages. Firstly, different constraint solvers can be applied to the same constraint: this is useful as different constraint solvers may use different techniques for finding a solution. And secondly, the constraint itself becomes more concise and comprehensible as it is not intermingled with details about the computation of its solution.

Each constraint solver expects a given constraint to be specified in a particular formal language: in the following, we refer to this language as constraint specification language. Similar to programming languages, constraint specification languages differ considerably in the level of abstraction they provide for specifying constraints:

1. A low-level constraint specification language has a restricted syntax and is designed for specifying constraints over less structured domains. Less structured domains often allow constraint solvers to apply very runtime-efficient search strategies in order to find a solution more quickly.

2. A high-level constraint specification language has a richer syntax providing features for specifying constraints on a more abstract level. High-level constraint specification languages are often useful for specifying constraints over more structured domains. While this makes corresponding constraint solvers more easily applicable to complex constraints from different areas, it complicates the runtime-efficient search for a solution.

The different levels of abstraction that are inherent to popular constraint specification languages reveal a fundamental trade-off between two requirements: the implementer of a constraint solver wants the language to be as restricted as possible in order to design a runtime-efficient search strategy, but the user wants a rich language that enables the specification of concise constraints on an abstract level.

Encoding constraints as satisfiability problems in propositional logic (SAT) constitutes a well-known method of constraint programming via a low-level constraint specification language that only supports Boolean variables and logical connectives [67][12]. For a given propositional formula, a SAT solver searches for a satisfying assignment of Boolean values to the variables in the formula. Since SAT solvers became powerful enough to handle propositional formulas with millions of variables and clauses, it is a promising technique to specify constraints from various areas as SAT problems [16][54].

Despite the popularity of constraint programming via SAT solving, it might be difficult to give a specification in propositional logic for complex constraints over structured domains. This is due to the restricted syntax of propositional formulas and the restricted domain of Boolean values. Therefore, this thesis introduces the constraint solver CO^4 (Complexity Concerned Constraint Compiler) whose constraint specification language is a subset of the purely declarative Haskell language [47]. This subset of Haskell includes user-defined algebraic data types, pattern matching, and recursive functions, as well as higher-order and polymorphic types. These are common features in the functional programming paradigm and they have proven to be useful concepts when writing concise and declarative programs. Recently, some of these features which used to be prevalent only in the functional programming paradigm were introduced even for imperative languages, e.g., algebraic data types and pattern matching in the Scala language [62]. For illustration, Listing 1.1 shows an excerpt of a constraint written in CO^4's constraint specification language.

This thesis aims at resolving the fundamental conflict between the expressiveness of high-level languages like CO^4's constraint specification language and the runtime-efficient search strategies available for low-level languages like SAT. Thus, the main contributions of this thesis are the following:

```
data List a = Nil | Cons a (List a)
data TRS = TRS (List Nat) (List (Pair Term Term))

constraint :: TRS -> List Nat -> Bool
constraint = \trs prec -> case trs of
  TRS symbols rules ->
    and2 (forall rules   (\rule -> ordered rule prec))
         (forall symbols (\sym  -> exists prec sym eqNat))

forall :: List a -> (a -> Bool) -> Bool
forall = \xs f -> case xs of
  Nil -> True
  Cons y ys -> and2 (f y) (forall ys f)
```

Listing 1.1: An excerpt from a constraint written in CO^4's constraint specification language. The complete constraint can be found in Appendix C.4 and is explained in detail in Section 7.1.2.

1. *We define the syntax and semantics of CO^4's purely declarative constraint specification language which is a syntactical subset of the Haskell language featuring user-defined algebraic data types, pattern matching, and polymorphic types, as well as higher-order and recursive functions.*
2. *For a constraint* c *written in CO^4's specification language, we define an automatic transformation into a propositional formula* f *and prove that a satisfying assignment for* f *can be decoded to a solution for* c.
3. *If the constraint* c *is satisfiable and its domain of discourse is finite, we prove that the propositional formula* f *generated by CO^4 is satisfiable as well.*

In order to find a solution for a constraint given in CO^4's constraint specification language, CO^4 transforms it into a satisfiability problem in propositional logic. Then, the generated propositional formula is solved by an external SAT solver and a solution from the domain of discourse is constructed from a satisfying assignment of the variables in the formula.

By providing such a transformation for constraints over structured domains like finite lists and trees, CO^4 leverages the power of SAT solvers for problems which to date have been hard to express as satisfiability problems in propositional logic. For example, Section 7.1.1 describes the application of CO^4 for finding looping derivations in term rewriting systems. To the author's best knowledge, this is the first time that SAT solving has been applied to analyzing non-termination of term rewriting systems beyond unary signatures.

Using a subset of Haskell as a purely declarative constraint specification language has many benefits compared to other approaches. First of all, Haskell is a high-level language with many useful features for specifying well-typed and concise constraints. For example, the mentioned application of CO^4 for finding looping derivations in term rewriting systems has been realized via a constraint of approx. 150 lines of code. Reusing an established programming language like Haskell for constraint programming lowers the barriers of applying CO^4 to real world problems because it is not necessary to learn a new constraint specification language if one is already familiar with Haskell.

Due to the declarative nature of constraint specifications in CO^4, constraints can be extended and combined with minimal overhead. For example, Section 7.1.3 shows how termination analysis by searching for lexicographic path orders can be easily combined with the semantic labelling of term rewriting systems. This illustrates the flexibility of CO^4 as a constraint solver. Furthermore, in Chapter 8, we give a detailed comparison of CO^4 to other constraint solvers with respect to the features of their corresponding constraint specification languages.

Note that the present approach is merely a prototypical implementation of solving constraints over structured domains via transformation to satisfiability problems in propositional logic. Thus, CO^4 is not intended to compete against manually generated propositional encodings in terms of runtime performance: by incorporating deep knowledge about a particular domain when manually crafting a propositional encoding for a given constraint, one can often outperform CO^4's runtime performance. As generating a propositional encoding by hand is a time-consuming and error-prone process, CO^4's automatic transformation offers a lot more flexibility when specifying constraints over complex domains. This situation is similar to the respective characteristics of programming in a high-level language like Haskell versus programming in Assembler: while Assembler helps developing fast and memory-efficient programs, a high-level language supports more abstract concepts that help developing applications on a large scale. Consequently, this thesis provides a foundation for realizing a competitive constraint solver which combines a high-level, purely declarative constraint specification language with the power of modern SAT solvers.

The constraint solver CO^4 is distributed under the terms of the GNU General Public License [31] and is available at

`http://abau.org/co4`

Outline The present thesis is structured as follows. In Chapter 2, we briefly introduce some scientific background, namely, constraint programming and the Haskell programming language. Chapter 3 specifies the constraint solver CO^4 in detail. Firstly, we give a conceptual overview of the solving process implemented in CO^4. Then, we define the syntax and semantics of concrete programs, which constitute the formal representation that CO^4 expects constraints to be specified in. As the solving process involves the compilation of a given concrete program into an intermediate representation called abstract program, we define the syntax and semantics for abstract programs as well. We highlight the relation and differences between concrete and abstract programs, and define a correctness criterion for the compilation that needs to hold in order for CO^4 to implement a correct solving procedure.

Chapter 4 defines the compilation from concrete to abstract programs and shows that it satisfies the aforementioned correctness criterion. We give an overview of the actual implementation of CO^4 and how it is applied for solving constraints. This chapter concludes with a discussion on design decisions in the implementation of CO^4.

Chapter 5 specifies the compilation of advanced language features like local abstractions, higher-order functions, and partial functions. The reason these are discussed separately is because programs that contain these features are merely transformed to concrete programs as they are specified in Chapter 3. This transformation is entirely independent from the compilation of concrete to abstract programs.

Chapter 6 covers several optimization strategies that aim at decreasing the size of the resulting propositional formula. That is reasonable because often the SAT solver's runtime is lower for smaller formulas.

Chapter 7 illustrates the results of applying CO^4 to constraints of two different areas: termination analysis of term rewriting systems and RNA design in bioinformatics.

Chapter 8 compares different aspects of CO^4 to related tools that also enable constraints to be specified in a high-level language.

Chapter 9 addresses directions for future work that improve different aspects of CO^4. This concerns the size of the resulting propositional formulas as well as the expressiveness of CO^4's constraint specification language.

Chapter 2

Background

This chapter briefly introduces the scientific background of the results given in the present thesis. In Section 2.1, we give an overview of constraint programming and relate different approaches to the constraint solver CO^4. In Section 2.2, we illustrate the Haskell programming language because a subset of Haskell constitutes the constraint specification language of CO^4.

2.1 Constraint Programming

The constraint programming paradigm includes a variety of different approaches for specifying and solving constraints [65]. Each of them has certain benefits which render it suitable for tackling a particular kind of constraint. In the following, we briefly introduce some classical approaches and relate them to the constraint solver CO^4.

Finite Domain Constraints Constraints whose variables range over finite domains are an important and well-researched class of constraints [5]. A solution for a finite domain constraint is a satisfying assignment that maps each variable to a value of its underlying domain. A classical approach for finding such a solution is to incrementally reduce the set of values that may be assigned to each variable in the constraint without violating the local consistency of these variables. Often, this is done by alternately performing constraint propagation and splitting. In general, constraint propagation alone does not lead to a solution; therefore, either the domain of a variable or the constraint itself is split in order to obtain two or more subproblems such that the union of all solutions for these subproblems is equivalent to the set of solutions for the original constraint. The Davis–Putnam–Logemann–Loveland algorithm illustrated in Appendix B is an example for applying constraint propagation and splitting to find a solution for satisfiability problems in propositional logic [14]. These problems are well-studied instances of finite domain constraints with many practical applications [16][56], e.g., for termination analysis of term rewriting systems [28].

In the context of solving finite domain constraints, the constraint solver CO^4 can be regarded as an initial compilation step where a given finite domain constraint written in a subset of Haskell is transformed into a satisfiability problem in propositional logic. However, CO^4 itself does not implement any search strategies for finding a solution; it merely transforms a given constraint into a propositional formula and applies the external SAT solver MiniSat [25] to find a satisfying assignment for the variables in that formula. Other targets than SAT are imaginable (cf. Section 9.4), but none of them were considered in the present thesis.

Satisfiability Modulo Theories Satisfiability modulo theories (SMT) is an alternative approach of constraint programming [53]. SMT constraints are formulas in first-order logic that may contain propositions from a certain theory, as well as variables that range over the domain of that theory. Exemplary theories are linear integer arithmetic [48] and bitvector arithmetic [32]. Finding a solution for an SMT constraint often involves a SAT solver that assigns a Boolean value to each theory-specific proposition indicating whether this proposition holds. Then, a theory solver checks whether the chosen assignment is consistent with the underlying theory. This alternating process of selecting an assignment and checking it against the theory is done until a consistent assignment is found. An alternative method for finding a solution is to transform the SMT constraint as a whole to a satisfiability problem in propositional logic. This approach enables the reuse of existing SAT solvers, but it is not equally feasible for all theories.

In general, the underlying theories in SMT only allow predicates over flat domains, e.g., bitvectors, whereas CO^4 allows the specification of constraints over structured domains. On the other hand, most SMT solvers support theories on infinite domains, e.g., integer/real linear arithmetic, while in CO^4 the search space of constraints over infinite domains must be restricted to a finite subset.

Constraint Logic Programming Constraint logic programming is a programming paradigm where logic programs are extended by predicates over certain domains [42][61], e.g., real numbers. These predicates paired with the ability of logic programs to specify relations between terms make powerful constraint programming languages, e.g., the Prolog language [30]. But for reasons concerning runtime performance, constraint specifications in Prolog are not purely declarative, e.g., they may contain non-declarative entities like the cut-operator for pruning the search tree. Such entities change the semantics of the constraint and are bound to the underlying search strategy.

CO^4 only deals with pure declarative constraints, i.e., constraints that do not restrict the search for a solution to a certain search strategy. This separation of a constraint's specification from the search for a solution allows that future CO^4 backends can be applied to all existing constraints.

On the other hand, Prolog's search strategy can handle constraints on infinite domains. As CO^4 transforms constraints to satisfiability problems in propositional logic, the domain of each involved variable must be finite.

Chapter 8 compares CO^4 to Prolog in more detail.

2.2 The Haskell Language

Haskell is a functional programming language whose design process was initiated in 1987 [43][71]. The Haskell ecosystem induces and benefits from active research in a lot of different areas, e.g., compiler construction [55], type theory [24], and high-performance computing [4]. These ongoing efforts are presented and published on annual conferences such as the International Conference on Functional Programming and the Haskell Symposium. There is also a growing number of commercial applications of Haskell, e.g., in finance [46].

The Haskell language follows a distinct set of principles which makes it well-suited as a constraint specification language, especially in comparison to even more popular programming languages like Java, C++, or Scala. In the following, we review its most important features and highlight their importance to CO^4.

Algebraic Data Types and Pattern Matching Algebraic data types (ADT) enable the definition of product and sum types in Haskell. An ADT specifies the a sum of one or more alternatives, each denoted via a constructor, where each alternative contains a product of zero or more values: the constructor's arguments. Example 2.1 illustrates some simple ADTs.

Example 2.1 Assume the following definition of four ADTs:

```
data Bool = False | True
data Maybe a = Nothing | Just a
data Pair a b = Pair a b
data List a = Nil | Cons a (List a)
```

The type `Bool` is an ADT with two constructors `False` and `True` where none of them contains any arguments.

The type `Maybe a` is polymorphic: it is parameterized by a type variable `a` which appears as the single argument of the `Just` constructor. Thus, `Maybe` is occasionally called a type operator [64] as it defines a whole set of types through instantiation of the type variable `a` by other types. Note that `Maybe a` is useful for specifying the possible absence of a particular value of type `a` (cf. Section 5.4).

The type `Pair a b` is polymorphic as well. Its single constructor (of the same name `Pair`) expects one argument for each of the type variables `a` and `b`. Therefore, `Pair a b` specifies a pair of values in the mathematical sense.

The type `List a` specifies an ordered sequence of values by recursively employing itself as a argument for the `Cons` constructor. The `Nil` constructor acts as the end of the sequence.

Applying the constructor C of a type T to values that correspond to the argument types of C generates a value of type T. For example, `Just False` is a value of the type `Maybe Bool` from Example 2.1.

Values of ADTs can be deconstructed via case distinctions. A case distinction is a Haskell expression that performs a pattern match on the value of the case distinction's discriminant. Listing 2.2 shows the definition of a Haskell function that computes the tail of a list using a case distinction.

```
1 data List a = Nil | Cons a (List a)
2
3 tail :: List a -> List a
4 tail = \list ->
5   case list of
6     Nil -> Nil
7     Cons x xs -> xs
```

Listing 2.2: The function `tail` computes the tail of the expression `list` using a case distinction.

The expression `list` in Listing 2.2 is the discriminant of the case distinction in Lines 5 to 7. The left-hand side of an arrow `->` denotes a pattern and the right-hand side denotes the corresponding branch that is evaluated in case that the pattern matches on the discriminant.

Case distinctions are very useful for writing programs on data that is structured via ADTs. In CO^4, ADTs and case distinctions play an important role as case distinctions are the only control flow feature in the subset of Haskell that is supported by CO^4. In the process of transforming a constraint into a satisfiability problem in propositional logic, the transformation of ADTs and case distinctions is a critical aspect and described in detail in Chapter 4.

Static type system In Haskell, the type of each expression is computed and checked at compilation time. This eliminates runtime type errors for well-typed programs, and is very useful for writing reliable and safe software.

A static type system is essential for the functioning of CO^4 because the transformation of a constraint into a satisfiability problem in propositional logic is guided by the types of the expressions in a given constraint. However, Haskell's type system is very rich and not all the features that it provides are supported by CO^4, e.g., type classes. Section 9.3 briefly discusses how CO^4 could benefit from supporting type classes in the future.

Non-strict evaluation In Haskell, expressions are evaluated according to a non-strict evaluation strategy, i.e., the value of an expression is not computed until it is actually required. This strategy is most apparent for function applications. Assume an application $f(x)$ of a function f to an argument x: by using a non-strict evaluation strategy for computing the value of $f(x)$, the argument x is not evaluated until its value is required for computing the value of $f(x)$ itself. This is contrary to programming languages that feature a strict evaluation strategy. In a strictly evaluated language, the value of x is always computed first before evaluating $f(x)$ itself.

Non-strict evaluation strategies have benefits for composing computer programs from non-related entities [44]. Assume a nested function application $f(g(x))$ where storing the result of $g(x)$ would require more memory than is available. In order to still compute the value of $f(g(x))$ using a strict evaluation strategy, one would have to explicitly interweave the definitions of f and g so that both functions are evaluated synchronously, i.e., g delivers only as much data as required for evaluating the next subexpression in f at any one time. Obviously, this would blur the logical separation of f and g. By using a non-strict evaluation strategy, the synchronous evaluation of f and g comes implicitly without needing to rewrite any of these functions.

Despite the usefulness of Haskell's non-strict evaluation strategy, CO^4 follows a strict evaluation strategy when transforming a constraint into a satisfiability problem in propositional logic. This is due to the strictness of modern SAT solvers: before running a SAT solver, in general, the complete formula to solve must be present. However, we have not experienced that a strict evaluation strategy would complicate the constraint specification using CO^4 for any of the studied applications (cf. Chapter 7).

While some SAT solvers use an incremental solving procedure (cf. Section 9.1), its benefits for specifying constraints with a non-strict evaluation strategy have not been researched in the scope of the present thesis.

Purely declarative Haskell is a purely declarative programming language, i.e., there are no implicit side effects. For example, a Haskell function is also a function in the mathematical sense: it can be evaluated for a given set of arguments, but it can neither change the value of a variable nor perform any input/output operation, e.g., print to screen, read a file, write to random memory. This might seem like a huge restriction compared to other programming languages, but actually has at least two benefits:

- Immutable variables are simply constants and enable certain compiler optimizations, e.g., constant propagation.
- Purity enforces all input/output operations to be explicitly specified, e.g., via type declarations. Again, this enables certain compiler optimizations, e.g., parallel evaluation of functions. Furthermore, a declaration concerning the presence of an input/output operation greatly improves the readability of a computer program.

The purity of Haskell is one of the main reasons for using it as the constraint specification language for CO^4. That is because using a language with implicit side effects in this context would vastly complicate its formal semantics in the scope of constraint programming as one would have to keep track of all the potential side effects when interpreting/compiling a given constraint.

Chapter 3

Specification of CO^4

This chapter gives a formal specification for the constraint solver CO^4. Section 3.1 defines the class of constraints that is handled by CO^4 and illustrates the solving process from a high-level point-of-view. Each of these constraints is represented by a concrete program. The language of concrete programs is defined in Section 3.2. When CO^4 is applied to a particular concrete program, it is compiled into an intermediate representation, called abstract program. Section 3.3 defines the language of abstract programs. In order to show that the compilation from concrete to abstract programs is reasonable, Section 3.4 specifies a correctness criterion.

3.1 Conceptual Overview

This section outlines the concept of the constraint solver CO^4 from a high-level perspective without regarding implementation-specific details. First of all, we define the type of constraint that CO^4 handles.

Definition 3.1 A *constraint* $c : \mathrm{P} \times \mathrm{U} \to \mathbb{B}$ is a parameterized predicate on a set U where P denotes the set of parameters and $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$.

Notation In the scope of this thesis, the domain U of a constraint $c : \mathrm{P} \times \mathrm{U} \to \mathbb{B}$ is denoted as *domain of discourse*, and the domain P is denoted as *parameter domain.*

Example 3.2 In the following constraint $c : \mathbb{N} \times \mathbb{N}^2 \to \mathbb{B}$, the domain of discourse is the set of pairs of natural numbers and the parameter domain is the set of natural numbers:

$$c(p, (a, b)) = \begin{cases} \mathsf{True} & \text{if } p = (a \cdot b) \wedge (a > 1) \wedge (b > 1) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

Most genuine constraints given in the present thesis are defined over more structured domains than illustrated in Example 3.2. For example, Section 7.1 describes an application of CO^4 to termination analysis of term rewriting systems, where the parameter domain is the set of term rewriting systems, and the domain of discourse is a subset of the set of (non-)termination proofs.

Some elements from the domain of discourse are solutions for a given constraint and a parameter.

Definition 3.3 An element $u \in \mathrm{U}$ denotes a *solution* for a constraint $c : \mathrm{P} \times \mathrm{U} \to \mathbb{B}$ and a parameter $p \in \mathrm{P}$ if $c(p, u) = \mathsf{True}$.

Example 3.4 $u = (5, 3)$ is a solution for constraint c and parameter $p = 15$ in Example 3.2 because $c(15, (5, 3)) = \mathsf{True}$.

From a high-level point-of-view CO^4 can be considered as a black box that takes a constraint c and a parameter p as input (cf. Figure 3.5) where c is represented by a concrete program (cf. Section 3.2). CO^4 either gives a solution $u \in \mathrm{U}$ so that $c(p, u) = \mathsf{True}$, or one of the special values Maybe and Unsat.

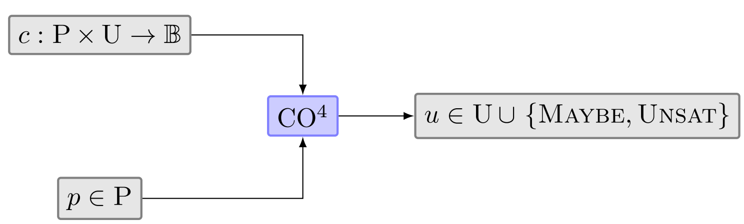


Figure 3.5: CO^4 as a black box

As CO^4 is an incomplete constraint solver (cf. Section 4.3.1), it gives the value Maybe if it is unable to find a solution for a constraint and a parameter. Maybe does not indicate the reason why CO^4 fails to produce a solution. On the other hand, CO^4 gives the value Unsat if there is no solution for a constraint $c : \mathrm{P} \times \mathrm{U} \to \mathbb{B}$ and a parameter $p \in \mathrm{P}$, i.e., there is no $u \in \mathrm{U}$ such that $c(p, u) = \mathsf{True}$. The output Unsat makes a more strict proposition than Maybe by indicating that the constraint is unsatisfiable at all for a particular parameter.

To find a solution, CO^4 generates a satisfiability problem in propositional logic. To do so, the constraint and the parameter are encoded as a formula $f \in \mathrm{F}$ where F denotes the set of propositional formulas. A SAT solver is applied to find a satisfying assignment $\sigma \in \mathbb{B}^{\mathsf{var}(f)}$ for the set $\mathsf{var}(f) \subseteq \mathrm{V}$ of propositional variables in f. Note that Appendix B gives an introduction to propositional logic and outlines the basics of SAT solving.

Figure 3.6 illustrates the process of constructing a propositional formula and decoding a satisfying assignment to a solution for a given constraint and a parameter.

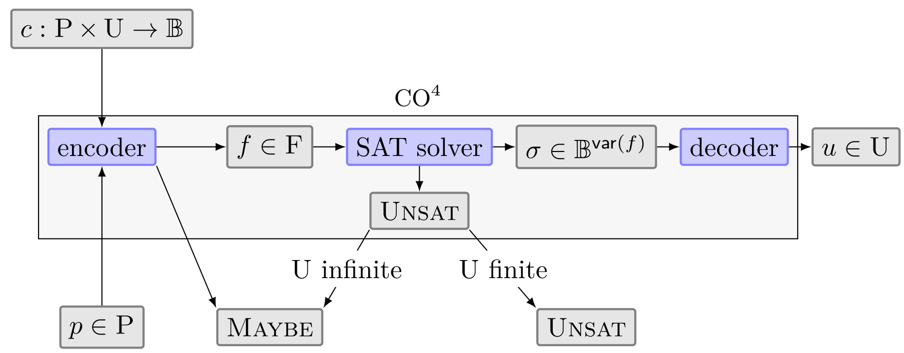


Figure 3.6: Encoding and decoding in CO^4

Encoding a constraint and a parameter as a propositional formula is done in two steps. During compilation time, an abstract program is generated from the concrete program that specifies the given constraint. This compilation is done in absence of any parameter. During runtime, evaluating the abstract program for a given parameter generates a propositional formula.

Because the compilation function in CO^4 does not depend on a given parameter value, each generated abstract program can generate different propositional formulas by passing different parameters during runtime. This avoids time-consuming recompilations in situations where the same constraint needs to be solved for multiple parameters. Without having such a parameter-independent compilation function, the same concrete program would need to be recompiled for each parameter.

Figure 3.7 illustrates this two-step derivation of propositional formulas.

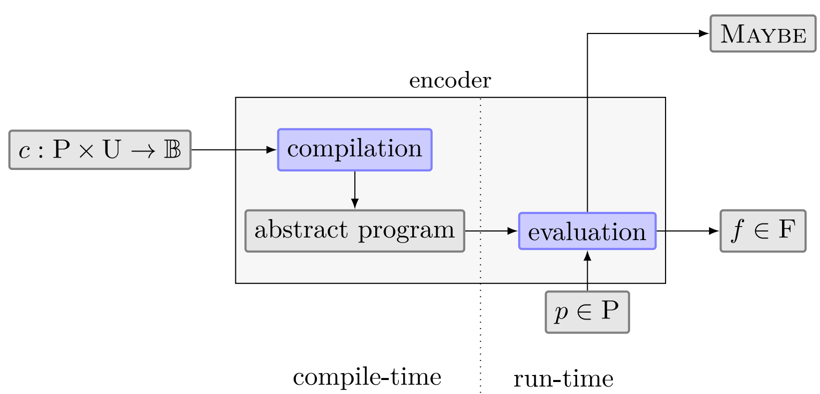


Figure 3.7: Generating a propositional formula is a two-step process

The essence of the present thesis is a specification of CO^4 so that it implements a correct solving procedure.

Theorem 3.8 The constraint solver CO^4 implements an incomplete and correct solving procedure, i.e., if CO^4 returns a value $u \in \mathrm{U}$ for a constraint $c : \mathrm{P} \times \mathrm{U} \to \mathbb{B}$ represented as a concrete program and a parameter $p \in \mathrm{P}$, then u is a solution for c and p, i.e., $c(p, u) = \mathsf{True}$.

Proof In order to show that Theorem 3.8 holds, we specify the necessary ingredients of CO^4 in the subsequent sections of this thesis. In Lemma 3.82, we show that the correctness of CO^4 depends on the correctness of two of its components:

1. There is a sound mapping from the values that concrete programs operate on to the values that abstract programs operate on, and vice-versa. Such a mapping is denoted as *encode/decode-pair* in Section 3.4. The proof of Lemma 4.31 shows that the mapping between both domains as it is implemented in CO^4 meets all requirements of an encode/decode-pair.

2. The compilation function of CO^4 that generates an abstract program from a given concrete program is correct according to the correctness criterion given in Definition 3.80. In Section 4.2, we specify the compilation function, and in Lemma 4.59 we show its correctness. ■

Note that Theorem 3.8 states that the solving procedure implemented by CO^4 is incomplete, i.e., CO^4 might not be able to provide a solution even if there is one. Example 4.63 illustrates a constraint whose obvious solution is not found by CO^4. But the incompleteness of CO^4 only concerns constraints whose domain of discourse is infinite, i.e., the solving procedure of CO^4 is complete for constraints on finite domains of discourse (cf. Theorem 4.62).

3.2 Language of Concrete Programs

This section defines the language used for specifying constraints. Each word of this language is called a concrete program. The syntax of concrete programs is a subset of Haskell's syntax [47], but their semantics differ from Haskell's semantics in the following ways:

1. Concrete programs are evaluated strictly whereas Haskell programs are evaluated non-strictly.

2. Concrete programs may contain n-ary functions and n-ary constructors for $n \in \mathbb{N}$. That is contrary to Haskell, where functions are curried by default, i.e., the application of an n-ary function is reduced to n applications of n unary functions.

3. Concrete programs are first-order, i.e., functions may neither be passed as arguments nor returned as results.

4. Functions in concrete programs may only be declared globally, i.e., there are no local abstractions.

By requiring concrete programs to be first-order, we also forbid partial function and constructor applications.

Example 3.9 shows an exemplary concrete program.

Example 3.9 Recall the constraint $c : \mathbb{N} \times \mathbb{N}^2 \to \mathbb{B}$ from Example 3.2:

$$c(p,(a,b)) = \begin{cases} \mathsf{True} & \text{if } p = (a \cdot b) \wedge (a > 1) \wedge (b > 1) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

The following concrete program is a correct specification of c according to the syntax and semantics defined in the subsequent sections:

```
data Bool     = False | True
data Nat      = Z | S Nat
data Pair a b = Pair a b

constraint :: Nat -> Pair Nat Nat -> Bool
constraint = \p u -> case u of
  Pair a b ->  and2 (greaterOne a)
              (and2 (greaterOne b)
                    (eq p (times a b)))

plus :: Nat -> Nat -> Nat
plus = \x y -> case x of
  Z -> y
  S x' -> S (plus x' y)

times :: Nat -> Nat -> Nat
times = \x y -> case x of
  Z -> Z
  S x' -> plus y (times x' y)

eq :: Nat -> Nat -> Bool
eq = \x y -> case x of
  Z -> case y of Z    -> True
                 S y' -> False
  S x' -> case y of Z    -> False
                    S y' -> eq x' y'

greaterOne :: Nat -> Bool
greaterOne = \x -> case x of
  Z -> False
  S x' -> case x' of
    Z -> False
    S x'' -> True

and2 :: Bool -> Bool -> Bool
```

```
and2 = \x y -> case x of
  False -> False
  True  -> y
```

In this concrete program, `Nat` models naturals as Peano numbers such that the value `Z` represents zero, and `S` represents the successor of another natural number. Because `Nat` is recursively defined, there are infinite many values of type `Nat`, i.e., without further restrictions, `constraint` specifies a constraint on an infinite domain of discourse. In order to solve such a constraint using CO^4, the domain of discourse must be restricted to a finite subset as explained in Section 4.1.5.

Example 6.16 shows a more efficient concrete program which also specifies the constraint c, but uses CO^4 built-in support for binary encoded natural numbers (cf. Section 6.3).

The following sections specify concrete concrete programs in detail: Section 3.2.1 introduces the syntax of concrete programs, which is then restricted by the static semantics defined in Section 3.2.2. Section 3.2.3 defines the dynamic semantics by specifying an evaluation function for concrete programs.

3.2.1 Syntax

In this section, we define the abstract syntax trees of concrete programs. We do not consider aspects related to the textual representation of concrete programs (i.e., the placement of parentheses) because due to the usage of Haskell-related tools in the present implementation of CO^4, namely Template-Haskell (cf. Section 4.3.2), the same rules apply as for the textual representation of Haskell programs.

A concrete program contains identifiers that name different entities. Each identifier stems from a common set of names.

Definition 3.10 The set $\textsc{Name}$ denotes the *set of names*.

Because concrete programs are statically typed, entities such as expressions and functions have at least one type. As in Haskell, types contain identifiers of two distinct sets: type variables and type constructors.

Definition 3.11 $\textsc{TypeVar} \subseteq \textsc{Name}$ denotes the *set of type variables*.

Definition 3.12 $\textsc{TypeCon} \subseteq \textsc{Name}$ denotes the *signature of type constructors*, where $\textsc{TypeVar} \cap \textsc{TypeCon} = \varnothing$.

Because $\textsc{TypeCon}$ denotes a signature, each type constructor $c \in \textsc{TypeCon}$ is associated with an arity $\mathsf{arity}(c) \in \mathbb{N}$ (cf. Definition A.20).

Notation In the context of types, names starting with a lowercase letter identify type variables. Names starting with an uppercase letter denote type constructors. The only exception is the type constructor `->`, which is used for constructing functional types.

Example 3.13 $\{\mathtt{a}, \mathtt{x}\} \subsetneq \textsc{TypeVar}$ and $\{\mathtt{List}, \mathtt{Either}, \mathtt{->}\} \subsetneq \textsc{TypeCon}$.

Types are built upon TYPEVAR and TYPECON.

Definition 3.14 The *syntax of types* TYPESYNTAX is defined by the following EBNF:

$$\begin{aligned} \textsc{TypeSyntax} \coloneqq\ & \textsc{TypeVar} \\ \mid\ & \textsc{TypeCon}\ \ \textsc{TypeSyntax}^* \qquad \text{(type application)} \end{aligned}$$

Notation The binary type constructor `->` is written in infix notation, e.g., `a -> b`, where `->` is right-associative and has a lower precedence than all the other type constructors.

For the code listings given in the present thesis, we assume that the concrete syntax of types allows subtypes to be parenthesized according to the same rules that apply for the Haskell language.

Example 3.15 Exemplary types are `x`, `Nat`, `List Nat`, `Either a b`, `List Nat -> Nat`, and `Pair (List Nat) Nat`.

A concrete program may contain polymorphic expressions and functions, i.e., expressions and functions that have more than one type. Polymorphism is expressed by free variables in types. Type schemes quantify over these free type variables.

Definition 3.16 The *syntax of type schemes* SCHEMESYNTAX is defined by the following EBNF:

$$\begin{aligned} \textsc{SchemeSyntax} \coloneqq\ & \textsc{TypeSyntax} \\ \mid\ & \texttt{"forall"}\ \textsc{TypeVar}^+\ \texttt{":"}\ \textsc{TypeSyntax} \end{aligned}$$

Notation As types in Haskell are implicitly quantified over all occurring type variables, we omit the explicit quantification in the concrete syntax of concrete programs as well. Thus, types and type schemes share the same concrete syntax in CO^4.

Example 3.17 A function that concatenates two lists has the type scheme:

```
List a -> List a -> List a
```

Expressions contain identifiers of two distinct sets: variables and constructors.

Definition 3.18 $\textsc{Var} \subseteq \textsc{Name}$ denotes the *set of variables*.

Definition 3.19 $\textsc{Con} \subseteq \textsc{Name}$ denotes the *signature of constructors*, where $\textsc{Var} \cap \textsc{Con} = \varnothing$.

Because $\textsc{Con}$ denotes a signature, each constructor $c \in \textsc{Con}$ is associated with an arity $\mathsf{arity}(c) \in \mathbb{N}$ (cf. Definition A.20). The set of terms over the signature of constructors that appear in a concrete program forms the set of concrete values (cf. Definition 3.50).

Notation To differentiate between elements of $\textsc{Var}$ and $\textsc{Con}$, variables are identified by names that start with a lowercase character. Names starting with an uppercase character denote constructors.

Example 3.20 $\{\mathtt{a}, \mathtt{x}\} \subsetneq \textsc{Var}$ and $\{\mathtt{True}, \mathtt{False}\} \subsetneq \textsc{Con}$.

Case distinctions enable conditional evaluation of sub-expressions (*branches*) based on the value of a particular expression: the case distinction's *discriminant*. Patterns match on the value of the discriminant in order to determine which branch to evaluate. A pattern consists of a constructor name and a sequence of variables.

Definition 3.21 The *syntax of patterns* $\textsc{PatSyntax}$ is defined by the following EBNF:

$$\textsc{PatSyntax} \coloneqq \textsc{Con}\ \ \textsc{Var}^*$$

Example 3.22 Exemplary patterns are `Nil` and `Cons x xs`.

For the sake of simplicity, CO^4 does not support nested patterns like `Cons x (Cons y Nil)`, which is contrary to Haskell.

A match consists of a pattern and a branch's expression, where an expression is either a variable, a constructor, an application, an abstraction, a case distinction, or a local binding.

Definition 3.23 The *syntax of matches* $\textsc{MatchSyntax}$ and the *syntax of expressions* $\textsc{ExpSyntax}$ are defined by the following EBNF:

$$
\begin{array}{lll}
\multicolumn{2}{l}{\textsc{MatchSyntax} \coloneqq \textsc{PatSyntax}\ \ \texttt{"->"}\ \ \textsc{ExpSyntax}} & \\
\multicolumn{2}{l}{\textsc{ExpSyntax} \coloneqq} & \\
 & \textsc{Var} & \\
 & \mid \textsc{Con} & \\
 & \mid \textsc{ExpSyntax}\ \textsc{ExpSyntax}^+ & \text{(application)} \\
 & \mid \texttt{"\textbackslash"}\ \textsc{Var}^+\ \texttt{"->"}\ \textsc{ExpSyntax} & \text{(abstraction)} \\
 & \mid \texttt{"case"}\ \textsc{ExpSyntax}\ \texttt{"of"}\ \textsc{MatchSyntax}^+ & \text{(case distinction)} \\
 & \mid \texttt{"let"}\ (\textsc{Var}\ \texttt{"="}\ \textsc{ExpSyntax})^+\ \texttt{"in"}\ \textsc{ExpSyntax} & \text{(local bindings)}
\end{array}
$$

As for types, we assume that the concrete syntax of expressions allows subexpressions to be parenthesized according to the same rules that apply for the Haskell language.

Example 3.24 An exemplary expression is

```
\f x -> case x of Nil -> Nil
                  Cons y ys -> f y
```

where `x` is the discriminant of a case distinction with two branches `Nil` and `f y`.

In a concrete program, types are defined by type declarations.

Definition 3.25 A *type declaration* lists the constructors of a type and is defined by the following EBNF:

$$\begin{aligned}\textsc{TypeDeclSyntax} \coloneqq\ & \texttt{"data"}\ \textsc{TypeCon}\ \textsc{TypeVar}^*\ \texttt{"="} \\ & (\textsc{Con}\ \textsc{TypeSyntax}^*\ \texttt{"|"})^*\ \textsc{Con}\ \textsc{TypeSyntax}^*\end{aligned}$$

Example 3.26 Exemplary type declarations are:

```
data Bool = False | True
data List a = Nil | Cons a (List a)
data Either a b = Left a | Right b
```

The identifier after the `data` keyword denotes the name of the type. It is followed by a possibly empty list of type variables that may occur in the constructor arguments. Each constructor may have several arguments where the number of arguments equals the constructor's arity. Note that it is perfectly legal for a constructor to have the same name as the corresponding type, which is common for types that have only one constructor.

Example 3.27 In the following, we declare a type `Pair` that has a single constructor of the same name:

```
data Pair a b = Pair a b
```

While the identifier `Pair` on the left-hand side of equality sign denotes a type name, the identifier `Pair` on the right-hand side denotes a constructor name.

Besides type declarations, a concrete program contains other declarations where names are bound to expressions and attributed with type schemes.

Definition 3.28 The *syntax of declarations* DeclSyntax is defined by the following EBNF:

$$\begin{aligned}\textsc{DeclSyntax} \coloneqq\ & \textsc{TypeDeclSyntax} \\ \mid\ & \textsc{Var}\ \texttt{"="}\ \textsc{ExpSyntax} \\ \mid\ & \textsc{Var}\ \texttt{"::"}\ \textsc{SchemeSyntax}\end{aligned}$$

The attribution $n :: s$ of a type scheme $s \in \textsc{SchemeSyntax}$ to a name $n \in \textsc{Var}$ is denoted as a *type signature* for n.

Example 3.29 Exemplary declarations are:

```
plus :: Nat -> Nat -> Nat
plus = \x y -> case x of Z -> y
                         S x' -> S (plus x' y)
```

Finally, a concrete program is defined as a non-empty sequence of declarations.

Definition 3.30 The *syntax of concrete programs* ProgSyntax is defined by the following EBNF:

$$\textsc{ProgSyntax} \coloneqq \ \textsc{DeclSyntax}^{+}$$

3.2.2 Static Semantics

This section restricts the previously specified syntax of concrete programs to the set of statically well-defined concrete programs.

Firstly, we define the static semantics of type declarations. Each type in a concrete program must be defined by a type declaration.

Definition 3.31 The *set of statically well-defined type declarations* TypeDecl is the set of all declarations of the form

$$\begin{array}{ll} \texttt{data}\ T\ v_1\ \ldots\ v_m\ \texttt{=} & C_1\ c_{11}\ \ldots\ c_{1n_1} \\ & \ldots \\ & \texttt{|}\ C_k\ c_{k1}\ \ldots\ c_{kn_k} \end{array}$$

in TypeDeclSyntax with $m \in \mathbb{N}$ type variables $v_1, \ldots, v_m$ and $k \in \mathbb{N}_{>0}$ constructors $C_1, \ldots, C_k$, all of the following properties hold:

1. the arity of T equals the number of type variables, i.e., $\mathsf{arity}(T) = m$,
2. all constructors are pairwise distinct, i.e.,
$$\forall (i,j) \in \{1 \ldots k\}^2 : i \neq j \implies C_i \neq C_j$$
3. the arity of each constructor equals the number of constructor arguments, i.e.,
$$\forall i \in \{1 \ldots k\} : n_i = \mathsf{arity}(C_i)$$
4. for all $i \in \{1 \ldots k\}$ and $j \in \{1 \ldots n_i\}$, the set of all variables appearing in the constructor argument c_{ij} is a subset of $\{v_1, \ldots, v_m\}$, and
5. for all $i \in \{1 \ldots k\}$ and $j \in \{1 \ldots n_i\}$, the type c_{ij} may not contain the type constructor `->`.

Example 3.32 The following declaration for type `Either` is not included in TypeDecl as the identifier `c` is no type variable of `Either`:

```
data Either a b = Left a | Right c
```

A type declaration that includes type variables defines a *type operator* [64]. Type operators may be applied to other types in order to construct new types.

The set of types equals the set of terms over the signature of type constructors. See Appendix A.2 for a brief introduction to signatures and terms.

Definition 3.33 The *set of statically well-defined types* Type is defined as the set of terms over TypeCon:

$$\textsc{Type} \coloneqq \mathsf{terms}(\textsc{TypeCon}, \textsc{TypeVar})$$

Example 3.34 Assume the following type declarations that define a type `Bool` and two type operators `List` and `Either`:

```
data Bool = False | True
data List a = Nil | Cons a (List a)
data Either a b = Left a | Right b
```

Here, `List Bool` $\in$ Type but `Either Bool` $\notin$ Type because the type operator `Either` has arity two but is only applied to one argument.

Throughout this thesis we often reference the constructors of a type declaration by their index.

Definition 3.35 The *constructor index* of a constructor $C \in \textsc{Con}$ of a type $T \in \textsc{Type}$ is a positive natural number in $\mathbb{N}_{>0}$ denoting the position of C in the sequence of constructors in the declaration of T.

Example 3.36 Assume the following type declaration that defines the type `Bool`:

```
data Bool = False | True
```

For the type `Bool`, the constructor `False` has index 1, and the constructor `True` has index 2.

We introduce the set of fully instantiated types as the subset of Type that do not contain type variables.

Definition 3.37 The *set of fully instantiated types* $\textsc{Type}_0 \subsetneq \textsc{Type}$ is the set of all types in Type that do not contain type variables:

$$\textsc{Type}_0 \coloneqq \mathsf{terms}(\textsc{TypeCon}, \varnothing)$$

Example 3.38 `List Bool` $\in \textsc{Type}_0$ but `List a` $\notin \textsc{Type}_0$.

Based on the previously defined set of types, we define the static semantics of type schemes:

Definition 3.39 The *set of statically well-defined type schemes* $\textsc{TypeScheme}$ is the set of all $S \in \textsc{SchemeSyntax}$ such that all of the following properties hold for S:

1. each type that appears in S is included in $\textsc{Type}$, and
2. if S is of the form `forall` $v_1 \dots v_n : T$ for $n \in \mathbb{N}_{>0}$ and $T \in \textsc{Type}$, then all type variables $v_1, \dots, v_n \in \textsc{TypeVar}$ are pairwise distinct and the set of type variables $\mathsf{var}(T)$ that appears in T is a subset of $\{v_1, \dots, v_n\}$.

Example 3.40 `forall x y` : `List z` is no type scheme as the type variable `z` is not bound by the quantifier.

We define the static semantics of patterns.

Definition 3.41 The *set of statically well-defined patterns* $\textsc{Pat}$ is the set of all $C\ v_1 \dots v_n \in \textsc{PatSyntax}$ with $C \in \textsc{Con}$ and all variables $v_1, \dots, v_n \in \textsc{Var}$ being pairwise distinct.

Example 3.42 `Cons x xs` $\in \textsc{Pat}$ and `Cons x x` $\notin \textsc{Pat}$.

We define the static semantics of expressions.

Definition 3.43 The *set of statically well-defined expressions* $\textsc{Exp}$ is the set of all $e \in \textsc{ExpSyntax}$ so that all of the following properties hold for e:

1. e is statically well-typed according to the Hindley–Damas–Milner type inference [22] so that all type signatures are respected,
2. e does not contain an abstraction as a strict subexpression,
3. if e is an application,
   (a) the application is total, i.e., the application of a n-ary function or constructor requires exactly n arguments for $n \in \mathbb{N}$,
   (b) no argument's type may contain the type constructor `->`,
4. if e is an abstraction `\` $v_1 \dots v_n$ `->` e' for $n \in \mathbb{N}_{>0}$, all variables $v_1, \dots, v_n \in \textsc{Var}$ are pairwise distinct,
5. if e is a case distinction on a discriminant of type $T \in \textsc{Type}$,
   (a) for each constructor $C \in \textsc{Con}$ in T there is a corresponding pattern $C\ v_1 \dots v_n$ with $n = \mathsf{arity}(C)$ in the matches of e,
   (b) no two patterns in the matches of e may contain the same constructor,

6. if e is a `let` expression that locally binds an expression $e' \in$ Exp, e' may only depend on values that have been bound in an enclosing scope, or in the same `let` block, but before e'.

Note that property 2 allows abstractions to be included in Exp, but not as a strict subexpression of another expression.

Example 3.44 For all expressions $e_1, e_2, \dots \in$ Exp and $v_1, v_2, \dots \in$ Var, the following expressions are not statically well-defined:

• `\` v_1 `->` $v_1 v_1$	violates property 1 in Def. 3.43
• `let` v_1 `=` `\` v_2 `->` e_1 `in` e_2	violates property 2 in Def. 3.43
• `\` v_1 v_1 `->` e_1	violates property 4 in Def. 3.43
• `data Bool = False | True` `case` e_1 `of False ->` e_2 `False ->` e_3	violates property 5a and 5b in Def. 3.43
• `let` v_1 `=` v_2 v_2 `=` v_1 `in` e_1	violates property 6 in Def. 3.43

The only requirement for statically well-defined matches is that each component must be statically well-defined as well.

Definition 3.45 The *set of statically well-defined matches* Match is the set of all matches in MatchSyntax that contain a pattern from Pat and an expression from Exp.

Similarly, we define the static semantics of declarations in a concrete program.

Definition 3.46 The *set of statically well-defined declarations* Decl is the set of all declarations in DeclSyntax that only contain statically well-defined types, expressions, and patterns.

Finally, we define the set of statically well-defined concrete programs.

Definition 3.47 The *set of statically well-defined concrete programs* Prog is the set of all programs $c \in$ ProgSyntax such that of the following properties hold for c:

1. c only contains statically well-defined declarations,
2. c contains exactly one type declaration of the form

```
data Bool = False | True
```

3. c contains exactly one declaration of the form `constraint` $= e$ with $e \in$ Exp so that

(a) e is of type P `->` U `-> Bool` for some types $P, U \in \textsc{Type}_0$,

(b) there is exactly one type declaration c for each of the types P and U,

4. each variable bound in c is only bound once, i.e., there are no two expressions that are bound to the same name.

The concrete program in Example 3.9 is statically well-defined.

Often we want to refer to the set of statically well-defined concrete programs that have a common type for their `constraint` declaration.

Definition 3.48 For the types $P, U \in \textsc{Type}_0$, the set $\textsc{Prog}_{PU}$ denotes the set of all concrete programs $c \in \textsc{Prog}$ that contain a declaration of the form `constraint` $= e$ with $e \in \textsc{Exp}$ being of type P `->` U `-> Bool`.

3.2.3 Dynamic Semantics

The dynamic semantics of concrete programs is given by an evaluation function that assigns a constraint (cf. Definition 3.1) to each concrete program. In contrast to Haskell, concrete programs are evaluated using a strict evaluation strategy, i.e., before evaluating the result of an application, each argument is evaluated. Note that we do not deal with problems related to non-termination: in the following we only give the semantics of terminating concrete programs.

The domain of values that a concrete program operates on equals the set of terms over the signature $\textsc{Con}$. See Appendix A.2 for a brief introduction to signatures and terms.

Definition 3.49 The set $\textsc{Universe}$ is defined as the set of terms over $\textsc{Con}$:

$$\textsc{Universe} := \mathsf{terms}(\textsc{Con}, \{\bot\})$$

$\bot$ is an exceptional value that denotes a failed computation.

The set of values that a particular concrete program operates on is a subset of $\textsc{Universe}$.

Definition 3.50 The function $\mathbb{C} : \textsc{Prog} \to 2^{\textsc{Universe}}$ maps a concrete program $c \in \textsc{Prog}$ to its *set of concrete values*:

$$\mathbb{C}(c) := \mathsf{terms}(C, \{\bot\})$$

with $C \subseteq \textsc{Con}$ being the set of constructor names that appear in c.

Notation For readability we omit the fact that $\mathbb{C}$ is a function on concrete programs. Because we always consider only a single concrete program $c \in \textsc{Prog}$ at a time, it is safe for us to denote $\mathbb{C}(c)$ by $\mathbb{C}$.

Example 3.51 Consider a concrete program containing the following type declarations:

```
data Bool = False | True
data Maybe a = Nothing | Just a
```

Then, $\{\texttt{False}, \texttt{True}, \texttt{Nothing}, \texttt{Just}\,\bot, \texttt{Just (Just True)}\} \subsetneq \mathbb{C}$.

In the following, we want to differentiate between concrete values according to their type.

Definition 3.52 $\mathbb{C}_T \subseteq \mathbb{C}$ is defined as the *set of values in* $\mathbb{C}$ *of type* $T \in \textsc{Type}_0$ so that:

$$\forall T \in \textsc{Type}_0 : \bot \in \mathbb{C}_T$$

Example 3.53 Consider a concrete program with the following type declarations:

```
data Bool = False | True
data Maybe a = Nothing | Just a
```

Then,

1. $\mathbb{C}_{\texttt{Bool}} = \{\texttt{False}, \texttt{True}, \bot\}$ and
2. $\mathbb{C}_{\texttt{Maybe Bool}} = \{\texttt{Nothing}, \texttt{Just False}, \texttt{Just True}, \texttt{Just}\,\bot, \bot\}$

We also want to differentiate types $T \in \textsc{Type}_0$ by the cardinality of the set $\mathbb{C}_T$.

Definition 3.54 A type $T \in \textsc{Type}_0$ is denoted to be *infinite* if $\mathbb{C}_T$ is infinite. Consequently, T is denoted to be *finite* if $\mathbb{C}_T$ is finite.

Example 3.55 Consider a concrete program with the following type declarations:

```
data Bool = False | True
data List a = Nil | Cons a (List a)
```

Then, `Bool` is finite because $\mathbb{C}_{\texttt{Bool}}$ has a cardinality of three, whereas `List Bool` is infinite because $\mathbb{C}_{\texttt{List Bool}}$ is infinite.

Now that we have specified concrete values, we define the dynamic semantics of expressions in a concrete program. In order to specify the dynamic semantics of case distinctions, we need to determine which branch to evaluate. This is done by matching the patterns of all branches against the value of the discriminant.

Definition 3.56 $\mathsf{matches} : \textsc{Pat} \times \mathbb{C} \rightarrow \mathbb{B}$ is defined as a binary predicate that holds if a pattern $p \in \textsc{Pat}$ matches a concrete value $v \in \mathbb{C}$:

$$\mathsf{matches}(p, v) \coloneqq \begin{cases} \mathsf{True} & \text{if } p = C\ p_1 \dots p_n \text{ and } v = C\ v_1 \dots v_n \\ & \text{where } C \in \textsc{Con} \text{ and } n = \mathsf{arity}(C) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

Example 3.57 $\mathsf{matches}$ holds for the following pairs of arguments:

$$(\texttt{Just x}, \texttt{Just False}) \qquad (\texttt{False}, \texttt{False})$$

In a case distinction, patterns not only determine which branch to evaluate but also bind constructor arguments to new variables.

Definition 3.58 $\mathsf{bindMatch} : \textsc{Pat} \times \mathbb{C} \nrightarrow \mathbb{C}^{\{p_1,\dots,p_n\}}$ gives a mapping from the set of variables $\{p_1, \dots, p_n\}$ contained in a pattern $C\ p_1 \dots p_n \in \textsc{Pat}$ to the concrete values that these variables bind to according to a concrete value $C\ v_1 \dots v_n \in \mathbb{C}$ with $n = \mathsf{arity}(C)$:

$$\mathsf{bindMatch}(C\ p_1 \dots p_n, C\ v_1 \dots v_n) \coloneqq \{(p_1, v_1), \dots, (p_n, v_n)\}$$

For all $(p, v) \in \textsc{Pat} \times \mathbb{C}$, $\mathsf{bindMatch}(p, v)$ is not defined if $v = \bot$ or $\mathsf{matches}(p, v) = \mathsf{False}$.

Example 3.59

1. $\mathsf{bindMatch}(\texttt{Just x}, \texttt{Just False}) = \{(\texttt{x}, \texttt{False})\}$
2. $\mathsf{bindMatch}(\texttt{False}, \texttt{False}) = \{\}$

We specify the evaluation of expressions in a concrete program.

Definition 3.60 $\mathsf{concrete\text{-}value}_{\textsc{Exp}} : \textsc{Prog} \times \mathbb{C}^{\textsc{Var}} \times \textsc{Exp} \rightarrow \mathbb{C}$ evaluates an expression $e \in \textsc{Exp}$ of a concrete program $c \in \textsc{Prog}$ in the context of an environment $E \in \mathbb{C}^{\textsc{Var}}$ such that

$$\textsf{concrete-value}_{\textsc{Exp}}(c, E, e) \coloneqq$$

$$\begin{cases}
E(e) & \text{if } e \in \textsc{Var} \text{ and } e \in \operatorname{dom}(E) \\
e & \text{if } e \in \textsc{Con} \\
\begin{array}{ll} C & \textsf{concrete-value}_{\textsc{Exp}}(c, E, e_1) \\ & \ldots \\ & \textsf{concrete-value}_{\textsc{Exp}}(c, E, e_n) \end{array} & \begin{array}{l}\text{if } e \text{ is a constructor} \\ \text{application } C\ e_1 \ldots e_n\end{array} \\
\begin{array}{ll} \multicolumn{2}{l}{\textsf{concrete-value}_{\textsc{Exp}}(c, E', e')} \\ \text{with} & v_1 = \textsf{concrete-value}_{\textsc{Exp}}(c, E, a_1) \\ & \quad \ldots \\ & v_n = \textsf{concrete-value}_{\textsc{Exp}}(c, E, a_n) \\ & E' = ((E[x_1/v_1]) \ldots)[x_n/v_n] \end{array} & \begin{array}{l}\text{if } e \text{ is an application} \\ f\ a_1 \ldots a_n \text{ with} \\ f \texttt{ = } \backslash x_1 \ldots x_n \texttt{ -> } e' \text{ being} \\ \text{a declaration in } c\end{array} \\
\begin{array}{ll} \multicolumn{2}{l}{\textsf{concrete-value}_{\textsc{Exp}}(c, E', y)} \\ \text{with} & v = \textsf{concrete-value}_{\textsc{Exp}}(c, E, x) \\ & E' = E[\textsf{bindMatch}(p, v)] \end{array} & \begin{array}{l}\text{if } e \text{ is a case distinction} \\ \texttt{case } x \texttt{ of } \ldots\ p \texttt{ -> } y\ \ldots \\ \text{with } v \neq \bot \text{ and } \textsf{matches } (p, v) \\ \text{holds for pattern } p \in \textsc{Pat}\end{array} \\
\begin{array}{ll} \multicolumn{2}{l}{\textsf{concrete-value}_{\textsc{Exp}}(c, E_n, e')} \\ \text{with} & v_1 = \textsf{concrete-value}_{\textsc{Exp}}(c, E, a_1) \\ & E_1 = E[x_1/v_1] \\ & v_2 = \textsf{concrete-value}_{\textsc{Exp}}(c, E_1, a_2) \\ & E_2 = E_1[x_2/v_2] \\ & \quad \ldots \\ & v_n = \textsf{concrete-value}_{\textsc{Exp}}(c, E_{n-1}, a_n) \\ & E_n = E_{n-1}[x_n/v_n] \end{array} & \begin{array}{l}\text{if } e \text{ is a local binding} \\ \texttt{let } x_1 \texttt{ = } a_1 \\ \qquad \ldots \\ \quad\ x_n \texttt{ = } a_n \\ \texttt{in } e'\end{array} \\
\bot & \text{otherwise}
\end{cases}$$

Note the following remarks:

1. $\textsf{concrete-value}_{\textsc{Exp}}$ gives $\bot$ for abstractions.
2. According to Definition 3.43 there is exactly one pattern $p \in \textsc{Pat}$ in a case distinction that matches the discriminant's value $v \in \mathbb{C}$ if $v \neq \bot$. Therefore, $\textsf{bindMatch}(p, v)$ is always defined.
3. For $(x, v) \in \textsc{Var} \times \mathbb{C}$, $E[x/v]$ denotes the update of E by the tuple (x, v) (cf. Definition A.17).
4. For $(p, v) \in \textsc{Pat} \times \mathbb{C}$, $E[\textsf{bindMatch}(p, v)]$ denotes the update of E by the assignment resulting from evaluating $\textsf{bindMatch}(p, v)$ (cf. Definition A.18).

Finally, we specify the dynamic semantics of concrete programs.

Definition 3.61 For two types $P, U \in \textsc{Type}_0$, $\textsf{concrete-value} : \textsc{Prog}_{PU} \to (\mathbb{C}_P \times \mathbb{C}_U \to \mathbb{C}_{\texttt{Bool}})$ evaluates a concrete program $c \in \textsc{Prog}_{PU}$ such that

$$\textsf{concrete-value}(c) := \{((v_p, v_u), \textsf{concrete-value}_{\textsc{Exp}}(c, \{(p, v_p), (u, v_u)\}, e)) \mid v_p \in \mathbb{C}_P \wedge v_u \in \mathbb{C}_U\}$$

where $e \in \textsc{Exp}$ denotes the expression bound in the `constraint` declaration of c:

```
constraint = \p u -> e
```

Note that the set $\{(p, v_p), (u, v_u)\}$ denotes the initial environment used for evaluating the expression e.

The result of evaluating a concrete program with $\textsf{concrete-value}$ is a constraint (cf. Definition 3.1) where the domain of discourse, the parameter domain, and the Boolean values are represented by sets of concrete values.

Example 3.62 Evaluating the concrete program $c \in \textsc{Prog}_{\texttt{Nat},\texttt{Pair Nat Nat}}$ from Example 3.9 gives the following constraint:

$$\textsf{concrete-value}(c) = \{((v_p, v_u), R(v_p, v_u)) \mid v_p \in \mathbb{C}_{\texttt{Nat}}, v_u \in \mathbb{C}_{\texttt{Pair Nat Nat}}\}$$

where $R(v_p, v_u) =$

$$\textsf{concrete-value}_{\textsc{Exp}}(c, \{(\texttt{p}, v_p), (\texttt{u}, v_u)\}, \texttt{case u of Pair a b -> ...})$$

The following values are included in $\textsf{concrete-value}_{\texttt{Nat},\texttt{Pair Nat Nat}}(c)$:

$$\textsf{concrete-value}_{\texttt{Nat},\texttt{Pair Nat Nat}}(c) \supsetneq \left\{\begin{array}{lll} ((\texttt{Z}, & \texttt{Pair Z Z}), & \texttt{False}), \\ ((\texttt{S Z}, & \texttt{Pair (S Z) (S Z)}), & \texttt{False}), \\ ((\texttt{S(S Z)}, & \texttt{Pair (S(S Z)) (S Z)}), & \texttt{False}), \\ ((\texttt{S(S(S(S Z)))}, & \texttt{Pair (S(S Z)) (S(S Z))}), & \texttt{True}), \\ ((\texttt{S(S(S(S Z)))}, & \texttt{Pair (S(S Z)) (S(S(S Z)))}), & \texttt{False}) \end{array}\right\}$$

Definition 3.3 already specified when an element of the domain of discourse is considered to be a solution for a constraint and a given parameter. Now that we have fixed the semantics of concrete programs, we lift this definition to specify the solution for a concrete program.

Definition 3.63 For two types $P, U \in \textsc{Type}_0$, a concrete value $v_u \in \mathbb{C}_U \setminus \{\bot\}$ is a *solution* for a concrete program $c \in \textsc{Prog}_{PU}$ and a parameter $v_p \in \mathbb{C}_P$ if

$$\textsf{concrete-value}(c)(v_p, v_u) = \texttt{True}$$

Notation Note that the parameter domain P and the domain of discourse U in Definition 3.1 are written in an upright font whereas the types $P, U \in \textsc{Type}_0$ in this section are written in italics. The rationale of the different notation is the following: the parameter domain P and the domain of discourse U are distinct sets which are not restricted in any way, i.e., their elements may have some arbitrary shape. On the other hand, P and U denote types whose corresponding sets of values $\mathbb{C}_P$ and $\mathbb{C}_U$ represent a particular parameter domain and domain of discourse, respectively. But as there are domains P and U which cannot be represented by the values $\mathbb{C}_P$ and $\mathbb{C}_U$ for any types $P, U \in \textsc{Type}_0$, we opt for a different notation of unrestricted sets and types, respectively.

3.3 Language of Abstract Programs

The first step in deriving a propositional formula from a concrete program and a parameter is the generation of an abstract program (cf. Figure 3.7). There are two essential differences between concrete and abstract programs:

1. abstract programs operate on the domain of abstract values, and
2. abstract programs do not contain case distinctions.

Both differences result from the fact that an abstract program deals with data that may be undetermined when evaluating the program. For example, only the type is known for the designated solution of a constraint, but not its value. On the other hand, the value of the constraint's parameter is always known when evaluating an abstract program. In order to handle both cases, a single abstract value represents a finite set of concrete values. Example 3.64 illustrates both cases in a simple concrete program.

Example 3.64 The following concrete program $c \in \textsc{Prog}$ specifies the conjunction of two Boolean variables.

```
data Bool = False | True

constraint :: Bool -> Bool -> Bool
constraint = \p u -> and p u

and :: Bool -> Bool -> Bool
and = \p u -> case p of False -> False
                        True  -> u
```

Compiling c to an abstract program $c_{\mathbb{A}}$ using the compilation function introduced in Chapter 4 transforms the domain of all values to the domain of abstract values. An abstract value representing the concrete value `u` actually represents both concrete values `False` and `True` because the exact value is not known when evaluating $c_{\mathbb{A}}$. On the other hand, the value of parameter p

is known when evaluating $c_{\mathbb{A}}$ (cf. Figure 3.7), thus, an abstract value representing p actually represents a single concrete value `False` or `True`, but not both of them.

The reason that an abstract value a may only represent a finite set of concrete values $C \subseteq \mathbb{C}$ is that the cardinality of C determines the number of propositional variables that are needed to encode a. If $|C| = n$ for $n \in \mathbb{N}$, then a can be encoded in binary using $\lceil \log_2 n \rceil$ propositional variables. Thus, C must be finite in order to encode a using a finite number of propositional variables.

The restriction to finite sets implies that there is no abstract value a that represents all concrete values $\mathbb{C}_T$ for $T \in \textsc{Type}_0$ if $\mathbb{C}_T$ is infinite. This applies if T is recursively defined, e.g., type `Nat` in Example 3.9. In this case, the set of concrete values that is represented by a needs to be restricted to a finite subset (cf. Section 4.1.5). This restriction induces that CO^4 is an incomplete constraint solver when dealing with recursively defined data types. Example 4.63 gives a concrete program whose obvious solution is not found by CO^4.

Besides the presence of abstract values, the lack of case distinctions is the second notable difference of abstract programs. That is because there is no way to evaluate case distinctions on potentially undetermined discriminants. While Chapter 4 shows in detail how concrete programs with case distinctions are compiled into abstract programs without case distinctions, Example 3.65 glances at the result of this compilation for an exemplary concrete program.

Example 3.65 The following abstract program $c_{\mathbb{A}}$ is the result of compiling the concrete program c from Example 3.64 using the compilation function introduced in Chapter 4:

```
constraint𝔸 = \p u -> and p u

and = \p u -> let v_d = p
              in valid_v_d ( let v_1 = cons_(1,2)
                                 v_2 = u
                             in merge_v_d v_1 v_2 )
```

The right-hand sides of the let-bindings to `v_1` and `v_2` represent the compiled branches of the case distinction in the definition of `and` from the concrete program c. According to the dynamic semantics of abstract programs (cf. Section 3.3.3), both compiled branches are evaluated and their values are eventually merged using the built-in function `merge`. The function `merge`$_{\texttt{v\_d}}$ `v_1 v_2` produces an abstract value that encodes the original case distinction in terms of propositional variables and logical connectives.

The built-in function `cons` simulates constructor calls: in this example, `cons`$_{\texttt{(1,2)}}$ gives an abstract value that represents the concrete value `False`. Note that the subscript `(1,2)` results from the fact that `False` is the first of two constructors of its corresponding type. Section 4.1.3 describes the semantics of `cons` in more detail.

The built-in function `valid` denotes a validity check for abstract values. This check is necessary for simulating the semantics of case distinctions on discriminants that evaluate to $\bot \in \mathbb{C}$: recall that in this situation, the case distinction evaluates to $\bot$ as well (cf. Definition 3.60). The function `valid` mimics this behavior for compiled case distinctions.

Note that the abstract program $c_{\mathbb{A}}$ does not contain any type signatures. That is because abstract programs operate only on abstract values and functions of abstract values. Thus, type signatures are neither required nor allowed in abstract programs as they do not reveal any additional information about the program.

The following sections in this chapter specify the abstract syntax trees and semantics of abstract programs. Again, we do not consider aspects related to the textual representation of abstract programs, e.g., the placement of parentheses.

3.3.1 Syntax

The syntax of abstract programs is more restricted than the syntax of concrete programs. An expression in an abstract program may either be a variable, a call to a built-in function (`arguments`, `cons`, `merge`, `valid`), a function call, an abstraction or a local binding.

Definition 3.66 The *syntax of abstract expressions* $\textsc{ExpSyntax}_{\mathbb{A}}$ is defined by the following EBNF:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textsc{ExpSyntax}_{\mathbb{A}} \coloneqq} \\
 & \textsc{Var} & \\
 | & \texttt{"arguments"}_{\mathbb{N}}\ \textsc{Var} & \text{(argument access)} \\
 | & \texttt{"cons"}_{(\mathbb{N},\mathbb{N})}\ \textsc{Var}^{*} & \text{(constructor call)} \\
 | & \texttt{"merge"}_{\textsc{Var}}\ \textsc{Var}^{+} & \text{(merge)} \\
 | & \texttt{"valid"}_{\textsc{Var}}\ \textsc{Var} & \text{(validity check)} \\
 | & \textsc{Var}\ \textsc{Var}^{+} & \text{(application)} \\
 | & \texttt{"\textbackslash"}\ \textsc{Var}^{+}\ \texttt{"->"}\ \textsc{ExpSyntax}_{\mathbb{A}} & \text{(abstraction)} \\
 | & \texttt{"let"}\ (\textsc{Var}\ \texttt{"="}\ \textsc{ExpSyntax}_{\mathbb{A}})^{+} & \text{(local binding)} \\
 & \texttt{"in"}\ \textsc{ExpSyntax}_{\mathbb{A}} &
\end{array}
$$

Note that there are no case distinctions in an abstract program.

Example 3.67 An exemplary abstract expression is

```
let x' = arguments₁ x
in
```

```
  let a_1 = plus x' y
  in
    cons(2,2) a_1
```

An abstract program consists of a sequence of declarations where each declaration binds an identifier to an abstract expression.

Definition 3.68 The *syntax of abstract declarations* $\textsc{DeclSyntax}_{\mathbb{A}}$ is defined by the following EBNF:

$$\textsc{DeclSyntax}_{\mathbb{A}} \coloneqq \textsc{Var} \texttt{ "=" } \textsc{ExpSyntax}_{\mathbb{A}}$$

Example 3.69 An exemplary abstract declaration is

```
f = \x -> g x x
```

An abstract program is a non-empty sequence of abstract declarations.

Definition 3.70 The *syntax of abstract programs* $\textsc{ProgSyntax}_{\mathbb{A}}$ is defined by the following EBNF:

$$\textsc{ProgSyntax}_{\mathbb{A}} \coloneqq \textsc{DeclSyntax}_{\mathbb{A}}^{+}$$

Example 3.71 shows an exemplary abstract program.

Example 3.71 The following listing shows an excerpt of the abstract program that is the result of compiling the concrete program from Example 3.9 using the compilation specified in Section 4.2.

```
constraint_𝔸 = \p u ->
  let v_d = u
  in
    valid_{v_d}
      ( let v_1 = let a = arguments_1 v_d
                      b = arguments_2 v_d
                  in
                    let v_2 = greaterOne a
                        v_3 =
                          let v_4 = greaterOne b
                              v_5 =
                                let v_6 = p
                                    v_7 =
                                      let v_8 = a
                                          v_9 = b
                                      in
                                        times v_8 v_9
                                in
                                  eq v_6 v_7
                          in
                            and2 v_4 v_5
                    in
                      and2 v_2 v_3
        in
          merge_{v_d} v_1 )

plus = \x y ->
  let v_d = x
  in
    valid_{v_d} ( let v_1 = y
                  v_2 = let x' = arguments_1 v_d
                        in
                          let a_1 =
                                let v_3 = x'
                                    v_4 = y
                                in
                                  plus v_3 v_4
                          in
                            cons_{(2,2)} a_1
              in
                merge_{v_d} v_1 v_2 )
```

The complete listing can be found in Appendix C.1.

3.3.2 Static Semantics

In contrast to concrete programs, which operate on concrete values, abstract programs operate on abstract values.

Definition 3.72 The *set of abstract values* $\mathbb{A}$ is defined as the least set A for which the following properties hold:

1. $\bot_{\mathbb{A}} \in A$,
2. $\forall(\vec{f},\vec{a}) \in \mathrm{F}^* \times A^* : (\vec{f},\vec{a}) \in A$.

Recall that F denotes the set of propositional formulas (cf. Definition B.4).

Notation Except for $\bot_{\mathbb{A}}$, an abstract value $(\vec{f},\vec{a}) \in \mathbb{A}$ consists of a sequence of propositional formulas $\vec{f}$ and a sequence of abstract values $\vec{a}$. In the following, we denote each element in $\vec{f}$ as *flag* and each element in $\vec{a}$ as *argument* of an abstract value.

The flags of an abstract value represent a constructor index (cf. Definition 3.35) in binary code under an assignment for all contained propositional variables. The arguments of an abstract value encode the corresponding constructor arguments. In Example 3.73, we show a simple abstract value and map it to different concrete values. Details about the transformation between abstract and concrete values are given in Section 4.1.

Example 3.73 Assume the following type declaration in a concrete program:

```
data Nat = Z | S Nat
```

From Definition 3.52 we know that $\{\mathtt{Z}, \mathtt{S}\,\mathtt{Z}, \mathtt{S}\,\bot\} \subsetneq \mathbb{C}_{\mathtt{Nat}}$. Furthermore, assume an abstract value $a_1 \in \mathbb{A}$ containing one flag $f_1 \in \mathrm{V}$ and one argument $a_2 \in \mathbb{A}$ that itself contains a single flag $f_2 \in \mathrm{V}$:

$$a_1 = \big((f_1),(a_2)\big)$$
$$a_2 = \big((f_2),()\big)$$

The abstract value a_1 can be mapped to different concrete values by assigning different truth values to f_1 and f_2:

f_1	f_2	concrete value
False	False	Z
False	True	Z
True	False	S Z
True	True	S $\bot$

Now that we have specified the values that abstract programs operate on, we define the set of statically well-defined abstract expressions.

Definition 3.74 The *set of statically well-defined abstract expressions* $\textsc{Exp}_{\mathbb{A}}$ is the set of all abstract expressions $e \in \textsc{ExpSyntax}_{\mathbb{A}}$ such that all of the following properties hold for e:

1. the type of e, according to Hindley–Damas–Milner type inference [22], is either $\mathbb{A}$ or a n-ary function on $\mathbb{A}$ for $n \in \mathbb{N}_{>0}$,
2. e may not contain an abstraction as a strict subexpression,
3. each application is total, and
4. the same rules apply for local bindings in abstract programs as for local bindings in concrete programs (cf. Definition 3.43), i.e., each bound expression may only depend on previously bound expressions.

We define the set of statically well-defined abstract declarations.

Definition 3.75 The *set of statically well-defined abstract declarations* $\textsc{Decl}_{\mathbb{A}}$ is the set of all declarations in $\textsc{DeclSyntax}_{\mathbb{A}}$ that bind an abstract expression from $\textsc{Exp}_{\mathbb{A}}$.

Finally, we define the set of statically well-defined abstract programs.

Definition 3.76 The *set of statically well-defined abstract programs* $\textsc{Prog}_{\mathbb{A}}$ is the set of all abstract programs $c_{\mathbb{A}} \in \textsc{ProgSyntax}_{\mathbb{A}}$ such that all of the following properties hold for $c_{\mathbb{A}}$:

1. $c_{\mathbb{A}}$ only contains declarations from $\textsc{Decl}_{\mathbb{A}}$, and
2. $c_{\mathbb{A}}$ contains exactly one declaration of the form

```
constraint𝔸 = \p u -> e
```

   where $e \in \textsc{Exp}_{\mathbb{A}}$.

The abstract program in Example 3.71 is statically well-defined.

3.3.3 Dynamic Semantics

We specify the dynamic semantics of abstract programs by providing an evaluation function for abstract programs and abstract expressions. Again, we do not deal with problems related to non-termination: we only give the semantics of terminating abstract programs.

Definition 3.77 $\mathsf{abstract\text{-}value}_{\textsc{Exp}} : \textsc{Prog}_{\mathbb{A}} \times \mathbb{A}^{\textsc{Var}} \times \textsc{Exp}_{\mathbb{A}} \to \mathbb{A}$ evaluates an expression $e \in \textsc{Exp}_{\mathbb{A}}$ of an abstract program $c \in \textsc{Prog}_{\mathbb{A}}$ in the context of an environment $E_{\mathbb{A}} \in \mathbb{A}^{\textsc{Var}}$ such that

$$\mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, e) :=$$

$$\begin{cases}
E_{\mathbb{A}}(e) & \text{if } e \in \textsc{Var} \\[2ex]
\mathsf{arguments}_i(v) & \text{if } e = \texttt{arguments}_i\ \ e' \\
\text{with } v = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, e') & \text{with } i \in \mathbb{N}_{>0} \\[2ex]
\mathsf{cons}_{(j,k)}(v_1, \ldots, v_n) & \text{if } e = \texttt{cons}_{(j,k)}\ \ a_1 \ldots a_n \\
\text{with } \ v_1 = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, a_1) & \text{with } j, k \in \mathbb{N}_{>0}, \\
\qquad\quad \ldots & j \in \{1 \ldots k\}, \text{ and } n \in \mathbb{N} \\
\qquad\ v_n = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, a_n) & \\[2ex]
\mathsf{merge}_{E_{\mathbb{A}}(v)}(v_1, \ldots, v_n) & \text{if } e = \texttt{merge}_v\ \ a_1 \ldots a_n \\
\text{with } \ v_1 = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, a_1) & \text{with } n \in \mathbb{N}_{>0} \\
\qquad\quad \ldots & \\
\qquad\ v_n = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, a_n) & \\[2ex]
\mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, e') & \text{if } e = \texttt{valid}_v\ \ e' \\
 & \text{with } E_{\mathbb{A}}(v) \neq \bot_{\mathbb{A}} \\[2ex]
\mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E'_{\mathbb{A}}, e') & \text{if } e \text{ is an application} \\
\text{with } \ \ v_1 = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, a_1) & f\ a_1 \ldots a_n \text{ with } n \in \mathbb{N} \text{ and} \\
\qquad\quad \ldots & f\, \texttt{=}\backslash\, x_1 \ldots x_n\ \texttt{->}\, e' \text{ being} \\
\qquad\ v_n = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, a_n) & \text{a declaration in } c \\
\qquad\ E'_{\mathbb{A}} = ((E_{\mathbb{A}}[x_1/v_1]) \ldots)[x_n/v_n] & \\[2ex]
\mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{n\mathbb{A}}, e') & \text{if } e \text{ is a local binding} \\
\text{with } \ \ \ v_1 = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, a_1) & \texttt{let}\ \ x_1\, \texttt{=}\ a_1 \\
\qquad E_{1\mathbb{A}} = E_{\mathbb{A}}[x_1/v_1] & \qquad\quad \ldots \\
\qquad\ \ v_2 = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{1\mathbb{A}}, a_2) & \qquad x_n\, \texttt{=}\ a_n \\
\qquad E_{2\mathbb{A}} = E_{1\mathbb{A}}[x_2/v_2] & \texttt{in}\ e' \\
\qquad\quad \ldots & \text{with } n \in \mathbb{N}_{>0} \\
\qquad\ \ v_n = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, E_{n-1\mathbb{A}}, a_n) & \\
\qquad E_{n\mathbb{A}} = E_{n-1\mathbb{A}}[x_n/v_n] & \\[2ex]
\bot_{\mathbb{A}} & \text{otherwise}
\end{cases}$$

Note the following remarks:

1. Because the functions arguments, cons, and merge (Definition 4.4, 4.32, and 4.41, respectively) depend on how concrete values are encoded as abstract values, we give their definitions not until we have introduced the encoding of concrete values in Section 4.1.

2. The validity check gives $\bot_{\mathbb{A}}$ if the checked variable evaluates to $\bot_{\mathbb{A}}$.
3. For $(x, v) \in \textsc{Var} \times \mathbb{A}$, $E_{\mathbb{A}}[x/v]$ denotes the update of $E_{\mathbb{A}}$ by the tuple (x, v) (cf. Definition A.17).

Finally, we specify the evaluation of abstract programs.

Definition 3.78 $\mathsf{abstract\text{-}value} : \textsc{Prog}_{\mathbb{A}} \to (\mathbb{A} \times \mathbb{A} \to \mathbb{A})$ evaluates an abstract program $c \in \textsc{Prog}_{\mathbb{A}}$ such that

$$\mathsf{abstract\text{-}value}(c) \coloneqq \\ \{((v_p, v_u), \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c, \{(p, v_p), (u, v_u)\}, e)) \mid v_p \in \mathbb{A} \land v_u \in \mathbb{A}\}$$

where c contains the following declaration

```
constraint𝔸 = \p u -> e
```

with $p, u \in \textsc{Var}$ and $e \in \textsc{Exp}_{\mathbb{A}}$. Note that the set $\{(p, v_p), (u, v_u)\}$ denotes the initial environment used for evaluating the expression e.

The result of evaluating an abstract program with $\mathsf{abstract\text{-}value}$ is a binary function on abstract values.

3.4 Correctness Criterion for Concrete and Abstract Programs

This section gives a correctness criterion that specifies if an abstract program is a correct compilation of a concrete program. A compilation function that meets this specification is a necessary requirement for proving Theorem 3.8. It is left to the subsequent chapters to provide an actual implementation of such a compilation function.

Recall that abstract programs operate on abstract values. Thus, we define encode/decode-pairs for mapping between concrete and abstract values.

Definition 3.79 An *encode/decode-pair* is a pair $(\mathfrak{E}, \mathfrak{D})$ where

$$\mathfrak{E} \coloneqq \big(\mathfrak{E}_T \mid T \in \textsc{Type}_0\big) \quad \text{and} \quad \mathfrak{D} \coloneqq \big(\mathfrak{D}_T \mid T \in \textsc{Type}_0\big)$$

are families of mappings $\mathfrak{E}_T : \mathbb{C}_T \to \mathbb{A}$ and $\mathfrak{D}_T : \mathbb{B}^{\mathrm{V}} \times \mathbb{A} \to \mathbb{C}_T$ for all types $T \in \textsc{Type}_0$ such that

1. $\mathfrak{E}_T(\bot) = \bot_{\mathbb{A}}$,
2. $\forall \sigma \in \mathbb{B}^{\mathrm{V}} : \mathfrak{D}_T(\sigma, \bot_{\mathbb{A}}) = \bot$, and
3. $\forall (v, \sigma) \in \mathbb{C}_T \times \mathbb{B}^{\mathrm{V}} : \mathfrak{D}_T(\sigma, \mathfrak{E}_T(v)) = v$.

For an encode/decode-pair $(\mathfrak{E}, \mathfrak{D})$ and a type $T \in \textsc{Type}_0$, Definition 3.79 requires that the value of $\mathfrak{D}_T(\sigma, a)$ is independent of any assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$ if $a \in \mathbb{A}$ is the result of $\mathfrak{E}_T(v)$ for a particular concrete value $v \in \mathbb{C}_T$.

In the following, we specify whether a concrete program has been correctly compiled into an abstract program with respect to a particular encode/decode-pair.

Definition 3.80 A compilation function $\mathsf{compile} : \textsc{Prog} \to \textsc{Prog}_{\mathbb{A}}$ is *correct with respect to an encode/decode-pair* $(\mathfrak{E}, \mathfrak{D})$ if the following property holds for each pair $(c, c_{\mathbb{A}}) \in \mathsf{compile}$ with $c \in \textsc{Prog}_{PU}$ and $P, U \in \textsc{Type}_0$:

$$\forall (p, u_{\mathbb{A}}, \sigma) \in \mathbb{C}_P \times \mathbb{A} \times \mathbb{B}^{\mathrm{V}} :$$
$$\mathfrak{D}_{\mathtt{Bool}}(\sigma, \mathsf{abstract\text{-}value}(c_{\mathbb{A}})(\mathfrak{E}_P(p), u_{\mathbb{A}})) = \mathsf{concrete\text{-}value}(c)(p, \mathfrak{D}_U(\sigma, u_{\mathbb{A}}))$$

For each triple $(p, u_{\mathbb{A}}, \sigma) \in \mathbb{C}_P \times \mathbb{A} \times \mathbb{B}^{\mathrm{V}}$ of a parameter p, an abstract value $u_{\mathbb{A}}$, and a propositional assignment σ, Definition 3.80 requires that both ways of evaluation lead to the same result:

1. Evaluating the abstract program $c_{\mathbb{A}} \in \textsc{Prog}_{\mathbb{A}}$ to $\mathsf{abstract\text{-}value}(c_{\mathbb{A}})$ $(\mathfrak{E}_P(p), u_{\mathbb{A}})$ in the first place, and then decoding the result.
2. Decoding $u_{\mathbb{A}}$ in the first place, and then evaluating the concrete program $c \in \textsc{Prog}_{PU}$ to $\mathsf{concrete\text{-}value}(c)(p, \mathfrak{D}_U(\sigma, u_{\mathbb{A}}))$.

Figure 3.81 illustrates both ways of evaluation in a commutative diagram.

$$\begin{tikzcd}
(p, u_{\mathbb{A}}, \sigma) \in \mathbb{C}_P \times \mathbb{A} \times \mathbb{B}^{\mathrm{V}} \arrow[r, "\mathsf{abstract\text{-}value}(c_{\mathbb{A}})(\mathfrak{E}_P(p), u_{\mathbb{A}})"] \arrow[d, "\mathfrak{D}_U(\sigma, u_{\mathbb{A}})"] & i_1 \in \mathbb{A} \arrow[d, "\mathfrak{D}_{\mathtt{Bool}}(\sigma, i_1)"'] \\
i_2 \in \mathbb{C}_U \arrow[r, "\mathsf{concrete\text{-}value}(c)(p, i_2)"'] & b \in \mathbb{C}_{\mathtt{Bool}}
\end{tikzcd}$$

Figure 3.81: $c_{\mathbb{A}} \in \textsc{Prog}_{\mathbb{A}}$ is a correct compilation of the concrete program $c \in \textsc{Prog}_{PU}$ if both evaluations lead to the same result $b \in \mathbb{C}_{\mathtt{Bool}}$ for all $(p, u_{\mathbb{A}}, \sigma) \in \mathbb{C}_P \times \mathbb{A} \times \mathbb{B}^{\mathrm{V}}$.

We show the following Lemma:

Lemma 3.82 Let $(\mathfrak{E}, \mathfrak{D})$ be an encode/decode-pair and $\mathsf{compile} : \textsc{Prog} \to \textsc{Prog}_{\mathbb{A}}$ a compilation function that is correct with respect to $(\mathfrak{E}, \mathfrak{D})$. Then, the function $\mathsf{compile}$ induces a correct solving procedure (cf. Theorem 3.8) for a constraint specified as a concrete program $c \in \textsc{Prog}_{PU}$ with $P, U \in \textsc{Type}_0$.

Proof Let

1. $P \in \textsc{Type}_0$ denote a parameter domain with $p \in \mathbb{C}_P$,
2. $U \in \textsc{Type}_0$ denote a domain of discourse,
3. $\mathsf{compile}$ denote a compilation function that is correct with respect to the encode/decode-pair $(\mathfrak{E}, \mathfrak{D})$,
4. $c \in \textsc{Prog}_{PU}$ denote a concrete program, and
5. $u_{\mathbb{A}} \in \mathbb{A}$ denote an abstract value.

Then,

$$\forall \sigma \in \mathbb{B}^{\mathrm{V}} : \mathfrak{D}_{\mathtt{Bool}}(\sigma, \mathsf{abstract\text{-}value}(\mathsf{compile}(c))(\mathfrak{E}_P(p), u_{\mathbb{A}})) = \mathtt{True}$$
$$\Longrightarrow$$
$$\mathsf{concrete\text{-}value}(c)(p, \mathfrak{D}_U(\sigma, u_{\mathbb{A}})) = \mathtt{True}$$

This result is directly implied by the definition of a correct compilation function: we just fixed the parameter p and the abstract value $u_{\mathbb{A}}$. Recall that Definition 3.80 requires that concrete and abstract evaluation give the same result, but in different domains. Thus, if abstract evaluation gives an abstract value that decodes to `True` under an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$, then the concrete evaluation gives `True` as well. As we can compute a solution $\mathfrak{D}_U(\sigma, u_{\mathbb{A}}) \in \mathbb{C}_U$ from such an assignment, we have a correct solving procedure for constraints that are specified as concrete programs. ■

Chapter 4

Compilation of Concrete Programs

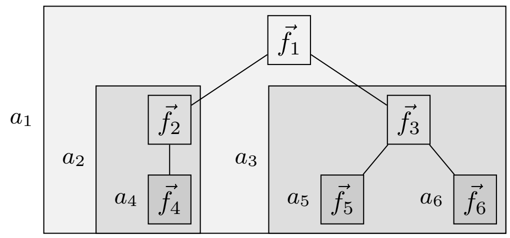
According to Figure 3.7, evaluating an abstract program for a given parameter results in a formula that represents the original constraint in terms of a satisfiability problem in propositional logic. As the constraint is specified as a concrete program, this chapter gives a compilation function from concrete programs to abstract programs. This compilation contains two essential parts: changing the underlying domain from concrete values to abstract values, and transforming case distinctions. Recall that case distinctions in concrete programs need to be handled specifically because they are not allowed in abstract programs.

Section 4.1 covers the transformation from concrete to abstract values and implements encoding and decoding functions. Section 4.2 describes the compilation function itself by specifying the compilation of all entities of a concrete program. In Section 4.2.1 we show that the compilation is correct according to Definition 3.80. This chapter concludes by illustrating the usage of the constraint solver CO^4 and addresses some aspects of its present implementation.

4.1 Data Transformation

Compiling a concrete into an abstract program changes the domain of values that is operated on: while concrete programs operate on concrete values, abstract programs operate on abstract values. Recall that each abstract value in $\mathbb{A}$ other than $\perp_{\mathbb{A}}$ is a tuple $(\vec{f}, \vec{a})$ containing a sequence of flags $\vec{f} \in \mathrm{F}^*$ and a sequence of arguments $\vec{a} \in \mathbb{A}^*$ (cf. Definition 3.72). Figure 4.1 illustrates an exemplary abstract value as a tree of flags.

The sequence of flags in an abstract value encodes the index of a constructor



$$
\begin{aligned}
a_1 &= (\vec{f}_1, (a_2, a_3)) \\
a_2 &= (\vec{f}_2, (a_4)) \\
a_3 &= (\vec{f}_3, (a_5, a_6)) \\
a_4 &= (\vec{f}_4, ()) \\
a_5 &= (\vec{f}_5, ()) \\
a_6 &= (\vec{f}_6, ()) \\
\vec{f}_1, \vec{f}_2, &\vec{f}_3, \vec{f}_4, \vec{f}_5, \vec{f}_6 \in \mathrm{F}^*
\end{aligned}
$$

Figure 4.1: A tree-shaped illustration of the abstract value $a_1 \in \mathbb{A}$

in binary code. As each flag is a propositional formula that may contain free variables, an abstract value represents a whole set of concrete values. By considering an assignment for these free variables, an abstract value can be decoded to a particular concrete value, depending on which constructor index (cf. Definition 3.35) is encoded by the flags under the given assignment. Section 4.1.1 covers the mapping between flags and constructor indices, i.e., natural numbers.

We define accessor functions to retrieve flags and arguments from a given abstract value.

Definition 4.2 $\mathsf{flags} : \mathbb{A} \to \mathrm{F}^*$ maps an abstract value $a \in \mathbb{A}$ to the topmost flags in a and is defined by:

$$\mathsf{flags}(a) := \begin{cases} () & \text{if } a = \bot_{\mathbb{A}} \\ \vec{f} & \text{if } a = (\vec{f}, \vec{a}) \end{cases}$$

$|\,\mathsf{flags}(a)|$ maps an abstract value $a \in \mathbb{A}$ to the number of its flags.

Notation Even though in general the flags $\mathsf{flags}(a)$ of an abstract value $a \in \mathbb{A}$ are not the only flags present in a, we will sloppily denote them as *the* flags of a in the remainder of this thesis.

Definition 4.3 $\mathsf{arguments} : \mathbb{A} \to \mathbb{A}^*$ maps an abstract value $a \in \mathbb{A}$ to its arguments and is defined by:

$$\mathsf{arguments}(a) := \begin{cases} () & \text{if } a = \bot_{\mathbb{A}} \\ \vec{a} & \text{if } a = (\vec{f}, \vec{a}) \end{cases}$$

$|\,\mathsf{arguments}(a)|$ maps an abstract value $a \in \mathbb{A}$ to the number of its arguments.

We are often interested in a particular argument of an abstract value.

Definition 4.4 For $i \in \mathbb{N}_{>0}$, $\mathsf{arguments}_i : \mathbb{A} \to \mathbb{A}$ maps an abstract value

$a \in \mathbb{A}$ to its i-th argument and is defined by:

$$\mathsf{arguments}_i(a) \coloneqq \begin{cases} a_i & \text{if } \mathsf{arguments}(a) = (a_1, \ldots, a_m) \text{ and } i \leq m \\ \bot_{\mathbb{A}} & \text{otherwise} \end{cases}$$

Often we are interested in the values of the flags of an abstract value under a certain assignment.

Definition 4.5 $\mathsf{eval}_{\mathsf{flags}} : \mathbb{B}^{\mathrm{V}} \times \mathbb{A} \setminus \{\bot_{\mathbb{A}}\} \to \mathbb{B}^*$ evaluates all $m \in \mathbb{N}$ flags $(f_1, \ldots, f_m) \in \mathrm{F}^m$ of an abstract value $a \in \mathbb{A} \setminus \{\bot_{\mathbb{A}}\}$, i.e., $\mathsf{flags}(a) = (f_1, \ldots, f_m)$, under an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$:

$$\mathsf{eval}_{\mathsf{flags}}(\sigma, a) \coloneqq (\mathsf{eval}_{\mathcal{B}}(\sigma, f_1), \ldots, \mathsf{eval}_{\mathcal{B}}(\sigma, f_m))$$

Note that $\mathcal{B}$ denotes the Boolean algebra given in Definition B.2.

Example 4.6 Given two propositional formulas $f_1 = x_1 \wedge \neg x_2$ and $f_2 = x_2 \vee x_3$, the flags of $a = ((f_1, f_2), ()) \in \mathbb{A}$ evaluate to $\mathsf{eval}_{\mathsf{flags}}(\sigma, a) = (\mathsf{True}, \mathsf{False})$ under assignment $\sigma = \{(x_1, \mathsf{True}), (x_2, \mathsf{False}), (x_3, \mathsf{False})\}$.

In the following, we want to specify an encode/decode-pair for abstract values. To do so, we start by giving a mapping between the flags of an abstract value and the natural numbers.

4.1.1 Encoding and Decoding of Constructor Indices

The flags of an abstract value $a \in \mathbb{A}$ encode the index of a constructor. As each flag is a propositional formula, a binary representation is reasonable. For decoding a to a concrete value $c \in \mathbb{C}_T$ with T having k constructors, we have to differentiate three cases for the number of flags in a:

1. $k > 2^{|\,\mathsf{flags}(a)|}$, i.e., there are not enough flags to encode k different indices. We can safely ignore this case for well-typed programs because Lemma 4.22 and 4.37 show that we always generate abstract values that hold enough flags to encode all constructors of a given type.
2. $k = 2^{|\,\mathsf{flags}(a)|}$, i.e., the flags encode exactly k different indices. We use a standard binary encoding in this case.
3. $k < 2^{|\,\mathsf{flags}(a)|}$, i.e., the flags can encode more than k different indices. If the flags of a happen to represent a constructor index greater than k, then a decoding for a would not be defined when using a naive binary encoding.

In the following, we give a prefix-free encoding for constructor indices that handles the latter two cases and yet is total.

Definition 4.7 For two elements $\vec{a}, \vec{b} \in \mathbb{B}^*$, $\vec{a}$ is a *prefix* of $\vec{b}$ if there is a sequence $\vec{c} \in \mathbb{B}^*$ so that $\vec{b} = \vec{a} \cdot \vec{c}$, where $\cdot$ denotes the concatenation of sequences (cf. Appendix A.1).

A set where no element is a prefix of another element is prefix-free.

Definition 4.8 A set $X \subseteq \mathbb{B}^*$ is *prefix-free* if for all pairs $(\vec{a}, \vec{b}) \in X^2$ with $\vec{a} \neq \vec{b}$, $\vec{a}$ is no prefix of $\vec{b}$.

For each $k \in \mathbb{N}_{>0}$, we define a particular prefix-free set of cardinality k.

Definition 4.9 For $k \in \mathbb{N}_{>0}$, the set $\mathrm{S}_k \subsetneq \mathbb{B}^*$ is defined by:

$$\mathrm{S}_k \coloneqq \begin{cases} \{()\} & \text{if } k = 1 \\ \{(\mathsf{False}) \cdot s \mid s \in \mathrm{S}_{\lceil k/2 \rceil}\} \cup \{(\mathsf{True}) \cdot s \mid s \in \mathrm{S}_{\lfloor k/2 \rfloor}\} & \text{if } k > 1 \end{cases}$$

Example 4.10

$$\begin{aligned}
\mathrm{S}_1 &= \{()\} \\
\mathrm{S}_2 &= \{(\mathsf{False}), (\mathsf{True})\} \\
\mathrm{S}_3 &= \{(\mathsf{False}, \mathsf{False}), (\mathsf{False}, \mathsf{True}), (\mathsf{True})\} \\
\mathrm{S}_4 &= \{(\mathsf{False}, \mathsf{False}), (\mathsf{False}, \mathsf{True}), (\mathsf{True}, \mathsf{False}), (\mathsf{True}, \mathsf{True})\} \\
\mathrm{S}_5 &= \ \{(\mathsf{False}, \mathsf{False}, \mathsf{False}), (\mathsf{False}, \mathsf{False}, \mathsf{True}) \\
&\quad ,(\mathsf{False}, \mathsf{True}), (\mathsf{True}, \mathsf{False}), (\mathsf{True}, \mathsf{True})\}
\end{aligned}$$

Lemma 4.11 For all $k \in \mathbb{N}_{>0}$, S_k is prefix-free and has a cardinality of k.■

For all $k \in \mathbb{N}_{>0}$, the set S_k can be used for constructing a binary representation to encode k different constructor indices. But we also want to handle the case where an abstract value holds more flags than necessary to encode k different constructor indices. Thus, we extend each set in S_k by additional elements.

Definition 4.12 For $k \in \mathbb{N}_{>0}$, the set $\mathrm{S}_{k\ldots}$ is defined by:

$$\mathrm{S}_{k\ldots} \coloneqq \{\vec{f} \cdot \vec{b} \mid \vec{f} \in \mathrm{S}_k \ \wedge \vec{b} \in \mathbb{B}^*\}$$

For all $k \in \mathbb{N}_{>0}$, the set $\mathrm{S}_{k\ldots}$ equals S_k with each sequence being extended by additional Boolean values.

Lemma 4.13 For all $k \in \mathbb{N}_{>0}$, each element in $\mathrm{S}_{k\ldots}$ has a unique prefix in S_k.

Proof Assume the contrary: for $k \in \mathbb{N}_{>0}$, let $\vec{p_1}, \vec{p_2} \in \mathrm{S}_k$ with $\vec{p_1} \neq \vec{p_2}$ be two prefixes of $\vec{a} \in \mathrm{S}_{k\ldots}$, i.e., $\vec{a} = \vec{p_1} \cdot \vec{s_1}$ and $\vec{a} = \vec{p_2} \cdot \vec{s_2}$ for two suffixes $\vec{s_1}, \vec{s_2} \in \mathbb{B}^*$. Without loss of generality, we assume that $\vec{p_1}$ contains more elements than $\vec{p_2}$. Thus, $\vec{p_2}$ is a prefix of $\vec{p_1}$, which contradicts Lemma 4.11. Therefore, Lemma 4.13 holds. ■

For each $k \in \mathbb{N}_{>0}$, we differentiate all sequences in $\mathrm{S}_{k\ldots}$ by their prefix in S_k.

Definition 4.14 Two elements $\vec{a}, \vec{b} \in \mathrm{S}_{k\ldots}$ are included in the binary relation $\sim_{k\ldots} \subsetneq \mathrm{S}_{k\ldots} \times \mathrm{S}_{k\ldots}$ if there is a common prefix $\vec{c} \in \mathrm{S}_k$ for $k \in \mathbb{N}_{>0}$, so that

1. $\exists \vec{u} \in \mathbb{B}^* : \vec{a} = \vec{c} \cdot \vec{u}$, and
2. $\exists \vec{v} \in \mathbb{B}^* : \vec{b} = \vec{c} \cdot \vec{v}$.

Example 4.15

1. $(\mathsf{False}, \mathsf{False}, \mathsf{True}) \sim_{3\ldots} (\mathsf{False}, \mathsf{False}, \mathsf{False})$ with $(\mathsf{False}, \mathsf{False})$ being the common prefix in S_3.
2. $\{(\mathsf{False}, \mathsf{False}, \mathsf{True}), (\mathsf{False}, \mathsf{False}, \mathsf{False})\} \subsetneq [(\mathsf{False}, \mathsf{False})]_{3\ldots}$

Because of Lemma 4.13, it is clear that for all $k \in \mathbb{N}_{>0}$, $\sim_{k\ldots}$ is an equivalence relation. The equivalence relation $\sim_{k\ldots}$ induces k different equivalence classes. By $[\vec{a}]_{k\ldots}$ we denote such an equivalence class with respect to $\sim_{k\ldots}$:

$$[\vec{a}]_{k\ldots} \coloneqq \{\vec{b} \mid \vec{b} \in \mathrm{S}_{k\ldots} \land \vec{a} \sim_{k\ldots} \vec{b}\}$$

For all $k \in \mathbb{N}_{>0}$, we want to specify a mapping from the elements in $\mathrm{S}_{k\ldots}$ into the natural numbers. To do so, we order the elements in $\mathrm{S}_{k\ldots}$ lexicographically.

Definition 4.16 $<_{\mathbb{B}^*} \subsetneq \mathbb{B}^* \times \mathbb{B}^*$ denotes the *lexicographic order on* $\mathbb{B}^*$, where $\vec{a} <_{\mathbb{B}^*} \vec{b}$ holds for two elements $\vec{a}, \vec{b} \in \mathbb{B}^*$ if

1. $\vec{a} \neq \vec{b}$ and $\vec{a}$ is a prefix of $\vec{b}$, or
2. $\vec{c} \in \mathbb{B}^*$ is the longest common prefix of $\vec{a}$ and $\vec{b}$ where
   (a) $\exists \vec{u} \in \mathbb{B}^* : \vec{a} = \vec{c} \cdot (\mathsf{False}) \cdot \vec{u}$, and
   (b) $\exists \vec{v} \in \mathbb{B}^* : \vec{b} = \vec{c} \cdot (\mathsf{True}) \cdot \vec{v}$

Note that $<_{\mathbb{B}^*}$ is a total order.

Example 4.17

1. $(\mathsf{False}, \mathsf{False}) <_{\mathbb{B}^*} (\mathsf{False}, \mathsf{False}, \mathsf{True})$
2. $(\mathsf{False}, \mathsf{False}) <_{\mathbb{B}^*} (\mathsf{True}, \mathsf{False}, \mathsf{True})$
3. $(\mathsf{True}, \mathsf{False}, \mathsf{False}) <_{\mathbb{B}^*} (\mathsf{True}, \mathsf{False}, \mathsf{True})$

For all $k \in \mathbb{N}_{>0}$, we define a mapping from $\mathrm{S}_{k\ldots}$ to the natural numbers.

Definition 4.18 For all $k \in \mathbb{N}_{>0}$, $\mathsf{numeric}_k : \mathrm{S}_{k\ldots} \to \{1 \ldots k\}$ maps a sequence $\vec{s} \in \mathrm{S}_{k\ldots}$ to a natural number $i \in \{1 \ldots k\}$ where

1. $\mathrm{S}_k = \{\vec{f_1}, \ldots, \vec{f_i}, \ldots, \vec{f_k}\}$,
2. $\vec{f_1} <_{\mathbb{B}^*} \cdots <_{\mathbb{B}^*} \vec{f_i} <_{\mathbb{B}^*} \cdots <_{\mathbb{B}^*} \vec{f_k}$, and
3. $\vec{s} \in [\vec{f_i}]_{k\ldots}$

Example 4.19 $\mathsf{numeric}_3(\mathsf{False}, \mathsf{True}, \mathsf{True}, \mathsf{True}) = 2$ because

1. $\mathrm{S}_3 = \{(\mathsf{False}, \mathsf{False}), (\mathsf{False}, \mathsf{True}), (\mathsf{True})\}$,

2. $(\mathsf{False}, \mathsf{False}) <_{\mathbb{B}^*} (\mathsf{False}, \mathsf{True}) <_{\mathbb{B}^*} (\mathsf{True})$, and
3. $(\mathsf{False}, \mathsf{True}, \mathsf{True}, \mathsf{True}) \in [(\mathsf{False}, \mathsf{True})]_{3\ldots}$

For all $k \in \mathbb{N}_{>0}$, we define a mapping from $\{1 \ldots k\}$ to S_k as well.

Definition 4.20 For all $k \in \mathbb{N}_{>0}$, $\mathsf{numeric}_k^- : \{1 \ldots k\} \to \mathrm{S}_k$ maps a natural number $i \in \{1 \ldots k\}$ to a prefix-free sequence $\vec{f_i} \in \mathrm{S}_k$ where

1. $\mathrm{S}_k = \{\vec{f_1}, \ldots, \vec{f_i}, \ldots, \vec{f_k}\}$, and
2. $\vec{f_1} <_{\mathbb{B}^*} \cdots <_{\mathbb{B}^*} \vec{f_i} <_{\mathbb{B}^*} \cdots <_{\mathbb{B}^*} \vec{f_k}$

Example 4.21 $\mathsf{numeric}_3^-(1) = (\mathsf{False}, \mathsf{False})$ because

1. $\mathrm{S}_3 = \{(\mathsf{False}, \mathsf{False}), (\mathsf{False}, \mathsf{True}), (\mathsf{True})\}$, and
2. $(\mathsf{False}, \mathsf{False}) <_{\mathbb{B}^*} (\mathsf{False}, \mathsf{True}) <_{\mathbb{B}^*} (\mathsf{True})$

Note the following relation between $\mathsf{numeric}_k$ and $\mathsf{numeric}_k^-$ for $k \in \mathbb{N}_{>0}$.

Lemma 4.22

$$\forall k \in \mathbb{N}_{>0} : \ \forall i \in \{1 \ldots k\} : \mathsf{numeric}_k(\mathsf{numeric}_k^-(i)) = i \qquad \blacksquare$$

Additionally, Lemma 4.22 guarantees that for all $k \in \mathbb{N}_{>0}$, $\mathsf{numeric}_k^-$ gives a sequence that contains enough elements to discriminate k different constructors.

4.1.2 Encoding and Decoding of Abstract Values

Now that we have specified how constructor indices are encoded, we define mappings between concrete and abstract values. These mappings must take into account the type of the concrete value, especially the number of constructors, the number of constructor arguments, and the constructor arguments' types. Thus, we introduce functions that give this information for a particular type.

Definition 4.23 $\mathsf{constructors} : \textsc{Type}_0 \to \textsc{Con}^*$ gives the sequence of constructors for a given type $T \in \textsc{Type}_0$ in the order of their occurrence in the declaration of T.

Although $\mathsf{constructors}(T)$ gives the sequence of constructors of type $T \in \textsc{Type}_0$, we occasionally treat that sequence as a set: this is valid as each constructor is unique within $\mathsf{constructors}(T)$.

Example 4.24 For the following type declaration

```
data Either a b = Left a | Right b
```

and two types $T_1, T_2 \in \textsc{Type}_0$, we have

1. $|\mathsf{constructors}(\texttt{Either}\ T_1\ T_2)| = 2$, and

2. $\mathsf{constructors}(\texttt{Either}\ T_1\ T_2) = (\texttt{Left}, \texttt{Right})$.

Mapping between an abstract and a concrete value of type $T \in \textsc{Type}_0$ not only depends on the constructors of T but also on its constructor arguments.

Definition 4.25 For all $i \in \mathbb{N}_{>0}$, $\mathsf{con\text{-}argtype}_i : \textsc{Con} \times \textsc{Type}_0 \nrightarrow \textsc{Type}_0$ maps the constructor $C \in \textsc{Con}$ of type $T \in \textsc{Type}_0$ to the type of its i-th constructor argument $\mathsf{con\text{-}argtype}_i(C, T) \in \textsc{Type}_0$.

Note that for all $(i, C, T) \in \mathbb{N}_{>0} \times \textsc{Con} \times \textsc{Type}$, $\mathsf{con\text{-}argtype}_i(C, T)$ is undefined if $i > \mathsf{arity}(C)$ or $C \notin \mathsf{constructors}(T)$.

Example 4.26 For the following type declaration

```
data Either a b = Left a | Right b
```

and two types $T_1, T_2 \in \textsc{Type}_0$, we have

1. $\mathsf{con\text{-}argtype}_1(\texttt{Left}, \texttt{Either}\ T_1\ T_2) = T_1$ and
2. $\mathsf{con\text{-}argtype}_1(\texttt{Right}, \texttt{Either}\ T_1\ T_2) = T_2$.

Now that we are able to query important features of types and constructors, we define mappings between abstract and concrete values. Recall that an abstract value $a \in \mathbb{A}$ represents a set of concrete values $C \subseteq \mathbb{C}$. In the following, we define a decoding from a to one of the concrete values in C. Which value in C a is decoded to is determined by an assignment for the propositional variables in the flags of a.

Definition 4.27 $\mathsf{decode}_T : \mathbb{B}^{\mathrm{V}} \times \mathbb{A} \to \mathbb{C}_T$ gives a concrete value $\mathsf{decode}_T(\sigma, a)$ of type $T \in \textsc{Type}_0$ for an abstract value $a \in \mathbb{A}$ and an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$:

$$\mathsf{decode}_T(\sigma, a) \coloneqq$$

$$\begin{cases} \bot & \text{if } a = \bot_{\mathbb{A}} \\ \bot & \text{if } k > 2^{|\,\mathsf{flags}(a)|} \\ C_j \quad \begin{array}{l} \mathsf{decode}_{T_1}(\sigma, \mathsf{arguments}_1(a)) \\ \dots \\ \mathsf{decode}_{T_n}(\sigma, \mathsf{arguments}_n(a)) \end{array} & \text{otherwise} \end{cases}$$

where

1. $k = |\,\mathsf{constructors}(T)|$ denotes the number of constructors of T,
2. $j = \mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, a))$ denotes the decoded constructor index,
3. $\mathsf{constructors}(T) = (C_1, \dots, C_j, \dots, C_k)$,
4. $n = \mathsf{arity}(C_j)$ denotes the arity of constructor C_j, and
5. for all $i \in \{1 \dots n\}$, $T_i = \mathsf{con\text{-}argtype}_i(C_j, T)$ denotes the type of the i-th argument of C_j.

Example 4.28 illustrates how an abstract value $a \in \mathbb{A}$ is decoded to different concrete values depending on the assignment of the propositional variables in the flags of a.

Example 4.28 Consider a concrete program with the following type declarations:

```
data RGB = Red | Green | Blue
data Maybe a = Nothing | Just a
```

Furthermore, assume two abstract values $a_1 = ((f_1), a_2)$ and $a_2 = ((f_2, f_3), ())$, and an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$ for all propositional variables in the formulas $f_1, f_2, f_3 \in \mathrm{F}$. The concrete shape of these formulas is not relevant here; we are only interested in their values. Let

$$n_1 = \mathsf{numeric}_2(\mathsf{eval}_{\mathsf{flags}}(\sigma, a_1))$$
$$n_2 = \mathsf{numeric}_3(\mathsf{eval}_{\mathsf{flags}}(\sigma, a_2))$$

Then,

$$\mathsf{decode}_{\texttt{Maybe RGB}}(\sigma, a_1) = \begin{cases} \texttt{Nothing} & \text{if } n_1 = 1 \\ \texttt{Just Red} & \text{if } n_1 = 2 \wedge n_2 = 1 \\ \texttt{Just Green} & \text{if } n_1 = 2 \wedge n_2 = 2 \\ \texttt{Just Blue} & \text{if } n_1 = 2 \wedge n_2 = 3 \end{cases}$$

Next we encode a concrete value $c \in \mathbb{C}_T$ of type $T \in \textsc{Type}_0$ as an abstract value $a \in \mathbb{A}$ that represents only the value c and no other concrete value. This is done by using only constant Boolean values in the flags of a.

Definition 4.29 For a concrete value $c \in \mathbb{C}_T$ of type $T \in \textsc{Type}_0$, $\mathsf{encode}_T : \mathbb{C}_T \to \mathbb{A}$ gives an abstract value such that:

$$\mathsf{encode}_T(c) :=$$
$$\begin{cases} \bot_{\mathbb{A}} & \text{if } c = \bot \\ (\mathsf{numeric}^-_k(j), (\mathsf{encode}_{T_1}(v_1), \dots, \mathsf{encode}_{T_n}(v_n))) & \text{if } c = C_j\ v_1 \dots v_n \end{cases}$$

where

1. $k = |\,\mathsf{constructors}(T)|$ denotes the number of constructors of T,
2. $\mathsf{constructors}(T) = (C_1, \dots, C_j, \dots, C_k)$,
3. $n = \mathsf{arity}(C_j)$ denotes the arity of constructor C_j, and
4. for all $i \in \{1 \dots n\}$, $T_i = \mathsf{con\text{-}argtype}_i(C_j, T)$ denotes the type of the i-th argument of C_j.

Example 4.30 Consider a concrete program with the following type declarations:

```
data RGB = Red | Green | Blue
data Maybe a = Nothing | Just a
```

Then,

$$\mathsf{encode}_{\texttt{RGB}}(\texttt{Green}) = ((\mathsf{False}, \mathsf{True}), ())$$
$$\mathsf{encode}_{\texttt{Maybe RGB}}(\texttt{Just Green}) = ((\mathsf{True}), ((\mathsf{False}, \mathsf{True}), ()))$$

The following lemma is the concluding result of this section.

Lemma 4.31 The tuple $(\mathsf{encode}, \mathsf{decode})$ denotes an encode/decode-pair.

Proof For $(\mathsf{encode}, \mathsf{decode})$ to denote an encode/decode-pair, the following property from Definition 3.79 must hold for all types $T \in \textsc{Type}_0$:

$$\forall (v, \sigma) \in \mathbb{C}_T \times \mathbb{B}^{\mathrm{V}} : \mathsf{decode}_T(\sigma, \mathsf{encode}_T(v)) = v$$

This follows from the definition of decode_T and encode_T and from the following relation between $\mathsf{numeric}_k$ and $\mathsf{numeric}_k^-$ for each $k \in \mathbb{N}_{>0}$ (cf. Lemma 4.22):

$$\forall i \in \{1 \dots k\} : \mathsf{numeric}_k(\mathsf{numeric}_k^-(i)) = i$$

As the other properties required for encode/decode-pairs are satisfied by definition of decode_T and encode_T, Lemma 4.31 holds. ■

4.1.3 Mimic Constructor Calls in Abstract Programs

Recall that according to Definition 3.77, an abstract program may call the built-in function `cons` whose semantics are given by a function cons. Applying cons to abstract values mimics a constructor call from the corresponding concrete program (cf. Example 3.71). Now that we have specified how concrete values can be encoded as abstract values, we are able to define cons.

Definition 4.32 $\mathsf{cons}_{(j,k)} : \mathbb{A}^* \to \mathbb{A}$ is defined by

$$\mathsf{cons}_{(j,k)}(a_1, \dots, a_n) \coloneqq (\mathsf{numeric}_k^-(j), (a_1, \dots, a_n))$$

for $j, k \in \mathbb{N}_{>0}$ and $j \in \{1 \dots k\}$.

Note how the definition of cons resembles the definition of encode (cf. Definition 4.29): but whereas encode gives an abstract value for a given concrete value, cons mimics the actual constructor call by taking all of its arguments as arguments of the generated abstract value.

Example 4.33 Assume an abstract value $a \in \mathbb{A}$. Then, $\mathsf{cons}_{(1,2)}(a) = ((\mathsf{False}), (a))$ mimics the call to the first of two constructors where its arity is assumed to be one.

4.1.4 Complete Abstract Values

An abstract value generated by encode is decoded to a unique concrete value. There are many ways to construct abstract values that can be decoded to more than one concrete value. In the following, we introduce complete abstract values as values that can be decoded to all concrete values of a particular set of types.

Definition 4.34 An abstract value $a \in \mathbb{A}$ is *complete* according to a set of types $\mathcal{T} \subseteq \text{TYPE}_0$ if every type in $\mathcal{T}$ is finite, and if for each concrete value $c \in \bigcup_{T \in \mathcal{T}} \mathbb{C}_T \setminus \{\perp\}$ there is an assignment of propositional variables such that a can be decoded to c:

$$\forall T \in \mathcal{T}: \ \forall c \in \mathbb{C}_T \setminus \{\perp\}: \ \exists \sigma \in \mathbb{B}^{\mathrm{V}}: \mathsf{decode}_T(\sigma, a) = c$$

Complete abstract values play an important role for CO^4 because a complete abstract value that can be decoded to all values of a finite type $T \in \text{TYPE}_0$ does only need to contain finite many propositional variables since $\mathbb{C}_T$ is finite as well (cf. Definition 3.54). We specify a function that generates such complete abstract values.

Definition 4.35 complete : $2^{\text{TYPE}_0} \setminus \varnothing \rightarrow \mathbb{A}$ gives the following abstract value for a non-empty set of finite types $\mathcal{T} \in 2^{\text{TYPE}_0} \setminus \varnothing$:

$$\mathsf{complete}(\mathcal{T}) \coloneqq ((f_1, \dots, f_m), (a_1, \dots, a_n))$$

where

1. $k = \max\{|\mathsf{constructors}(T)| \mid T \in \mathcal{T}\}$ denotes the maximum number of constructors over all types in $\mathcal{T}$,
2. $m = \lceil \log_2 k \rceil$ denotes the number of flags needed to encode k constructors,
3. each flag is a fresh propositional variable: $\forall i \in \{1 \dots m\} : f_i \in \mathrm{V}$,
4. $n = \max\{\mathsf{arity}(C) \mid T \in \mathcal{T} \wedge C \in \mathsf{constructors}(T)\}$ denotes the highest arity in the union of all constructors of all types in $\mathcal{T}$, and
5. for all $i \in \{1 \dots n\}$, the abstract value $a_i = \mathsf{complete}\ (\mathcal{T}_i)$ is complete for the types of all i-th constructor arguments in the union of all constructors of all types in $\mathcal{T}$:

$$\begin{aligned} \mathcal{T}_i = \{\mathsf{con\text{-}argtype}_i(C, T) \mid T \in \mathcal{T} \\ \wedge\ C \in \mathsf{constructors}(T) \\ \wedge\ i \leq \mathsf{arity}(C)\} \end{aligned}$$

This implementation of complete abstract values overlaps the encoding of constructor arguments, i.e., all first (second, third, etc.) arguments of all constructors are encoded by the same complete abstract value. Figure 4.36 illustrates

```
data A = A1 C
       | A2 E
data B = B1 E D
data C = C1
       | C2 E
data D = D1 E E
       | D2 E E
data E = E1
       | E2
```

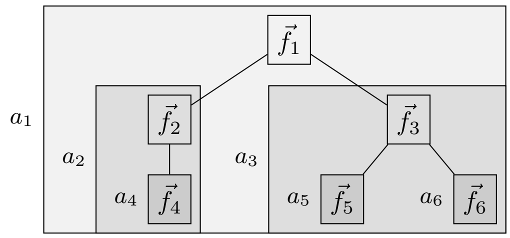


Figure 4.36: Visual representation of an abstract value $a_1 = \mathsf{complete}(\{\mathtt{A}, \mathtt{B}\}) = (\vec{f}_1, ((\vec{f}_2, (\vec{f}_4, ())), (\vec{f}_3, ((\vec{f}_5, ()), (\vec{f}_6, ())))))$ that can be decoded to all values in $\mathbb{C}_{\mathtt{A}} \cup \mathbb{C}_{\mathtt{B}}$. Note the overlapping of the following constructor arguments: a_2 encodes the first argument of `A1`, `A2`, and `B1`; a_5 encodes the first argument of `D1` and `D2`; and a_6 encodes the second argument of `D1` and `D2`.

this feature for an exemplary abstract value generated by $\mathsf{complete}$. In Section 4.3.4, we discuss alternative encodings for complete abstract values.

Note that $\mathsf{complete}(\mathcal{T})$ does not terminate if $\mathcal{T} \in 2^{\textsc{Type}_0} \setminus \varnothing$ contains an infinite type (cf. Definition 3.54).

The following lemma follows immediately from Definition 4.35:

Lemma 4.37 For each set $\mathcal{T} \subseteq \textsc{Type}_0$, the abstract value $\mathsf{complete}(\mathcal{T}) \in \mathbb{A}$ is complete if all types in $\mathcal{T}$ are finite. ∎

We give an example for decoding an abstract value generated by $\mathsf{complete}$.

Example 4.38 Consider a program with the following type declarations:

```
data Bool = False | True
data RGB = Red | Green | Blue
data Either a b = Left a | Right b
```

Furthermore, assume two complete abstract values $a_1, a_2 \in \mathbb{A}$ containing the propositional variables $f_1, f_2, f_3 \in \mathrm{V}$ as flags:

$$\begin{aligned} \mathsf{complete}(\{\mathtt{Either\ Bool\ RGB}\}) = a_1 &= ((f_1), a_2) \\ a_2 &= \mathsf{complete}(\{\mathtt{Bool}, \mathtt{RGB}\}) \\ &= ((f_2, f_3), ()) \end{aligned}$$

Note that a_2 encodes the argument of both constructors of `Either Bool RGB`.

For a given assignment $\sigma \in \mathbb{B}^{\{f_1,f_2,f_3\}}$, let

$$\begin{aligned} n_1 &= \mathsf{numeric}_2(\mathsf{eval}_{\mathsf{flags}}(\sigma, a_1)) \\ n_2 &= \mathsf{numeric}_2(\mathsf{eval}_{\mathsf{flags}}(\sigma, a_2)) \\ n_3 &= \mathsf{numeric}_3(\mathsf{eval}_{\mathsf{flags}}(\sigma, a_2)) \end{aligned}$$

Then,

$$\mathsf{decode}_{\texttt{Either Bool RGB}}(\sigma, a_1) = \begin{cases} \texttt{Left False} & \text{if } n_1 = 1 \wedge n_2 = 1 \\ \texttt{Left True} & \text{if } n_1 = 1 \wedge n_2 = 2 \\ \texttt{Right Red} & \text{if } n_1 = 2 \wedge n_3 = 1 \\ \texttt{Right Green} & \text{if } n_1 = 2 \wedge n_3 = 2 \\ \texttt{Right Blue} & \text{if } n_1 = 2 \wedge n_3 = 3 \end{cases}$$

The overlapped encoding of constructor arguments illustrated in Figure 4.36 is beneficial for reducing the number of variables in the propositional formula generated by CO^4. The savings gained by applying this scheme depend on the structure of the encoded data type. In Example 4.39, we show how the order of the constructor arguments affects the resulting complete abstract value.

Example 4.39 Consider the following type declarations:

```
data Bool = False | True
data RGB  = Red | Green | Blue
data Foo  = Foo1 Bool RGB | Foo2 RGB Bool
data Bar  = Bar1 Bool RGB | Bar2 Bool RGB
```

Note that the order of arguments in the second constructor is the only difference between both types `Foo` and `Bar`. Both values $\mathsf{complete}(\{\texttt{Foo}\})$ and $\mathsf{complete}(\{\texttt{Bar}\})$ result in a similar shaped abstract value:

$$\begin{aligned} \mathsf{complete}(\{\texttt{Foo}\}) &= (\vec{f_1}, ((\vec{f_2}, ()), (\vec{f_3}, ()))) \\ \mathsf{complete}(\{\texttt{Bar}\}) &= (\vec{g_1}, ((\vec{g_2}, ()), (\vec{g_3}, ()))) \end{aligned}$$

Whereas both vectors $\vec{f_2}$ and $\vec{f_3}$ contain two flags each, $\vec{g_2}$ only contains one flag but $\vec{g_3}$ contains two flags. That is because $\mathsf{complete}$ exploits the fact that the types of the first (resp. second) argument in both constructors `Bar1` and `Bar2` match, and therefore all flags in $\vec{g_2}$ and $\vec{g_3}$ are shared. On the other hand, each of the vectors $\vec{f_2}$ and $\vec{f_3}$ contain one flag that is not shared because the overlapping arguments have different types. Thus, more propositional variables are necessary for encoding all values of type `Foo` in comparison to encoding the same number of values of type `Bar`.

4.1.5 Incomplete Abstract Values

An abstract value that is complete according to a singleton set $\{T\}$ with $T \in$ $\textsc{Type}_0$ represents all values in $\mathbb{C}_T$. Complete abstract values can be generated using the function $\mathsf{complete}$ (cf. Definition 4.35). However, if $\mathbb{C}_T$ is infinite, $\mathsf{complete}(\{T\})$ is undefined because it does not terminate. This implies that the function $\mathsf{complete}$ cannot be used when dealing with constraints over infinite domains of discourse. Therefore, we generate an abstract value that represents only a finite subset of the infinite set $\mathbb{C}_T$. How to restrict $\mathbb{C}_T$ to a finite subset in a reasonable manner depends on the shape of the type T. We give an example in which T denotes the set of lists of Booleans.

Example 4.40 Assume the following type declarations:

```
data Bool = False | True
data List a = Nil | Cons a (List a)
```

As `List Bool` is a recursive type, $\mathsf{complete}(\{\texttt{List Bool}\})$ is not defined. But we can generate an abstract value $\mathsf{abstract\text{-}bool\text{-}list}(n)$ that represents lists of Booleans up to a particular length $n \in \mathbb{N}$:

$$\mathsf{abstract\text{-}bool\text{-}list}(n) :=$$
$$\begin{cases} \mathsf{encode}_{\texttt{List Bool}}(\texttt{Nil}) & \text{if } n = 0 \\ ((f), (\,\mathsf{complete}(\{\texttt{Bool}\}) & \text{if } n > 0 \text{ and } f \in \mathrm{V} \text{ is a fresh} \\ \quad\quad , \mathsf{abstract\text{-}bool\text{-}list}(n-1))) & \text{propositional variable} \end{cases}$$

$\mathsf{abstract\text{-}bool\text{-}list}$ gives:

$$\begin{aligned} a_0 &= \mathsf{abstract\text{-}bool\text{-}list}(0) = ((\mathsf{False}), ()) \\ a_1 &= \mathsf{abstract\text{-}bool\text{-}list}(1) = ((f_1), (((b_1), ()), a_0)) \quad\quad \text{with } f_1, b_1 \in \mathrm{V} \\ a_2 &= \mathsf{abstract\text{-}bool\text{-}list}(2) = ((f_2), (((b_2), ()), a_1)) \quad\quad \text{with } f_2, b_2 \in \mathrm{V} \\ &\ldots \end{aligned}$$

For all $n \in \mathbb{N}$, $\mathsf{abstract\text{-}bool\text{-}list}(n)$ generates an abstract value that represents all lists in $\mathbb{C}_{\texttt{List Bool}}$ whose length is less or equal n (where the length is defined as the number of involved `Cons` constructors).

Functions similar to $\mathsf{abstract\text{-}bool\text{-}list}$ can be constructed for other recursively defined types, e.g., in Example 6.10 we generate incomplete abstract values that represent bounded natural numbers.

4.1.6 Merging Abstract Values

Now that we have introduced abstract values as a representation for sets of concrete values, we specify a merge operation for abstract values. As we have described in the introduction of Section 3.3, a call to the built-in function `merge` gives an abstract value that simulates the result of a case distinction in terms of propositional variables and logical connectives.

Recall that according to Definition 3.77, the dynamic semantics of a call to `merge` are given by a function merge which we have not yet specified.

> **Definition 4.41** $\mathsf{merge}_{v_d} : \mathbb{A}^* \to \mathbb{A}$ merges $k \in \mathbb{N}_{>0}$ abstract values $(v_1, \ldots, v_k) \in \mathbb{A}^k$ according to the value $v_d \in \mathbb{A}$ and is defined by:
>
> $$\mathsf{merge}_{v_d}(v_1, \ldots, v_k) = \begin{cases} \bot_{\mathbb{A}} & \text{if } v_d = \bot_{\mathbb{A}} \\ r & \text{otherwise} \end{cases}$$
>
> so that $r \in \mathbb{A}$ is specified by
>
> $$\begin{aligned} &\forall(\sigma, i) \in \mathbb{B}^{\mathrm{V}} \times \{1 \ldots k\} : \\ &\quad (\mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, v_d)) = i) \implies (\mathsf{decode}_T(\sigma, r) = \mathsf{decode}_T(\sigma, v_i)) \end{aligned}$$
>
> with $T \in \textsc{Type}_0$ being the type of the case distinction in the concrete program.

Definition 4.41 of merge does not induce any concrete implementation but specifies its result in relation to its arguments: for $v_d, v_1, \ldots, v_k \in \mathbb{A}$, $k \in \mathbb{N}_{>0}$, and $\sigma \in \mathbb{B}^{\mathrm{V}}$, the decoding of the abstract value $\mathsf{merge}_{v_d}(v_1, \ldots, v_k)$ equals the decoding of v_i for $i \in \{1 \ldots k\}$ if the flags of v_d index the i-th constructor under the assignment σ. Note that Definition 4.41 does not specify the result of $\mathsf{merge}_{v_d}(v_1, \ldots, v_k)$ if the flags of the abstract value v_d do not evaluate to a value in $\{1 \ldots k\}$. The reason is that such a situation does not occur if the original concrete program is well-typed.

Example 4.42 illustrates a simple merge of two abstract values.

> **Example 4.42** Assume the following abstract values $v_d, v_1, v_2 \in \mathbb{A}$ that only contain a single flag each:
>
> $$\begin{aligned} v_d &= ((f_d), ()) \text{ with } f_d \in \mathrm{F} \\ v_1 &= ((f_1), ()) \text{ with } f_1 \in \mathrm{F} \\ v_2 &= ((f_2), ()) \text{ with } f_2 \in \mathrm{F} \end{aligned}$$
>
> Each of these values represents concrete values of type `Bool` where:
>
> ```
> data Bool = False | True
> ```
>
> In order to satisfy the property stated in Definition 4.41, it is sufficient for the value of $\mathsf{merge}_{v_d}(v_1, v_2)$ to contain only a single flag because this merge needs to differentiate between two values v_1 and v_2 only. Thus,
>
> $$\mathsf{merge}_{v_d}(v_1, v_2) = r = ((f_r), ()) \text{ with } f_r \in \mathrm{F}$$

For the resulting value r and its single flag f_r, the following must hold for all assignments $\sigma \in \mathbb{B}^{\mathrm{V}}$:

$$\begin{aligned} &(\mathsf{numeric}_2(\mathsf{eval}_{\mathsf{flags}}(\sigma, v_d)) = 1 \implies \mathsf{decode}_{\mathtt{Bool}}(\sigma, r) = \mathsf{decode}_{\mathtt{Bool}}(\sigma, v_1)) \\ \wedge\ &(\mathsf{numeric}_2(\mathsf{eval}_{\mathsf{flags}}(\sigma, v_d)) = 2 \implies \mathsf{decode}_{\mathtt{Bool}}(\sigma, r) = \mathsf{decode}_{\mathtt{Bool}}(\sigma, v_2)) \end{aligned}$$

which equals:

$$\begin{aligned} &(\mathsf{eval}_{\mathcal{B}}(\sigma, f_d) = \mathsf{False} \implies \mathsf{eval}_{\mathcal{B}}(\sigma, f_r) = \mathsf{eval}_{\mathcal{B}}(\sigma, f_1)) \\ \wedge\ &(\mathsf{eval}_{\mathcal{B}}(\sigma, f_d) = \mathsf{True} \implies \mathsf{eval}_{\mathcal{B}}(\sigma, f_r) = \mathsf{eval}_{\mathcal{B}}(\sigma, f_2)) \end{aligned}$$

Furthermore, assume that $f_1 = \mathsf{True}$ and $f_2 = \mathsf{False}$. In this case we have:

$$\begin{aligned} &(\mathsf{eval}_{\mathcal{B}}(\sigma, f_d) = \mathsf{False} \implies \mathsf{eval}_{\mathcal{B}}(\sigma, f_r) = \mathsf{True}) \\ \wedge\ &(\mathsf{eval}_{\mathcal{B}}(\sigma, f_d) = \mathsf{True} \implies \mathsf{eval}_{\mathcal{B}}(\sigma, f_r) = \mathsf{False}) \end{aligned}$$

For example, this property holds if $f_r \Leftrightarrow \neg f_d$.

Now, we inspect a special case of merge. The following lemma states that $\mathsf{merge}_{v_d}(v_1, \ldots, v_k)$ projects onto one of its $k \in \mathbb{N}_{>0}$ arguments $v_1, \ldots, v_k \in \mathbb{A}$ if the flags $\mathsf{flags}(v_d)$ of v_d index a constant constructor index, regardless of any assignment for the propositional variables in $\mathsf{flags}(v_d)$.

Lemma 4.43 If there is an abstract value $v_d \in \mathbb{A}$ and a number $n \in \{1 \ldots k\}$ for some $k \in \mathbb{N}_{>0}$ so that

$$\forall \sigma \in \mathbb{B}^{\mathrm{V}} : \mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, v_d)) = n$$

then

$$\forall (v_1, \ldots, v_k) \in \mathbb{A}^k : \mathsf{merge}_{v_d}(v_1, \ldots, v_k) = v_n \qquad \blacksquare$$

Lemma 4.43 immediately follows from the Definition 4.41 of merge. The consequences of Lemma 4.43 are that in some situations, compiled case distinctions can be evaluated without generating new subformulas. This is not always possible as the flags in the abstract value v_d in general do not index a constant constructor. In Section 9.2, we illustrate the difference between both types of case distinctions and discuss their importance for estimating the complexity of constraints specified by concrete programs.

4.2 Program Transformation

This section specifies a compilation function from concrete to abstract programs that is correct according to Definition 3.80. The essential part of this compilation concerns case distinctions. Recall that, in contrast to concrete programs, there are no case distinctions in abstract programs. Thus, case distinctions are handled in a special way by the compilation function.

Notation In the following, we define the result of a compilation by specifying the abstract counterpart a for some entity of the concrete program where a is given using the syntax of abstract programs. Enclosing a particular term $f(x)$ in brackets $[\![f(x)]\!]$ in a denotes that not $f(x)$ appears in a but the result of evaluating $f(x)$.

Firstly, we define the compilation of expressions $\mathsf{compile}_{\text{EXP}} : \text{EXP} \rightarrow \text{EXP}_{\mathbb{A}}$. For some expressions, the compilation to their abstract counterparts is trivial, e.g., variables are just copied to the abstract program as they appear in the concrete program.

Definition 4.44 The *compilation of variables* is defined by:

$$\forall v \in \text{VAR} : \mathsf{compile}_{\text{EXP}}(v) := v$$

Local bindings are compiled by compiling all subexpressions but their structure remains.

Definition 4.45 The *compilation of local bindings* is defined by:

$$\begin{aligned}
&\forall (v, e_1, e_2) \in \text{VAR} \times \text{EXP} \times \text{EXP} : \\
\mathsf{compile}_{\text{EXP}}(\texttt{let } v \texttt{ = } e_1 \texttt{ in } e_2) := {} & \texttt{let } v \texttt{ = } [\![\mathsf{compile}_{\text{EXP}}(e_1)]\!] \\
& \texttt{in } [\![\mathsf{compile}_{\text{EXP}}(e_2)]\!]
\end{aligned}$$

Similarly to local bindings, an abstraction is compiled by compiling the subexpression of the abstraction. Again, the program structure remains.

Definition 4.46 The *compilation of abstractions* is defined by:

$$\begin{gathered}
\forall (v, e) \in \text{VAR} \times \text{EXP} : \\
\mathsf{compile}_{\text{EXP}}(\texttt{\textbackslash} v \texttt{ -> } e) := \texttt{\textbackslash} v \texttt{ -> } [\![\mathsf{compile}_{\text{EXP}}(e)]\!]
\end{gathered}$$

Compiling function applications slightly changes the program's structure: each argument is bound to a fresh name inside a block of local bindings. This change just accounts for the syntax of function applications in abstract programs (cf. Definition 3.66) where each argument of an application must have been bound to a name.

Definition 4.47 The *compilation of function applications* is defined by

$$\begin{aligned}
&\forall (f, n) \in \text{VAR} \times \mathbb{N}_{>0} : \ \forall (e_1, \ldots, e_n) \in \text{EXP}^n : \\
\mathsf{compile}_{\text{EXP}}(f\ e_1 \ldots e_n) := {} & \texttt{let } v_1 \texttt{ = } [\![\mathsf{compile}_{\text{EXP}}(e_1)]\!] \\
& \qquad \ldots \\
& \qquad v_n \texttt{ = } [\![\mathsf{compile}_{\text{EXP}}(e_n)]\!] \\
& \texttt{in } f\ v_1 \ldots v_n
\end{aligned}$$

where $v_1, \ldots, v_n \in \text{VAR}$ are fresh variable names.

The compilation of a constructor application resembles the compilation of function applications, but an abstract value $a \in \mathbb{A}$ is explicitly generated so that a's flags index the called constructor.

Definition 4.48 The *compilation of a constructor application* $C\ e_1 \ldots e_n$ is defined by

$$\begin{aligned}\mathsf{compile}_{\text{EXP}}(C\ e_1 \ldots e_n) \coloneqq\ & \texttt{let}\ \ v_1\ \texttt{=}\ [\![\mathsf{compile}_{\text{EXP}}(e_1)]\!] \\ & \qquad \ldots \\ & \qquad v_n\ \texttt{=}\ [\![\mathsf{compile}_{\text{EXP}}(e_n)]\!] \\ & \texttt{in} \\ & \quad \texttt{cons}_{(j,k)}\, v_1\ \ldots\ v_n\end{aligned}$$

where

1. $C \in \textsc{Con}$ is the j-th constructor of a type $T \in \textsc{Type}$,
2. $k = |\,\mathsf{constructors}(T)|$ denotes the number of constructors of T,
3. $n = \mathsf{arity}(C)$ denotes the arity of the constructor C,
4. $e_1, \ldots, e_n \in \textsc{Exp}$ are n constructor arguments, and
5. $v_1, \ldots, v_n \in \textsc{Var}$ denote fresh variable names.

The compilation of case distinctions is more complex. Note that according to Definition 3.60, the value of evaluating a case distinction is determined by evaluating the one branch whose pattern matches the discriminant. That is in general not doable in the domain of abstract values because the flags of the discriminant's abstract value $d \in \mathbb{A}$ may contain propositional variables. Thus, there is no way to determine which pattern matches on d. Therefore, all branches are evaluated in an abstract case distinction and the final result is obtained by merging the branches' abstract values.

Definition 4.49 The *compilation of a case distinction* $e \in \textsc{Exp}$

$$\begin{aligned}e = \texttt{case}\ d\ \texttt{of}\ & C_1\ a_{11} \ldots a_{1n_1}\ \texttt{->}\ e_1 \\ & \ldots \\ & C_k\ a_{k1} \ldots a_{kn_k}\ \texttt{->}\ e_k\end{aligned}$$

is defined by

$$\begin{aligned}\mathsf{compile}_{\text{EXP}}(e) \coloneqq\ & \texttt{let}\ \ v_d\ \texttt{=}\ [\![\mathsf{compile}_{\text{EXP}}(d)]\!] \\ & \texttt{in valid}_{v_d}\ \texttt{(}\ \texttt{let}\ v_1\ \texttt{=}\ [\![\mathsf{compile\text{-}branch}_{v_d}(e_1)]\!] \\ & \qquad\qquad\qquad \ldots \\ & \qquad\qquad\qquad v_k\ \texttt{=}\ [\![\mathsf{compile\text{-}branch}_{v_d}(e_k)]\!] \\ & \qquad\qquad \texttt{in} \\ & \qquad\qquad\quad \texttt{merge}_{v_d}\, v_1 \ldots v_k\ \texttt{)}\end{aligned}$$

where

1. the value of discriminant d is of type $T \in \textsc{Type}$,
2. $k = |\,\mathsf{constructors}(T)|$ denotes the number of constructors of T,
3. for all $i \in \{1 \dots k\}$, $n_i = \mathsf{arity}(C_i)$ denotes the arity of constructor C_i,
4. $v_d, v_1, \dots, v_k \in \textsc{Var}$ are fresh variable names, and
5. for all $i \in \{1 \dots k\}$, $\mathsf{compile\text{-}branch}_{v_d} : \textsc{Exp} \to \textsc{Exp}_{\mathbb{A}}$ compiles a single branch $e_i \in \textsc{Exp}$ of a case distinction:

$$\begin{aligned}\mathsf{compile\text{-}branch}_{v_d}(e_i) \coloneqq \texttt{let}\ & a_{i1} = \texttt{arguments}_1\ v_d \\ & \dots \\ & a_{in_i} = \texttt{arguments}_{n_i}\ v_d \\ \texttt{in}\ & \\ & [\![\mathsf{compile}_{\textsc{Exp}}(e_i)]\!]\end{aligned}$$

(if $n_i = 0$, then $\mathsf{compile\text{-}branch}_{v_d}(e_i)$ reduces to $[\![\mathsf{compile}_{\textsc{Exp}}(e_i)]\!]$)

Note that the compilation of a case distinction contains a validity check $\texttt{valid}_{v_d}$ on the value v_d of the evaluated discriminant $\mathsf{compile}_{\textsc{Exp}}(d)$ (cf. the dynamic semantics of `valid` in Definition 3.77). This check is necessary for simulating the dynamic semantics of case distinctions on undefined discriminants, i.e., when d evaluates to $\perp$ in a concrete program. According to Definition 3.60, the case distinction as a whole evaluates to $\perp$ in this situation.

For each branch e_i with $i \in \{1 \dots k\}$, $\mathsf{compile\text{-}branch}_{v_d}$ retrieves the n_i constructor arguments from the discriminant's abstract value v_d and binds them to the same names as in the concrete program. After evaluating each branch, all the results $v_1, \dots, v_k$ are merged into a single abstract value by calling `merge` (cf. Section 4.1.6).

We illustrate the compilation of case distinctions in Example 4.50.

Example 4.50 Consider the following declarations:

```
data Bool = False | True
not = \x -> case x of False -> True
                      True -> False
```

The corresponding abstract declaration is

```
not = \x -> let v_d = x
            in
              valid_v_d ( let v_1 = cons_(2,2)
                              v_2 = cons_(1,2)
                          in
                              merge_v_d v_1 v_2 )
```

In this case, `v_d` (resp. `v_1` and `v_2`) consists of a single flag f_d (resp. f_1 and f_2) because it represents a Boolean value. For the result $r \in \mathbb{A}$ of the corresponding merge it is sufficient to contain a single flag $f_r \in \mathrm{F}$ as well because it needs to differentiate between two values only. According to Example 4.42, $f_r = \neg f_d$ is a valid result because $(f_1) = \mathsf{numeric}_2^-(2) = (\mathsf{True})$ and $(f_2) = \mathsf{numeric}_2^-(1) = (\mathsf{False})$. The equality $f_r = \neg f_d$ shows how the negation of a value of type `Bool` is represented in terms of propositional formulas.

Finally, we specify $\mathsf{compile}_{\textsc{Exp}}$ by the union of all cases that we have covered so far.

Definition 4.51 $\mathsf{compile}_{\textsc{Exp}} : \textsc{Exp} \rightarrow \textsc{Exp}_{\mathbb{A}}$ compiles an expression of a concrete program to an expression of an abstract program, so that $\mathsf{compile}_{\textsc{Exp}}$ complies with the Definitions 4.44, 4.45, 4.46, 4.47, 4.48, and 4.49.

Now that we have specified the compilation of expressions, we apply $\mathsf{compile}_{\textsc{Exp}}$ in order to compile declarations and concrete programs.

Each declaration in a concrete program that binds an expression to a name is compiled to a declaration in the abstract program. Type declarations and type signatures are removed. That is because abstract programs operate on abstract values that are implicitly defined for each abstract program.

Definition 4.52 $\mathsf{compile}_{\textsc{Decl}} : \textsc{Decl} \nrightarrow \textsc{Decl}_{\mathbb{A}}$ transforms a declaration $d \in \textsc{Decl}$ that binds an expression $e \in \textsc{Exp}$ to a name $v \in \textsc{Var}$:

$$\mathsf{compile}_{\textsc{Decl}}(d) :=$$
$$\begin{cases} \texttt{constraint}_{\mathbb{A}}\ \texttt{=}\ [\![\mathsf{compile}_{\textsc{Exp}}(e)]\!] & \text{if } d = \texttt{constraint = } e \\ v\ \texttt{=}\ [\![\mathsf{compile}_{\textsc{Exp}}(e)]\!] & \text{if } d = v\ \texttt{=}\ e \end{cases}$$

Note that for most cases the $\mathsf{compile}_{\textsc{Decl}}$ keeps bound names, e.g., if expression $e \in \textsc{Exp}$ is bound to name $v \in \textsc{Var}$ in the concrete program, then $\mathsf{compile}_{\textsc{Exp}}(e)$ is bound to v in the abstract program. The single exception concerns the name `constraint`, which is changed to $\texttt{constraint}_{\mathbb{A}}$ in order to differentiate between both functions.

The compilation of concrete programs is reduced to the compilation of declarations that bind expressions to names.

Definition 4.53 The *compilation function* $\mathsf{compile} : \textsc{Prog} \rightarrow \textsc{Prog}_{\mathbb{A}}$ compiles all $n \in \mathbb{N}_{>0}$ declarations $d_1, \ldots, d_n \in \textsc{Decl}$ in a concrete program $c \in \textsc{Prog}$ that bind expressions to names, i.e., all declarations of the form $n\ \texttt{=}\ e$ for some pair $(n, e) \in \textsc{Var} \times \textsc{Exp}$:

$$\begin{aligned} \mathsf{compile}(c) := {} & [\![\mathsf{compile}_{\textsc{Decl}}(d_1)]\!] \\ & \ldots \\ & [\![\mathsf{compile}_{\textsc{Decl}}(d_n)]\!] \end{aligned}$$

Example 4.54 The listing in Appendix C.1 shows the result of compiling the concrete program from Example 3.9 using the function $\mathsf{compile}$.

4.2.1 Correctness of Compilation

In this section we show that the compilation function given in Definition 4.53 is correct. First of all, we define a correctness criterion for compiling concrete expressions, which is a variant of the correctness criterion for compiling concrete programs given in Definition 3.80.

Definition 4.55 The compilation function $\mathsf{compile}_{\textsc{Exp}} : \textsc{Exp} \to \textsc{Exp}_{\mathbb{A}}$ is *correct with respect to the encode/decode-pair* $(\mathfrak{E}, \mathfrak{D})$ if the following property holds for each program $c \in \textsc{Prog}$ and each expression $e \in \textsc{Exp}$ in c:

$$\forall (E_{\mathbb{A}}, \sigma) \in \mathbb{A}^{\textsc{Var}} \times \mathbb{B}^{\mathrm{V}} :$$
$$\mathfrak{D}_T(\sigma, \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}})) = \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathfrak{D}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), e)$$

where

1. $c_{\mathbb{A}} = \mathsf{compile}(c)$ denotes a correct compilation of c,
2. $e_{\mathbb{A}} = \mathsf{compile}_{\textsc{Exp}}(e)$ denotes a correct compilation of e,
3. $T \in \textsc{Type}_0$ denotes the type of expression e, and
4. $\mathfrak{D}_{\mathsf{Env}} : \mathbb{B}^{\mathrm{V}} \times \mathbb{A}^{\textsc{Var}} \to \mathbb{C}^{\textsc{Var}}$ decodes an abstract environment $E_{\mathbb{A}} \in \mathbb{A}^{\textsc{Var}}$ to a concrete environment under an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$:

$$\mathfrak{D}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}) \coloneqq \{(v, \mathfrak{D}_{T_a}(\sigma, a)) \mid (v, a) \in E_{\mathbb{A}}\}$$

where $T_a \in \textsc{Type}_0$ denotes the type of a in the concrete expression e (recall that concrete and abstract programs share the same set of variable names $\textsc{Var}$).

Note how Definition 4.55 resembles the correctness criterion for concrete and abstract programs (cf. Definition 3.80): we just added environments because concrete and abstract expressions are evaluated in the context of an environment (cf. Sections 3.2.3 and 3.3.3).

As we have seen in the previous section, the compilation of most expressions does neither change the program structure nor its semantics, thus, the correctness criterion is satisfied. In Example 4.56, we illustrate this for the compilation of constructor applications.

Example 4.56 The compilation function $\mathsf{compile}_{\textsc{Exp}}$ introduced in Definition 4.51 compiles a constructor application $C\ e_1 \ldots e_n$ of type $T \in \textsc{Type}_0$ in a concrete program $c \in \textsc{Prog}$ by generating the following abstract expression (cf. Definition 4.48) in an abstract program $c_{\mathbb{A}} \in \textsc{Prog}_{\mathbb{A}}$

$$\begin{array}{l}\texttt{let}\ v_1\ \texttt{=}\ e_{1\mathbb{A}}\\ \qquad \texttt{...}\\ \qquad v_n\ \texttt{=}\ e_{n\mathbb{A}}\\ \texttt{in}\\ \quad \texttt{cons}_{(j,k)}\ v_1\ \ldots\ v_n\end{array}$$

where

1. constructor $C \in \textsc{Con}$ is the j-th of type T,
2. $k = |\,\mathsf{constructors}(T)|$ denotes the number of constructors of type T,
3. $n = \mathsf{arity}(C)$ denotes the arity of constructor C,
4. for all $i \in \{1 \ldots n\}$, $e_{i\mathbb{A}} \in \textsc{Exp}_{\mathbb{A}}$ denotes a correct compilation of $e_i \in \textsc{Exp}$, and
5. for all $i \in \{1 \ldots n\}$, $T_i \in \textsc{Type}$ denotes the type of e_i.

In order to show that the compilation of constructor applications is correct with respect to the encode/decode-pair $(\mathsf{encode}, \mathsf{decode})$, we inspect the value that $\texttt{cons}_{(j,k)}\, v_1 \ldots v_n$ evaluates to. Therefore, we firstly evaluate all n compiled constructor arguments to the values $a_1, \ldots, a_n \in \mathbb{A}$ in the context of a fixed environment $E_{\mathbb{A}} \in \mathbb{A}^{\textsc{Var}}$:

$$\forall i \in \{1 \ldots n\} : a_i = \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{i\mathbb{A}})$$

By $E'_{\mathbb{A}} \in \mathbb{A}^{\textsc{Var}}$ we denote an updated environment with

$$E'_{\mathbb{A}} = E_{\mathbb{A}}[\{(v_1, a_1), \ldots, (v_n, a_n)\}]$$

According to Definition 3.77, $\texttt{cons}_{(j,k)}\, v_1 \ldots v_n$ evaluates to the abstract value $\mathsf{cons}_{(j,k)}(a_1, \ldots, a_n)$ whose semantics are given by Definition 4.32. Thus, for the present constructor application, the left-hand side of the equality in Definition 4.55 gives:

$$\begin{aligned} \forall \sigma \in \mathbb{B}^{\mathrm{V}} :\ & \mathsf{decode}_T(\sigma, \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E'_{\mathbb{A}}, \texttt{cons}_{(j,k)}\, v_1 \ldots v_n)) \\ =\ & \mathsf{decode}_T(\sigma, \mathsf{cons}_{(j,k)}(a_1, \ldots, a_n)) \\ =\ & C\ \ \mathsf{decode}_{T_1}(\sigma, a_1) \ldots \mathsf{decode}_{T_n}(\sigma, a_n) \end{aligned}$$

For the right-hand side of the equality in Definition 4.55, we have

$$\begin{aligned} \forall \sigma \in \mathbb{B}^{\mathrm{V}} :\ & \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E'_{\mathbb{A}}), C\ e_1 \ldots e_n) \\ =\ & C\ \ \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E'_{\mathbb{A}}), e_1) \\ & \quad \ldots \\ & \quad \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E'_{\mathbb{A}}), e_n) \end{aligned}$$

As the following holds by induction over the constructor arguments e_i for $i \in \{1 \ldots n\}$

$$\forall \sigma \in \mathbb{B}^{\mathrm{V}} : \mathsf{decode}_{T_i}(\sigma, a_i) = \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E'_{\mathbb{A}}), e_i)$$

we have shown that constructor applications are compiled correctly according to Definition 4.55.

There are similar proofs for variables, function applications and local bindings. The only exception concerns the compilation of case distinctions. Before we prove their correctness, we show that expressions that have been compiled using the compilation function given in Definition 4.51 correctly deal with failed computations. This is merely a special case of the correctness criterion specified in Definition 4.55, but it is worthwhile to be proven separately as it simplifies the correctness proof for compiled case distinctions.

Lemma 4.57 The compilation function $\mathsf{compile}_{\mathrm{EXP}}$ given in Definition 4.51 is correct with respect to failed computations, i.e., for a concrete expression $e \in \mathrm{EXP}$ of type $T \in \mathrm{TYPE}_0$ in a concrete program $c \in \mathrm{PROG}$ and an abstract expression $e_{\mathbb{A}} \in \mathrm{PROG}_{\mathbb{A}}$ in an abstract program $c_{\mathbb{A}} \in \mathrm{PROG}_{\mathbb{A}}$ with $(e, e_{\mathbb{A}}) \in \mathsf{compile}_{\mathrm{EXP}}$, the following equivalence holds:

$$\begin{aligned}\forall (E_{\mathbb{A}}, \sigma) \in \mathbb{A}^{\mathrm{VAR}} \times \mathbb{B}^{\mathrm{V}} : {} & \mathsf{decode}_T(\sigma, \mathsf{abstract\text{-}value}_{\mathrm{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}})) = \bot \\ & \Leftrightarrow \mathsf{concrete\text{-}value}_{\mathrm{EXP}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), e) = \bot\end{aligned}$$

Proof We show both directions of the equivalence in Lemma 4.57 for a fixed environment $E_{\mathbb{A}} \in \mathbb{A}^{\mathrm{VAR}}$ and a fixed assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$.

1. "$\Leftarrow$": According to the dynamic semantics of concrete expressions (cf. Definition 3.60), there are two reasons for e to evaluate directly to $\bot$, i.e., without evaluating any subexpressions: e is either an abstraction or a variable that is bound to $\bot$ in $\mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}})$. The former reason can be omitted for statically well-typed programs. For the latter case, the following holds:

$$\begin{aligned}\mathsf{concrete\text{-}value}_{\mathrm{EXP}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), e) &= \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}})(e) \\ &= \mathsf{decode}_T(\sigma, E_{\mathbb{A}}(e)) \\ &= \mathsf{decode}_T(\sigma, \bot_{\mathbb{A}}) \\ &= \bot\end{aligned}$$

In this case, the abstract expression $e_{\mathbb{A}}$ denotes the same variable (cf. Definition 4.44) but in the context of the abstract program $c_{\mathbb{A}}$. Thus, Lemma 4.57 holds because of the following equality:

$$\begin{aligned}\mathsf{decode}_T(\sigma, \mathsf{abstract\text{-}value}_{\mathrm{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}})) &= \mathsf{decode}_T(\sigma, E_{\mathbb{A}}(e_{\mathbb{A}})) \\ &= \mathsf{decode}_T(\sigma, \bot_{\mathbb{A}}) \\ &= \bot\end{aligned}$$

All other ways of evaluating the concrete expression e to $\bot$ involve the evaluation of subexpressions. The correctness of Lemma 4.57 for these cases can be proven by induction over the involved subexpressions.

2. "⇒": According to the dynamic semantics of abstract expressions (cf. Definition 3.77) that have been compiled by $\mathsf{compile}_{\mathrm{EXP}}$, there are three reasons for $e_{\mathbb{A}}$ to evaluate to $\perp_{\mathbb{A}}$:

1. In the first case, $e_{\mathbb{A}}$ is a variable bound to $\perp_{\mathbb{A}}$ in $E_{\mathbb{A}}$, i.e., $(e_{\mathbb{A}}, \perp_{\mathbb{A}}) \in E_{\mathbb{A}}$. Then, Lemma 4.57 holds because of

$$\begin{aligned}\mathsf{decode}_T(\sigma, \mathsf{abstract\text{-}value}_{\mathrm{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}})) &= \mathsf{decode}_T(\sigma, E_{\mathbb{A}}(e_{\mathbb{A}})) \\ &= \mathsf{decode}_T(\sigma, \perp_{\mathbb{A}}) \\ &= \perp\end{aligned}$$

and

$$\begin{aligned}\mathsf{concrete\text{-}value}_{\mathrm{EXP}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), e) &= \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}})(e) \\ &= \mathsf{decode}_T(\sigma, E_{\mathbb{A}}(e)) \\ &= \mathsf{decode}_T(\sigma, \perp_{\mathbb{A}}) \\ &= \perp\end{aligned}$$

2. In the remaining two cases, $e_{\mathbb{A}}$ denotes a compiled case distinction (cf. Definition 4.49):

$$\begin{aligned}e_{\mathbb{A}} = \ & \mathtt{let}\ v_d\ \mathtt{=}\ d_{\mathbb{A}} \\ & \mathtt{in}\ \mathtt{valid}_{v_d}\ \mathtt{(}\ \mathtt{let}\ v_1\,\mathtt{=}\,\mathtt{let}\ a_{11}\,\mathtt{=}\ \mathtt{arguments}_1\, v_d \\ & \qquad\qquad\qquad\qquad\qquad\quad \ldots \\ & \qquad\qquad\qquad\qquad\quad\ \mathtt{in}\ e_{1\mathbb{A}} \\ & \qquad\qquad\qquad\qquad \ldots \\ & \qquad\qquad\qquad\quad\ v_k\,\mathtt{=}\,\mathtt{let}\ a_{k1}\,\mathtt{=}\ \mathtt{arguments}_1\, v_d \\ & \qquad\qquad\qquad\qquad\qquad\quad \ldots \\ & \qquad\qquad\qquad\qquad\quad\ \mathtt{in}\ e_{k\mathbb{A}} \\ & \qquad\qquad\quad \mathtt{in} \\ & \qquad\qquad\quad\ \ \mathtt{merge}_{v_d}\, v_1\ \ldots\ v_k\ \mathtt{)}\end{aligned}$$

This abstract expression has been compiled from the following original case distinction e:

$$\begin{aligned}e = \mathtt{case}\ d\ \mathtt{of}\ & C_1\ a_{11} \ldots\ \mathtt{->}\ e_1 \\ & \ldots \\ & C_k\ a_{k1} \ldots\ \mathtt{->}\ e_k\end{aligned}$$

We assume the discriminant d being of type $T_d \in \textsc{Type}_0$ with $k = |\,\mathsf{constructors}(T_d)|$.

Now, there are two reasons why $e_{\mathbb{A}}$ might evaluate to $\perp_{\mathbb{A}}$:

(a) Because of the semantics of $\texttt{valid}_{v_d}$ (cf. Definition 3.77), $e_{\mathbb{A}}$ evaluates to $\bot_{\mathbb{A}}$ if $d_{\mathbb{A}}$ evaluates to $\bot_{\mathbb{A}}$, which gives the left-hand side of the equivalence in Lemma 4.57:

$$\begin{aligned}&\textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, d_{\mathbb{A}}) = \bot_{\mathbb{A}}\\ &\qquad\Longrightarrow \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}}) = \bot_{\mathbb{A}}\\ &\qquad\Longrightarrow \textsf{decode}_{T}(\sigma, \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}})) = \bot\end{aligned}$$

By induction over d and $d_{\mathbb{A}}$, we have

$$\begin{aligned}&\textsf{decode}_{T_d}(\sigma, \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, d_{\mathbb{A}})) = \bot\\ &\qquad\Leftrightarrow \textsf{concrete-value}_{\text{EXP}}(c, \textsf{decode}_{\textsf{Env}}(\sigma, E_{\mathbb{A}}), d) = \bot\end{aligned}$$

which implies the right-hand side of the equivalence in Lemma 4.57 due to the dynamic semantics of case distinctions (cf. Definition 3.60):

$$\begin{aligned}&\textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, d_{\mathbb{A}}) = \bot_{\mathbb{A}}\\ &\qquad\Longrightarrow \textsf{decode}_{T_d}(\sigma, \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, d_{\mathbb{A}})) = \bot\\ &\qquad\Longrightarrow \textsf{concrete-value}_{\text{EXP}}(c, \textsf{decode}_{\textsf{Env}}(\sigma, E_{\mathbb{A}}), d) = \bot\\ &\qquad\Longrightarrow \textsf{concrete-value}_{\text{EXP}}(c, \textsf{decode}_{\textsf{Env}}(\sigma, E_{\mathbb{A}}), e) = \bot\end{aligned}$$

Thus, Lemma 4.57 holds if the compilation $d_{\mathbb{A}}$ of the concrete discriminant d evaluates to $\bot_{\mathbb{A}}$.

(b) Because of the semantics of $\texttt{merge}_{v_d}$ (cf. Definition 4.41), $e_{\mathbb{A}}$ evaluates to $\bot_{\mathbb{A}}$ if all compiled branches $e_{i\mathbb{A}}$ evaluate to $\bot_{\mathbb{A}}$ for $i \in \{1 \dots k\}$. This gives the left-hand side of the equivalence in Lemma 4.57:

$$\begin{aligned}&\big(\forall i \in \{1 \dots k\} : \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{i\mathbb{A}}) = \bot_{\mathbb{A}}\big)\\ &\qquad\Longrightarrow \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}}) = \bot_{\mathbb{A}}\\ &\qquad\Longrightarrow \textsf{decode}_{T}(\sigma, \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}})) = \bot\end{aligned}$$

By induction over e_i and $e_{i\mathbb{A}}$ for all $i \in \{1 \dots k\}$, we have

$$\begin{aligned}&\textsf{decode}_{T}(\sigma, \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{i\mathbb{A}})) = \bot\\ &\qquad\Leftrightarrow \textsf{concrete-value}_{\text{EXP}}(c, \textsf{decode}_{\textsf{Env}}(\sigma, E_{\mathbb{A}}), e_i) = \bot\end{aligned}$$

which implies that each branch in the original case distinction e evaluates to $\bot$ in the present case. Then, e evaluates to $\bot$ as well:

$$\begin{pmatrix}\begin{aligned}&\forall i \in \{1 \dots k\} : \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{i\mathbb{A}}) = \bot_{\mathbb{A}}\\ &\qquad\Longrightarrow \textsf{decode}_{T}(\sigma, \textsf{abstract-value}_{\text{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{i\mathbb{A}})) = \bot\\ &\qquad\Longrightarrow \textsf{concrete-value}_{\text{EXP}}(c, \textsf{decode}_{\textsf{Env}}(\sigma, E_{\mathbb{A}}), e_i) = \bot\end{aligned}\end{pmatrix}$$

$$\Longrightarrow \textsf{concrete-value}_{\text{EXP}}(c, \textsf{decode}_{\textsf{Env}}(\sigma, E_{\mathbb{A}}), e) = \bot$$

Thus, Lemma 4.57 holds if the compilation $e_{i\mathbb{A}}$ of the concrete branch e_i evaluates to $\bot_{\mathbb{A}}$ for all $i \in \{1 \dots k\}$. ■

Now that we have proven the correctness of the compilation with respect to failed computations, we show the correctness of compiled case distinctions.

Lemma 4.58 For the compilation of case distinctions (cf. Definition 4.49), the compilation function $\mathsf{compile}_{\mathrm{EXP}}$ introduced in Definition 4.51 is correct according to the correctness criterion given in Definition 4.55 with respect to the encode/decode-pair $(\mathsf{encode}, \mathsf{decode})$.

Proof Assume a case distinction $e \in \textsc{Exp}$ of type $T \in \textsc{Type}_0$ in a concrete program $c \in \textsc{Prog}$

$$\begin{aligned} e = \texttt{case}\ d\ \texttt{of}\ & C_1\ a_{11} \dots a_{1n_1}\ \texttt{->}\ e_1 \\ & \dots \\ & C_k\ a_{k1} \dots a_{kn_k}\ \texttt{->}\ e_k \end{aligned}$$

where the concrete discriminant $d \in \textsc{Exp}$ is of type $T_d \in \textsc{Type}_0$ with $k = |\mathsf{constructors}(T_d)|$ and $\mathsf{constructors}(T_d) = (C_1, \dots, C_k)$. For all $i \in \{1 \dots k\}$, $n_i = \mathsf{arity}(C_i)$ denotes the arity of constructor C_i.

According to Definition 4.49, the compiled case distinction $e \in \textsc{Exp}_{\mathbb{A}}$ has the following shape:

$$\begin{aligned} e_{\mathbb{A}} = \ & \texttt{let}\ v_d\ \texttt{=}\ d_{\mathbb{A}} \\ & \texttt{in valid}_{v_d}\ \texttt{(}\ \texttt{let}\ v_1 \texttt{=} \texttt{let}\ a_{11} \texttt{=}\ \texttt{arguments}_1\ v_d \\ & \qquad\qquad\qquad\qquad\qquad \dots \\ & \qquad\qquad\qquad\qquad\qquad a_{1n_1} \texttt{=}\ \texttt{arguments}_{n_1}\ v_d \\ & \qquad\qquad\qquad\qquad\ \ \texttt{in}\ e_{1\mathbb{A}} \\ & \qquad\qquad\qquad\quad \dots \\ & \qquad\qquad\qquad\quad v_k \texttt{=} \texttt{let}\ a_{k1} \texttt{=}\ \texttt{arguments}_1\ v_d \\ & \qquad\qquad\qquad\qquad\qquad \dots \\ & \qquad\qquad\qquad\qquad\qquad a_{kn_k} \texttt{=}\ \texttt{arguments}_{n_k}\ v_d \\ & \qquad\qquad\qquad\qquad\ \ \texttt{in}\ e_{k\mathbb{A}} \\ & \qquad\qquad \texttt{in} \\ & \qquad\qquad\ \ \texttt{merge}_{v_d}\ v_1 \dots v_k\ \texttt{)} \end{aligned}$$

Furthermore, we assume all of the following:

1. The abstract expression $d_{\mathbb{A}} \in \textsc{Exp}_{\mathbb{A}}$ is a correct compilation of d.
2. The abstract expression $e_{i\mathbb{A}} \in \textsc{Exp}_{\mathbb{A}}$ is a correct compilation of the branch e_i for $i \in \{1 \dots k\}$.
3. For a fixed environment $E_{\mathbb{A}} \in \mathbb{A}^{\textsc{Var}}$, $e_{\mathbb{A}}$ evaluates to an abstract value other than $\bot_{\mathbb{A}}$, i.e., $\mathsf{abstract\text{-}value}_{\mathrm{EXP}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}}) \neq \bot_{\mathbb{A}}$. The opposing case is handled in Lemma 4.57.

In order to show that the compilation of the case distinction $e_{\mathbb{A}}$ is correct with respect to a fixed environment $E_{\mathbb{A}} \in \mathbb{A}^{\textsc{Var}}$, an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$, and the encode/decode-pair $(\mathsf{encode}, \mathsf{decode})$, we inspect the values that both expressions e and $e_{\mathbb{A}}$ evaluate to according to the value of their respective discriminant.

The value of the compiled expression $e_{\mathbb{A}}$ equals the value of $\texttt{merge}_{v_d}\, v_1 \ldots v_k$. According to Definition 4.41, $\texttt{merge}_{v_d}\, v_1 \ldots v_k$ decodes to the same value as $e_{i\mathbb{A}}$ does if the decoded flags of the abstract value of $d_{\mathbb{A}}$ represent the natural number $i \in \{1 \ldots k\}$:

$$\begin{gathered}\mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}, d_{\mathbb{A}}))) = i \\ \Longrightarrow \\ \begin{aligned}&\mathsf{decode}_T(\sigma, \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{\mathbb{A}})) \\ &\quad = \mathsf{decode}_T(\sigma, \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}[E_{i\mathbb{A}}], e_{i\mathbb{A}}))\end{aligned}\end{gathered}$$

Note that e_i is evaluated under the environment $\mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}[E_{i\mathbb{A}}])$ that contains the bounded constructor arguments:

$$E_{i\mathbb{A}} = \bigcup_{j \in \{1 \ldots n_i\}} (a_{ij}, \mathsf{arguments}_j(\mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}, d_{\mathbb{A}})))$$

The value of the original case distinction e equals the value of the i-th branch e_i if the value of the discriminant d matches on the i-th constructor $C_i \in \textsc{Con}$ of type T_d for $i \in \{1 \ldots k\}$ (cf. Definition 3.60):

$$\begin{gathered}\mathsf{matches}(C_i\ a_{k1} \ldots a_{kn_k}, \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), d)) \\ \Longrightarrow \\ \begin{aligned}&\mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), e) \\ &\quad = \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}[E_{i\mathbb{A}}]), e_i)\end{aligned}\end{gathered}$$

As

$$\begin{gathered}\mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}, d_{\mathbb{A}}))) = i \\ \Leftrightarrow \\ \mathsf{matches}(C_i\ a_{i1} \ldots a_{in_i}, \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), d))\end{gathered}$$

and

$$\begin{aligned}&\mathsf{decode}_T(\sigma, \mathsf{abstract\text{-}value}_{\textsc{Exp}}(c_{\mathbb{A}}, E_{\mathbb{A}}, e_{i\mathbb{A}})) \\ &\qquad\qquad = \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, \mathsf{decode}_{\mathsf{Env}}(\sigma, E_{\mathbb{A}}), e_i)\end{aligned}$$

holds for all $i \in \{1 \ldots k\}$ by induction over d, $d_{\mathbb{A}}$, e_i, and $e_{i\mathbb{A}}$, the compilation $e_{\mathbb{A}}$ of the case distinction e is correct according to Definition 4.55 with respect to the encode/decode-pair $(\mathsf{encode}, \mathsf{decode})$. ■

Now that we have proven the correctness of the compilation of concrete expressions, we show that this is sufficient for correctly compiling concrete programs.

Lemma 4.59 For two types $P, U \in \textsc{Type}_0$ and the set $\textsc{Prog}_{PU}$ of concrete programs, the compilation function $\mathsf{compile}$ given in Definition 4.53 is correct according to Definition 3.80 with respect to the encode/decode pair $(\mathsf{encode}, \mathsf{decode})$.

Proof Lemma 4.59 immediately follows from the correctness of the compilation function $\mathsf{compile}_{\text{EXP}}$ for concrete expressions. That is because all entities other than concrete expressions are either removed during the compilation of concrete programs (e.g., type declarations) or maintain their structure and dynamic semantics (e.g., declarations that bind values to identifiers). ■

4.3 Solving Constraints with CO^4

In this section, we address the present implementation of the constraint solver CO^4. We start by illustrating how the concepts we introduced in the previous sections act together in order to constitute CO^4's central solving algorithm (cf. Figure 4.60). In order to find a solution for a constraint specified by a concrete program $c \in \textsc{Prog}_{PU}$ for $P, U \in \textsc{Type}_0$ and a parameter $p \in \mathbb{C}_P$, the algorithm expects three inputs: c, p, and an abstract value $u_{\mathbb{A}} \in \mathbb{A}$. In case that the domain of discourse U is finite, $u_{\mathbb{A}}$ may be equal to $\mathsf{complete}(\{U\})$; otherwise, it is an incomplete abstract value (cf. Section 4.1.5).

The algorithm may have three different outcomes:

1. $\textsc{Unsat}$ if there is no solution and U is finite,
2. $\textsc{Maybe}$ if there is no solution and U infinite, or
3. a solution $u \in \mathbb{C}_U$ such that $\mathsf{concrete\text{-}value}(c)(p, u) = \mathtt{True}$.

The algorithm itself works as follows: at first, the concrete program c is compiled to an abstract program and the parameter p is encoded as an abstract value. Secondly, a propositional formula $f \in \mathrm{F}$ is derived by evaluating the abstract program, which then is feed into an external SAT solver. The application of an external SAT solver is a crucial point in CO^4's solving algorithm: due to the correctness of the compilation function $\mathsf{compile}$, finding a satisfying assignment $\sigma \in \mathbb{B}^{\mathsf{var}(f)}$ for the propositional variables $\mathsf{var}(f)$ in formula f implies that there is a solution $u = \mathsf{decode}_U(\sigma, u_{\mathbb{A}})$ for the concrete program c and the parameter p such that $\mathsf{concrete\text{-}value}(c)(p, u) = \mathtt{True}$. In order to highlight where the specified concepts are incorporated in the generation of the propositional formula f, Figure 4.61 depicts an instance of Figure 3.7 from Section 3.1.

Input

1. Concrete program $c \in \textsc{Prog}_{PU}$ for $P, U \in \textsc{Type}_0$
2. Abstract value $u_{\mathbb{A}} \in \mathbb{A}$
3. Concrete value $p \in \mathbb{C}_P$ representing a parameter

Output

1. Either $\textsc{Unsat}$, $\textsc{Maybe}$, or a solution $u \in U$ so that

$$\mathsf{concrete\text{-}value}(c)(p, u) = \texttt{True}$$

Algorithm

1. Generate an abstract program $c_{\mathbb{A}} \in \textsc{Prog}_{\mathbb{A}}$ with $c_{\mathbb{A}} = \mathsf{compile}(c)$
2. Encode the parameter p to $p_{\mathbb{A}} = \mathsf{encode}_P(p)$
3. Compute $r_{\mathbb{A}} = \mathsf{abstract\text{-}value}(c_{\mathbb{A}})(p_{\mathbb{A}}, u_{\mathbb{A}})$
4. Let $f \in \mathrm{F}$ denote the first and only flag of the abstract value $r_{\mathbb{A}}$, i.e., $\mathsf{flags}(r_{\mathbb{A}}) = (f)$
5. Apply an external SAT solver to find a satisfying assignment $\sigma \in \mathbb{B}^{\mathsf{var}(f)}$ for formula f where $\mathsf{var}(f)$ denotes the set of propositional variables in f:
   - σ exists: return the solution $u = \mathsf{decode}_U(\sigma, u_{\mathbb{A}})$ and terminate
   - σ does not exist:
     - U is finite: return $\textsc{Unsat}$ and terminate
     - U is infinite: return $\textsc{Maybe}$ and terminate

Figure 4.60: The solving algorithm implemented in CO^4.

4.3.1 Concerning the Completeness of CO^4

In this section, we show that CO^4's solving procedure is complete for constraints on finite domains of discourse. To illustrate CO^4's incompleteness for constraints on infinite domains of discourse, we also give an example of a constraint with an obvious solution that is not found by CO^4.

Theorem 4.62 The constraint solver CO^4 implements a complete solving procedure for constraints on finite domains of discourse, i.e., if a constraint $c : \mathrm{P} \times \mathrm{U} \to \mathbb{B}$ specified as a concrete program has a solution $u \in \mathrm{U}$ for a parameter $p \in \mathrm{P}$ and a finite domain of discourse U, then CO^4 finds a solution.

Proof For two types $P, U \in \textsc{Type}_0$, assume a concrete program $c \in \textsc{Prog}_{PU}$ where P represents the parameter domain, and U represents the domain of discourse. If U is finite, then there is a complete abstract value $u_{\mathbb{A}} = \mathsf{complete}(\{U\})$ that represents all values $\mathbb{C}_U \setminus \{\bot\}$, i.e., $u_{\mathbb{A}}$ can be decoded to all values in $\mathbb{C}_U \setminus \{\bot\}$:

$$\forall u \in \mathbb{C}_U \setminus \{\bot\} : \ \exists \sigma \in \mathbb{B}^{\mathrm{V}} : \mathsf{decode}_U(\sigma, u_{\mathbb{A}}) = u$$

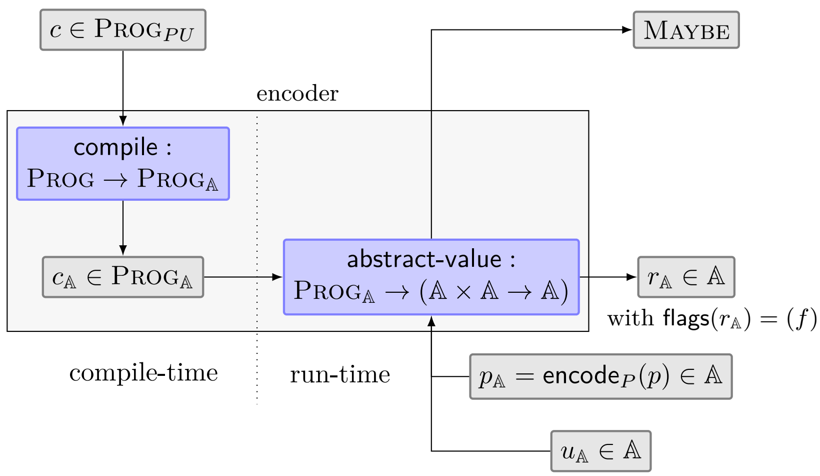


Figure 4.61: Generating a propositional formula $f \in \mathrm{F}$ using the concepts that have been introduced in the Chapters 3 and 4.

Because CO^4's compilation function $\mathsf{compile} : \textsc{Prog} \to \textsc{Prog}_{\mathbb{A}}$ is correct (cf. Lemma 4.59) with respect to the encode/decode-pair $(\mathsf{encode}, \mathsf{decode})$, the following equality holds for all parameters $p \in \mathbb{C}_P$, values $u \in \mathbb{C}_U$, and variable assignments $\sigma \in \mathbb{B}^{\mathrm{V}}$:

$$\forall (p, u, \sigma) \in \mathbb{C}_P \times \mathbb{C}_U \times \mathbb{B}^{\mathrm{V}} : \mathsf{concrete\text{-}value}(c)(p, u) \\ = \mathsf{decode}_{\mathtt{Bool}}(\sigma, \mathsf{abstract\text{-}value}(c_{\mathbb{A}})(\mathsf{encode}_P(p), \mathsf{encode}_U(u)))$$

Thus, if there is a solution $u \in \mathbb{C}_U$ for the concrete program c and a particular parameter $p \in \mathbb{C}_P$, then there also is an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$ so that

$$\mathsf{decode}_{\mathtt{Bool}}(\sigma, \mathsf{abstract\text{-}value}(c_{\mathbb{A}})(\mathsf{encode}_P(p), u_{\mathbb{A}})) = \mathtt{True}$$

and the complete abstract value $u_{\mathbb{A}}$ can always be decoded to u:

$$\mathsf{decode}_U(\sigma, u_{\mathbb{A}}) = u$$

As the SAT solver applied in the algorithm depicted in Figure 4.60 is complete as well, such an assignment is always found if there is a solution. Thus, the solving procedure implemented in CO^4 is complete for constraints represented by concrete programs on finite domains of discourse. ■

If the domain of discourse U is infinite (cf. Definition 3.54), CO^4's solving procedure is incomplete. In Example 4.63, we give a constraint that has an obvious solution that is not found by CO^4.

Example 4.63 Assume the following concrete program $c \in \textsc{Prog}$ encoding a constraint that is satisfied for the single element of the domain of discourse `Nat` that equals the given parameter:

```
data Bool = False | True
data Nat  = Z | S Nat

constraint = \p u -> equals p u

equals = \p u -> case p of
  Z -> case u of Z -> True
                 S y -> False
  S x -> case u of Z -> False
                   S y -> equals x y
```

Because `Nat` is infinite, we need to construct an incomplete abstract value (cf. Section 4.1.5) that represents a finite subset of $\mathbb{C}_{\texttt{Nat}}$. As `Nat` encodes the natural numbers, it is reasonable to restrict the designated solution's range to the first $n \in \mathbb{N}$ natural numbers. Thus, $\mathsf{abstract\text{-}nat}(n)$ gives an abstract value that encodes the natural numbers less or equal to n:

$$\mathsf{abstract\text{-}nat}(n) :=$$
$$\begin{cases} \mathsf{encode}_{\texttt{Nat}}(\texttt{Z}) & \text{if } n = 0 \\ ((f), (\mathsf{abstract\text{-}nat}(n-1))) & \text{if } n > 0 \text{ and } f \in \mathrm{V} \text{ is a fresh} \\ & \text{propositional variable} \end{cases}$$

Now, applying the algorithm from Figure 4.60 to a parameter $p = \texttt{S}\,\texttt{Z}$ and an abstract value $u_{\mathbb{A}} = \mathsf{abstract\text{-}nat}(0)$ always gives the result MAYBE because:

$$\begin{aligned} \forall \sigma \in \mathbb{B}^{\mathrm{V}} : & \mathsf{concrete\text{-}value}(c)(\texttt{S}\,\texttt{Z}, \mathsf{decode}_{\texttt{Nat}}(\sigma, u_{\mathbb{A}})) \\ = \; & \mathsf{concrete\text{-}value}(c)(\texttt{S}\,\texttt{Z}, \texttt{Z}) \\ = \; & \mathsf{concrete\text{-}value}_{\mathrm{EXP}}(c, \{(p, \texttt{S}\,\texttt{Z}), (u, \texttt{Z})\}, \texttt{equals p u}) \\ = \; & \mathsf{concrete\text{-}value}_{\mathrm{EXP}}(c, \{(p, \texttt{S}\,\texttt{Z}), (u, \texttt{Z})\}, \texttt{False}) \\ = \; & \mathsf{False} \end{aligned}$$

That is because $u_{\mathbb{A}}$ represents only the value `Z` which is not equal to p. Therefore, CO^4 does not find the obvious solution `S Z`. To resolve this problem, the range of values represented by the designated solution $u_{\mathbb{A}}$ must be increased, e.g., by setting $u_{\mathbb{A}}$ to $\mathsf{abstract\text{-}nat}(1)$.

4.3.2 Usage of CO^4

CO^4 is implemented as a library written in Haskell. It consists of approximately 5500 lines of Haskell code and contains two essential parts: a compilation pipeline for generating abstract programs on compile-time, and a library for producing propositional formulas during run-time.

As concrete and abstract programs are syntactic subsets of Haskell, compile-time generation of abstract programs is done using the Template-Haskell [68] library.

Template-Haskell provides access to the abstract syntax tree of Haskell code, which can be programmatically transformed and extended. When compiling a Haskell module $\mathcal{H}$ containing a concrete program $c \in$ Prog, the following steps are performed:

1. Template-Haskell provides CO^4 with the abstract syntax tree of c.
2. The compilation pipeline in CO^4 produces an abstract syntax tree for the resulting abstract program $\mathsf{compile}(c)$.
3. Template-Haskell writes the abstract syntax tree of $\mathsf{compile}(c)$ back to $\mathcal{H}$.
4. $\mathcal{H}$ is compiled by the Glasgow Haskell Compiler.

This approach conveniently embeds the generation of abstract programs into the compilation of Haskell programs. In order to realize the semantics of abstract programs, they are implemented as monadic Haskell programs in the present implementation of CO^4. Thus, their actual syntax differs from what is specified in Section 3.3.1.

In Example 4.64, we illustrate how a solution for an exemplary concrete program is searched for using the present implementation of CO^4.

Example 4.64 The following Haskell module $\mathcal{H}$ applies CO^4 to find the solution of a trivial constraint that is represented by the concrete program $c \in$ Prog, which is written between Template-Haskell's quotation marks `[d|...|]`.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
module Main where

import           Prelude hiding (Bool(..))
import qualified Data.Maybe as M
import           Language.Haskell.TH (runIO)
import           System.Environment (getArgs)
import qualified Satchmo.Core.SAT.Minisat
import qualified Satchmo.Core.Decode
import           CO4

$( [d| data Bool       = False | True deriving Read
       data Color      = Red | Green | Blue
                         deriving Show
       data Monochrome = Black | White deriving Show
       data Pixel      = Colored Color
                       | Background Monochrome
                         deriving Show

```

```
       constraint :: Bool -> Pixel -> Bool
       constraint p u = case p of
        False -> case u of Background _ -> True
                           _            -> False
        True -> isBlue u

       isBlue :: Pixel -> Bool
       isBlue u = case u of
        Background _ -> False
        Colored    c -> case c of
          Blue -> True
          _    -> False

   |] >>= compile []
  )

main :: IO ()
main = do
  [ p ]  <- getArgs
  result <- solveAndTestP (read p) complete
            encConstraint constraint
  putStrLn (show result)
```

The abstract syntax tree of c is passed to CO^4's `compile` function that generates the abstract syntax tree of the resulting abstract program. The resulting syntax tree is written back to $\mathcal{H}$ using Template-Haskell's splice operator `$(...)`.

To find a solution for c, the function `solveAndTestP` is called in the `main` function of $\mathcal{H}$. We pass four arguments to `solveAndTestP`:

1. a parameter $\texttt{p} \in \mathbb{C}_{\texttt{Bool}}$ that is read from the command line when `main` is invoked,
2. an abstract value `complete` that denotes a complete abstract value (cf. Definition 4.34) for all values in the domain of discourse `Pixel`,
3. the top-level declaration `encConstraint` of the abstract program, and
4. the top-level declaration `constraint` of the concrete program.

Compiling $\mathcal{H}$ gives an executable program $\mathcal{H}$`.exe`. Running $\mathcal{H}$`.exe` with a given parameter evaluates the abstract program $\mathsf{compile}(c)$, generates a propositional formula $f \in \mathrm{F}$ and calls the SAT solver MiniSat to find a satisfying assignment for the variables $\mathsf{var}(f)$ in f. If there is such an assignment, CO^4 constructs a solution from the domain `Pixel`. For running $\mathcal{H}$`.exe` with `True` as the command line argument, CO^4 prints the following output:

```
Start producing CNF
Number of shared values: 0
Allocator: #variables: 3, #clauses: 0
Toplevel: #variables: 0, #clauses: 1
CNF finished
#variables: 5, #clauses: 7, #literals: 17,
  clause density: 1.4
#variables (Minisat): 5, #clauses (Minisat): 6,
  clause density: 1.2
#clauses of length 1: 1
#clauses of length 2: 2
#clauses of length 3: 4

Starting solver
Solver finished in 0.0 seconds (result: True)
Starting decoder
Decoder finished
Test: True
Just (Colored Blue)
```

The last line shows the actual solution `Colored Blue`. The next-to-last line gives the result of evaluating `constraint True (Colored Blue)`. Recall that this test must always succeed, otherwise the compilation was not correct. The remaining output shows profiling information that is described in more detail in Section 6.1.

4.3.3 Implementation Details

In the present implementation of CO^4, the compilation from concrete to abstract programs happens as specified in Section 4.2, but the representation of propositional formulas differs from the specification in Definition B.4: propositional formulas are stored not as trees but as as directed acyclic graphs (DAG) where each vertex either represents a variable or a connective of several subformulas (cf. Figure 4.65). This representation allows subformulas to be shared, which is reasonable as propositional formulas may become huge for more complex constraints. This is essential as performance would be poor in terms of memory consumption if they were actually stored in a tree-shaped representation. The original tree-shape can always be reconstructed by unravelling the nodes of the DAG.

In the present implementation of CO^4, the DAG representation of a propositional formula is managed by an intermediate library called Satchmo-core. During the runtime of CO^4, Satchmo-core transparently transforms subformulas into their conjunctive normal form (CNF, cf. Definition B.10) using Tseitin's transformation (cf. Definition B.13). Such a transformation is necessary as most SAT solvers expect propositional formulas to be in CNF.

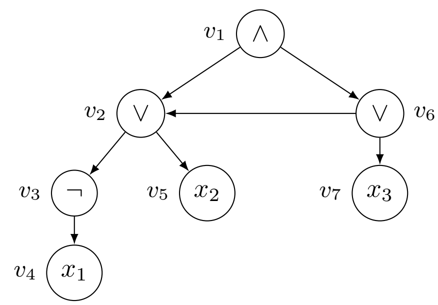


Figure 4.65: Representation of $(\neg x_1 \vee x_2) \wedge ((\neg x_1 \vee x_2) \vee x_3) \in \mathrm{F}$ with $x_1, x_2, x_3 \in \mathrm{V}$ as a DAG with vertex set $\{v_1 \dots v_7\}$. Note the sharing of the subformula $\neg x_1 \vee x_2$, which is represented by vertex v_2.

For each subformula $f \in \mathrm{F}$, Tseitin's transformation generates a fresh propositional variable $v \in \mathrm{V}$ such that $v \Leftrightarrow f$. Each vertex containing a logical connective in the DAG in Figure 4.65 corresponds to such a variable v. Thus, each vertex is a representative for the respective subformula f. As the variable v is semantically equivalent to the subformula f, it is sufficient to store the propositional variables generated by Tseitin's transformation in the flags of an abstract value $a \in \mathbb{A}$. This way, f is automatically shared during the following operations on the value a (cf. the dynamic semantics of abstract expressions in Definition 3.77):

1. The value a is bound to a new name or parameter inside a local binding or a function application.
2. One of the arguments of a is extracted via $\mathsf{arguments}_i(a)$ for $i \in \mathbb{N}_{>0}$ (cf. Definition 4.4).
3. A new abstract value $a' \in \mathbb{A}$ is constructed with a being one of its arguments such that $a' = \mathsf{cons}_{(j,k)}(\dots, a, \dots)$ for $j, k \in \mathbb{N}_{>0}$ and $j < k$ (cf. Definition 4.32).
4. A new abstract value $a' \in \mathbb{A}$ is constructed by merging a such that $a' = \mathsf{merge}_d(\dots, a, \dots)$ with $d \in \mathbb{A}$ being an abstract value whose flags do not contain any propositional variables, i.e., Lemma 4.43 holds.

Beyond these operations, running CO^4 without further optimizations might generate identical subformulas which are not getting shared. Therefore, Section 6.2 introduces an optimization technique called memoization which enables sharing of subformulas even for merging operations that are not covered by Lemma 4.43.

Besides managing the DAG representation of a propositional formula, using an intermediate library like Satchmo-core has another benefit: it hides the details of communicating with a particular SAT solver like MiniSat behind a consistent programming interface. This way, CO^4 itself becomes solver-agnostic and can be used with any solver supported by Satchmo-Core.

4.3.4 Alternative Encodings for Abstract Values

Abstract values generated by `complete` have two special properties: they are tree-shaped and their encoded constructor arguments may share flags (cf. Definition 4.35). Alternative encodings may omit one or both of these properties.

Flat Encoding Storing all flags of an abstract value as a flat sequence resembles the binary serialization of structured data. While managing a sequence is simpler than managing a tree, the access to the flags of a particular argument is more cumbersome because the size of the other arguments has to be taken into account. Figure 4.66 illustrates the encoding of a simple value by a sequence of its flags.

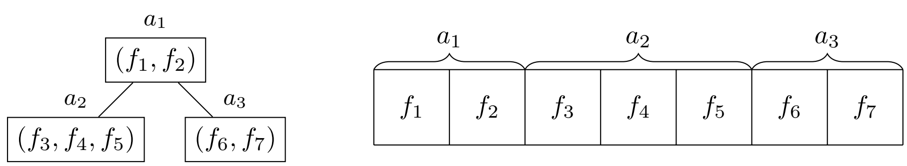


Figure 4.66: Flags may be organized in a tree or sequence, but accessing arguments in a sequence requires knowledge about the global layout, e.g., for accessing a_3 one needs to know the size of the other values a_1 and a_2.

The computational overhead required for accessing arguments and merging values are the reasons this encoding is not applied in CO^4.

Non-overlapping Encoding Another way of encoding abstract values using a tree of flags is to omit any overlappings. Then, each constructor index is represented by a distinct sequence of flags. Figure 4.67 illustrates this encoding of simple abstract value in comparison to the specification of complete abstract values given in Definition 4.35.

```
data Bool = False
          | True
data Either a b = Left a
                | Right b
```

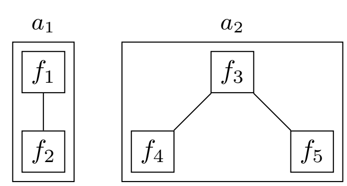


Figure 4.67: Two tree-shaped encodings for all values of type `Either Bool Bool`. Note that a_1 is encoded according to Definition 4.35, i.e., f_2 encodes the argument of `Left` and `Right`, while a_2 uses distinctive flags for encoding the argument of `Left` and `Right`.

A non-overlapping encoding seems more intuitive in the first place. It also simplifies the merging of abstract values that is necessary for transforming case distinctions. However, the main drawback is that this approach uses more flags than the overlapping one. Thus, a non-overlapping encoding immediately increases the size of the generated formulas, which is not acceptable for non-trivial constraints.

Chapter 5

Compilation of Advanced Language Features

In Chapter 4, we have specified the compilation of concrete programs that conform to the syntax and semantics defined in Chapter 3. These concrete programs are subject to certain restrictions: they must be first-order, total, and may not feature local abstractions. In this chapter, we lift these restrictions by introducing the compilation of local abstractions, a restricted form of higher-order functions, and partial functions. These features increase the expressiveness of concrete programs so that constraints can be specified in a more concise way.

Firstly, this chapter introduces extended concrete programs, i.e., concrete programs that feature the aforementioned concepts. Then, we give an overview of the compilation of extended concrete programs to abstract programs. This compilation is merely a reduction to concrete programs as they have been introduced in Chapter 3, i.e., the compilation function itself does not change.

This chapter also gives simple examples that illustrate the usefulness of the advanced language features present in extended concrete programs. Chapter 7 shows more comprehensive use-cases that use these features.

5.1 Extended Concrete Programs

This section introduces extended concrete programs as superset of concrete programs. We define three kinds of extended concrete programs $\textsc{Prog}_1$, $\textsc{Prog}_2$, and $\textsc{Prog}_3$ where

1. each kind denotes a particular feature set, and
2. $\textsc{Prog} \subsetneq \textsc{Prog}_1 \subsetneq \textsc{Prog}_2 \subsetneq \textsc{Prog}_3$.

The compilation process for extended concrete programs (cf. Figure 5.1) firstly reduces a given concrete program in $\textsc{Prog}_3$ to a concrete program in $\textsc{Prog}$. Then, the resulting concrete program is compiled to an abstract program in $\textsc{Prog}_{\mathbb{A}}$ by applying the function compile. The compilation process as it has been specified in Chapter 4 is not changed.

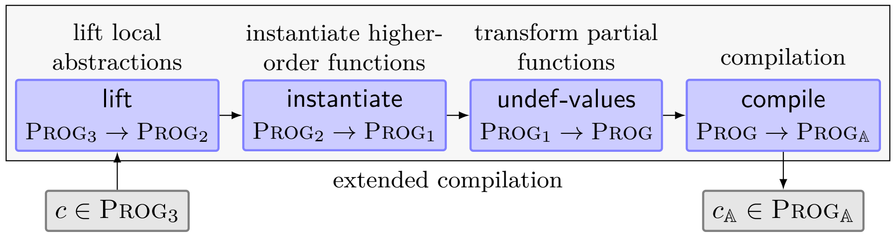


Figure 5.1: Compilation of extended concrete programs $\textsc{Prog}_3$ to abstract programs $\textsc{Prog}_{\mathbb{A}}$.

In the following, we define all three kinds of extended concrete programs in the reverse order of their compilation.

Extended Concrete Programs of Kind 1 Extended concrete programs of kind 1 denote a superset of concrete programs that includes partial functions. We will use the distinct identifier `undefined` to specify partial functions. Example 5.2 illustrates a simple function that is undefined for certain arguments.

Example 5.2 `f` defines a function that is undefined for argument `T3`:

```
data Bool = False | True
data T = T1 Bool | T2 | T3

f :: T -> Bool
f = \t -> case t of T1 b -> b
                    T2   -> True
                    T3   -> undefined
```

We give a specification for extended concrete programs of kind 1.

Definition 5.3 The *set of extended concrete programs of kind 1* $\textsc{Prog}_1 \supsetneq \textsc{Prog}$ is a superset of concrete programs $\textsc{Prog}$ so that each program in $\textsc{Prog}_1$ may contain a free identifier `undefined` $\in \textsc{Var}$, i.e., `undefined` may not be bound to any value.

The dynamic semantics of extended concrete programs of kind 1 equal the dynamic semantics of concrete programs. The special identifier `undefined` always evaluates to $\bot \in \mathbb{C}$ because it may not be bound according to Definition 5.3.

Section 5.4 illustrates how extended concrete programs of kind 1 are reduced to concrete programs.

Extended Concrete Programs of Kind 2 Extended concrete programs of kind 2 extend $\textsc{Prog}_1$ in order to feature restricted support for higher-order functions. In the context of CO^4, higher-order functions are functions that expect at least one of their arguments to be a function. Higher-order functions are useful because they provide a powerful way of composing simple functions into more complex ones. Example 5.4 illustrates a simple higher-order function.

Example 5.4 The following function `mapMaybe` takes another function that is applied to the constructor argument of `Just`:

```
data Maybe a = Nothing | Just a

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe = \f m -> case m of
  Nothing -> Nothing
  Just a -> Just (f a)
```

We give a specification for extended concrete programs of kind 2.

Definition 5.5 The *set of extended concrete programs of kind 2* $\textsc{Prog}_2 \supsetneq \textsc{Prog}_1$ is a superset of extended concrete programs of kind 1 $\textsc{Prog}_1$ so that each program in $\textsc{Prog}_2$ may contain higher-order functions, i.e., Property 3b of Definition 3.43 does not apply to the programs in $\textsc{Prog}_2$.

Section 5.3 illustrates how extended concrete programs of kind 2 are reduced to extended concrete programs of kind 1.

Extended Concrete Programs of Kind 3 Extended concrete programs of kind 3 may contain local abstractions. Local abstractions allow functions to be defined in the context of another expression and provide several benefits. First of all, they avoid cluttering up the top-level namespace by allowing functions to be defined where they are actually needed. Doing so obeys the Separation-of-Concerns paradigm because related functions are encapsulated in a shared scope that is hidden from other parts of the program. The second advantage of local abstractions are their ability to access values that were bound in an outer scope. This reduces the number of arguments that need to be passed to the abstraction. Example 5.6 illustrates a concrete program that features two locally defined functions.

Example 5.6 The following function `f` binds two local abstractions `not` and `g`:

```
data Bool = False | True
constraint = \x y ->
  let not = \a -> case a of False -> True
                            True -> False
      g = \a -> case x of False -> not a
                          True -> a
  in
    g y
```

Note that `g` captures the variable `x` of its enclosing scope.

We give a specification for extended concrete programs of kind 3.

Definition 5.7 The *set of extended concrete programs of kind 3* $\textsc{Prog}_3 \supsetneq \textsc{Prog}_2$ is a superset of extended concrete programs of kind 2 $\textsc{Prog}_2$ so that each program in $\textsc{Prog}_3$ may contain local abstractions, i.e., Property 2 of Definition 3.43 does not apply to the programs in $\textsc{Prog}_3$.

Section 5.2 illustrates how extended concrete programs of kind 3 are reduced to extended concrete programs of kind 2.

Notation In this section, we denoted extended concrete programs of kind 1, 2, and 3 by $\textsc{Prog}_1$, $\textsc{Prog}_2$, and $\textsc{Prog}_3$, respectively. Similarly, we will denote certain entities of these extended concrete programs with the corresponding subscript. For example, $\textsc{Exp}_i$ denotes the set of expressions for extended concrete programs of kind i where $i \in \{1, 2, 3\}$. Entities that are equal in all kinds of extended concrete programs are not annotated by a subscript, e.g., the set of variables $\textsc{Var}$.

5.2 Local Abstractions

This section defines a transformation $\mathsf{lift} : \textsc{Prog}_3 \to \textsc{Prog}_2$ from extended concrete programs of kind 3 to extended concrete programs of kind 2. As local abstractions are the distinctive feature of programs in $\textsc{Prog}_3$, we give a definition for local abstractions.

Definition 5.8 An abstraction $e \in \textsc{Exp}_3$ is denoted as *local* in an extended concrete program $c \in \textsc{Prog}_3$ if e occurs in c and there is no declaration v `=` e in c for some name $v \in \textsc{Var}$.

The transformation lift lifts local abstractions to declarations [45]. Local abstractions may capture values from the context in which they are defined. These values must be passed explicitly as additional arguments to the lifted abstraction if they are not bound in another declaration in the same concrete program.

Definition 5.9 $\mathsf{free} : \textsc{Exp}_3 \to 2^{\textsc{Var}}$ gives all variables that appear *free* in an expression $e \in \textsc{Exp}_3$ and is defined by:

$$\forall n \in \mathbb{N}_{>0} : \mathsf{free}(e) \coloneqq$$

$$\begin{cases} \{e\} \setminus \{\texttt{undefined}\} & \text{if } e \in \textsc{Var} \\ \{\,\} & \text{if } e \in \textsc{Con} \\ \mathsf{free}(e') \cup \bigcup_{1 \le i \le n} \mathsf{free}(e_i) & \text{if } e = e'\ e_1 \dots e_n \\ \mathsf{free}(e') \setminus \{v_1, \dots, v_n\} & \text{if } e = \texttt{\textbackslash}\, v_1 \dots v_n \texttt{->} e' \\ \mathsf{free}(e') \cup \bigcup_{1 \le i \le n} \mathsf{free}_{\textsc{Match}}(p_i, e_i) & \text{if } e = \texttt{case}\ e'\ \texttt{of}\ \begin{array}[t]{l} p_1 \texttt{->} e_1 \\ \dots \\ p_n \texttt{->} e_n \end{array} \\ \big(\mathsf{free}(e') \cup \bigcup_{1 \le i \le n} \mathsf{free}(e_i)\big) \setminus \{v_1, \dots, v_n\} & \text{if } e = \begin{array}[t]{l} \texttt{let}\ v_1 \texttt{=} e_1 \\ \quad \dots \\ \quad v_n \texttt{=} e_n \\ \texttt{in}\ e' \end{array} \end{cases}$$

where

$$\forall n \in \mathbb{N} : \mathsf{free}_{\textsc{Match}}(C\ v_1 \dots v_n, e) \coloneqq \mathsf{free}(e) \setminus \{v_1, \dots, v_n\}$$

Example 5.10

1. $\mathsf{free}(\texttt{\textbackslash x y -> f (g x) (h y)}) = \{\texttt{f}, \texttt{g}, \texttt{h}\}$
2. $\mathsf{free}(\texttt{case x of C y -> f x y z}) = \{\texttt{x}, \texttt{f}, \texttt{z}\}$
3. $\mathsf{free}(\texttt{let x = f y in g x}) = \{\texttt{f}, \texttt{y}, \texttt{g}\}$

For a local abstraction $e \in \textsc{Exp}_3$ in an extended concrete program $c \in \textsc{Prog}_3$, each variable in $\mathsf{free}(e)$ that is not bound by some declaration of c must be explicitly passed as a argument to the lifted version of e. Thus, the set of names that are bound by c needs to be determined in the first place.

Definition 5.11 $\mathsf{global} : \textsc{Prog}_3 \to 2^{\textsc{Var}}$ computes the set of names that are bound by declarations in a given extended concrete program $c \in \textsc{Prog}_3$ and is defined by:

$$\mathsf{global}(c) \coloneqq \{n \mid n \texttt{=} e \text{ is a declaration in } c\}$$

Now we can specify how a local abstraction is lifted to a declaration.

Definition 5.12 For $n \in \mathbb{N}_{>0}$, $\mathsf{lift}_{\textsc{Exp}} : \textsc{Prog}_3 \times \textsc{Exp}_3 \nrightarrow \textsc{Decl}_3$ transforms a local abstraction $\texttt{\textbackslash}\, v_1 \dots v_n \texttt{->} e' \in \textsc{Exp}_3$ in an extended concrete program $c \in \textsc{Prog}_3$ to a declaration and is defined by:

$$\mathsf{lift}_{\textsc{Exp}}(c, \texttt{\textbackslash}\, v_1 \dots v_n \texttt{->} e') = v \texttt{=}\ \texttt{\textbackslash}\, v_1 \dots v_n\ u_1 \dots u_m \texttt{->} e'$$

where

1. v denotes the name e' is locally bound to (if e' was not locally bound to a name, then v denotes a fresh name), and
2. $\{u_1, \ldots, u_m\} = \mathsf{free}(\backslash\, v_1 \ldots v_n\, \texttt{->}\, e') \setminus \mathsf{global}(c)$

After we have lifted an abstraction $e \in \textsc{Exp}_3$ of an extended concrete program $c \in \textsc{Prog}_3$ to a declaration, we need to consider that $\mathsf{lift}_{\textsc{Exp}}(c, e)$ expects more arguments than e if e contains free variables that are not bound by any declaration in c. In this case, we pass the values of these free variables to each call of $\mathsf{lift}_{\textsc{Exp}}(e, c)$. Example 5.13 illustrates the lifting of a simple abstraction and shows the consequences for calls to this abstraction.

Example 5.13 Assume an extended concrete program $c \in \textsc{Prog}_3$ that contains the following expression $e \in \textsc{Exp}_3$:

$$
\begin{aligned}
e = {} & \texttt{let f = \textbackslash x y -> g x y z} \\
& \texttt{in f a b}
\end{aligned}
$$

Note that $\mathsf{free}(\texttt{\textbackslash x y ->} \ldots) = \{\texttt{g}, \texttt{z}\}$. Assuming that $\texttt{g} \in \mathsf{global}(c)$ and $\texttt{z} \notin \mathsf{global}(c)$, lifting the local abstraction gives:

$$\mathsf{lift}_{\textsc{Exp}}(c, \texttt{\textbackslash x y ->} \ldots) = \texttt{f = \textbackslash x y z -> g x y z}$$

Because the lifted version of `f` expects an additional argument, we need to adjust every call to `f` in e accordingly.

By repeatedly applying $\mathsf{lift}_{\textsc{Exp}}$ we are able to lift all local abstractions to declarations.

Definition 5.14 $\mathsf{lift} : \textsc{Prog}_3 \to \textsc{Prog}_2$ lifts each local abstraction in an extended concrete program $c \in \textsc{Prog}_3$ by applying the following algorithm:

$\mathsf{lift}(c) =$
  if (c contains local abstraction $e \in \textsc{Exp}_3$) **then**
    if (e is bound to name $f \in \textsc{Var}$ in a `let` expression) **then**
      $c \leftarrow$ remove binding of f
      $c \leftarrow$ add arguments $\mathsf{free}(e) \setminus \mathsf{global}(c)$ to each application of f in c
      $c \leftarrow$ add declaration $\mathsf{lift}_{\textsc{Exp}}(c, e)$ to c
      return $\mathsf{lift}(c)$
    else
      $c \leftarrow$ replace e by fresh name $f \in \textsc{Var}$ in c
      $c \leftarrow$ add arguments $\mathsf{free}(e) \setminus \mathsf{global}(c)$ to the application of f in c
      $c \leftarrow$ add declaration $\mathsf{lift}_{\textsc{Exp}}(c, e)$ to c
      return $\mathsf{lift}(c)$
  else
    return c

Note that the algorithm illustrated in Definition 5.14 does not account for some corner cases:

1. If a binding is removed from a `let` expression `let ... in` e with a single binding, then the `let` expression must be replaced by e (cf. Example 5.13).

2. The order in which local abstractions are lifted affects the number of additional arguments that are added by $\mathsf{lift}_{\mathrm{EXP}}$: to avoid introducing unnecessary arguments, local abstractions should be lifted in a top-down fashion, i.e., from outer scopes to inner scopes.

Example 5.15 shows how the result of lift depends on the order in which local abstractions are lifted.

Example 5.15 Assume the following declaration of `foo` that defines two local abstractions `f` and `g`:

```
foo = \x ->
  let f = \y -> ...
  in
    let g = \z -> ... f z ...
    in
      ...
```

Lifting `g` in the first place introduces a new parameter for the free variable `f`. Thus, the final result of lifting `g` and `f` is:

```
foo = \x -> ...
f = \y -> ...
g = \z f -> ... f z ...
```

Lifting `f` before `g` avoids the introduction of a new parameter for variable `f` because when `g` is lifted, `f` is already bound in a declaration.

Example 5.16 illustrates the lifting of all local abstractions in an extended concrete program of kind 3.

Example 5.16 For the following extended concrete program $c \in \textsc{Prog}_3$

```
data Bool = False | True
constraint = \x y ->
  let not = \a -> case a of False -> True
                            True -> False
      g = \a -> case x of False -> not a
                          True -> a
  in
    g y
```

$\mathsf{lift}(c)$ gives the following program:

```
data Bool = False | True
constraint = \x y -> g y x

not = \a -> case a of False -> True
                      True -> False

g = \a x -> case x of False -> not a
                      True -> a
```

5.3 Higher-Order Functions

This section defines a transformation $\mathsf{instantiate} : \textsc{Prog}_2 \rightarrow \textsc{Prog}_1$ from extended concrete programs of kind 2 to extended concrete programs of kind 1. This transformation instantiates higher-order functions to first-order functions [59]. At first, we give a definition for a higher-order function in the context of extended concrete programs of kind 2.

Definition 5.17 For all $n \in \mathbb{N}_{>0}$, an abstraction $\backslash\, v_1 \ldots v_n$ `->` $e \in \textsc{Exp}_2$ specifies a *higher-order function* if the type T_i of v_i for $i \in \{1 \ldots n\}$ is a functional value, i.e.,

$$\exists i \in \{1 \ldots n\} : \mathsf{rootsym}(T_i) = \texttt{->}$$

The variable v_i denotes a *higher-order parameter*, and an expression that is bound to v_i denotes a *higher-order argument*.

In the following, we assume that each higher-order function f has exactly one higher-order parameter $v \in \textsc{Var}$ where v is the first parameter of f. This simplification is merely done for readability and can be trivially extended to more complex higher-order functions.

For each application of a higher-order function, a corresponding first-order instance is generated by substituting each higher-order parameter with the name that the higher-order argument is bound to. There are two reasons why we can safely assume that each higher-order argument has already been bound to a name:

1. When instantiating higher-order functions, all local abstractions already have been lifted to declarations. Thus, each local abstraction that potentially denotes a higher-order argument was replaced by the name of the declaration that was introduced by lift (cf. Definition 5.14).

2. Partially applied functions are not covered by the static semantics of concrete programs (cf. Section 3.2.2), i.e., a higher-order argument may not be a partially applied function.

Because of these reasons, all we have to do is to introduce a new instance for each application of a higher-order function where the higher-order parameter is replaced by the name that the higher-order argument is bound to.

Definition 5.18 For all $n \in \mathbb{N}$, $\mathsf{instantiate}_{\textsc{Decl}} : \textsc{Decl}_2 \times \textsc{Var} \rightarrow \textsc{Decl}_2$ instantiates a higher-order function bound in an extended declaration

$$f \texttt{ = } \backslash\, v_1\ v_2 \ldots v_n \texttt{ -> } e \in \textsc{Decl}_2$$

by a higher-order argument bound to variable $h \in \textsc{Var}$:

$$\begin{gathered}\mathsf{instantiate}_{\textsc{Decl}}(f \texttt{ = } \backslash\, v_1\ v_2 \ldots v_n \texttt{ -> } e, h) \coloneqq \\ f' \texttt{ = } \backslash\, v_2 \ldots v_n \texttt{ -> } e'\end{gathered}$$

where

1. $v_1 \in \textsc{Var}$ denotes the single higher-order parameter of f,
2. $f' \in \textsc{Var}$ denotes a fresh name for the instantiated declaration, and
3. $e' \in \textsc{Exp}_2$ equals e, but with all occurrences of v_1 replaced by h.

Note that $\mathsf{instantiate}_{\textsc{Decl}}$ merely performs a variable substitution.

An extended concrete program in $\textsc{Prog}_1$ that features no higher-order functions can now be obtained by repeatedly applying $\mathsf{instantiate}_{\textsc{Decl}}$ for all applications of higher-order functions.

Definition 5.19 $\mathsf{instantiate} : \textsc{Prog}_2 \rightarrow \textsc{Prog}_1$ applies $\mathsf{instantiate}_{\textsc{Decl}}$ to each application of a higher-order function in an extended concrete program $c \in \textsc{Prog}_2$ and is defined by:

```
instantiate(c) =
  if (c contains application f e1 e2 ... en ∈ Exp2 where
        1. e1 ∈ Exp2 is a higher-order argument, and
        2. f ∈ Var is bound by declaration d in c)
  then
    d' ← instantiate_Decl(d, e1)
    c ← replace application f e1 e2 ... en by f' e2 ... en in c
          where f' ∈ Var denotes the name of declaration d'
    c ← add declaration d' to c
    return instantiate(c)
  else
    c ← delete all declarations of higher-order functions in c
    return c
```

The algorithm illustrated in Definition 5.19 does not terminate for recursively defined higher-order functions. This issue can be resolved by memoization: if a declaration $d \in \textsc{Decl}$ needs to be instantiated by some higher-order argument $h \in \textsc{Var}$, a new declaration d' is only generated if d has not been already instantiated by h.

We give an example for instantiating higher-order functions using instantiate.

Example 5.20 The function `mapMaybe` is applied to a higher-order argument `not`:

```
data Maybe a = Nothing | Just a
data Bool = False | True

constraint = \p u -> ... mapMaybe not u ...

mapMaybe = \f m -> case m of Nothing -> Nothing
                             Just a -> Just (f a)

not = \a -> case a of False -> True
                      True -> False
```

The algorithm illustrated in Definition 5.19 introduces a new instance for `mapMaybe` and gives the following extended concrete program of kind 1.

```
data Maybe a = Nothing | Just a
data Bool = False | True

constraint = \p u -> ... mapMaybe_1 u ...

mapMaybe_1 = \m -> case m of Nothing -> Nothing
                             Just a -> Just (not a)

not = \a -> case a of False -> True
                      True -> False
```

Restrictions The algorithm illustrated in Definition 5.19 only enables the usage of higher-order functions in some restricted cases. For example, it does not handle

1. functions that return a function as resulting value, nor

2. higher-order constructor arguments.

There are more sophisticated algorithms that enable the usage of higher-order functions even in these cases [59]. But as CO^4 is not able to encode functions as abstract values, these algorithms are not applied here.

5.4 Partial Functions

Concrete programs as they have been specified in Section 3.2 only consist of total functions, i.e., functions that are defined for all elements of their respective domain. In order to work around the lack of partial functions, one may introduce a data type that consists of an additional constructor that explicitly denotes an undefined value. In Example 5.21, we show an application of this work-around.

Example 5.21 In the following concrete program, the type `Optional a` contains a constructor `Undefined` that models the non-existence of a value. For example, the partial function `f` returns `Undefined` if it is applied to an argument for which `f` is not defined.

```
data Optional a = Undefined | Defined a
data Bool = False | True
data T = T1 Bool | T2 | T3

f :: T -> Optional Bool
f = \t -> case t of T1 b -> Defined b
                    T2   -> Defined True
                    T3   -> Undefined

g :: Bool -> Bool
g = \b -> case b of False -> True
                    True -> False

constraint = ...
```

Using a type like `Optional` is perfectly fine for modeling partial functions, but it comes with a drawback: it complicates the composition of functions. For example, the result of `f` cannot be directly passed to `g` because `f` results in a value of type `Optional Bool` whereas `g` expects a value of type `Bool`.

We can fix this issue by wrapping all values in a concrete program by `Optional` and adding an additional case distinction on `Optional` before each case distinction in the original program:

```
data Optional a = Undefined | Defined a
data Bool = False | True
data T = T1 (Optional Bool) | T2 | T3

f :: Optional T -> Optional Bool
f = \ot -> case ot of Undefined -> Undefined
                      Defined t -> case t of
                        T1 b -> b
                        T2 -> Defined True
                        T3 -> Undefined
```

```

g :: Optional Bool -> Optional Bool
g = \ob -> case ob of Undefined -> Undefined
                      Defined b -> case b of
                        False -> Defined True
                        True  -> Defined False

constraint = ...
```

Now the result of `f` can be directly passed to `g`.

The idea of modeling exceptions using a special type like `Optional` is well-known [69]. But it comes with an obvious flaw: because we now deal with potentially undefined values, the code is much more verbose and tedious to write. Thus, extended concrete programs of kind 1 feature a distinct identifier `undefined` to denote partial functions. Example 5.22 illustrates this for the concrete program from Example 5.21.

Example 5.22 The following concrete program shows the concrete program from Example 5.21 specified as extended concrete program of kind 1:

```
data Bool = False | True
data T = T1 Bool | T2 | T3

f :: T -> Bool
f = \t -> case t of T1 b -> b
                    T2   -> True
                    T3   -> undefined

g :: Bool -> Bool
g = \b -> case b of False -> True
                    True -> False

constraint = ...
```

Evaluating `undefined` in an extended concrete program of kind 1 results in the value $\bot$. That is because `undefined` may not be bound to a value (cf. Definition 5.3), therefore, $\mathsf{concrete\text{-}value}_{\textsc{Exp}}$ gives $\bot$ when evaluating `undefined` (cf. Definition 3.60).

In the following, we specify a reduction of extended concrete programs of kind 1 to concrete programs $\textsc{Prog}$. To do so, we incorporate the main ideas of the work-around in Example 5.21. Function $\mathsf{undef\text{-}values}_{\textsc{Exp}} : \textsc{Exp}_1 \to \textsc{Exp}$ maps concrete expressions of kind 1 to concrete expressions on possibly undefined values `Optional` so that

1. each call to `undefined` is replaced by a constructor call to `Undefined`,

2. each call to a constructor is wrapped by an application of the constructor `Defined`, and

3. each case distinction is wrapped by a case distinction on `Optional`.

Definition 5.23 $\mathsf{undef\text{-}values}_{\textsc{Exp}} : \textsc{Exp}_1 \rightarrow \textsc{Exp}$ maps a concrete expression $e \in \textsc{Exp}_1$ of kind 1 to a concrete expression in $\textsc{Exp}$ and is defined by:

$$
\mathsf{undef\text{-}values}_{\textsc{Exp}}(e) :=
\begin{cases}
\texttt{Undefined} & \text{if } e \in \textsc{Var} \text{ and } e = \texttt{undefined} \\
e & \text{if } e \in \textsc{Var} \text{ and } e \neq \texttt{undefined} \\
\texttt{Defined}\ e & \text{if } e \in \textsc{Con} \\
\begin{array}{l} \texttt{Defined}\ (C\ \ [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(e_1)]\!] \\ \qquad \ldots \\ \qquad [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(e_n)]\!]) \end{array} & \begin{array}{l} \text{if } e \text{ is an application } C\ e_1 \ldots e_n \\ \text{with } C \in \textsc{Con} \text{ and } n \in \mathbb{N} \end{array} \\
\begin{array}{l} f\ \ [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(a_1)]\!] \\ \quad \ldots \\ \quad [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(a_n)]\!] \end{array} & \begin{array}{l} \text{if } e \text{ is an application } f\ a_1 \ldots a_n \\ \text{with } f \in \textsc{Var} \text{ and } n \in \mathbb{N}_{>0} \end{array} \\
\begin{array}{l} \texttt{case}\ d\ \texttt{of} \\ \quad \texttt{Undefined -> Undefined} \\ \quad \texttt{Defined}\ d'\ \texttt{->}\ \texttt{case}\ d'\ \texttt{of} \\ \qquad p_1\ \texttt{->}\ [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(e_1)]\!] \\ \qquad \ldots \\ \qquad p_n\ \texttt{->}\ [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(e_n)]\!] \end{array} & \begin{array}{l} \text{if } e \text{ is a case distinction} \\ \texttt{case}\ d\ \texttt{of}\ p_1\ \texttt{->}\ e_1 \\ \qquad\qquad \ldots \\ \qquad\qquad p_n\ \texttt{->}\ e_n \\ \text{with } n \in \mathbb{N}_{>0}\text{, and } d' \in \textsc{Var} \\ \text{denotes a fresh name} \end{array} \\
\begin{array}{l} \texttt{let}\ \ x_1 = [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(a_1)]\!] \\ \qquad \ldots \\ \qquad x_n = [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(a_n)]\!] \\ \texttt{in}\ [\![\mathsf{undef\text{-}values}_{\textsc{Exp}}(e')]\!] \end{array} & \begin{array}{l} \text{if } e \text{ is a local binding} \\ \texttt{let}\ \ x_1 = a_1 \\ \qquad \ldots \\ \qquad x_n = a_n \\ \texttt{in}\ e' \\ \text{with } n \in \mathbb{N}_{>0} \end{array}
\end{cases}
$$

Transforming expressions with the function $\mathsf{undef\text{-}values}_{\textsc{Exp}}$ changes their type. Thus, types that appear explicitly in a concrete program, e.g., in type declarations and type signatures, need to be transformed accordingly.

Definition 5.24 $\mathsf{undef\text{-}values}_{\textsc{Type}} : \textsc{Type} \rightarrow \textsc{Type}$ prepends every type constructor in a given type $T \in \textsc{Type}$ with the type constructor `Optional` and is defined by:

$$\mathsf{undef\text{-}values}_{\textsc{Type}}(T) :=$$

$$\begin{cases} T & \text{if } T \in \textsc{TypeVar} \\ [\![\mathsf{undef\text{-}values}_{\textsc{Type}}(T_1)]\!] \texttt{ -> } [\![\mathsf{undef\text{-}values}_{\textsc{Type}}(T_2)]\!] & \text{if } T = T_1 \texttt{ -> } T_2 \\ \texttt{Optional } (C\ [\![\mathsf{undef\text{-}values}_{\textsc{Type}}(T_1)]\!] & \text{if } T = C\ T_1 \dots T_n \\ \qquad\qquad \dots & \text{with } C \in \textsc{TypeCon} \\ \qquad\qquad [\![\mathsf{undef\text{-}values}_{\textsc{Type}}(T_n)]\!]) & \text{and } n \in \mathbb{N} \end{cases}$$

In order to transform type schemes as well, we introduce $\mathsf{undef\text{-}values}_{\textsc{TypeScheme}}$ as a lifted form of $\mathsf{undef\text{-}values}_{\textsc{Type}}$.

Definition 5.25 $\mathsf{undef\text{-}values}_{\textsc{TypeScheme}} : \textsc{TypeScheme} \to \textsc{TypeScheme}$ transforms the type in type scheme $S \in \textsc{TypeScheme}$ using $\mathsf{undef\text{-}values}_{\textsc{Type}}$ and is defined by:

$$\mathsf{undef\text{-}values}_{\textsc{TypeScheme}}(S) :=$$

$$\begin{cases} [\![\mathsf{undef\text{-}values}_{\textsc{Type}}(S)]\!] & \text{if } S \in \textsc{Type} \\ \forall\ v_1 \dots v_n : [\![\mathsf{undef\text{-}values}_{\textsc{Type}}(T)]\!] & \text{if } S = \forall\ v_1 \dots v_n : T \text{ for } n \in \mathbb{N}_{>0} \end{cases}$$

After applying $\mathsf{undef\text{-}values}_{\textsc{Exp}}$, $\mathsf{undef\text{-}values}_{\textsc{Type}}$, and $\mathsf{undef\text{-}values}_{\textsc{TypeScheme}}$ to all expressions, types, and type schemes, respectively, the resulting program is technically no concrete program according to Definition 3.47. That is because the introduction of possibly undefined values changes the type of function `constraint` from P `->` U `-> Bool` for some types $P, U \in \textsc{Type}_0$ to

$$[\![\mathsf{undef\text{-}values}_{\textsc{Type}}(P)]\!] \texttt{ -> } [\![\mathsf{undef\text{-}values}_{\textsc{Type}}(U)]\!] \texttt{ -> Optional Bool}$$

As Definition 3.47 requires `constraint` to return a value of type `Bool`, we need to fix `constraint` after applying $\mathsf{undef\text{-}values}_{\textsc{Exp}}$, $\mathsf{undef\text{-}values}_{\textsc{Type}}$, and $\mathsf{undef\text{-}values}_{\textsc{TypeScheme}}$. This fix merely consists of an additional case distinction that ensures that the fixed version of `constraint` gives the Boolean value `False` if the unfixed version of `constraint` evaluates to `Undefined`.

Definition 5.26 $\mathsf{fix\text{-}constraint} : \textsc{Prog}_1 \to \textsc{Prog}$ rewrites declaration

```
constraint = \p u -> e
```

in an extended concrete program of kind 1 to

```
constraint = \p u -> case e of
      Undefined -> False
      Defined b -> b
```

where $b \in \textsc{Var}$ denotes a fresh variable and expression e does not contain partial functions, i.e., $e \in \textsc{Exp}$.

Finally, we can specify an algorithm that transforms an extended concrete program of kind 1 to a concrete program.

Definition 5.27 undef-values : $\textsc{Prog}_1 \rightarrow \textsc{Prog}$ transforms an extended concrete program $c \in \textsc{Prog}_1$ to a concrete program in $\textsc{Prog}$ and is defined by:

```
undef-values(c) =
  add type data Optional a = Undefined | Defined a to c
  forall type signatures v :: s ∈ Decl_1 in c
    replace s by undef-values_TypeScheme(s)
  forall expressions e ∈ Exp_1 in c
    replace e by undef-values_Exp(e)
  forall type declarations d ∈ TypeDecl in c
    forall constructor argument types T ∈ Type in d
      replace T by undef-values_Type(T)
  return fix-constraint(c)
```

We give an example of applying undef-values to an extended concrete program of kind 1.

Example 5.28 Assume the following concrete program $c \in \textsc{Prog}_1$:

```
data Bool = False | True
data Unit = Unit
data T = T1 Bool | T2 | T3

constraint :: Unit -> T -> Bool
constraint = \p u -> f u

f :: T -> Bool
f = \u -> case u of T1 b -> b
                    T2   -> True
                    T3   -> undefined
```

undef-values(c) gives the following total program which explicitly deals with `Optional` values:

```
data Optional a = Undefined | Defined a
data Bool = False | True
data Unit = Unit
data T = T1 (Optional Bool) | T2 | T3

constraint :: Optional Unit -> Optional T -> Bool
constraint = \p u -> case (f u) of
                       Undefined -> False
                       Defined b -> b

```

```
f :: Optional T -> Optional Bool
f = \u -> case u of Undefined  -> Undefined
                    Defined u' -> case u' of
                      T1 b -> b
                      T2   -> Defined True
                      T3   -> Undefined
```

For concrete programs $c \in \textsc{Prog}$ that do not contain partial functions, we want the semantics of c to be identical to $\textsf{undef-values}(c)$.

Lemma 5.29 For two types $P, U \in \textsc{Type}_0$, the following equality holds for all concrete programs $c \in \textsc{Prog}_{PU}$ without partial functions:

$$\begin{aligned}&\forall (p,u) \in \mathbb{C}_P \times \mathbb{C}_U : \; p \neq \bot \wedge u \neq \bot \implies \textsf{concrete-value}(c)(p,u) \\ &\quad = \textsf{concrete-value}(\textsf{undef-values}(c))(\textsf{undef-values}_{\mathbb{C}}(p), \textsf{undef-values}_{\mathbb{C}}(u))\end{aligned}$$

where $\textsf{undef-values}_{\mathbb{C}} : \mathbb{C} \to \mathbb{C}$ maps a concrete value $v \in \mathbb{C}$ to a possibly undefined value:

$$\textsf{undef-values}_{\mathbb{C}}(v) :=$$

$$\begin{cases} \texttt{Undefined} & \text{if } v = \bot \\ \texttt{Defined}\ (C\ \textsf{undef-values}_{\mathbb{C}}(v_1) & \text{if } v = C\ v_1 \dots v_n \\ \qquad\quad \dots & \text{with } C \in \textsc{Con} \text{ and } n \in \mathbb{N} \\ \qquad\quad \textsf{undef-values}_{\mathbb{C}}(v_n)) & \end{cases}$$

Proof In order to prove Lemma 5.29 for a particular concrete program $c \in \textsc{Prog}$, we show that the dynamic semantics of each expression $e \in \textsc{Exp}$ does not change:

$$\begin{aligned}&\forall E \in \mathbb{C}^{\textsc{Var}} : \textsf{undef-values}_{\mathbb{C}}(\textsf{concrete-value}_{\textsc{Exp}}(c, E, e)) = \\ &\textsf{concrete-value}_{\textsc{Exp}}(\textsf{undef-values}(c), E', \textsf{undef-values}_{\textsc{Exp}}(e))\end{aligned} \tag{5.30}$$

where

$$E' = \{(n, \textsf{undef-values}_{\mathbb{C}}(v)) \mid (n,v) \in E\}$$

Informally, if the expression e in the program c evaluates to the value $v \in \mathbb{C}$ in context of an environment $E \in \mathbb{C}^{\textsc{Var}}$, then the expression $\textsf{undef-values}_{\textsc{Exp}}(e)$ in the program $\textsf{undef-values}(c)$ evaluates to the value $\textsf{undef-values}_{\mathbb{C}}(v)$ in context of the environment E'. We show this by induction over concrete expressions:

1. If $e \in \textsc{Var}$, then $e \neq \texttt{undefined}$. The left-hand side of (5.30) gives

$$\textsf{undef-values}_{\mathbb{C}}(\textsf{concrete-value}_{\textsc{Exp}}(c, E, e)) = \textsf{undef-values}_{\mathbb{C}}(E(e))$$

which equals its right-hand side

$$\begin{aligned}&\textsf{concrete-value}_{\textsc{Exp}}(\textsf{undef-values}(c), E', \textsf{undef-values}_{\textsc{Exp}}(e)) \\ =\ &\textsf{concrete-value}_{\textsc{Exp}}(\textsf{undef-values}(c), E', e) \\ =\ &E'(e)\end{aligned}$$

by definition of E'.

2. If $e \in \textsc{Con}$, then the left-hand side of (5.30) gives

$$\begin{aligned}\mathsf{undef\text{-}values}_{\mathbb{C}}(\mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, E, e)) &= \mathsf{undef\text{-}values}_{\mathbb{C}}(e)\\ &= \texttt{Defined}\ e\end{aligned}$$

which equals its right-hand side

$$\begin{aligned}&\mathsf{concrete\text{-}value}_{\textsc{Exp}}(\mathsf{undef\text{-}values}(c), E', \mathsf{undef\text{-}values}_{\textsc{Exp}}(e))\\ =\ &\mathsf{concrete\text{-}value}_{\textsc{Exp}}(\mathsf{undef\text{-}values}(c), E', \texttt{Defined}\ e)\\ =\ &\texttt{Defined}\ e\end{aligned}$$

3. If e is an application $C\ e_1 \ldots e_n$ with $C \in \textsc{Con}$, then the left-hand side of (5.30) gives

$$\begin{aligned}&\mathsf{undef\text{-}values}_{\mathbb{C}}(\mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, E, e))\\ =\ &\mathsf{undef\text{-}values}_{\mathbb{C}}(C\ \ \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, E, e_1)\\ &\qquad\qquad\qquad\quad \ldots\\ &\qquad\qquad\qquad\quad \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, E, e_n))\\ =\ &\texttt{Defined}\ (C\ \ \mathsf{undef\text{-}values}_{\mathbb{C}}(\mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, E, e_1))\\ &\qquad\qquad\quad \ldots\\ &\qquad\qquad\quad \mathsf{undef\text{-}values}_{\mathbb{C}}(\mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, E, e_n)))\end{aligned}$$

which equals its right-hand side

$$\begin{aligned}&\mathsf{concrete\text{-}value}_{\textsc{Exp}}(\mathsf{undef\text{-}values}(c), E', \mathsf{undef\text{-}values}_{\textsc{Exp}}(e))\\ =\ &\mathsf{concrete\text{-}value}_{\textsc{Exp}}\ (\ \ \mathsf{undef\text{-}values}(c), E'\\ &\qquad\qquad\qquad\quad ,\ \texttt{Defined}\ (C\ \ \mathsf{undef\text{-}values}_{\textsc{Exp}}(e_1)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{undef\text{-}values}_{\textsc{Exp}}(e_n)))\\ =\ &\texttt{Defined}\ (C\ \ \mathsf{concrete\text{-}value}_{\textsc{Exp}}\ (\ \ \mathsf{undef\text{-}values}(c), E'\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad ,\ \mathsf{undef\text{-}values}_{\textsc{Exp}}(e_1))\\ &\qquad\qquad\quad \ldots\\ &\qquad\qquad\quad \mathsf{concrete\text{-}value}_{\textsc{Exp}}\ (\ \ \mathsf{undef\text{-}values}(c), E'\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad ,\ \mathsf{undef\text{-}values}_{\textsc{Exp}}(e_n)))\end{aligned}$$

by induction on the constructor arguments $e_1, \ldots, e_n$.

4. If e is a case distinction `case` d `of` $\ldots$ p `->` e' $\ldots$, then we assume that $v_d = \mathsf{concrete\text{-}value}_{\textsc{Exp}}(c, E, d)$ denotes the value of the discriminant $d \in \textsc{Exp}$. Note that $v_d \neq \bot$ because c contains no partial functions, $p \neq \bot$, and $u \neq \bot$. We furthermore assume that the pattern $p \in \textsc{Pat}$

matches the value v_d, i.e., $\mathsf{matches}\ (p, v_d)$ holds. According to the dynamic semantics of case distinctions (cf. Definition 3.60), the left-hand side of (5.30) evaluates to:

$$\begin{aligned} & \mathsf{undef\text{-}values}_{\mathbb{C}}(\mathsf{concrete\text{-}value}_{\mathrm{EXP}}(c, E, e)) \\ = \ & \mathsf{undef\text{-}values}_{\mathbb{C}}(\mathsf{concrete\text{-}value}_{\mathrm{EXP}}(c, E[\mathsf{bindMatch}(p, v_d)], e')) \end{aligned}$$

Note that $E[\mathsf{bindMatch}(p, v_d)]$ denotes the update of E by the assignment resulting from evaluating $\mathsf{bindMatch}(p, v_d)$ (cf. Definition A.18).

In the transformed program $\mathsf{undef\text{-}values}(c)$, the discriminant $\mathsf{undef\text{-}values}_{\mathrm{EXP}}(d)$ evaluates to

$$v'_d = \mathsf{concrete\text{-}value}_{\mathrm{EXP}}(\mathsf{undef\text{-}values}(c), E', \mathsf{undef\text{-}values}_{\mathrm{EXP}}(d))$$

where $\mathsf{undef\text{-}values}_{\mathbb{C}}(v_d) = v'_d$ holds by induction over d. Note that $\mathsf{rootsym}(v'_d) = \mathtt{Defined}$ because $v_d \neq \perp$. Thus, the right-hand side of (5.30) gives

$$\begin{aligned} & \mathsf{concrete\text{-}value}_{\mathrm{EXP}}(\mathsf{undef\text{-}values}(c), E', \mathsf{undef\text{-}values}_{\mathrm{EXP}}(e)) \\ = \ & \mathsf{concrete\text{-}value}_{\mathrm{EXP}}(\mathsf{undef\text{-}values}(c)\ ,\ E'[\mathsf{bindMatch}(\mathtt{Defined}\ p, v'_d)] \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ,\ \mathsf{undef\text{-}values}_{\mathrm{EXP}}(e')) \end{aligned}$$

which equals its left-hand side by induction over e'.

5. As $\mathsf{undef\text{-}values}_{\mathrm{EXP}}$ does not change the structure of function applications and local bindings, Lemma 5.29 holds by induction over the involved subexpressions. ■

Note that Lemma 5.29 excludes the cases where $p = \perp$ or $u = \perp$ when evaluating the concrete program c. The reason is that $\mathsf{concrete\text{-}value}(c)(p, u)$ may give $\perp$ in these situations while

$$\mathsf{concrete\text{-}value}(\mathsf{undef\text{-}values}(c))(\mathsf{undef\text{-}values}_{\mathbb{C}}(p), \mathsf{undef\text{-}values}_{\mathbb{C}}(u))$$

never evaluates to $\perp$.

Chapter 6

Optimization of Abstract Programs

Recall that evaluating an abstract program which has been compiled from a concrete program $c \in \textsc{Prog}$ gives an abstract value containing a single propositional formula $f \in \mathrm{F}$. A solution for c can be decoded from a satisfying assignment for f. As described in Section 4.3, an external SAT solver is applied in order to find such an assignment.

For most non-trivial concrete programs, the runtime of the SAT solver determines the runtime of CO^4 as a whole. Therefore, it is crucial to minimize the solver's runtime. From the perspective of CO^4, we do not have any insight in the design and the inner workings of the applied SAT solver, thus we cannot perform any specific optimizations to reduce its runtime.

Nonetheless, this chapter illustrates general techniques which might reduce the solver's runtime without giving any proof that they actually work for a given SAT solver. These techniques are motivated by the idea that a solver's runtime depends especially on the size of the formula to be solved, where the definition of the size of a formula is vague as well. Thus, we aim to reduce the size of the formulas that are generated by evaluating an abstract program. Firstly, Section 6.1 gives an overview of the profiling capabilities of CO^4, which are useful for gathering statistics about the generated propositional formula. Using these statistics we can evaluate the benefits of the optimization strategies introduced in the subsequent sections.

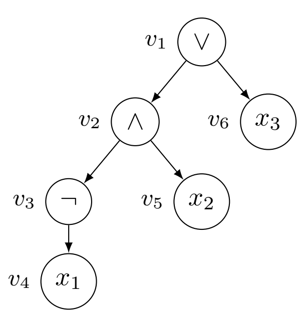
Section 6.2 illustrates the memoization of function applications. Memoization is an optimization strategy that stores the results of function applications in order to reuse them if a function is repeatedly applied to the same arguments. Furthermore, Section 6.3 shows a more efficient encoding of natural numbers in abstract programs. This optimization is reasonable because natural numbers are often included in the domain of discourse or the parameter domain of real-world constraints. Therefore, it is important to provide a more efficient encoding than the unary encoding illustrated in Example 3.9. We conclude this chapter by a brief overview of further optimizations.

6.1 Profiling in CO^4

CO^4 offers basic profiling features that are useful for inspecting not only the size of the generated propositional formula but also how each section of the concrete program contributes to the formula. Firstly, we give definitions for the different size outputs provided by CO^4.

> **Definition 6.1** The *number of variables* of a propositional formula $f \in \mathrm{F}$ equals $|\,\mathsf{var}(\mathsf{tseitin}(f))|$ where $\mathsf{tseitin} : \mathrm{F} \to \mathrm{CNF}$ maps a propositional formula to an equisatisfiable conjunctive normal form (cf. Definition B.13).

Note that the number of variables $|\,\mathsf{var}(\mathsf{tseitin}(f))|$ equals the number of vertices in the directed acyclic graph (DAG) that represents a propositional formula $f \in \mathrm{F}$ in the present implementation of CO^4 (cf. Section 4.3.3). That is because $\mathsf{tseitin}$ generates a fresh propositional variable for each subformula of f that is no variable. Figure 6.2 shows an example of this relation between the number of vertices in the DAG representation and the number of variables in conjunctive normal form.



$$\mathsf{tseitin}(f) =$$

$$\left\{\begin{array}{l} \quad \{v_1\} \\ , \{v_2, x_3, \neg v_1\}, \{\neg v_2, v_1\}, \{\neg x_3, v_1\} \\ , \{\neg v_3, \neg x_2, v_2\}, \{v_3, \neg v_2\}, \{x_2, \neg v_2\} \\ , \{v_3, x_1\}, \{\neg v_3, \neg x_1\} \end{array}\right\}$$

where

$$\mathsf{var}(\mathsf{tseitin}(f)) = \{v_1, v_2, v_3, x_1, x_2, x_3\}$$

Figure 6.2: DAG representation of $f = (\neg x_1 \wedge x_2) \vee x_3$ and its conjunctive normal form $\mathsf{tseitin}(f)$. Note that the number of vertices in the DAG equals the number of variables in $\mathsf{tseitin}(f)$.

Besides the number of variables, CO^4 also gives the number of clauses and the number of literals for a propositional formula.

> **Definition 6.3** The *number of clauses* of a propositional formula $f \in \mathrm{F}$ equals the number of clauses in $\mathsf{tseitin}(f)$ (cf. Definition B.9) and the *num-*

ber of literals of f equals the sum of literal occurrences over all clauses in $\mathsf{tseitin}(f)$.

Example 6.4 The number of clauses for the formula $(\neg x_1 \wedge x_2) \vee x_3$ in Figure 6.2 is 9 and the number of literals is 19.

Additionally, CO^4 gives the clause density of a propositional formula.

Definition 6.5 The *clause density* of a propositional formula $f \in \mathrm{F}$ denotes the ratio of clauses to variables.

For a given propositional formula $f \in \mathrm{F}$, the clause density of f is often considered to be an indicator for the hardness of finding a satisfying assignment for f or proving f to be unsatisfiable [58].

The number of variables, clauses, and literals of a propositional formula only give a narrow overview of the characteristics of a particular constraint. CO^4 also provides more fine-grained profiling information for function applications and case distinctions.

Recall that in general, evaluating compiled case distinctions in an abstract program generates new subformulas because of the merge operation on all evaluated branches (cf. Definition 4.49). In order to inspect the amount that each case distinction contributes to the final formula, CO^4 tracks each evaluation of a compiled case distinction. There are two important numbers for each compiled case distinction: the number of *total evaluations* and the number of *constant evaluations*. The number of constant evaluations is especially useful to know as it denotes how often a particular compiled case distinction could be evaluated without generating new subformulas (cf. Lemma 4.43). Knowing these numbers is useful for inspecting which case distinction causes the most merge operations.

In Example 6.6, we illustrate CO^4's profiling features for a trivial concrete program. Appendix C.5 shows the actual profiling log of CO^4 for the more complex Example 7.11.

Example 6.6 Assume a constraint $c : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ where

$$c(p, u) = \begin{cases} \mathsf{False} & \text{if } p = \mathsf{False} \\ \neg u & \text{if } p = \mathsf{True} \end{cases}$$

c is specified by the following concrete program:

```
data Bool = False | True

constraint = \p u -> case p of
  False -> False
  True -> case u of False -> True
                    True -> False
```

	$\mathtt{p} = \mathtt{False}$	$\mathtt{p} = \mathtt{True}$
#variables	0	2
#clauses	0	3
#literals	0	5
clause density	-	1.5
#total evaluations of $d_{\mathtt{p}}$	1	1
#constant evaluations of $d_{\mathtt{p}}$	1	1
#total evaluations of $d_{\mathtt{u}}$	0	1
#constant evaluations of $d_{\mathtt{u}}$	0	0

Table 6.7: Profiling information for solving constraint c.

There are two case distinctions which we denote by $d_{\mathtt{p}}$ and $d_{\mathtt{u}}$: $d_{\mathtt{p}}$ matches on `p`, $d_{\mathtt{u}}$ matches on `u` if $\mathtt{p} = \mathtt{True}$. Table 6.7 shows the profiling information given by CO^4 for both values of parameter `p`.

For $\mathtt{p} = \mathtt{False}$, CO^4 generates a propositional formula without any variables because there is no non-constant evaluation of a case distinction. This formula evaluates to False, therefore, constraint c is unsatisfiable for parameter False.

For $\mathtt{p} = \mathtt{True}$, CO^4 generates a propositional formula $\neg f_u$ where $f_u \in \mathrm{V}$ denotes the single flag in the abstract value that represents value `u` (cf. Example 4.42). The values in Table 6.7 correspond to Definition 6.1 because:

$$\mathsf{tseitin}(\neg f_u) = \big\{\{v\}, \{f_u, v\}, \{\neg f_u, \neg v\}\big\}$$

where $v = \mathsf{fresh}(\neg f_u)$ denotes a fresh propositional variable (cf. Definition B.13).

6.2 Memoization of Function Applications

Memoization refers to a general optimization technique where sub-results of an algorithm are stored so that they only need to be computed once and may be reused on future occasions [57]. Memoization is often applied in order to improve the runtime and/or space complexity of an algorithm. In the scope of this thesis, we aim to reduce the size of the generated propositional formula, which often leads to shorter solver runtimes. Thus, this chapter shows how memoization can be applied to the evaluation of abstract programs.

Memoization can be applied in various ways to the evaluation of abstract programs. As illustrated in the previous section, evaluating compiled case distinctions on non-constant discriminants is expensive because it leads to new propositional variables and clauses in the generated formula. Thus, compiled case distinctions are obvious candidates for memoization. As the value of a case distinction not only depends on the value of its discriminant but also on the value of all free variables that appear in the case distinction, all these free variables have to be taken into account when implementing memoization on case distinctions. Memoization of function applications is a simpler choice because the value of a function application in an abstract program only depends on the value of its arguments and on the definition of the applied function.

Example 6.8 motivates why memoizing function applications is reasonable.

Example 6.8 Assume the following constraint $c : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$

$$c(p, u) = \begin{cases} \mathsf{True} & \text{if } p = \mathsf{fib}(u) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

with $\mathsf{fib} : \mathbb{N} \to \mathbb{N}$ being a mapping from naturals to Fibonacci numbers so that $\mathsf{fib}(n)$ gives the n-th Fibonacci number for $n \in \mathbb{N}$:

$$\mathsf{fib}(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathsf{fib}(n-2) + \mathsf{fib}(n-1) & \text{otherwise} \end{cases}$$

The following concrete program specifies constraint c and contains a straightforward implementation of fib.

```
data Nat = Z | S Nat

constraint p n = eq p (fib n)

fib = \x -> case x of
  Z -> Z
  S x’ -> case x’ of
    Z -> S Z
    S x’’ -> let f1 = fib x’
                 f2 = fib x’’
             in
               add f1 f2

eq = \x y -> case x of
  Z -> case y of Z -> True
                 S y’ -> False
  S x’ -> case y of Z -> False
                    S y’ -> eq x’ y’

add = \x y -> case x of
  Z -> y
  S x’ -> S (add x’ y)
```

This implementation leads to identical evaluations of `fib` for certain arguments, e.g., when evaluating `fib (S(S(S(S Z))))`:

$$\texttt{fib (S(S(S(S Z))))}\begin{cases}\texttt{fib (S(S(S Z)))}\begin{cases}\texttt{fib (S(S Z))}\begin{cases}\texttt{fib (S Z)}\\ \texttt{fib Z}\end{cases}\\ \texttt{fib (S Z)}\end{cases}\\ \texttt{fib (S(S Z))}\begin{cases}\texttt{fib (S Z)}\\ \texttt{fib Z}\end{cases}\end{cases}$$

`fib (S(S Z))`, `fib (S Z)`, and `fib Z` are evaluated multiple times. Storing the results of these applications would allow them to be shared among subsequent applications of `fib` to identical arguments.

Example 6.8 illustrates that memoizing function applications in concrete programs saves evaluating applications that already have been evaluated before. In the following, we apply memoization of function applications to the evaluation of abstract programs by introducing a global cache that stores the results of function applications. This is a straightforward extension of the abstract evaluation as it has been introduced in Section 3.3.3.

Definition 6.9 The result $r \in \mathbb{A}$ of evaluating a function application $f\ a_1 \dots a_n$ in an abstract program $c \in \textsc{Prog}_{\mathbb{A}}$ in the context of an environment $E_{\mathbb{A}} \in \mathbb{A}^{\textsc{Var}}$ and a cache $\gamma : \textsc{Var} \times \mathbb{A}^* \nrightarrow \mathbb{A}$ is defined by the following inference rules:

$$\frac{(f, (a_1, \dots, a_n)) \in \mathrm{dom}(\gamma)}{r = \gamma(f, (a_1, \dots, a_n))}$$

$$\frac{(f, (a_1, \dots, a_n)) \notin \mathrm{dom}(\gamma)}{r = \textsf{abstract-value}_{\textsc{Exp}}(c, E_{\mathbb{A}}, f\ a_1 \dots a_n) \quad \gamma(f, (a_1, \dots, a_n)) = r}$$

Evaluation of other abstract expressions remains unchanged. Evaluation of abstract programs starts with an empty cache, i.e., $\gamma = \varnothing$, but remains unchanged otherwise.

We show how memoization decreases the size of the generated propositional formula for the concrete program in Example 6.8.

Example 6.10 In the concrete program of Example 6.8, the domain of discourse `Nat` is specified using a recursive type, thus, we cannot compute a complete abstract value for the solution. Instead we define an incomplete abstract value $\textsf{abstract-nat}(n)$ that represents the natural numbers less or

	without memoization	with memoization
# variables	29	24
# clauses	72	60
# literals	182	154
clause density	2.48	2.5
# cache hits	0	2

Table 6.11: Profiling information for finding a solution for the concrete program in Example 6.8.

equal $n \in \mathbb{N}$:

$$\mathsf{abstract\text{-}nat}(n) :=$$
$$\begin{cases} \mathsf{encode}_{\mathtt{Nat}}(\mathtt{Z}) & \text{if } n = 0 \\ ((f), (\mathsf{abstract\text{-}nat}(n-1))) & \text{if } n > 0 \text{ and } f \in \mathrm{V} \text{ is a fresh} \\ & \text{propositional variable} \end{cases}$$

See Section 4.1.5 for more information on incomplete abstract values.

Table 6.11 shows some profiling information for finding a solution for the concrete program in Example 6.8 with CO^4. In this example, we fix parameter $\mathtt{p} \in \mathbb{C}_{\mathtt{Nat}}$ to $\mathtt{p} = \mathtt{S(S(S\ Z))}$. For the designated solution $\mathtt{u}$, we use an abstract value $\mathsf{abstract\text{-}nat}(4)$.

When solving Example 6.8 without memoization, CO^4 generates a propositional formula in conjunctive normal form with 29 variables, 72 clauses, and a total of 182 literals. Solving with memoization reduces the size of the generated formula to a total of 154 literals. For such small formulas, it rarely makes any difference in terms of SAT solver runtimes, but this changes dramatically for more complex constraints.

Note the two cache hits while solving with memoization: these hits indicate that two function applications could be evaluated by querying the cache because they have already been evaluated before.

Section 7.1.2 illustrates a more complex use-case that benefits from memoization.

6.3 Built-In Natural Numbers

Example 3.9 and Example 6.8 use natural numbers in the domain of discourse. So far, we modeled natural numbers as Peano numbers, which essentially results in a unary encoding because an incomplete abstract value representing the natural numbers less or equal $n \in \mathbb{N}$ contains a total of n flags (cf. function

$\mathsf{abstract\text{-}nat} : \mathbb{N} \to \mathbb{A}$ in Example 6.10). A unary encoding of natural numbers is easy to implement and may even lead to better solver runtimes for certain constraints in comparison to a binary encoding [17]. On the other hand, a binary encoding is superior in terms of space complexity, which often results in better solver runtimes for other constraints. Because a binary encoding is more complex to implement, CO^4 provides a built-in type `Nat` that represents binary encoded natural numbers, as well as functions that operate on values of type `Nat` (cf. Table 6.12).

Name	Type	Semantics
`eqNat`	`Nat -> Nat -> Bool`	Equality check
`gtNat`	`Nat -> Nat -> Bool`	Greater-than check
`plusNat`	`Nat -> Nat -> Nat`	Addition
`timesNat`	`Nat -> Nat -> Nat`	Multiplication

Table 6.12: Some predefined functions on natural numbers in CO^4.

Binary encoded natural numbers integrate seamlessly into the compilation pipeline because they are compiled to abstract values just as ordinary concrete values.

Definition 6.13 $\mathsf{encode}_{\mathtt{Nat}} : \mathbb{N} \to \mathbb{A}$ maps a natural number $n \in \mathbb{N}$ to an abstract value containing $w = \lceil \log_2(n) \rceil$ flags that represent n in binary:

$$\mathsf{encode}_{\mathtt{Nat}}(n) := (\mathsf{numeric}^{-}_{2^w}(n+1), ())$$

Note that we have abused the function $\mathsf{numeric}^{-}$ to generate a sequence of w flags that represent the natural number n (cf. Definition 4.20). Note furthermore that $\mathsf{numeric}^{-}_{2^w}$ is defined only for the values in $\{1 \dots 2^w\}$, which is why we need to add one when converting n into its binary representation.

For $n \in \mathbb{N}$, decoding the abstract value $\mathsf{encode}_{\mathtt{Nat}}(n) \in \mathbb{A}$ simply reduces to converting its flags back to a natural number in decimal.

Definition 6.14 $\mathsf{decode}_{\mathtt{Nat}} : \mathbb{B}^{\mathrm{V}} \times \mathbb{A} \to \mathbb{N}$ decodes an abstract value $a \in \mathbb{A}$ to a natural number in decimal with respect to an assignment $\sigma \in \mathbb{B}^{\mathrm{V}}$:

$$\mathsf{decode}_{\mathbb{N}}(\sigma, a) := \mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, a))$$

where $k = 2^{|\,\mathsf{flags}(a)|}$.

CO^4 provides built-in arithmetic functions for natural numbers encoded in binary (cf. Table 6.12). These functions directly operate on the flags of their operands and for that reason they are directly implemented in the solver. In the following, we give an example that illustrates the addition of two abstract values that each represent a binary encoded natural number.

Example 6.15 $+_{\mathbb{A}} : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ gives an abstract value $u +_{\mathbb{A}} v$ that represents the sum of two binary encoded naturals $u, v \in \mathbb{A}$ where $m \in \mathbb{N}_{>0}$

denotes the number of flags in u and v, i.e., $|\,\mathsf{flags}(u)| = |\,\mathsf{flags}(v)| = m$. $+_{\mathbb{A}}$ is defined by:

$$((u_1, \ldots, u_m), ()) +_{\mathbb{A}} ((v_1, \ldots, v_m), ()) \coloneqq \; ((r_1, \ldots, r_m), ())$$

where

$$\begin{aligned}
(r_m, c_m) &= \mathsf{half\text{-}add}(u_m, v_m) \\
(r_{m-1}, c_{m-1}) &= \mathsf{full\text{-}add}(u_{m-1}, v_{m-1}, c_m) \\
&\cdots \\
(r_1, c_1) &= \mathsf{full\text{-}add}(u_1, v_1, c_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{full\text{-}add}(u_i, v_i, c_i) &= (r'_2, c'_1 \vee c'_2) \text{ with } &(r'_1, c'_1) &= \mathsf{half\text{-}add}(u_i, v_i) \\
& &(r'_2, c'_2) &= \mathsf{half\text{-}add}(r'_1, c_i) \\
\mathsf{half\text{-}add}(u_i, v_i) &= (u_i \oplus v_i, u_i \wedge v_i)
\end{aligned}$$

$+_{\mathbb{A}}$ is implemented using a series of full adders where each full adder consists of two half adders. There are semantically equivalent implementations for $+_{\mathbb{A}}$ that use fewer logical connectives [50] but are more complex. Note that $+_{\mathbb{A}}$ ignores the final carry flag c_1, i.e., overflows are not detected. In the present implementation of CO^4, built-in functions on natural numbers are partial (cf. Section 5.4) so that an overflow induces an exceptional value.

In Example 6.16, we show a specification of the constraint from Example 3.9 using CO^4's built-in natural numbers.

Example 6.16 Recall the constraint $c : \mathbb{N} \times \mathbb{N}^2 \to \mathbb{B}$ from Example 3.9:

$$c(p, (a, b)) = \begin{cases} \mathsf{True} & \text{if } p = (a \cdot b) \wedge (a > 1) \wedge (b > 1) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

The following concrete program applies CO^4's built-in natural numbers and is a correct specification of c.

```
data Bool     = False | True
data Pair a b = Pair a b

constraint :: Nat -> Pair Nat Nat -> Bool
constraint = \p u -> case u of
  Pair a b ->  and2 (gtNat a)
               (and2 (gtNat b)
                     (eqNat p (timesNat a b)))

and2 :: Bool -> Bool -> Bool
and2 = \x y -> case x of
  False -> False
```

```
  True  -> y
```

We give an example that compares the unary encoding of naturals to an explicit binary encoding and to CO^4's built-in binary encoding.

Example 6.17 Assume the following constraint $c : \mathbb{N} \times \mathbb{N}^2 \to \mathbb{B}$:

$$c(p,(a,b)) = \begin{cases} \mathsf{True} & \text{if } p = a + b \\ \mathsf{False} & \text{otherwise} \end{cases}$$

Firstly, we specify c by representing natural numbers using a unary encoding:

```
data Bool = False | True
data Nat = Z | S Nat
data Pair a b = Pair a b

constraint :: Nat -> Pair Nat Nat -> Bool
constraint = \p u -> case u of
  Pair a b -> let ab = plus a b
              in
                eq p ab

plus :: Nat -> Nat -> Nat
plus = \x y -> case x of Z -> y
                         S x' -> S (plus x' y)

eq :: Nat -> Nat -> Bool
eq = \x y -> case x of
  Z -> case y of Z -> True
                 S y' -> False
  S x' -> case y of Z -> False
                    S y' -> eq x' y'
```

The following listing shows an excerpt of a concrete program that specifies c using an explicit binary encoding where each natural number is represented by a list of Booleans. Note that the head of the list denotes the most significant bit. The complete listing can be found in Appendix C.2.

```
data Bool = False | True
data List a = Nil | Cons a (List a)
data Pair a b = Pair a b

constraint :: List Bool -> Pair (List Bool) (List Bool)
           -> Bool
constraint = \p u -> case u of
  Pair x y -> case add x y of
```

```
    Pair sum carry -> and (eqNat sum p) (not carry)

add :: List Bool -> List Bool -> Pair (List Bool) Bool
add = \x y ->
  let add' pair accu = case pair of
        Pair u v -> case accu of
         Pair bits carry -> case fullAdder u v carry of
          Pair sum carry' -> Pair (Cons sum bits) carry'
  in
    foldr add' (Pair Nil False) (zip x y)

fullAdder :: Bool -> Bool -> Bool -> Pair Bool Bool
fullAdder = \x y carry -> case halfAdder x y of
  Pair sum1 carry1 -> case halfAdder sum1 carry of
    Pair sum2 carry2 -> Pair sum2 (or carry1 carry2)

halfAdder :: Bool -> Bool -> Pair Bool Bool
halfAdder = \x y -> Pair (xor x y) (and x y)
```

The following concrete program specifies c and encodes natural numbers using CO^4's built-in binary encoding.

```
data Bool = False | True
data Pair a b = Pair a b

constraint :: Nat -> Pair Nat Nat -> Bool
constraint = \p u -> case u of
  Pair a b -> eqNat p (plusNat a b)
```

Table 6.18 shows some profiling information for finding solutions for all three concrete programs with CO^4 and MiniSat (version 2.2) on a 3.2 GHz CPU. For all three programs, we fixed the parameter `p` so that it represents the natural number 1002 using the respective encoding. For the parameter $\mathtt{u} \in \mathbb{N}$, we generated an incomplete abstract value that represents a pair of naturals $(a, b) \in \mathbb{N} \times \mathbb{N}$ with $a < 1024$ and $b < 1024$ (cf. Section 4.1.5).

Even for a trivial example like this, binary encoding of natural numbers has a significant impact on the size of the generated propositional formula and the runtime of the SAT solver. The differences between both binary encodings are less significant. While CO^4's built-in encoding uses fewer variables and clauses, the clause density is higher. This could indicate that the resulting satisfiability problem for the explicit encoding is under-constrained, so that it might be solved more easily by SAT solvers [58]. But that is speculative and highly depends on the implementation of the solver and the respective constraint. As for this simple example, there is no difference in solver runtime for both binary encodings.

	unary encoding	explicit binary encoding	CO^4's binary encoding
# variables	1050603	111	43
# clauses	4193219	222	150
# literals	12576601	521	510
clause density	3.99	2.0	3.43
solver runtime	11 s	0.1 s	0.1 s

Table 6.18: Profiling information for finding a solution for constraint c using three different concrete programs.

6.4 Further Optimizations

The optimizations introduced in the previous sections are reasonable because they decrease the size of the generated propositional formula. There are several more optimization strategies. In the following, we briefly introduce two of them.

Merging Abstract Values Recall that case distinctions in a concrete program are compiled to an abstract expression where all branches are evaluated and eventually merged into a single resulting abstract value. As it has been shown in Section 6.1 and 6.2, evaluating compiled case distinctions is, in general, an expensive operation. To reduce their costs, we give an optimization for compiled case distinctions whose branches do not contain an equal number of flags.

In Definition 4.41, we specified the function $\mathsf{merge}_{v_d} : \mathbb{A}^* \to \mathbb{A}$ that merges $k \in \mathbb{N}_{>0}$ abstract values $v_1, \dots, v_k \in \mathbb{A}$ into a single value $\mathsf{merge}_{v_d}(v_1, \dots, v_k) = r \in \mathbb{A}$ with $v_d \in \mathbb{A} \setminus \{\perp_{\mathbb{A}}\}$ denoting the abstract value of the case distinction's discriminant so that the following holds for all assignments $\sigma \in \mathbb{B}^{\mathrm{V}}$:

$$\begin{aligned}&\forall i \in \{1 \dots k\} : \\ &\qquad (\mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, v_d)) = i) \implies (\mathsf{decode}_T(\sigma, r) = \mathsf{decode}_T(\sigma, v_i))\end{aligned}$$

For the original case distinction of type $T \in \textsc{Type}_0$, the merged values $v_1, \dots, v_k$ represent the abstract values of all k evaluated branches.

For readability, we only consider case distinctions with two branches ($k = 2$) where the abstract value v_d of the discriminant contains a single flag, i.e., $\mathsf{flags}(v_d) = (f_d)$ with $f_d \in \mathrm{F}$. In this scenario, the result r of the merge must satisfy the following property for all assignments $\sigma \in \mathbb{B}^{\mathrm{V}}$ (cf. Example 4.42):

$$\begin{aligned}&(\mathsf{eval}_{\mathcal{B}}(\sigma, f_d) = \mathsf{False} \implies \mathsf{decode}_T(\sigma, r) = \mathsf{decode}_T(\sigma, v_1)) \\ \wedge\ &(\mathsf{eval}_{\mathcal{B}}(\sigma, f_d) = \mathsf{True} \implies \mathsf{decode}_T(\sigma, r) = \mathsf{decode}_T(\sigma, v_2))\end{aligned}$$

Furthermore, we assume that v_1 and v_2 are abstract values with the following properties:

1. $\mathsf{flags}(v_1) = (f_{11}, \ldots, f_{1m})$ and $\mathsf{flags}(v_2) = (f_{21}, \ldots, f_{2n})$ with $m, n \in \mathbb{N}_{>0}$ and $m \neq n$, and
2. $|\,\mathsf{arguments}(v_1)| = |\,\mathsf{arguments}(v_2)| = 0$

Informally, v_1 and v_2 both have no arguments and a different number of flags.

It might seem counter-intuitive that two abstract values that represent concrete values of the same type T can contain a different number of flags; here m and n with $m \neq n$. But such a situation can occur because of the overlapping encoding of constructor arguments (cf. Section 4.1.4). Merging those values must result in an abstract value that contains $\max(m, n)$ flags in order to represent both branches of the case distinction.

Merging both values $v_1 = ((f_{11}, \ldots, f_{1m}), ())$ and $v_2 = ((f_{21}, \ldots, f_{2n}), ())$ into the value $\mathsf{merge}_{v_d}(v_1, v_2) = ((r_1, \ldots, r_{\max(m,n)}), ())$ using the discriminant's abstract value $v_d = ((f_d), ())$ can be implemented as follows:

$$\forall i \in \{1 \ldots \max(m, n)\} :$$
$$r_i \Leftrightarrow \begin{cases} (\neg f_d \implies f_{1i}) \wedge (f_d \implies f_{2i}) & \text{if } i \leq m \text{ and } i \leq n \\ (\neg f_d \implies f_{1i}) \wedge (f_d \implies \mathsf{False}) & \text{if } i \leq m \text{ and } i > n \\ (\neg f_d \implies \mathsf{False}) \wedge (f_d \implies f_{2i}) & \text{if } i > m \text{ and } i \leq n \end{cases}$$

In this implementation, the abstract value that contains fewer flags is hypothetically expanded by additional flags of value False. While that is a reasonable approach, it generates unnecessary clauses in the resulting propositional formula whenever $i > n$ or $i > m$. The following variant is equally correct but leads to fewer clauses in the resulting formula:

$$\forall i \in \{1 \ldots \max(m, n)\} :$$
$$r_i \Leftrightarrow \begin{cases} (\neg f_d \implies f_{1i}) \wedge (f_d \implies f_{2i}) & \text{if } i \leq m \text{ and } i \leq n \\ f_{1i} & \text{if } i \leq m \text{ and } i > n \\ f_{2i} & \text{if } i > m \text{ and } i \leq n \end{cases}$$

In this variant, clauses are only generated for the resulting flags r_i with $i \leq \min(m, n)$. All residual flags are simply copied from $\mathsf{flags}(v_1)$ and $\mathsf{flags}(v_2)$. That is reasonable because when decoding an abstract value to a concrete value with $k \in \mathbb{N}_{>0}$ constructors, only the first $\lceil \log_2(k) \rceil$ flags are used (cf. Definition 4.18). All additional flags are ignored.

This example shows that it is beneficial to exploit patterns of flags when merging abstract values.

Primitive Operations on Booleans Often we want to use Booleans in the domain of discourse of a constraint. Thus, we implement functions on Booleans as well. When compiling these functions, we can exploit the fact that CO^4 compiles concrete programs to satisfiability problems in propositional logic. In the following, we give an example of a specially optimized compilation for a function that implements the conjunction of two Boolean values.

```
data Bool = False | True

and2 :: Bool -> Bool -> Bool
and2 = \x y -> case x of False -> False
                         True -> y
```

The compilation of `and2`, as it has been introduced in Section 4.2, results in an abstract declaration that contains a merge of two values: the abstract counterpart of the constant `False` and the abstract value that represents the concrete value `y`. In the following, $v_x \in \mathbb{A}$ and $v_y \in \mathbb{A}$ denote the abstract values that respectively represent the concrete values `x` and `y` with $\mathsf{flags}(v_x) = (f_x)$ and $\mathsf{flags}(v_y) = (f_y)$ (note that we assume $v_x \neq \bot_{\mathbb{A}}$ and $v_y \neq \bot_{\mathbb{A}}$). The merge $\mathsf{merge}_{v_x}(\mathsf{encode}_{\mathtt{Bool}}(\mathtt{False}), v_y) = r = ((r_1), ())$ that results from compiling the function `and2` is specified according to Definition 4.41 for all assignments $\sigma \in \mathbb{B}^{\mathbb{V}}$:

$$\begin{aligned} &(\mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, v_x)) = 1 \implies \mathsf{decode}_{\mathtt{Bool}}(\sigma, r) = \mathtt{False}) \\ \land\ &(\mathsf{numeric}_k(\mathsf{eval}_{\mathsf{flags}}(\sigma, v_x)) = 2 \implies \mathsf{decode}_{\mathtt{Bool}}(\sigma, r) = \mathsf{decode}_{\mathtt{Bool}}(\sigma, v_y))) \end{aligned}$$

We give a naive implementation of this merge for the resulting flag r_1:

$$r_1 \Leftrightarrow (\neg f_x \implies \mathsf{False}) \land (f_x \implies f_y)$$

While this implementation is correct according to the definition of merge, it generates unnecessary clauses in the final propositional formula after performing Tseitin's transformation (cf. Definition B.13). A better implementation can be obtained by applying the implication elimination rule (modus ponens) beforehand, which gives:

$$\begin{aligned} r_1 &\Leftrightarrow (\neg f_x \implies \mathsf{False}) \land (f_x \implies f_y) \\ &\Leftrightarrow f_x \land (f_x \implies f_y) \\ &\Leftrightarrow f_x \land f_y \end{aligned}$$

This implementation generates fewer clauses in the final propositional formula after performing Tseitin's transformation because there is only a single conjunction of two propositional formulas.

In order to efficiently compile conjunctions of lists of Booleans, this scheme can be generalized so that more than two values are handled. There are equivalent optimizations for merges that result from compiling other functions on Boolean values, e.g., disjunctions.

Chapter 7

Applications

In this chapter, we illustrate two use-cases where the CO^4 constraint solver is applied to constraint satisfaction problems from two different domains. These examples emphasize different strengths of CO^4.

In Section 7.1, CO^4 is applied to different problems related to termination analysis of term rewriting systems. In Section 7.1.1, we inspect looping derivations, which prove a rewriting system to be non-terminating. As looping derivations are non-flat structures, this example shows how CO^4 handles constraints over complex and highly structured domains. Section 7.1.2 illustrates a use-case where the termination of a given term rewriting system is shown by finding a compatible lexicographic path order over the terms of that system. This use-case illustrates how the textbook definition of a proof strategy can be directly translated into a concrete program. In order to search termination proofs for term rewriting systems that do not admit a lexicographic path order, we extend this proof strategy in Section 7.1.3 by applying the semantic labelling transformation on term rewriting systems.

Finally, Section 7.2 shows an application of CO^4 for the RNA design problem in bioinformatics. As the RNA design problem is the inverse of the RNA secondary structure prediction problem, we show how CO^4 can be applied to tackle both problems simply by swapping the parameter domain and the domain of discourse.

7.1 Termination Analysis of Term Rewriting Systems

Term rewriting constitutes a Turing-complete computation model and has strong relations to many programming paradigms, e.g., functional programming [33].

A term rewriting system is a simple but powerful formalism for representing computations on structured data. Note that Appendix A.3 gives a brief introduction to term rewriting.

Proving either termination or non-termination of term rewriting systems is an area of active research with applications for termination analysis of computer programs [37][75][29]. Constraint-based analyses are a well-known approach where the task of finding the parameter of a termination proof is considered as a constraint satisfaction problem [33], i.e., for a given term rewriting system and a parameterized proof strategy, one is looking for a parameter that induces an actual proof of termination for the given rewriting system. In this section, we illustrate that CO^4 allows concise and high-level specifications for the parameters of different proof strategies [8].

Firstly, we define when a term rewriting system is considered to be terminating.

Definition 7.1 A term rewriting system (Σ, X, R) is *terminating* if there are no infinite rewrite chains $t_1 \to_R t_2 \to_R t_3 \to_R \ldots$ of the rewrite relation $\to_R$.

Consequently, a term rewriting system does not terminate if there is an infinite rewrite chain.

7.1.1 Looping Derivations in Term Rewriting Systems

The existence of a looping derivation induces an infinite rewrite chain; thus, it proves a term rewriting system to be non-terminating. In this section, we use CO^4 for finding looping derivations in term rewriting systems. Finding looping derivations in string rewriting systems by manually constructing a satisfiability problem in propositional logic is a well-known method [76].

Definition 7.2 A term rewriting system (Σ, X, R) is *looping* if there are a term $t \in \mathsf{terms}(\Sigma, X)$ and a looping derivation

$$t \to_R t_1 \to_R \cdots \to_R t_n[\widehat{\sigma}(t)]_p$$

of the rewrite relation $\to_R$ so that after $n \in \mathbb{N}_{>0}$ rewrite steps, the term $t_n[\widehat{\sigma}(t)]_p \in \mathsf{terms}(\Sigma, X)$ contains the term $\widehat{\sigma}(t) \in \mathsf{terms}(\Sigma, X)$ at position $p \in \mathrm{Pos}_{t_n}$ for some substitution $\widehat{\sigma} : \mathsf{terms}(\Sigma, X) \to \mathsf{terms}(\Sigma, X)$.

Example 7.3 The term rewriting system $(\{f, g, 0, 1\}, \{x, y\}, R)$ [72] with

$$R = \left\{ \begin{array}{rcl} f(0,1,x) & \to & f(x,x,x), \\ g(x,y) & \to & x, \\ g(x,y) & \to & y \end{array} \right\}$$

induces a looping derivation $t \to_R t_1 \to_R t_2 \to_R t_3$ with

$$\begin{aligned} t &= f(g(y,0), g(1,0), g(1,0)) \\ t_1 &= f(g(y,0), 1, g(1,0)) \\ t_2 &= f(0, 1, g(1,0)) \\ t_3 &= f(g(1,0), g(1,0), g(1,0)) \end{aligned}$$

where $t_3|_{()} = t_3 = \widehat{\sigma}(t)$ and $\sigma = \{(y,1)\}$.

In Listing 7.4, we give an excerpt of a concrete program $c \in \textsc{Prog}$ that implements a specification for looping derivations. The parameter domain of c is the set of term rewriting systems, where each system is represented by a list of rules. Each rule is a pair of terms, where the first (resp. second) component denote the rule's left-hand (resp. right-hand) side. Note that we do not fix a certain signature. Instead, we denote variable and function symbols by natural numbers `Nat` in order to allow this concrete program to be applied to term rewriting systems with different signatures (cf. Section 6.3 for CO^4's built-in implementation `Nat` of natural numbers). But for indexing subterms, the concrete program c does also utilize unary encoded natural numbers (cf. Line 7). This is reasonable because the recursive definition of `Unary` makes it easy to write functions that operate on an indexed element of some recursive structure, e.g., the function `replace` at Line 105 in Appendix C.3.

The domain of discourse of c is the set of looping derivations over a set of terms $\mathsf{terms}(\Sigma, X)$. According to Definition 7.2, each loop of length $n \in \mathbb{N}_{>0}$ contains three components (cf. the type `LoopingDerivation` in Listing 7.4):

1. a list of n intermediate terms $t_1, \ldots, t_n \in \mathsf{terms}(\Sigma, X)$,
2. a position $p \in \textsc{Pos}_{t_n}$ (represented as list of unary numbers), and
3. a substitution $\widehat{\sigma} : \mathsf{terms}(\Sigma, X) \to \mathsf{terms}(\Sigma, X)$ (represented as list of pairs of `Nat` and `Term`).

Each of the intermediate terms is represented by a derivation step (cf. `Step` in c) where each step consists of

1. an input term,
2. the rule that is applied in this step,
3. a position (represented as list of unary numbers),
4. a substitution (represented as list of pairs of `Nat` and `Term`), and
5. a resulting term.

The concrete program c checks if a looping derivation is compatible with the given term rewriting system.

In the following example, we apply CO^4 to find a solution for the term rewriting system in Example 7.3 using the aforementioned concrete program.

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

data Term = Var Nat
          | Node Nat (List Term)

data Unary = Z | S Unary

data Step = Step Term
                 (Pair Term Term)
                 (List Unary)
                 (List (Pair Nat Term))
                 Term

data LoopingDerivation = LoopingDerivation
                           (List Step)
                           (List Unary)
                           (List (Pair Nat Term))

constraint :: List (Pair Term Term) -> LoopingDerivation
           -> Bool
constraint = \trs deriv ->
  isCompatibleLoopingDerivation trs deriv

isCompatibleLoopingDerivation :: List (Pair Term Term)
                              -> LoopingDerivation
                              -> Bool
isCompatibleLoopingDerivation = \trs loopDeriv ->
  case loopDeriv of
    LoopingDerivation deriv lastPos lastSub ->
      case deriv of
        Nil -> False
        Cons step steps -> case step of
          Step t0 rule pos sub t1 ->
            let last    = deriveTerm trs t0 deriv
                subterm = getSubterm lastPos last
                t0'     = applySubstitution lastSub t0
            in
              eqTerm t0' subterm
```

Listing 7.4: An excerpt of a concrete program $c \in \textsc{Prog}$ that implements a specification for looping derivations. The complete program can be found in Appendix C.3.

Example 7.5 For the term rewriting system in Example 7.3, CO^4 finds the mentioned looping derivation by generating a propositional formula with 78165 variables, 243090 clauses, and 661534 literals. This formula is solved by MiniSat (version 2.2) in 0.5 seconds on a 3.2 GHz CPU. As the domain of discourse is infinite, it has been restricted to three derivation steps on terms with a maximum depth of three and at most three subterms.

Evaluation

As we have stated in the introduction, achieving competitive runtimes for CO^4 in comparison to domain-specific solvers is beyond the scope of the present thesis. However, we want to evaluate the runtime-performance of CO^4 with respect to the search for looping derivations, which represent a highly structured domain of discourse. Therefore, we compare the propositional encoding generated by CO^4 with the term-unfolding strategy `unfold` [63] implemented in the Tyrolean Termination Tool 2 version 1.16 ($\mathsf{T_TT_2}$) [41][52]. The $\mathsf{T_TT_2}$ software is a termination analyzer for term rewriting systems and it provides a flexible strategy language for configuring the proof strategy used for proving termination.

Furthermore, we want to evaluate the impact of the memoization optimization presented in Section 6.2 on the size of the propositional encoding generated by CO^4. In order to limit this size with regard to a timeout of 60 seconds for finding a looping derivation in a term rewriting system, we restrict the domain of discourse to derivations containing up to three steps where each step may involve terms with a maximum depth of three and at most three subterms. For $\mathsf{T_TT_2}$, we run with its default configuration, i.e., neither the length of the derivation nor the size of the involved terms are restricted.

We evaluate against the 1463 term rewriting systems in the `TRS_Standard` category of the Termination Problems Data Base version 8.0.7 [1]; a standard set of benchmarks used in the annual Termination Competition [38]. With a timeout of 60 seconds for each of the 1463 term rewriting systems, $\mathsf{T_TT_2}$ finds a loop in 160 systems, whereas CO^4 finds a loop in 127 systems. There are 101 systems for which both tools find a loop. Note that the system from Example 7.3 is among the 26 systems for which CO^4 finds a loop within the given timeout, but $\mathsf{T_TT_2}$'s `unfold` strategy does not. In Table 7.6, we compare the solver runtimes with respect to the common 101 systems on a 3.2 GHz CPU.

The times given in Table 7.6 reveal that, on average, CO^4 runs around one order of magnitude slower than $\mathsf{T_TT_2}$. While this shows that there is still work to be done in order to provide a constraint solver that can compete with modern domain-specific solvers, it is still a promising result for a prototypical implementation of a general-purpose constraint solver that has no domain-specific optimizations available.

Note that the propositional encoding generated by CO^4 does not benefit much from memoization when searching for looping derivations. Note furthermore

Tool	Solving time [s] total	avg	max
$\mathsf{T_TT_2}$	11.93	0.1	0.18
CO^4	143.69	1.39	5.41
CO^4 (no memoization)	143.87	1.42	5.45
CO^4 (only SAT solver)	47.04	0.47	2.51

Table 7.6: Total, average and maximum solver runtimes for finding looping derivations in 101 term rewriting systems of the Termination Problems Data Base.

that most of CO^4's runtime is spent on generating the propositional formula. This result indicates that future work should not only consider optimizing the generated propositional formulas but also the runtime-performance of CO^4's implementation itself, e.g., it might be of interest how the garbage collection procedure of Haskell's runtime-system affects the runtime of CO^4. The accumulated solver runtimes shown in Figure 7.7 illustrate the gap between CO^4's total runtime and the runtime of the external SAT solver MiniSat.

7.1.2 Lexicographic Path Orders

In this section, we aim to prove termination of a term rewriting system by finding a lexicographic path order [6]. A lexicographic path order is a relation on terms that is induced by a strict order of the symbols of a signature.

Definition 7.8 The *lexicographic path order* $>_{\text{lpo}} \subseteq \mathsf{terms}(\Sigma, X)^2$ induced by a strict partial order $>_{\text{prec}} \subseteq \Sigma^2$ is a binary relation on terms $s, t \in \mathsf{terms}(\Sigma, X)$ over a variable set X so that $s >_{\text{lpo}} t$ holds if:

1. $t \in \mathsf{var}(s)$ and $s \neq t$, or
2. $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$, and
   (a) $s_i \geq_{\text{lpo}} t$ for some $i \in \{1 \ldots m\}$, or
   (b) $f >_{\text{prec}} g$ and $s >_{\text{lpo}} t_j$ for all $j \in \{1 \ldots n\}$, or
   (c) $f = g$, $s >_{\text{lpo}} t_j$ for all $j \in \{1 \ldots n\}$, and there is an index $i \in \{1 \ldots m\}$, so that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_i >_{\text{lpo}} t_i$.

Note that $\geq_{\text{lpo}}$ denotes the reflexive closure of $>_{\text{lpo}}$. In the following, we name the strict partial order $>_{\text{prec}} \subseteq \Sigma^2$ a *precedence* over the symbols in Σ.

The lexicographic path order induced by a precedence on function symbols may prove termination of term rewriting systems.

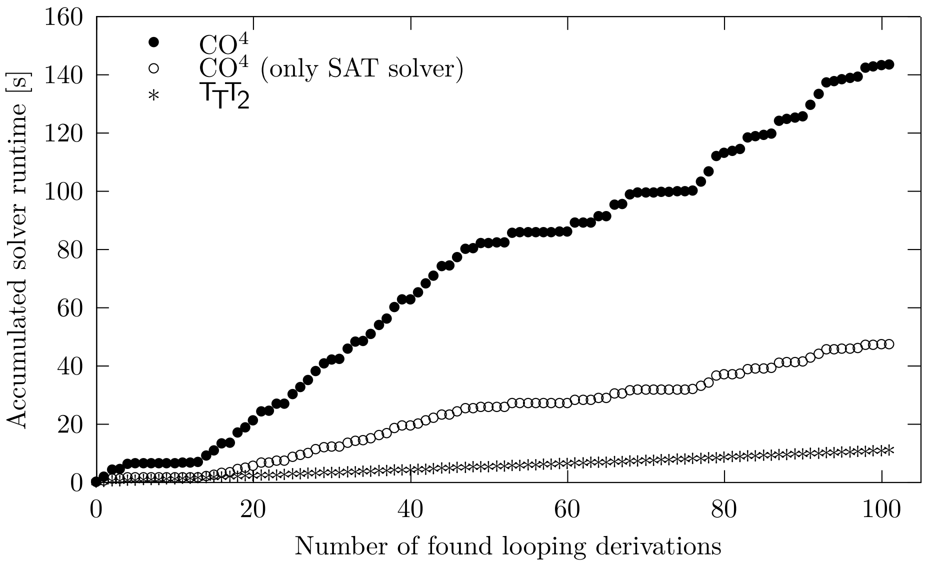


Figure 7.7: Accumulated solver runtimes for finding looping derivations in 101 term rewriting systems of the Termination Problems Data Base. Note that the order in which the systems were solved corresponds to the lexicographic order of their file names in the Termination Problems Data Base. For readability, the graph does not show the runtimes for CO^4 without memoization as the search for looping derivations does not benefit from this optimization.

Theorem 7.9 A term rewriting system (Σ, X, R) is terminating if there is a precedence $>_{\mathrm{prec}} \subseteq \Sigma^2$ inducing a lexicographic path order $>_{\mathrm{lpo}} \subseteq \mathsf{terms}(\Sigma, X)^2$ such that $l >_{\mathrm{lpo}} r$ for all $(l \to r) \in R$ [6]. ■

Finding a LPO-inducing partial order $>_{\mathrm{prec}} \subseteq \Sigma^2$ by specifying its properties as a satisfiability problem in propositional logic is a well-known approach for proving a term rewriting system to be terminating [18] [19].

In Listing 7.10, we give an excerpt of a concrete program that specifies a constraint on precedences so that the induced lexicographic path order is compatible with a given rewriting system. Note that the function `lpo` in Listing 7.10 is almost a direct translation of the textbook definition of lexicographic path orders (cf. Definition 7.8). The parameter domain is the set of term rewriting systems, where each system is represented by a list of function symbols and a list of rules. Each rule is a pair of terms, where the first (resp. second) component denotes the rule's left-hand (resp. right-hand) side. As described in Section 7.1.1, we represent function and variable symbols by natural numbers.

The domain of discourse for a term rewriting system (Σ, X, R) is the set of precedences over $|\Sigma|$ different function symbols. The variable `prec` represents the underlying strict partial order using a list of $|\Sigma|$ function symbols sorted in descending order of their respective precedence. Note that `prec` actually represents a total order, and therefore slightly diverges from Definition 7.8. This is not a problem as Definition 7.8 equally holds if $>_{\text{prec}}$ is total.

The function `lpo` recursively traverses its arguments `s` and `t` using case distinctions in Line 22, 23, 24, and 29. However, in the corresponding abstract program, the compiled case distinctions can be evaluated without generating new subformulas (cf. Lemma 4.43). That is because all traversed terms stem from the term rewriting system that is the parameter of the concrete program, i.e., all terms are known when evaluating the abstract program. On the other hand, functions that depend on the unknown precedence contribute to the resulting propositional formula. That especially applies to the function `ord`, which computes the relation between two function symbols by looking up their precedences in the list `prec`.

In Example 7.11, we show the termination of an exemplary term rewriting system.

Example 7.11 Assume a term rewriting system (Σ, X, R) that specifies the Ackermann function:

$$\begin{aligned}
\Sigma = \{a, s, n\} \text{ with } & \mathsf{arity}(a) = 2 \\
& \mathsf{arity}(s) = 1 \\
& \mathsf{arity}(n) = 0 \\
X = \{x, y\} & \\
R = \left\{ \begin{array}{rcl} a(n, y) & \to & s(y), \\ a(s(x), n) & \to & a(x, s(n)), \\ a(s(x), s(y)) & \to & a(x, a(s(x), y)) \end{array} \right\} &
\end{aligned}$$

CO^4 finds the precedence $a >_{\text{prec}} n >_{\text{prec}} s$ by generating a propositional formula with 172 variables, 417 clauses, and 989 literals.

CO^4's profiling output given in Appendix C.5 actually confirms that the compiled counterparts of the case distinctions in Line 22, 23, 24, and 29 do not contribute to the final propositional formula.

Note that the present implementation of lexicographic path orders heavily benefits from memoization of function applications (cf. Section 6.2). With memoization, function `ord` needs to be evaluated at most once for each pair of function symbols. Without memoization, CO^4 finds the aforementioned precedence by generating a propositional formula with 531 variables, 1420 clauses, and 3433 literals.

```
data Bool = False | True
data Pair a b = Pair a b
data List a = Nil | Cons a (List a)

data Term = Var Nat | Node Nat (List Term)

data Order = Gr | Eq | NGe

data TRS = TRS (List Nat) (List (Pair Term Term))

constraint :: TRS -> List Nat -> Bool
constraint = \trs prec -> case trs of
  TRS symbols rules ->
    and2 (forall rules   (\rule -> ordered rule prec))
         (forall symbols (\sym  -> exists prec sym eqNat))

ordered :: Pair Term Term -> List Nat -> Bool
ordered = \rule prec -> case rule of
  Pair lhs rhs -> eqOrder (lpo prec lhs rhs) Gr

lpo :: List Nat -> Term -> Term -> Order
lpo = \prec s t -> case t of
 Var x -> case eqTerm s t of
   False -> case varOccurs x s of
              False -> NGe
              True  -> Gr
   True -> Eq

 Node g ts -> case s of
  Var v -> NGe
  Node f ss ->
   case forall ss (\si -> eqOrder (lpo prec si t) NGe) of
    False -> Gr
    True  -> case ord prec f g of
     Gr ->
      case forall ts (\ti -> eqOrder (lpo prec s ti) Gr) of
        False -> NGe
        True  -> Gr
     Eq ->
      case forall ts (\ti -> eqOrder (lpo prec s ti) Gr) of
        False -> NGe
        True  -> lex (\xs ys -> lpo prec xs ys) ss ts
     NGe -> NGe

ord :: List Nat -> Nat -> Nat -> Order
ord = \prec a b -> ...
```

Listing 7.10: An excerpt of a concrete program that specifies a constraint on precedences so that the induced lexicographic path order is compatible with a given rewriting system. The complete listing can be found in Appendix C.4.

Evaluation

Similar to the search for looping derivations that we have illustrated in the previous section, we want to evaluate the size of the propositional encoding and the overall runtime-performance of CO^4 with respect to the search for compatible lexicographic path orders.

Therefore, we compare both aspects to the size of the manually derived propositional encoding implemented in the `lpo` processor of the Tyrolean Termination Tool 2 version 1.16 ($\mathsf{T_TT_2}$) [41][52] and the overall runtime-performance of $\mathsf{T_TT_2}$, respectively. Furthermore, we want to evaluate the impact of the memoization optimization presented in Section 6.2 on the size of the propositional encoding generated by CO^4.

Both tools find a compatible lexicographic path order for 144 of 1463 term rewriting systems from the `TRS_Standard` category of the Termination Problems Data Base version 8.0.7 [1]. Table 7.12 shows the runtimes and sizes of the generated propositional formulas for both tools.

	Solving time [s]			#variables		#clauses	
Tool	total	avg	max	avg	max	avg	max
$\mathsf{T_TT_2}$	16.27	0.1	0.15	96	755	139	1104
CO^4	4.14	0.02	0.21	581	6724	1654	21096
CO^4 (no memoiz.)	20.2	0.14	2.09	3987	52186	13174	182400

Table 7.12: Total, average and maximum runtimes for $\mathsf{T_TT_2}$ and CO^4, as well as formula sizes for finding compatible lexicographic path orders in 144 term rewriting systems of the Termination Problems Data Base.

Table 7.12 shows that the propositional formulas generated by CO^4 are about one order of magnitude larger than the propositional encoding provided by $\mathsf{T_TT_2}$ with respect to the number of variables and clauses. That is not surprising as domain-specific tools like $\mathsf{T_TT_2}$ may incorporate deep knowledge about their respective domains into the generation of propositional encodings. However, CO^4's runtime is, on average, slightly lower than $\mathsf{T_TT_2}$'s runtime, which might result from a more sophisticated preprocessing done by $\mathsf{T_TT_2}$, which eventually results in the generation of smaller propositional encodings.

The numbers in Table 7.12 confirm that the search for a compatible lexicographic path order benefits from memoization (cf. Example 7.11), i.e., when disabling memoization, CO^4's runtime performance degrades and it generates larger propositional formulas. Figure 7.13 illustrates the benefits of memoization for the accumulated solver runtimes over all 144 term rewriting systems.

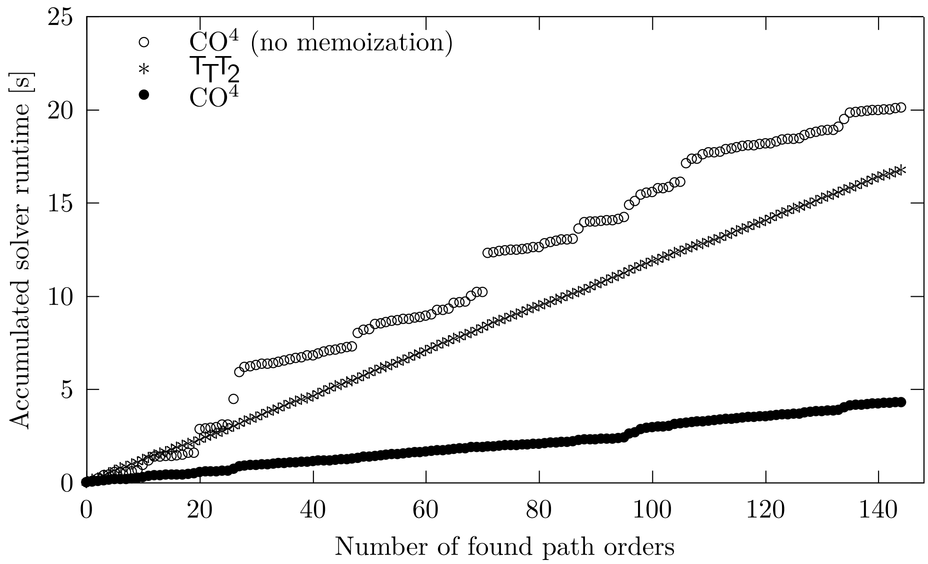


Figure 7.13: Accumulated solver runtimes for finding compatible lexicographic path orders in 144 term rewriting systems of the Termination Problems Data Base. Note that the order in which the systems were solved corresponds to the lexicographic order of their file names in the Termination Problems Data Base.

Overall, these results are quite promising: they not only show the benefits of optimizations like memoization, but also indicate that for certain applications, CO^4's runtime is already competitive compared to domain-specific tools. However, the differences in the sizes of the generated propositional encodings show that there is much room for improvement. While this potential should be evaluated in future work, a general-purpose approach like CO^4 will always lack domain-specific knowledge needed for generating an optimal propositional encoding. This situation is similar to the respective characteristics of programming in a high-level language like Haskell versus programming in Assembler: while Assembler helps developing fast and memory-efficient programs, a high-level language supports more abstract concepts that help developing applications on a large scale.

7.1.3 Semantic Labelling

The term rewriting system in Example 7.11 contains only three function symbols; thus, there are only few semantically different precedences. The problem of finding a compatible lexicographic path order becomes harder for more complex rewriting systems. In this section, we introduce semantic labelling [77], a transformation of term rewriting systems that deliberately increases the signature and the number of rules of a given rewriting system. While such a transformation seems counterintuitive in the first place, it is often necessary in order to apply proof strategies like lexicographic path orders. Example 7.14 shows a term rewriting system that admits no compatible lexicographic path order.

Example 7.14 Assume the following term rewriting system (Σ, X, R) [77]:

$$\begin{aligned} \Sigma = \{+, *, f, g, a\} \text{ with } \mathsf{arity}(+) &= 2 \\ \mathsf{arity}(*) &= 2 \\ \mathsf{arity}(f) &= 1 \\ \mathsf{arity}(g) &= 2 \\ \mathsf{arity}(a) &= 0 \end{aligned}$$

$$X = \{x, y, z\}$$

$$R = \left\{ \begin{array}{rcl} (x * y) * z & \to & x * (y * z), \\ (x + y) * z & \to & (x * z) + (y * z), \\ x * (y + f(z)) & \to & g(x, z) * (y + a) \end{array} \right\}$$

For readability we have written the symbols $+$ and $*$ in infix notation.

There is no compatible lexicographic path order for this system.

In order to prove termination of the term rewriting system in Example 7.14, we specify semantic labelling as a transformation of term rewriting systems. The signature of a labelled term rewriting system has more symbols than the original rewriting system. In the end, this will allow us to apply proof strategies like the lexicographic path order to the labelled system, which is not possible for the original rewriting system.

Definition 7.15 The *signature of labelled function symbols* $\overline{\Sigma}$ is defined by

$$\overline{\Sigma} = \{s_l \mid s \in \Sigma \wedge l \in L_s\}$$

where L_s denotes a non-empty set of labels for the symbol $s \in \Sigma$.

Next, we need to label the rules of the given term rewriting system. To do so, we fix a Σ-algebra (cf. Definition A.26) that we require to be a model for the given rewriting system.

Definition 7.16 A Σ-algebra $\mathcal{M} = (M, [.])$ is a *model for a term rewriting system* (Σ, X, R) with a variable set X if:

$$\forall \sigma \in M^X : \forall (l \to r) \in R : \mathsf{eval}_{\mathcal{M}}(\sigma, l) = \mathsf{eval}_{\mathcal{M}}(\sigma, r)$$

Example 7.17 The following algebra $(M, [.])$ is a model for the term rewriting system in Example 7.14:

$$\begin{aligned} M &= \{1, 2\} & [+](x, y) &= y \\ [*](x, y) &= 1 & [f](x) &= 2 \\ [g](x, y) &= 2 & [a] &= 1 \end{aligned}$$

For every symbol $s \in \Sigma$, we choose an n-ary mapping $\pi_s : M^n \to L_s$ with $n = \mathsf{arity}(s)$. The mapping π_s determines the label for the symbol s according to the values of its arguments as interpreted in the model $\mathcal{M} = (M, [.])$. Based on these mappings we define a labelling function for terms.

Definition 7.18 The *labelling function* $\mathsf{lab} : \mathsf{terms}(\Sigma, X) \times M^X \to \mathsf{terms}(\overline{\Sigma}, X)$ gives the labelled term for a term $t \in \mathsf{terms}(\Sigma, X)$ and a mapping $\sigma : X \to M$ from the variable set X to the carrier set M of a model $\mathcal{M}$:

$$\mathsf{lab}(t, \sigma) :=$$
$$\begin{cases} t & \text{if } t \in X \\ f_{\pi_f(\mathsf{eval}_{\mathcal{M}}(\sigma, t_1), \ldots, \mathsf{eval}_{\mathcal{M}}(\sigma, t_n))}(\mathsf{lab}(t_1, \sigma), \ldots, \mathsf{lab}(t_n, \sigma)) & \text{if } t = f(t_1, \ldots, t_n) \\ & \text{for } n \in \mathbb{N} \end{cases}$$

Using the labelling function, we can compute a labelled term rewriting system by applying lab to all rules of a given rewriting system.

Definition 7.19 The set of rules $\overline{R} \subseteq \mathsf{terms}(\overline{\Sigma}, X)^2$ of the *labelled term rewriting system* $(\overline{\Sigma}, X, \overline{R})$ for a given term rewriting system (Σ, X, R) and a model $\mathcal{M}$ with carrier set M is defined by:

$$\overline{R} = \left\{ \mathsf{lab}(\sigma, l) \to \mathsf{lab}(\sigma, r) \mid (l \to r) \in R \wedge \sigma \in M^X \right\}$$

Example 7.20 For the rewriting system (Σ, X, R) in Example 7.14 and the model $\mathcal{M}$ in Example 7.17 we fix the following:

$$\begin{aligned} \overline{\Sigma} &= \{+_1, *_1, *_2, f_1, g_1, a_1\} \\ \pi_*(x, y) &= y \\ \pi_+(x, y) &= \pi_f(x) = \pi_g(x, y) = \pi_a = 1 \end{aligned}$$

Then, the set of rules $\overline{R}$ of the labelled term rewriting system is:

$$\overline{R} = \left\{ \begin{array}{rcl} (x *_1 y) *_1 z & \to & x *_1 (y *_1 z), \\ (x *_1 y) *_2 z & \to & x *_1 (y *_2 z), \\ (x *_2 y) *_1 z & \to & x *_1 (y *_1 z), \\ (x *_2 y) *_2 z & \to & x *_1 (y *_2 z), \\ (x +_1 y) *_1 z & \to & (x *_1 z) +_1 (y *_1 z), \\ (x +_1 y) *_2 z & \to & (x *_2 z) +_1 (y *_2 z), \\ x *_2 (y +_1 f_1(z)) & \to & g_1(x, z) *_1 (y +_1 a) \end{array} \right\}$$

Due to Theorem 7.21, we can prove termination of a given term rewriting system by proving termination of the labelled term rewriting system.

Theorem 7.21 [77] A term rewriting system (Σ, X, R) is terminating if the labelled term rewriting system $(\overline{\Sigma}, X, \overline{R})$ is terminating for

1. a model $\mathcal{M} = (M, [.])$,
2. a non-empty set of labels L_s for each symbol $s \in \Sigma$, and
3. a mapping $\pi_s : M^n \to L_s$ with $n = \mathsf{arity}(s)$ for each symbol $s \in \Sigma$. ■

Example 7.22 For the labelled term rewriting system in Example 7.20, CO^4 finds a precedence $>_{\text{prec}}$ with

$$*_2 >_{\text{prec}} *_1 >_{\text{prec}} g_1 >_{\text{prec}} f_1 >_{\text{prec}} a_1 >_{\text{prec}} +_1$$

that induces a compatible lexicographic path order. CO^4 generates a propositional formula with 328 variables, 1133 clauses, and 3117 literals.

Note that the precedence in Example 7.22 for the labelled system from Example 7.22 was found using the exact same concrete program as it was used in Example 7.11 without semantic labelling. This is possible because we did not fix a particular signature in the concrete program.

In Figure 7.23, we informally specify an algorithm that combines the search for a compatible lexicographic path order with semantic labelling in order to prove termination of unlabelled term rewriting systems. This approach has been published as *SAT Compilation for Termination Proofs via Semantic Labelling and Unlabelling* at the Workshop on Termination in 2014 [10].

1. Specify a model $\mathcal{M}$ with a carrier set M over the signature Σ.
2. For each symbol $s \in \Sigma$, pick an appropriate mapping $\pi_s : M^n \to L_s$ so that $n = \mathsf{arity}(s)$ and L_s denotes a non-empty set of labels for s.
3. Construct a labelled term rewriting system over $\overline{\Sigma}$.
4. Specify a precedence over the symbols in $\overline{\Sigma}$ that induces a lexicographic path order for the labelled term rewriting system.

Figure 7.23: Algorithm for combining the search for a compatible lexicographic path order with semantic labelling in CO^4.

In Listing 7.24, we give an excerpt of a concrete program implementing a specification of a precedence which induces a lexicographic path order over a labelled term rewriting system.

Evaluation

Using the algorithm given in Figure 7.23 leads to a single SAT solver run for finding a model as well as a precedence. This extends previous approaches [9] that only work on string rewriting systems. This algorithm has been implemented as a module in the Matchbox termination prover [74] for participating in the Termination Competition 2014. We could produce the first certified proof for the

```
constraint :: Triple (TRS Symbol)
                     (List (Labelled Symbol))
                     (List Sigma)
           -> Pair (Precedence (Labelled Symbol))
                   (Interpretation Symbol)
           -> Bool
constraint = \p u ->
  let eqSymbol         = eqNat
      eqLabelledSymbol = eqLabelled eqNat
  in
    case p of Triple trs lsymbols assigns ->
      case u of Pair prec interp ->
        case trs of Pair symbols rules ->
          let lrules = labelledRules eqNat interp
                                     assigns rules
              ltrs   = Pair lsymbols lrules
          in
            and2 (lpoConstraint eqLabelledSymbol ltrs prec)
                 (isModel eqNat interp assigns trs)

labelledRules :: (a -> a -> Bool) -> Interpretation a
              -> List Sigma -> List (Rule a)
              -> List (Rule (Labelled a))
labelledRules = \eq interp assigns rules ->
  concat’ (map’ (\rule -> case rule of
    Pair lhs rhs -> map’
      (\sigma -> Pair (labelledTerm eq interp sigma lhs)
                      (labelledTerm eq interp sigma rhs)
      ) assigns) rules)

isModel :: (a -> a -> Bool) -> Interpretation a
        -> List Sigma -> TRS a -> Bool
isModel = \eq interp assigns trs -> case trs of
  Pair symbols rules ->
    forall assigns (\sigma ->
      forall rules (\(Pair lhs rhs) ->
        eqNat (eval eq interp sigma lhs)
              (eval eq interp sigma rhs)))
```

Listing 7.24: An excerpt of a concrete program implementing a specification of a precedence which induces a lexicographic path order over a labelled term rewriting system. The complete program can be found in Appendix C.6.

`TRS_Standard/AProVE_04/JFP_Ex31.xml` system of the Termination Problems Data Base.

In the following example, we use the algorithm from Figure 7.23 to prove termination of the term rewriting system from Example 7.14.

> **Example 7.25** For the unlabelled term rewriting system in Example 7.14, CO^4 finds the model from Example 7.17 and a precedence for the symbols in the labelled system by generating a propositional formula with 18286 variables and 53808 clauses, which is solved by MiniSat (version 2.2) in 0.11 s on a 3.2 GHz CPU. Note that we have fixed π_s to equal the identity function for all symbols $s \in \Sigma$. Furthermore, we restricted the model's carrier set to contain exactly two elements.

Using the same setup as described in Example 7.25 and a timeout of 300 seconds, CO^4 is able to prove termination using the algorithm given in Figure 7.23 for 210 term rewriting systems from the `TRS_Standard` category of the Termination Problems Data Base version 8.0.7 [1], which contains 1463 systems in total.

When restricting the model's carrier set to contain exactly one element, the algorithm in Figure 7.23 reduces to the search for a compatible lexicographic path order as it has been introduced in Section 7.1.2. Unsurprisingly, in this case, CO^4 finds a path order for the exact same term rewriting systems as with the concrete program given in Listing 7.10.

7.2 RNA Design

In this section, we illustrate an application of the constraint solver CO^4 for design problems of ribonucleic acids (RNA). The results of this application have been published as *RNA Design by Program Inversion via SAT Solving* at the Workshop on Constraint-Based-Methods for Bioinformatics in 2013 [11].

RNA molecules play an important role for many biological processes. They are uniquely represented by chains of organic bases.

> **Definition 7.26** The *primary structure* $\mathcal{S}_1 \in \{A, C, G, U\}^n$ of an RNA molecule of length $n \in \mathbb{N}_{>0}$ is a tuple of length n over the four bases denoted as A, C, G, and U.

While these linear molecules fold into tertiary structures, i.e., three-dimensional shapes, many aspects of RNA structures are commonly studied at the level of RNA's secondary structure, i.e., the sequence of pairs of bases.

> **Definition 7.27** The *secondary structure* $\mathcal{S}_2 \subseteq \{1 \dots n\}^2$ of an RNA molecule with the primary structure $\mathcal{S}_1 = (p_1, \dots, p_n)$ of length $n \in \mathbb{N}_{>0}$ is a set of pairs of indices so that all of the following properties hold:

1. each pair indexes a *canonical base pair*:

$$\forall(i,j) \in \mathcal{S}_2 : (p_i, p_j) \in \{(A,U),(U,A),(C,G),(G,C),(G,U),(U,G)\}$$

2. no two pairs $(i,j),(u,v) \in \mathcal{S}_2$ are *crossing*:

$$\big(\{i \ldots j\} \subseteq \{u \ldots v\}\big) \vee \big(\{i \ldots j\} \supseteq \{u \ldots v\}\big)$$

The folding of an RNA's primary structure $\mathcal{S}_1$ into a secondary structure $\mathcal{S}_2$ is associated with a certain amount of free energy $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2) \in \mathbb{Z} \cup \{\infty\}$ whose exact value depends on the underlying energy model. As biological systems strive to minimize the amount of free energy, $\mathcal{S}_1$ is more likely to fold into a secondary structure $\mathcal{S}_2$ that minimizes $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2)$. Consequently, if $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2) = \infty$, $\mathcal{S}_1$ will not fold into the structure $\mathcal{S}_2$ under any circumstances.

Definition 7.28 For a given primary structure $\mathcal{S}_1$, the *RNA secondary structure prediction problem* asks for a secondary structure $\mathcal{S}_2$ so that the amount of free energy $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2)$ is minimized:

$$\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2) = \min\{\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2') \mid \mathcal{S}_2' \in \text{secondary structures}\}$$

The RNA secondary structure prediction problem is an elementary problem of bioinformatics [2]. Assuming non-crossing structures, this problem can be solved efficiently using dynamic programming [36].

A similarly fundamental problem is known as RNA design, which asks for a primary structure of an RNA that folds optimally into a given secondary structure. The RNA secondary structure design problem is most naturally phrased as the exact inverse of the structure prediction problem.

Definition 7.29 For a given secondary structure $\mathcal{S}_2$, the *RNA design* problem asks for a primary structure $\mathcal{S}_1$ so that $\mathcal{S}_2$ is the solution of the structure prediction problem for $\mathcal{S}_1$.

To simplify our implementation of the RNA design problem, we change the objective function of the RNA secondary structure prediction: instead of minimizing the free energy, we aim to maximize the bound energy. This change is reasonable because a biological system binding a maximum amount of energy equally minimizes the amount of free energy that is inherent to that system.

Our energy model is rather simple: for each canonical base pair in a secondary structure, we assign an energy value representing the amount of energy bound by that pair. Non-canonical base pairs are excluded by fixing their bound energy to $-\infty$; this simulates an infinite amount of free energy.

Definition 7.30 $\mathsf{engy}_{\mathsf{base}} : \{A,C,G,U\}^2 \to \mathbb{N} \cup \{-\infty\}$ assigns an energy value to a pair $(x, y) \in \{A, C, G, U\}^2$ of bases and is defined by:

$$\mathsf{engy}_{\mathsf{base}}(x, y) := \begin{cases} 1 & \text{if } (x, y) = (G, U) \vee (x, y) = (U, G) \\ 2 & \text{if } (x, y) = (A, U) \vee (x, y) = (U, A) \\ 3 & \text{if } (x, y) = (C, G) \vee (x, y) = (G, C) \\ -\infty & \text{otherwise} \end{cases}$$

The energy values $\mathbb{N} \cup \{-\infty\}$ form a semiring, i.e., a set with associated addition and multiplication functions:

- Adding two energy values is done by taking the maximum of both values with $-\infty$ being the identity element.
- The product of two energy values is the (classic) sum of both values where $-\infty$ is the absorbing element.

In our energy model, the total energy bound by folding a primary structure into a secondary structure equals the semiring-product of the energy values $\mathsf{engy}_{\mathsf{base}}(p_i, p_j)$ for each base pair $(p_i, p_j) \in \{A, C, G, U\}^2$ in the given structure.

Definition 7.31 $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2) \in \mathbb{N} \cup \{-\infty\}$ computes the energy bound by folding the primary structure $\mathcal{S}_1 = (p_1, \dots, p_n)$ of length $n \in \mathbb{N}_{>0}$ into the secondary structure $\mathcal{S}_2 \subseteq \{1 \dots n\}^2$ and is defined by:

$$\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2) := \prod_{(i,j) \in \mathcal{S}_2} \mathsf{engy}_{\mathsf{base}}(p_i, p_j)$$

Listing 7.32 shows an excerpt of a straightforward implementation for $\mathsf{engy}_{\mathsf{base}}$ in a concrete program. Note that the type `Nat` denotes CO^4's built-in encoding for natural numbers (cf. Section 6.3).

```
data Base   = A | C | G | U
data Energy = MinusInfinity | Finite Nat

energyBase :: Base -> Base -> Energy
energyBase = \b1 b2 -> case b1 of
  A -> case b2 of
    A -> MinusInfinity
    C -> MinusInfinity
    G -> MinusInfinity
    U -> Finite (nat 2)
  ...
```

Listing 7.32: An excerpt of an exemplary implementation of $\mathsf{engy}_{\mathsf{base}}$.

Listing 7.33 illustrates how the implementation for the sum and product in the energy-semiring uses the built-in functions `maxNat` and `plusNat` for computing the maximum and sum of two natural numbers.

```
plusE :: Energy -> Energy -> Energy
plusE = \e f -> case e of
  Finite x -> case f of
    Finite y -> Finite (maxNat x y)
    MinusInfinity -> e
  MinusInfinity -> f

timesE :: Energy -> Energy -> Energy
timesE = \e f -> case e of
  Finite x -> case f of
    Finite y -> Finite (plusNat x y)
    MinusInfinity -> f
  MinusInfinity -> e
```

Listing 7.33: The implementation of the sum and product in the energy-semiring.

As the secondary structure is required to be non-crossing, we represent it as a well-formed list of parentheses in the concrete program where a `Blank` character indexes a base that is not part of any base pair. The energy $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2)$ bound by a given primary structure $\mathcal{S}_1$ that is folded into a given secondary structure $\mathcal{S}_2$ is computed using a stack automaton in the function `parse` shown in Listing 7.34.

In order to specify the RNA design problem as a concrete program, we exploit its relation to RNA secondary structure prediction (cf. Definition 7.28), i.e., instead of directly tackling the design problem, we are going to solve the structure prediction problem for an unknown primary structure $\mathcal{S}_1 = (p_1, \ldots, p_n)$ with $n \in \mathbb{N}_{>0}$. For a given secondary structure $\mathcal{S}_2$, we apply the Algebraic Dynamic Programming (ADP) framework [36] for computing the maximal amount of bound energy. To do so, we specify a matrix E over energy values with dimension $(n+1)\times(n+1)$. Note that E and the matrices mentioned below are indexed in a zero-based manner, i.e., their indices range from $(0,0)$ to (n,n). $E_{(i,j)}$ contains the maximally bound energy for the sequence $(p_{i+1}, \ldots, p_{j-1}, p_j)$ of $\mathcal{S}_1$ where $i, j \in \{0 \ldots n\}$. In general, E has the following form:

$$E = \begin{bmatrix} -\infty & 0 & E_{(0,2)} & E_{(0,3)} & E_{(0,4)} & \ldots & E_{(0,n)} \\ -\infty & -\infty & 0 & E_{(1,2)} & E_{(1,3)} & \ldots & E_{(1,n)} \\ -\infty & -\infty & -\infty & 0 & E_{(2,3)} & \ldots & E_{(2,n)} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

```
data Paren = Open | Close | Blank

boundEnergy :: List Base -> List Paren -> Energy
boundEnergy = \p s -> parse Nil p s

parse :: List Base -> List Base -> List Paren -> Energy
parse = \stack p s -> case s of
  Nil -> case stack of
    Nil -> Finite (nat 0)
    Cons z zs -> MinusInfinity
  Cons y ys -> case p of
    Nil -> MinusInfinity
    Cons x xs ->
      let stack' = case y of
            Blank -> stack
            Open  -> Cons x stack
            Close -> tail stack
          here = case y of
            Blank -> Finite (nat 0)
            Open  -> Finite (nat 0)
            Close -> energyBase (head stack) x
      in
        timesE here (parse stack' xs ys)
```

Listing 7.34: The computation of the energy bound by a primary structure `p` that is folded into a secondary structure `s` is computed using a stack automaton. Note that the functions `head` and `tail` give the first element and the trailing elements of the stack, respectively. Furthermore, the function `parse` does not check whether the list of parentheses is actually well-formed: it evaluates to $\perp$ if it is not, which effectively renders the constraint unsatisfiable (cf. Section 5.4).

According to the ADP framework, E is the pointwise least solution of the following equation:

$$E = I + E \cdot E + \sum_{x,y \in \{A,C,G,U\}} \mathsf{engy}_{\mathsf{base}}(x, y) \cdot I_x \cdot I_y \cdot G_3 \tag{7.35}$$

Note the following:

1. $+$ and $\cdot$ denote the matrix addition and multiplication over the energy semiring.
2. For $x \in \{A, C, G, U\}$, I_x denotes a $(n+1) \times (n+1)$ matrix where

$$\forall (i, j) \in \{0 \dots n\}^2 : I_{x(i,j)} = \begin{cases} 0 & \text{if } i+1 = j \text{ and } x = p_j \\ -\infty & \text{otherwise} \end{cases}$$

For example, if $\mathcal{S}_1 = (U, C, C, A)$, then

$$I_C = \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

I_x is used for selecting all bases x present in the primary structure $\mathcal{S}_1$.

3. I is defined as:

$$I = \sum_{x \in \{A,C,G,U\}} I_x$$

For example, if $\mathcal{S}_1$ contains 4 bases, then

$$I = \begin{bmatrix} -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 \\ -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

4. For $l \in \mathbb{N}_{>0}$, G_l denotes a $(n+1) \times (n+1)$ matrix where

$$\forall (i,j) \in \{0 \dots n\}^2 : G_{l(i,j)} = \begin{cases} E_{(i,j)} & \text{if } i + l \leq j \\ -\infty & \text{otherwise} \end{cases}$$

G_l enforces that at least l bases of the primary structure $\mathcal{S}_1$ are folded along its secondary structure (minimal hairpin length). For example, if $\mathcal{S}_1$ contains 4 bases and $l = 3$, then

$$G_3 = \begin{bmatrix} -\infty & -\infty & -\infty & E_{(0,3)} & E_{(0,4)} \\ -\infty & -\infty & -\infty & -\infty & E_{(1,4)} \\ -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

We rewrite Equation (7.35) to

$$E = I + E \cdot E + C \odot G_3' \tag{7.36}$$

which requires fewer operations. Note the following:

1. $A \odot B$ denotes the pointwise multiplication of two $(n+1) \times (n+1)$ matrices A and B over the energy semiring:

$$\forall (i,j) \in \{0 \dots n\}^2 : (A \odot B)_{(i,j)} = A_{(i,j)} \cdot B_{(i,j)}$$

2. C denotes a $(n+1) \times (n+1)$ matrix that contains the energy values according to $\mathsf{engy}_{\mathsf{base}}$ for all valid pairings of the bases in $\mathcal{S}_1$:

$$\forall (i,j) \in \{0 \ldots n\}^2 : C_{(i,j)} = \begin{cases} \mathsf{engy}_{\mathsf{base}}(p_{i+1}, p_j) & \text{if } i+1 < j \\ -\infty & \text{otherwise} \end{cases}$$

For example, if $\mathcal{S}_1$ contains 4 bases, then

$$C = \begin{bmatrix} -\infty & -\infty & \mathsf{engy}_{\mathsf{base}}(p_1,p_2) & \mathsf{engy}_{\mathsf{base}}(p_1,p_3) & \mathsf{engy}_{\mathsf{base}}(p_1,p_4) \\ -\infty & -\infty & -\infty & \mathsf{engy}_{\mathsf{base}}(p_2,p_3) & \mathsf{engy}_{\mathsf{base}}(p_2,p_4) \\ -\infty & -\infty & -\infty & -\infty & \mathsf{engy}_{\mathsf{base}}(p_3,p_4) \\ -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

3. For $l \in \mathbb{N}_{>0}$, G'_l denotes a $(n+1) \times (n+1)$ matrix that is an index-shifted version of G_l:

$$\forall (i,j) \in \{0 \ldots n\}^2 : G'_{l(i,j)} = \begin{cases} G_{l(i+1,j-1)} & \text{if } i < n \text{ and } j > 0 \\ -\infty & \text{otherwise} \end{cases}$$

For example, if $\mathcal{S}_1$ contains 4 bases, then

$$G'_l = \begin{bmatrix} -\infty & G_{l(1,0)} & G_{l(1,1)} & G_{l(1,2)} & G_{l(1,3)} \\ -\infty & G_{l(2,0)} & G_{l(2,1)} & G_{l(2,2)} & G_{l(2,3)} \\ -\infty & G_{l(3,0)} & G_{l(3,1)} & G_{l(3,2)} & G_{l(3,3)} \\ -\infty & G_{l(4,0)} & G_{l(4,1)} & G_{l(4,2)} & G_{l(4,3)} \\ -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

Now that we have seen how to compute the energy matrix, we give the constraint c_{design} for solving the aforementioned instance of the RNA design problem through RNA secondary structure prediction.

$$c_{\mathsf{design}}(\mathcal{S}_2, (\mathcal{S}_1, E)) \coloneqq \begin{cases} \mathsf{True} & \text{if } E \text{ is the pointwise least solution of} \\ & \text{Equation (7.36) and } \mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2) = E_{(0,n)} \\ \mathsf{False} & \text{otherwise} \end{cases}$$

Note that c_{design} expects the primary structure $\mathcal{S}_1$ containing $n \in \mathbb{N}_{>0}$ bases. Listing 7.37 shows an excerpt of a concrete program that implements c_{design}.

Evaluation

In Example 7.38, we show an exemplary primary structure found by CO^4 for a given secondary structure using the concrete program in Appendix C.7.

```
constraint :: List Paren
           -> Pair (List Base) (List (List Energy))
           -> Bool
constraint = \secondary u -> case u of
  Pair primary e ->
    let c1 = geEnergy (boundEnergy primary secondary)
                      (upright e)
        c2 = matrixAll eqEnergy e
                       (energyM primary e)
        c3 = matrixAll eqEnergy e
                       (gap (S Z) MinusInfinity e)
    in
      and2 c1 (and2 c2 c3)

energyM :: List Base -> List (List Energy)
        -> List (List Energy)
energyM = \p m ->
  let mInfty = MinusInfinity
  in sum
    (Cons (item mInfty (Finite (nat 0)) p)
    (Cons (product (Cons m (Cons m Nil)))
    (Cons (pointwise timesE
            (costM MinusInfinity p)
            (matrixShift mInfty (gap (S (S (S Z)))
                                     mInfty m)))
     Nil)))
```

Listing 7.37: An excerpt of the implementation of c_{design}. The complete listing can be found in Appendix C.7.

Example 7.38 Assume the secondary structure

$$\mathcal{S}_2 = \left\{ \begin{array}{c} (1,30),(2,29),(3,28),(4,27),(5,26), \\ (9,24),(10,23),(11,22),(12,21),(13,20) \end{array} \right\}$$

that corresponds to the following list of parentheses:

$$(((((\_\_\_(((((\_\_\_\_\_\_)))))\_)))))$$

For $n = 30$, CO^4 finds the primary structure

$$\mathcal{S}_1 = \begin{pmatrix} U,U,U,G,A,G,G,G,G,G,A,U,G,G,G, \\ U,G,G,G,U,A,U,U,U,G,U,C,G,G,G \end{pmatrix}$$

by generating a propositional formula with 227151 variables, 1157736 clauses, and 3677312 literals, which is solved by MiniSat (version 2.2) in 43 s on a

3.2 GHz CPU. The bound energy is $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2) = 15$. The complete energy matrix can be found in Appendix C.8.

Using CO^4 actually allows us to exploit the inverse relation of RNA secondary structure prediction and RNA design simply by switching the primary and secondary structure in the constraint c_{design}.

$$c_{\mathsf{prediction}}(\mathcal{S}_1, (\mathcal{S}_2, E)) \coloneqq c_{\mathsf{design}}(\mathcal{S}_2, (\mathcal{S}_1, E))$$

Example 7.39 shows a solution of the secondary structure prediction problem by implementing $c_{\mathsf{prediction}}$ using the almost identical concrete program that implements c_{design}.

Example 7.39 Assume the primary structure $\mathcal{S}_1$ found in Example 7.38. CO^4 finds a secondary structure

$$\mathcal{S}_2 = \left\{ \begin{array}{c} (2,30),(3,29),(4,26),(5,23),(11,12), \\ (14,22),(16,18),(20,21),(24,25),(27,28) \end{array} \right\}$$

by generating a propositional formula with 202287 variables, 1056154 clauses, and 3376251 literals, which is solved by MiniSat (version 2.2) in 0.1 s on a 3.2 GHz CPU. The found secondary structure corresponds to the following list of parentheses:

_((((_____()_(_(_)_()))())()))

Unsurprisingly, the bound energy $\mathsf{engy}(\mathcal{S}_1, \mathcal{S}_2)$ equals 15 again, although a different secondary structure is found than in Example 7.38.

Note that we do not evaluate the present approach to domain-specific tools from the area of bioinformatics as they apply more complex energy models that we do not support in this proof of concept.

Chapter 8

Related Work

In this section, we compare CO^4 to the following techniques and tools used for constraint solving: Ersatz, MiniZinc, Prolog, and Answer Set Programming (ASP). To a certain degree, each of them provides a way of specifying constraints in a high-level constraint specification language. Thus, our comparison is based on the features inherent to the constraint specification languages of these solvers. Note that we do not compare runtime-performances. Table 8.1 gives an overview of all surveyed solvers with respect to a selected set of language features.

In Section 8.1, we describe the considered set of features and motivate their relevance and benefits for a constraint specification language. In Section 8.2, each solver is briefly introduced in order to highlight their respective advantages and differences to CO^4. For comparing the constraint specification languages of the surveyed constraint solvers, we specify two easy constraints in each language:

1. The specification of the constraint

$$c(p,(a,b)) = \begin{cases} \mathsf{True} & \text{if } p = (a \cdot b) \wedge (a > 1) \wedge (b > 1) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

   has been introduced in Example 6.16 and serves as introductory example that does not incorporate any structured data, and therefore is no typical use-case for the CO^4 constraint solver. We use it here nonetheless as it is concise and comprehensible.

2. Listing 8.2 shows the CO^4 specification of the second constraint, which is equally trivial but incorporates a structured domain of discourse `Pixel` and parameter domain `Bool`, where

$$\mathbb{C}_{\mathtt{Bool}} = \{\mathtt{False}, \mathtt{True}\}$$

$$\mathbb{C}_{\mathtt{Pixel}} = \left\{ \begin{matrix} \mathtt{Foreground\ Red}, \mathtt{Foreground\ Green}, \mathtt{Foreground\ Blue}, \\ \mathtt{Background\ Black}, \mathtt{Background\ White} \end{matrix} \right\}$$

	Structured types	Pattern matching	Autom. support for user-defined types	Purely declarative	Infinite domains	Static type system	Polymorphic types	Partial functions	Higher-order functions	Local abstractions	Module system
CO^4	●	●	●	●	P	●	P	P	P	P	
Ersatz	P	P	P	●	P	●	P	P	P	P	●
MiniZinc						●	P	P			●
Prolog	●	●			●		-	-	-	-	●
ASP	○	-		●			-	-	-	-	○

Table 8.1: Comparison of CO^4 with other constraint solvers with respect to the features of their constraint specification languages. For each feature, the symbols have the following meaning: ● denotes full feature support, P denotes full feature support but only for the parameter domain, and ○ denotes partial feature support. Whereas a missing symbol denotes a lack of support for the particular feature, the symbol - indicates that a feature is missing for conceptual reasons.

8.1 Surveyed Language Features

In this section, we briefly describe the features of the constraint specification languages considered in the comparison of CO^4 with other constraint solvers. Although some of them have already been introduced in this thesis, we will restate them here for the sake of completeness.

Structured types A constraint specification language that supports structured types allows specifying constraints that incorporate structured and hierarchical data like lists and trees. If those types are even supported for the domain of discourse, then the corresponding solver can solve constraints on structured data as well. Without such support for structured domains of discourse, any non-flat data is required to be manually flattened, which is not only error-prone but also introduces complexity to the constraint specification.

Pattern matching Pattern matching is very useful for inspecting the shape of data and for the conditional evaluation of expressions (cf. Section 2.2). This is especially helpful when matching on structured data as inspecting such data without the help of pattern matching would require manual traversals, which are tedious to write and error-prone (especially without a static type system). Thus,

```
data Bool       = False | True
data Color      = Red | Green | Blue
data Monochrome = Black | White
data Pixel      = Foreground Color
                | Background Monochrome

constraint :: Bool -> Pixel -> Bool
constraint p u = case p of
  False -> case u of Background b -> True
                     Foreground f -> False
  True -> isBlue u

isBlue :: Pixel -> Bool
isBlue u = case u of
  Background b -> False
  Foreground f -> case f of
    Red   -> False
    Green -> False
    Blue  -> True
```

Listing 8.2: The CO^4 specification of the second constraint used for comparing the surveyed constraint specification languages. For the parameter `p` = `False`, the concrete values `Background Black` and `Background White` are solutions. For the parameter `p` = `True`, the concrete value `Foreground Blue` is the only solution.

pattern matching supports specifications of constraints on structured domains of discourse.

Automatic support for user-defined types Each of the surveyed constraint specification languages supports a set of built-in data types. Additionally, users often want to define problem-specific types for a particular constraint. If user-defined types are supported, then the built-in types can be composed and combined in order to define new types. If there is no support for user-defined types, then each value in a constraint must be represented using the built-in types. Depending on the structure of the value, this may become tedious and introduce type-errors.

Purely declarative A constraint specification language that is purely declarative does not contain entities that entail implicit side-effects. For example, constraints written in a declarative language do not impose any strategy for finding a solution. Therefore, purely declarative languages often have more intuitive semantics. On the other hand, some constraint specification languages which are *almost* purely declarative still contain a few non-declarative entities in order to speed up the solving process. As this is often the only way of using these languages for non-trivial constraints, we do not consider these languages as purely declarative in the present comparison.

Infinite domains A language that allows to specify constraints over infinite domains does not impose any size restrictions on the domain of discourse. This is useful for specifying constraints over recursively defined domains such as lists and trees. For constraint specification languages that do not feature infinite domains of discourse, the domain of discourse must either be finite by design or restricted to a finite subset.

Static type system For a constraint specification language that is governed by a static type system, the type of the value that each expression evaluates to can be derived statically, i.e., before running the actual program or solving the constraint. For each constraint written in a statically typed specification language, the type system guarantees that there will not be any type errors during runtime. This is a very strong assumption that cannot be made for dynamically typed languages. Thus, a static type system supports the development of complex software by eliminating a large number of potential runtime errors. The type inference of most functional programming languages stems from the Hindley–Damas–Milner algorithm [22].

Polymorphic types A constraint specification language that is governed by a type system supporting polymorphic types allows a single entity in a constraint to have multiple types. There are different kinds of polymorphism, e.g., parametric polymorphism and ad-hoc polymorphism, with each of them providing different benefits [64]. Parametric polymorphism allows a single entity to be generically typed while having the same operational semantics for all type instances, e.g., a function that operates in the same way on data of different types (cf. function `mapMaybe` in Example 5.4). On the other hand, ad-hoc polymorphism associates a single entity to different operational semantics for different types, e.g., function overloading is a popular form of ad-hoc polymorphism. In general, polymorphic types support the expression of abstract concepts by hiding unnecessary details.

Partial functions A constraint specification language that supports partial functions allows functions to be undefined for certain arguments. Not requiring totality for each function is often necessary when specifying non-trivial constraints in a concise manner. A lack of support for partial functions can be worked around by introducing a distinct value that indicates a missing function value, but this solution has drawbacks on its own (cf. Section 5.4).

Higher-order functions In a constraint specification language that supports higher-order functions, a function may be passed as an argument to another function, or may be returned as the result of a function. Higher-order functions are a powerful tool that increases the modularity and composability of constraints (cf. Example 5.4). Thus, many modern programming languages (e.g., Haskell) support higher-order functions. In a language lacking higher-order functions, each argument must be a non-functional value.

Local abstractions In a constraint specification language supporting local abstractions, abstractions may appear not only at the top-level of a specification, but may also be nested with other expressions (cf. Example 5.6). This is useful for expressing a tight coupling of several interconnected constraint entities. Without local abstractions, each abstraction must be defined on the top-level of a specification.

Module system A constraint specification language providing a module system allows constraints being composed of multiple modules. Bundling interconnected entities in modules supports the Separation-of-Concerns paradigm of modern software engineering. In a language lacking a module system, the whole constraint must be defined in a single file, which clearly becomes confusing for more complex constraints.

8.2 Surveyed Constraint Solvers

In this section, we briefly introduce the constraint solvers that have been listed in Table 8.1.

8.2.1 Ersatz

Ersatz [51] is a Haskell library for specifying constraints using an embedded domain specific language. Similar to CO^4, constraints written in Ersatz are solved by transformation to propositional formulas.

Example 8.3 shows a simple constraint written in Ersatz.

Example 8.3 We give a Haskell program that implements the constraint from Example 3.9 using the Ersatz library.

```
import Prelude hiding ((&&))
import Ersatz
import Ersatz.Bits
import Control.Monad

constraint :: Bits -> (Bits, Bits) -> Bit
constraint p (u1, u2) = (u1 /== 1) && (u2 /== 1)
                                   && (u1 * u2 === p)

main :: IO ()
main = do
  (Satisfied, Just solution) <- solveWith minisat $ do
    let p = encode 143
    u1 <- liftM Bits (replicateM 5 exists)
    u2 <- liftM Bits (replicateM 5 exists)
```

```
  assert (constraint p (u1,u2))
  return (p,u1,u2)

putStrLn $ show solution
```

Note that Ersatz uses non-standard comparison operators, e.g., `/==`, and logical connectives.

Unlike concrete programs for CO^4, constraints written using Ersatz are not compiled into an intermediate representation. Instead, they are immediately compiled by the Haskell compiler. This has several advantages. Most importantly, Ersatz enables users to use a larger subset of Haskell than the present implementation of CO^4 does. As Ersatz constraints are standard, purely declarative Haskell code, they not only share Haskell's syntax but its semantics as well. This is beneficial as it enables even advanced features of the Haskell language for constraint programming, e.g., type classes. Furthermore, it is straightforward for users to apply Ersatz if they are familiar with Haskell.

Language Features

We discuss Ersatz' language features with respect to Table 8.1.

Structured types In Ersatz, the domain of discourse may only be a Boolean, a list of Booleans of some fixed length, or a natural number. Thus, structured types are only supported for the parameter domain.

Pattern matching Due to the lack of support for structured domains of discourse, pattern matching is not supported for the domain of discourse. Note that pattern matching is supported for all values not included in the domain of discourse.

Automatic support for user-defined types Ersatz does not support user-defined types. Thus, structured data must be flattened manually and represented using the natively supported types (cf. Example 8.4).

Example 8.4 As Ersatz does not support values of user-defined algebraic data types, we have to simulate them by explicitly encoding their constructor indices using natural numbers (cf. Definition 3.35). The following Ersatz constraint specifies the CO^4 constraint given in Listing 8.2. In order to represent the unknown value from the original domain of discourse `Pixel`, we introduce three variables `pixel`, `color`, and `monochrome` with each encoding a constructor index in binary. The values of these variables are to be determined by a SAT solver.

```
import Prelude hiding ((&&))
import Ersatz
import Ersatz.Bits
```

```
import Control.Monad

foreground = encode 0 :: Bits
background = encode 1 :: Bits

red   = encode 0 :: Bits;  black = encode 0 :: Bits
green = encode 1 :: Bits;  white = encode 1 :: Bits
blue  = encode 2 :: Bits

constraint :: Bool -> (Bits, Bits, Bits) -> Bit
constraint p (pixel, color, monochrome) =
  if p then isBlue (pixel, color)
       else pixel === background

isBlue :: (Bits, Bits) -> Bit
isBlue (pixel, color) =
  ((pixel === background) ==> false) &&
  ((pixel === foreground) ==> color === blue)

main :: IO ()
main = do
  (Satisfied, Just solution) <- solveWith minisat $ do
    let p = True
    pixel <- liftM Bits (replicateM 1 exists)
    color <- liftM Bits (replicateM 2 exists)
    monochrome <- liftM Bits (replicateM 1 exists)
    assert (constraint p (pixel, color, monochrome))
    return (pixel, color, monochrome)

  putStrLn $ show solution
```

As Ersatz constraints neither support case distinctions nor `if-then-else` expressions on values of the domain of discourse, the original case distinction in the function `isBlue` has been replaced by an expression that simulates the semantics of CO^4's built-in function merge (cf. Definition 4.41).

Purely declarative As Ersatz provides a domain specific language embedded in Haskell, it inherits Haskell's purity with respect to implicit side-effects. For the same reason, Ersatz supports all the powerful features of Haskell's **static type system** (including advanced concepts like type classes), **polymorphic types**, **partial functions**, **higher-order functions**, and **local abstractions** as long as they do not appear in the domain of discourse. Consequently, Haskell's **module system** is supported as well.

Infinite domains As Ersatz constraints are transformed into satisfiability problems in propositional logic, infinite domains of discourse are not supported. Note that the parameter domain may be infinite.

8.2.2 MiniZinc

MiniZinc 2.0 [60] is an interpreted constraint specification language that was designed in order to establish a standard language for constraint solvers. Example 8.5 gives a simple constraint specified in MiniZinc.

Example 8.5 We give an implementation of the constraint from Example 3.9 in the MiniZinc language:

```
par int: p;
var int: a;
var int: b;

constraint ((a*b) == p) /\ (a > 1) /\ (b > 1);

solve satisfy;
```

Note that MiniZinc differentiates between variables (keyword `var`) and parameters (keyword `par`). Whereas variables are to be determined by the solver, the value of each parameter is fixed by the user. This concept resembles the differentiation between the domain of discourse and the parameter domain in CO^4.

A MiniZinc constraint is a predicate on an arbitrary set of variables. In general, the specification of the MiniZinc language does not enforce any particular solving procedure for finding satisfying assignments for the involved variables, but non-declarative and solver-specific annotations may be added to the constraint in order to influence the solving procedure. Before solving a constraint written in MiniZinc, the constraint is transformed to an intermediate representation called FlatZinc, which is then solved by an external solver. Example 8.6 shows an exemplary FlatZinc constraint.

Example 8.6 The following listing shows the constraint from Example 8.5 after transforming it to its FlatZinc representation with the parameter `p` being fixed to the value 102:

```
var int: a:: output_var;
var int: b:: output_var;
var int: X_INTRODUCED_0 ::var_is_introduced ::
  is_defined_var;

constraint int_eq(X_INTRODUCED_0,102);
constraint int_le(2,a);
constraint int_le(2,b);
```

```
constraint int_times(a,b,X_INTRODUCED_0)::
  defines_var(X_INTRODUCED_0);

solve satisfy;
```

Note that parameters in MiniZinc constraints always need to be fixed when transforming the constraint to FlatZinc. This is contrary to CO^4, where the compilation of concrete to abstract programs is independent of the parameter's value.

Similar to MiniZinc, CO^4 provides a similar decoupling of a constraint's specification from the search for its solution. But whereas CO^4 only aims at SAT solvers, any solver that supports FlatZinc can be used when solving constraints specified in MiniZinc, irrespective of its solving strategy. And because of FlatZinc being much more expressive than SAT, there may exist a number of viable search strategies for solving a given constraint. It is a great advantage of MiniZinc that all these different strategies can be applied without additional efforts. For this very reason, Section 9.4 briefly discusses the possible benefits of a FlatZinc backend for CO^4.

Language Features

We discuss MiniZinc's language features with respect to Table 8.1.

Structured types MiniZinc does not support structured types.

Pattern matching MiniZinc does not support pattern matching.

Automatic support for user-defined types MiniZinc does not support user-defined types. Thus, structured data must be flattened manually and represented using the natively supported types (cf. Example 8.7).

Example 8.7 The following MiniZinc constraint specifies the CO^4 constraint given in Listing 8.2. Similar to Example 8.4, we have to simulate values of user-defined algebraic data types by explicitly encoding their constructor indices using natural numbers (cf. Definition 3.35) because MiniZinc does not support such values natively. In order to represent the unknown value from the original domain of discourse `Pixel`, we introduce three integer variables `pixel`, `color`, and `monochrome` with each encoding a constructor index. The values of these variables are to be determined by the constraint solver. A solution for the following constraint is an assignment for these three variables such that the predicate `constraint` holds.

```
par bool: p = true;
var 0..1: pixel;
var 0..2: color;
var 0..1: monochrome;
```

```

par int: red   = 0;    par int: black = 0;
par int: green = 1;    par int: white = 1;
par int: blue  = 2;

par int: foreground = 0;
par int: background = 1;

constraint   if p
               then isBlue
               else ( pixel == background )
             endif;

var bool: isBlue =  if pixel == background
                      then false
                      else ( color == blue )
                    endif;
solve satisfy;
```

Note how the original case distinctions have been replaced by `if-then-else` expressions and integer comparisons.

Purely declarative MiniZinc allows constraints to contain non-declarative annotations for controlling the solving process of particular solver backends. Thus, MiniZinc does not provide a purely declarative constraint specification language.

Infinite domains MiniZinc neither supports infinite domains of discourse nor infinite parameter domains.

Static type system MiniZinc provides a static type system supporting the following types: primitive types like Booleans, integer numbers, and real numbers, as well as compound types like sets and arrays, where compound types may not be nested. Additionally, a compound type similar to the type `Optional` from Example 5.21 for modeling the non-existence of a primitive value has been added in MiniZinc 2.0.

Polymorphic types MiniZinc supports ad-hoc polymorphism for functions and predicates, i.e., both may be overloaded with different parameter types. However, parametric polymorphism is not supported.

Partial functions MiniZinc allows functions and predicates to be defined partially.

Higher-order functions MiniZinc does not support higher-order functions.

Local abstractions While MiniZinc allows local declarations of variables, it does not support local abstractions.

Module system MiniZinc allows constraint specifications to include other files.

This is helpful for accessing libraries and splitting constraints into separate modules.

8.2.3 Prolog

Prolog is a general-purpose programming language for logic programming [30]. In Prolog, constraints are specified via rules and facts in first order logic which are expressed as Horn clauses over terms containing variables, numbers, and atoms. The solving process for a constraint is initiated by providing a query, which itself is a conjunction of terms containing possibly free variables. A solution for the given query either consists of the answer YES together with an assignment for the free variables in the query, or the answer NO in case that no solution could be found. An answer YES indicates that the query, with all variables replaced by the values of the returned assignment, is a logical consequence of the constraint. An answer NO indicates that no assignment for the variables in the query could be found so that the instantiated query is a logical consequence of the present constraint. Example 8.8 gives a simple constraint specified in Prolog.

Example 8.8 The following Prolog constraint specifies the CO^4 constraint given in Listing 8.2.

```
is_blue(foreground(blue)).

constraint(true,U) :- is_blue(U).
constraint(false,background(black)).
constraint(false,background(white)).
```

For the query `constraint(true,U)`, the solution `U = foreground(blue)` is inferred. For the query `constraint(false,U)`, both solutions

`U = background(black)` and
`U = background(white)`

are inferred.

Checking if a query is a logical consequence of a given constraint is a semi-decidable problem because validity and unsatisfiability in first-order logic is semi-decidable as well [66]. Thus, the unsatisfiability of the query clauses in conjunction with the given constraint implies that the query is a logical consequence of that constraint. In Prolog, the proof of unsatisfiability is directed by a process called SLD resolution as resolving an empty clause for a first-order formula proves that the formula is unsatisfiable. The SLD resolution as implemented in popular Prolog interpreters dictates a deterministic search strategy: the search space is traversed in a depth-first manner by resolving query clauses from left to right with constraint clauses in the order of their appearance in the constraint.

Note that in general, Prolog does not allow the specification of finite domain constraints. However, the CLPFD library, which is available in most modern Prolog systems, enables the specification of constraints over integer domains (cf. Example 8.9).

Example 8.9 In the following, we specify the constraint from Example 3.9 in Prolog:

```
:- use_module(library(clpfd)).

constraint(P,(A,B)) :- A #> 1, B #> 1, P #= A * B.
```

The operators `#>` and `#=` are provided by the CLPFD library and specify arithmetic constraints over integers. For the query

```
constraint(20,(A,B)), labeling([ff],[A,B]).
```

the solution `A = 2, B = 10` is computed. Note that the final `labeling` predicate is required for resolving residual goals and picking an actual solution from the finite domains of the given variables `A` and `B`.

Language Features

We discuss Prolog's language features with respect to Table 8.1.

Structured types Prolog features terms (cf. Appendix A.2) for representing structured data.

Pattern matching The SLD resolving algorithm implemented in Prolog's search strategy relies on term unification [66] which subsumes pattern matching. Note that each solution in Example 8.8 assigns a value to the free variable `U` in the respective query. This value is computed by unifying terms of the query with terms in the Horn clauses from the constraint. Term unification is more powerful than pattern matching as the latter allows free variables to appear only in the pattern to match, but not in the discriminant. When unifying two terms, free variables are allowed to appear in both terms.

Automatic support for user-defined types Prolog does not support user-defined types.

Purely declarative Prolog is not a purely declarative constraint specification language for at least two reasons:

1. Constraints written in Prolog may contain constructs that affect the search for a solution, e.g., the cut operator. When specifying non-trivial constraints, these constructs are often mandatory in order to achieve competitive solver runtimes.

2. As SLD resolution resolves constraint clauses in the order of their appearance in the constraint, the search for a solution can easily be trapped in a left-recursion. For example, the following constraints

   ```
   descend(X,Y) :- child(X,Y).
   descend(X,Y) :- child(X,Z), descend(Z,Y).
   ```

   and

   ```
   descend(X,Y) :- child(X,Y).
   descend(X,Y) :- descend(Z,Y), child(X,Z).
   ```

   and

   ```
   descend(X,Y) :- descend(Z,Y), child(X,Z).
   descend(X,Y) :- child(X,Y).
   ```

   all have different runtime behaviors. The first one works as expected together with a set of `child` facts. The second one finds all solutions by enumerating the set of `child` facts, and then loops due to the left-recursion in the second rule. The third constraints loops immediately without finding any solution due to the left-recursion in the first rule.

Infinite domains Prolog supports infinite domains through lists and trees.

Static type system Prolog does not provide a static type system.

Polymorphic types Due to the lack of a static type system, there are no polymorphic types in Prolog.

Partial functions As constraint specifications in Prolog only consist of rules and facts, there are no functions definitions. Consequently, the concept of partial functions cannot be applied to Prolog.

Higher-order functions For the same reason, the concept of higher-order functions cannot be applied to Prolog.

Local abstractions For the same reason, the concept of local abstractions cannot be applied to Prolog.

Module system Prolog allows constraint specifications to include other files. This is helpful for accessing libraries and splitting constraints into separate modules.

8.2.4 Answer Set Programming

Answer set programming (ASP) is a declarative constraint specification paradigm for logic programming [27]. Similar to Prolog, constraints are specified via rules and facts over terms containing variables, numbers, and atoms. Solutions for grounded, i.e., variable-free, ASP constraints are expressed in terms of answer sets [35]. An answer set of an ASP constraint is a minimal Herbrand model for that constraint, i.e., a set of ground terms such that no proper subset of an answer set is a Herbrand model itself. Consequently, the search for a solution of an ASP constraint is reduced to the search for one or more answer sets, which is an NP-complete problem. Searching for answer sets has certain advantages to the resolution-based approach provided by Prolog [21]:

- ASP constraints are purely declarative, i.e., the order of rules and clauses contained in an ASP constraint does not affect the search for answer sets.
- The search for answer sets always terminates. This is contrary to the SLD resolution in Prolog, which might get caught in left-recursions.
- The answer set semantics are more intuitive than Prolog is with respect to constraints that feature logical negations. In order to handle negations, Prolog extends SLD resolution to SLDNF resolution that supports *negation as failure* by providing a `not` predicate such that `not`(a) holds for a term a if the truth of a cannot be inferred via SLDNF resolution. But this approach fails in certain situations, e.g., when dealing with mutually recursive negations. However, grounded ASP constraints featuring mutually recursive negations usually have several stable models which can be found by an ASP solver.

As answer sets provide semantics for logic programs, ASP constraints resemble constraints in Prolog. Example 8.10 illustrates this for a simple constraint. However, genuine ASP constraints usually contain advanced constructs, e.g., disjunctive rules (i.e., rules with disjunctions in their heads), which are not featured by Prolog.

Example 8.10 The following ASP constraint specifies the CO^4 constraint given in Listing 8.2 and is identical to the Prolog constraint given in Example 8.8.

```
is_blue(foreground(blue)).

constraint(true,U) :- is_blue(U).
constraint(false,background(black)).
constraint(false,background(white)).
```

As this constraint contains no negations, it has exactly one stable model

$$\left\{ \begin{array}{l} \texttt{is\_blue(foreground(blue)),} \\ \texttt{constraint(true,foreground(blue)),} \\ \texttt{constraint(false,background(black)),} \\ \texttt{constraint(false,background(white))} \end{array} \right\}$$

The solutions of interest for the predicate `constraint` are included in the found stable model.

Most ASP solvers require the ASP constraint to be grounded. Grounding is a semantics-preserving operation that generates a variable-free ASP constraint and is often performed via an external grounding tool like Lparse [70] or Gringo [34]. As grounding may increase the size of the ASP constraint dramatically, domain-restricting predicates are often required in order to mitigate the increase in size (cf. Example 8.11). Furthermore, an ASP constraint, in general, cannot be specified over an infinite domain of discourse as this would lead to infinite many rules during grounding the original constraint [15]. This problem is similar to specifying a CO^4 constraint over an infinite domain while still being required to generate a finite propositional formula: as described in Section 4.1.5, we generate an incomplete abstract value in this case, which basically restricts the infinite domain of discourse to a finite subset. For ASP, there are efforts to support constraints on infinite domains for certain situations, e.g., through finitary programs [13].

Example 8.11 In the following, we specify the constraint from Example 3.9 as an ASP constraint:

```
p_domain(0..255).
a_domain(2..255).
b_domain(2..255).

#hide p_domain(X).
#hide a_domain(X).
#hide b_domain(X).

constraint(P,A,B) :- p_domain(P), a_domain(A),
                     b_domain(B), P == A * B.
```

The predicates `p_domain`, `a_domain`, and `b_domain` are domain predicates which restrict the domain of the variables `P`, `A` and `B`, respectively. The `#hide` declarations instruct the ASP solver to hide all terms matching the specified predicates when printing the found answer sets. Therefore, an ASP solver would output the following answer sets for the above constraint:

$$\left\{ \begin{array}{l} \texttt{constraint(4,2,2)}, \texttt{constraint(6,3,2)}, \texttt{constraint(8,4,2)}, \\ \texttt{constraint(10,5,2)}, \texttt{constraint(12,6,2)}, \quad \ldots \end{array} \right\}$$

Language Features

As different grounding tools support different extensions for classic logic programs, we only consider the language that is supported by Lparse [70] in the following overview (cf. Table 8.1).

Structured types ASP constraints may contain terms (cf. Appendix A.2) for representing structured data. However, in general, ASP constraints may not feature recursively defined domains as this would lead to infinite many instances during grounding.

Pattern matching ASP constraints do not require pattern matching, because due to grounding, ASP solvers operate on variable-free constraints.

Automatic support for user-defined types ASP constraints do not support user-defined types.

Purely declarative ASP constraints are specified in a purely declarative language.

Infinite domains Due to the grounding procedure, ASP constraints may only be specified over finite domains.

Static type system ASP does not provide a static type system.

Polymorphic types Due to the lack of a static type system, there are no polymorphic types in an ASP constraint.

Partial functions As ASP constraints only consist of rules and facts, there are no function definitions. Consequently, the concept of partial functions cannot be applied to ASP.

Although grounding tools like Lparse can be dynamically linked against shared libraries written in C or C++ in order to support user-defined functions, we deliberately ignore such capabilities in this overview as they are not part of the actual constraint specification language.

Higher-order functions For the same reason, the concept of higher-order functions cannot be applied to ASP.

Local abstractions For the same reason, the concept of local abstractions cannot be applied to ASP.

Module system ASP does not support a module system, but grounding tools like Lparse offer grounding of several constraints into a single resulting constraint. This is helpful for splitting constraints into separate files.

Chapter 9

Directions for Future Work

This section addresses several features that are missing in the present implementation of CO^4. Section 9.1 describes an alternative evaluation strategy for abstract programs that incrementally queries the backend SAT solver. This strategy might generate smaller propositional formulas, which could lead to shorter solver runtimes.

Section 9.2 illustrates a static complexity analysis of concrete programs in order to predict the expected size of the resulting propositional formula. Such a complexity analysis is useful for evaluating how much each part of a given concrete program contributes to the resulting propositional formula.

Section 9.3 covers advanced language features that are not supported in the present implementation of CO^4, e.g., type-classes known from the Haskell language. Adding support for these features in the compilation from concrete to abstract programs would allow the user to specify even more expressive and concise constraints.

Finally, Section 9.4 illustrates how CO^4 could benefit from supporting other solver backends than SAT solvers.

9.1 Incremental Solving

According to Section 4.3, finding a solution for a concrete program $c \in \textsc{Prog}$ and a given parameter essentially consists of four steps:

1. compiling c to an abstract program $c_{\mathbb{A}} \in \textsc{Prog}_{\mathbb{A}}$,
2. evaluating $c_{\mathbb{A}}$ to an abstract value $a \in \mathbb{A}$,
3. using a SAT solver to solve the propositional formula $f \in \mathrm{F}$ represented by the single flag in a, and

4. decoding the final solution from a satisfying assignment for f.

This section briefly introduces a technique for mitigating the separation of Step 2 from Step 3 where the SAT solver will already be queried when evaluating the abstract program $c_{\mathbb{A}}$. This is especially useful when evaluating compiled branches of case distinctions. We believe that incorporating the SAT solver into the process of evaluating $c_{\mathbb{A}}$ leads to smaller formulas that can be solved faster.

Recall that for evaluating a compiled case distinction, all compiled branches are evaluated and their respective results are eventually merged (cf. Definition 4.49). Evaluating all branches is necessary because, in general, the abstract value $v_d \in \mathbb{A}$ of the case distinction's discriminant may represent more than one concrete value. In case v_d is representing a single concrete value, only the corresponding branch needs to be evaluated (cf. Lemma 4.43).

But there are situations where v_d represents more than one concrete value, and yet not all branches of the compiled case distinction need to be evaluated. Example 9.1 illustrates the general pattern where such an optimization is applicable: when evaluating the compiled branch of a case distinction, the discriminant can be assumed to have a fixed value, thus, any nested case distinction on that discriminant can be simplified.

Example 9.1 Consider the following (hypothetical) declarations where $d, f, g, h \in$ Exp denote arbitrary sub-expressions.

```
data T = A | B

e = case d of
      A -> f
      B -> case d of
        A -> g
        B -> h
```

Note that the value of `e` does not depend on the value of g at all because it will not be evaluated for any value of discriminant d. Consequently, `e` can be rewritten to `e'` without changing its dynamic semantics:

```
data T = A | B

e' = case d of
       A -> f
       B -> h
```

For this reason, it is not necessary to evaluate the corresponding branch in the compilation of `e`. Depending on the complexity of g, not evaluating the compilation of g may save variables and clauses in the resulting propositional formula.

We believe that these kind of situations frequently occur when evaluating abstract programs, albeit in more complex contexts than illustrated in Example 9.1.

The trivial case distinction depicted in Example 9.1 could actually be rewritten in a preprocessing step similar to dead code elimination techniques found in optimizing compilers [3]. Such a preprocessing step is not feasible for more complex situations. Thus, we propose a strategy of evaluating abstract programs where the value of a case distinction's discriminant is used for determining whether a particular compiled branch needs to be evaluated.

One thinkable approach is to apply the incremental solving feature found in SAT solvers like MiniSat. Incremental solving allows assumptions to be propagated to the solver which are only valid for a certain number of solver invocations [26]. The solver can then be queried for the satisfiability status of the generated propositional formula under the propagated set of assumptions. Such a query is often constrained by resource restrictions, e.g., a fixed number of propagation steps or a timeout, in order to prevent a complete solver run. Because of these resource restrictions, the solver might not be able to give a definite answer.

In Example 9.2, we illustrate how incremental solving can be applied for evaluating compiled case distinctions.

Example 9.2 We give the compilation of the original formulation of the case distinction in Example 9.1 according to Definition 4.49:

$$
\begin{array}{l}
\texttt{let}\ v_{d1}\ \texttt{=}\ [\![\mathsf{compile}_{\mathrm{EXP}}(d)]\!] \\
\texttt{in valid}\ v_{d1}\ \texttt{(} \\
\quad \texttt{let}\ v_1\ \texttt{=}\ [\![\mathsf{compile\text{-}branch}_{v_{d1}}(f)]\!] \\
\qquad\ \ v_2\ \texttt{=}\quad \texttt{let}\ v_{d2}\ \texttt{=}\ [\![\mathsf{compile}_{\mathrm{EXP}}(d)]\!] \\
\qquad\qquad\qquad \texttt{in valid}\ v_{d2}\ \texttt{(let}\ v_3\ \texttt{=}\ [\![\mathsf{compile\text{-}branch}_{v_{d2}}(g)]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ v_4\ \texttt{=}\ [\![\mathsf{compile\text{-}branch}_{v_{d2}}(h)]\!] \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{merge}_{v_{d2}}(v_3, v_4)\ \texttt{)} \\
\quad \texttt{in} \\
\qquad \mathsf{merge}_{v_{d1}}(v_1, v_2) \\
\texttt{)}
\end{array}
$$

When evaluating the right-hand side of v_2, we want to temporarily fix the value of v_{d1} to $\mathsf{encode}_{\mathsf{T}}(\mathtt{B})$ because the value of v_2 corresponds to the branch where discriminant d matches value `B` in the concrete expression. Such a fixing induces an assignment $\sigma_{v_2} = \{(x_1, b_1), \ldots, (x_n, b_n)\}$ of Boolean values $b_1, \ldots, b_n \in \mathbb{B}$ to the $n \in \mathbb{N}$ propositional variables $x_1, \ldots, x_n \in \mathrm{V}$ in the flags of v_{d1}. The assignment σ_{v_2} denotes the set of assumptions for evaluating v_2. We use the SAT solver for checking whether the propositional formula $f \in \mathrm{F}$, which has been generated so far, is still satisfiable under the assumptions denoted by σ_{v_2}. We only evaluate v_2 if the solver does not give UNSAT.

Let us assume that f is not unsatisfiable under σ_{v_2}. Eventually we want to evaluate the right-hand side of v_3. This time we fix the value of v_{d2} to $\mathsf{encode}_{\mathtt{T}}(\mathtt{A})$ because the value of v_3 corresponds to the branch where discriminant d matches value `A` in the original case distinction. Again, this induces an assignment σ_{v_3} of Boolean values to the propositional variables in the flags of v_{d2}. Now we query the satisfiability of f under σ_{v_2} and σ_{v_3}. As v_{d1} equals v_{d2}, both sets of assumptions cannot hold simultaneously. Thus, the query yields UNSAT and we do not need to evaluate the right-hand side of v_3.

Again, we believe that this optimized evaluation of compiled case distinctions is beneficial, especially for constraints over highly-structured domains, as these constraints often contain many nested case distinctions.

9.2 Static Complexity Analysis

In this section, we briefly illustrate an approach for a static complexity analysis on concrete programs. By developing such an analysis, we hope to be able to estimate the complexity of the propositional formula that is generated by CO^4 for a given concrete program.

For the following overview, we do not define a particular complexity measure. Thinkable measures include the number of variables or clauses in the final propositional formula.

Deriving the complexity of a concrete program essentially requires an analysis of the structure of compiled case distinctions in the corresponding abstract program. That is because case distinctions are the single control-flow feature in concrete programs. In the following, we briefly address the importance of case distinctions and their discriminants for complexity analysis.

In general, case distinctions in a concrete program are compiled in such a way that all branches are evaluated and the resulting abstract values are eventually merged (cf. Section 4.2). The flags of the resulting abstract value contain propositional formulas that represent the result of the original case distinction in terms of propositional variables and logical connectives (cf. Example 4.42). Thus, the merge operation is the single source that increases the complexity of the final propositional formula.

Because of Lemma 4.43, evaluating compiled case distinctions on discriminants that represent only a single concrete value can be simplified to the evaluation of a single branch, i.e., no merge operation is necessary. This also reduces the complexity of the resulting propositional formula to the complexity induced by that single branch. In general, this simplification is not possible for compiled case distinctions whose discriminants represent more than one concrete value. Thus, the complexity of the propositional formula that is passed to the SAT solver significantly depends on the kind of discriminants occurring in an abstract program.

Example 9.3 illustrates both kinds of case distinctions.

Example 9.3 In the following concrete program $c_1 \in \textsc{Prog}$, the case distinction on discriminant `u` has two branches $f, g \in \textsc{Exp}$:

```
data U = U1 | U2

constraint = \p u -> case u of
  U1 -> f
  U2 -> g
```

Compare c_1 to the following concrete program $c_2 \in \textsc{Prog}$ that contains the same case distinction, but with parameter `p` being its discriminant:

```
data P = P1 | P2

constraint = \p u -> case p of
  P1 -> f
  P2 -> g
```

Both corresponding abstract programs $c_{\mathbb{A}_1}, c_{\mathbb{A}_2} \in \textsc{Prog}_{\mathbb{A}}$ are almost identical. $c_{\mathbb{A}_1}$ is

```
constraint_𝔸 = \p u ->
  let v_d = u
  in
    valid_v_d ( let v_1 = ⟦compile-branch_v_d(f)⟧
                    v_2 = ⟦compile-branch_v_d(g)⟧
                in
                  merge_v_d v_1 v_2 )
```

and $c_{\mathbb{A}_2}$ is

```
constraint_𝔸 = \p u ->
  let v_d = p
  in
    valid_v_d ( let v_1 = ⟦compile-branch_v_d(f)⟧
                    v_2 = ⟦compile-branch_v_d(g)⟧
                in
                  merge_v_d v_1 v_2 )
```

The main difference between $c_{\mathbb{A}_1}$ and $c_{\mathbb{A}_2}$ concerns their dynamic semantics. When evaluating $c_{\mathbb{A}_2}$, Lemma 4.43 applies because `v_d` = `p` and the abstract value `p` represents only a single concrete value (cf. Section 4.3). On the other hand, Lemma 4.43 does not apply for $c_{\mathbb{A}_1}$ because `u` may represent more than one concrete value.

The challenge of analyzing the complexity of a concrete program is the compile-time identification of case distinctions whose compilations will be affected by Lemma 4.43. Being able to differentiate these case distinctions from case distinctions on discriminants that represent more than one concrete value allows an estimate for the complexity introduced by the merge operation. Note that CO^4 already provides profiling information about both kinds of case distinctions (cf. Section 6.1).

9.3 Compilation of More Advanced Language Features

The language of concrete programs illustrated in Section 3.2 is a syntactic subset of the Haskell language. In order to make this language even more expressive, it would be beneficial to incorporate more language features of Haskell. In the following, we discuss type classes [39] as an especially useful feature that CO^4 is lacking.

Type classes are a realization of ad-hoc polymorphism. In contrast to parametric polymorphism, where a single implementation operates in the context of different types, ad-hoc polymorphism allows to write different implementations for a single interface so that the correct implementation for a particular type is chosen based on a distinct set of rules.

In its present implementation, CO^4 only supports functions that are either monomorphic or parametrically polymorphic (cf. Example 9.4).

Example 9.4 The function `tail` is polymorphic in regard to the type of the elements in the list `xs`: it operates in the exact same way for all lists.

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

tail :: List a -> List a
tail = \xs -> case xs of
  Nil -> Nil
  Cons y ys -> ys
```

Haskell provides ad-hoc polymorphism through type classes. Example 9.5 shows Haskell code where a type class is used for defining the structural equality of data.

Example 9.5 In the following Haskell code, we declare a type class `Eq` that consists of a function `eq`. Here, `eq` defines a common signature for all functions that compare two values of the same type. It is followed by a class instance declaration where `eq` is implemented for a user-defined type `Nat`.

```
data Bool = False | True
data Nat = Z | S Nat

class Eq a where
  eq :: a -> a -> Bool

instance Eq Nat where
  eq x y = case x of
    Z -> case y of
      Z -> True
      S v -> False
    S u -> case y of
      Z -> False
      S v -> eq u v
```

Values of type `Nat` can now be compared using the function `eq`.

Type classes can be hierarchic, i.e., a type class may have parent classes. This allows the construction of type hierarchies where each type implements a certain set of classes.

An interesting feature related to type classes is the automatic derivation of class instances for user-defined types. This is very useful for type classes whose instances all have a similar structure. For example, most instances for the `Eq` class in Example 9.5 are inductions over all constructor arguments of a given type. Therefore, they are very tedious to write, but rather easy to be automatically derived by the compiler.

Both features, type classes and an automatic derivation of class instances, would tremendously increase the expressiveness of the language of concrete programs, and the benefit that is provided by CO^4 as a general purpose constraint solver.

9.4 Additional Solver Backends

In the following, we discuss ideas related to CO^4's solver backend. So far, evaluating an abstract program in CO^4 eventually generates a propositional formula that is passed to an external SAT solver. The present implementation of CO^4 provides a backend only for the MiniSat solver. It would be interesting to inspect how other SAT solvers perform when solving formulas that are generated by CO^4. Thus, it would be reasonable to extend CO^4's solver interface Satchmo-core (cf. Section 4.3.3) in order to support additional SAT solvers.

While supporting other SAT solvers is an obvious extension to CO^4's solver backend, more fundamental expansions are thinkable. One of them concerns alternative representations of propositional formulas like and-inverter graphs, binary decision diagrams, and pseudo-Boolean constraints. As there are tools for finding solutions for formulas given in either representation, a comparison between all these representations would be interesting. Such a comparison should cover the runtimes needed for finding solutions for a set of benchmark problems, as well as the space complexity of each representation.

Furthermore, abstract evaluation could be changed so that a first-order formula over a certain theory is generated instead of a propositional formula. Promising theories for such a CO^4-*modulo-theory* approach are the theory of fixed-size bitvectors and the theory of linear integer arithmetic. For example, generating a first-order formula over the theory of linear integer arithmetic may prove useful for specifying constraints over infinite domains of discourse without needing to restrict the search space to a finite subset. For the theories included in the SMT-LIB standard [7], there are several tools and solvers available for solving constraints encoded in such a way.

Due to the declarative semantics of constraints in the answer set programming (ASP) paradigm (cf. Section 8.2.4), generating an ASP constraint during the abstract evaluation is an equally interesting approach. However, recursively defined types as lists and trees, which are quite common in non-trivial CO^4 constraints, must be treated specifically as the domains that appear in an ASP constraint may, in general, not be defined recursively because this would lead to infinite many instances during the grounding procedure.

In order to make use of solvers that support constraints specified using the FlatZinc language, a CO^4 backend is thinkable that emits either MiniZinc or FlatZinc constraints (cf. Section 8.2.2). Both are medium-level constraint representations that are suitable targets for compiling concrete programs into. Due to their support for primitive types, a corresponding CO^4 backend would allow the user to specify constraints that feature highly structured data, as well as primitive values like integer and real numbers.

Chapter 10

Conclusion

In the present thesis, we introduced the constraint solver CO^4 that enables a subset of Haskell to be used as a constraint specification language. We specified the essential steps of the solving process: a parameterized constraint implemented as a concrete program is firstly compiled into an intermediate representation called abstract program. Evaluating the abstract program for a given parameter gives a propositional formula that is solved by an external SAT solver. In case there is a satisfying assignment for this formula, a solution in the domain of discourse is constructed.

As it has been shown, this approach for solving constraints over structured domains is feasible and has advantages over other strategies, e.g., reuse of the established programming language Haskell for specifying concise and expressive constraints. By specifying constraints that originate from different domains (cf. Chapter 7), we demonstrated that CO^4 is a general-purpose constraint solver which makes the power of modern SAT solvers available for constraints that are hard to specify in other languages.

Because of these benefits, it is suggested to invest in the continuation of the development of CO^4. Further work is necessary in order to implement a competitive tool. While there are few tools that provide an equally expressive way of specifying constraints, manually crafted propositional encodings often outperform the encodings generated by CO^4 in terms of solver runtimes. Thus, further work should favor developments that aim at reducing these runtimes either by improving CO^4's propositional encodings or even by switching to another backend (cf. Section 9.4). As the latter option is far more invasive, the former one should be preferred.

Appendix A

Notations

A.1 Basic Notations

In this section, we introduce basic notations for common concepts used throughout this thesis.

Sets

In the present thesis, we assume familiarity with the basics of set theory. We use the common symbols for functions and relations on sets: $\cap$ (intersection), $\cup$ (union), $\setminus$ (difference), $\in$ (element-of), $\subseteq$ (subset), $\subsetneq$ (strict subset), $\supseteq$ (superset), and $\supsetneq$ (strict superset).

Additionally, we give the following notation for power sets.

Definition A.1 2^A denotes the *set of all subsets (power set)* of set A and is defined by:

$$2^A := \{A' \mid A' \subseteq A\}$$

We define some essential sets.

Definition A.2 $\mathbb{N}$ denotes the *set of natural numbers including* 0.

Definition A.3 $\mathbb{N}_{>i}$ denotes the *set of natural numbers greater than* $i \in \mathbb{N}$.

Definition A.4 $\mathbb{Z}$ denotes the *set of integer numbers.*

Definition A.5 $\mathbb{B}$ denotes the *set of Boolean values* and is defined by:

$$\mathbb{B} := \{\mathsf{False}, \mathsf{True}\}$$

Tuples

A tuple is an ordered collection of elements. In the present thesis, we use the terms *tuple* and *sequence* synonymously.

Definition A.6 $A_1 \times \cdots \times A_n$ denotes the *Cartesian product of* $A_1, \ldots, A_n$ for $n \in \mathbb{N}_{>0}$ and is defined by:

$$A_1 \times \cdots \times A_n \coloneqq \{(a_1, \ldots, a_n) \mid a_1 \in A_1 \ \wedge \ \ldots \ \wedge \ a_n \in A_n\}$$

We also deal with empty tuples.

Definition A.7 $()$ denotes the *empty tuple*.

We define the length of a tuple as the number of elements it contains.

Definition A.8 $n \in \mathbb{N}$ denotes the *length of a tuple* $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$.

For readability, we introduce the following shortcut for denoting sequences of some fixed length over a particular set.

Definition A.9 A^n denotes the *set of n-tuples over A* for $n \in \mathbb{N}$ and is defined by:

$$A^n \coloneqq \begin{cases} \{()\} & \text{if } n = 0 \\ \underbrace{A \times \cdots \times A}_{n\text{-times}} & \text{otherwise} \end{cases}$$

Occasionally, we do not want to fix the length of sequences, but deal with sequences of arbitrary length.

Definition A.10 A^* denotes the *set of all tuples over A* and is defined by:

$$A^* \coloneqq \bigcup_{i \in \mathbb{N}} A^i$$

Sometimes we need to concatenate two tuples

Definition A.11 $(x_1, \ldots, x_m) \cdot (y_1, \ldots, y_n)$ denotes the *concatenation of two tuples* for $m, n \in \mathbb{N}$ and is defined by:

$$(x_1, \ldots, x_m) \cdot (y_1, \ldots, y_n) \coloneqq (x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Relations and Functions

We define relations as sets of tuples.

Definition A.12 Any subset of $A_1 \times \cdots \times A_n$ denotes an n-ary *relation* between the sets $A_1, \ldots, A_n$ for $n \in \mathbb{N}_{>1}$.

A partial function is a binary relation that maps elements from one set to another.

Definition A.13 The relation $f \subseteq A \times B$ is a *partial function* from set A to set B if:

$$\forall((a_1, b_1), (a_2, b_2)) \in f : a_1 = a_2 \implies b_1 = b_2$$

Notation By $f : A \nrightarrow B$ we denote a partial function f from set A to B.

Functions have a particular domain.

Definition A.14 The *domain* $\mathrm{dom}(f) \subseteq A$ of a function f from set A to set B is defined by:

$$\mathrm{dom}(f) \coloneqq \{a \mid (a, b) \in f\}$$

Instead of being partial, functions can be total.

Definition A.15 The function $f \subseteq A \times B$ is a *total function* from set A to set B if $\mathrm{dom}(f) = A$.

Notation By $f : A \rightarrow B$ we denote a total function f from set A to B.

Functions can be applied to arguments.

Definition A.16 $f(a) \in B$ denotes the *application* of a total or partial function f (from set A to set B) to an argument $a \in A$ and is defined by:

$$f(a) = b \Leftrightarrow (a, b) \in f$$

Note that the application of a partial function $f : A \nrightarrow B$ to argument $a \in A$ is only defined if $a \in \mathrm{dom}(f)$.

Notation When applying a function $f : A_1 \times \cdots \times A_n \rightarrow B$ to a n-tuple $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$ for $n \in \mathbb{N}_{>1}$, we only type a single pair of parentheses, e.g., $f(a_1, \ldots, a_n)$ instead of $f\big((a_1, \ldots, a_n)\big)$.

For a given function, we sometimes want to update one of its tuples, or add a new tuple. We define a common notation for both operations.

Definition A.17 $f[c/d]$ denotes the *update* of a total or partial function f from set A to set B by a tuple $(c, d) \in A \times B$ such that:

$$\forall a \in A : f[c/d](a) = \begin{cases} d & \text{if } a = c \\ b & \text{if } a \neq c \wedge (a, b) \in f \end{cases}$$

We generalize Definition A.17 so that we can update a function using another function.

Definition A.18 $f[g]$ denotes the *update* of a total or partial function f from set A to set B by a total or partial function g from the same set A to set B:

$$f[\{(a_1, b_1), \ldots, (a_n, b_n)\}] = ((f[a_1/b_1]) \ldots)[a_n/b_n]$$

Sometimes we want to refer to the set of all total functions that map from one particular set to another set.

Definition A.19 B^A denotes the *set of all total functions* $f : A \to B$.

A.2 Terms and Algebras

In the following, we introduce terms and algebras [6] as underlying concepts of term rewriting systems (cf. Section 7.1) and propositional logic (Appendix B).

A signature captures a set of symbols and their arity.

Definition A.20 A *signature* Σ is a set of symbols where each symbol in Σ is associated with a value $\mathsf{arity} : \Sigma \to \mathbb{N}$ that denotes its arity. For all $n \in \mathbb{N}$, Σ_n denotes the greatest subset of Σ so that all symbols in Σ_n have arity n:

$$\Sigma_n \coloneqq \{f \mid f \in \Sigma \ \wedge \ \mathsf{arity}(f) = n\}$$

The set of terms is constructed using a signature and a set of variables.

Definition A.21 Given a signature Σ and a variable set X so that $\Sigma \cap X = \varnothing$, the set of Σ-*terms over* X, denoted as $\mathsf{terms}(\Sigma, X)$, equals the least set T for which the following properties hold:

1. $X \subseteq T$, i.e., each variable is a Σ-term, and
2. the application of an n-ary function symbol to n Σ-terms is a Σ-term as well:

$$\forall n \in \mathbb{N} : \ \forall (f, (t_1, \ldots, t_n)) \in \Sigma_n \times T^n : f(t_1, \ldots, t_n) \in T$$

The variable set of a Σ-term $t \in \mathsf{terms}(\Sigma, X)$ is a subset of X.

Definition A.22 The *variable set* $\mathsf{var} : \mathsf{terms}(\Sigma, X) \to 2^X$ of a Σ-term $t \in \mathsf{terms}(\Sigma, X)$ is defined by:

$$\mathsf{var}(t) \coloneqq \begin{cases} \{t\} & \text{if } t \in X \\ \bigcup_{i=1}^{n} \mathsf{var}(t_i) & \text{if } t = f(t_1, \ldots, t_n) \text{ for some } n \in \mathbb{N} \end{cases}$$

We define the root symbol of a Σ-term.

Definition A.23 The *root symbol* $\mathsf{rootsym} : \mathsf{terms}(\Sigma, X) \nrightarrow \Sigma$ of a non-variable Σ-term $t \in \mathsf{terms}(\Sigma, X)$ is defined by:

$$\forall n \in \mathbb{N} : \mathsf{rootsym}(f(t_1, \ldots, t_n)) \coloneqq f$$

Sometimes we are interested in the set of subterms of a Σ-term.

Definition A.24 $\mathsf{subterms} : \mathsf{terms}(\Sigma, X) \to 2^{\mathsf{terms}(\Sigma, X)}$ gives the *set of subterms* of a Σ-term $t \in \mathsf{terms}(\Sigma, X)$ and is defined by:

$$\mathsf{subterms}(t) \coloneqq \begin{cases} \{t\} & \text{if } t \in X \\ \{t\} \cup \bigcup_{i \in \{1 \dots n\}} \mathsf{subterms}(t_i) & \text{if } t = f(t_1, \dots, t_n) \\ & \text{for some } n \in \mathbb{N} \end{cases}$$

The depth of a term denotes the number of nestings.

Definition A.25 $\mathsf{depth} : \mathsf{terms}(\Sigma, X) \to \mathbb{N}$ gives the *depth* of a Σ-term $t \in \mathsf{terms}(\Sigma, X)$ and is defined by:

$$\mathsf{depth}(t) \coloneqq \begin{cases} 0 & \text{if } t \in X \\ 0 & \text{if } t = f() \\ 1 + \max\{\mathsf{depth}(t_1), \dots, \mathsf{depth}(t_n)\} & \text{if } t = f(t_1, \dots, t_n) \\ & \text{for some } n \in \mathbb{N}_{>0} \end{cases}$$

In order to provide some semantics for Σ-terms, we introduce Σ-algebras.

Definition A.26 For a given signature Σ, a Σ-*algebra* $\mathcal{A} = (A, [.])$ consists of

1. a carrier set (or domain) A and
2. a mapping for each n-ary function symbol $f \in \Sigma_n$ to its n-ary interpretation $[f] : A^n \to A$

Using a Σ-algebra $\mathcal{A}$ and an assignment of variables, we can evaluate Σ-terms to values of the carrier set of $\mathcal{A}$.

Definition A.27 For a given Σ-algebra $\mathcal{A} = (A, [.])$ over a signature Σ and a variable set X, $\mathsf{eval}_{\mathcal{A}} : A^X \times \mathsf{terms}(\Sigma, X) \to A$ evaluates a Σ-term $t \in \mathsf{terms}(\Sigma, X)$ to a value in A under an assignment $\sigma \in A^X$:

$$\mathsf{eval}_{\mathcal{A}}(\sigma, t) \coloneqq$$

$$\begin{cases} \sigma(t) & \text{if } t \in X \\ [f](\mathsf{eval}_{\mathcal{A}}(\sigma, t_1), \dots, \mathsf{eval}_{\mathcal{A}}(\sigma, t_n)) & \text{if } f \in \Sigma_n \text{ and } t = f(t_1, \dots, t_n) \\ & \text{for some } n \in \mathbb{N} \end{cases}$$

A.3 Term Rewriting

Term rewriting is a computational model where subterms are substituted according to a system of rules [6]. Subterms are identified by a position.

Definition A.28 $\textsc{Pos}$ denotes the *set of positions* and is defined by:

$$\textsc{Pos} \coloneqq \mathbb{N}^*_{>0}$$

Each position in $\textsc{Pos}$ is a sequence of positive natural numbers denoting the path from the root symbol of a term down to a particular subterm.

A Σ-term with a finite set of subterms has a finite set of positions.

Definition A.29 $\textsc{Pos}_t \subsetneq \textsc{Pos}$ denotes the *set of positions of term* $t \in \mathsf{terms}(\Sigma, X)$ and equals the least set P so that

1. $() \in P$ and
2. $\{(i) \cdot p \mid p \in \textsc{Pos}_{t_i}\} \subseteq P$ if $t = f(t_1, \ldots, t_n)$ for all $n \in \mathbb{N}_{>0}$ and $i \in \{1 \ldots n\}$.

Note that $\cdot$ denotes the concatenation of tuples (cf. Definition A.11).

Example A.30 For term $t = f(g(x), y)$ and $t \in \mathsf{terms}(\{f, g\}, \{x, y\})$, the set of positions $\textsc{Pos}_t$ is $\{(), (1), (1, 1), (2)\}$.

Given a position in a term, we compute the subterm at that position.

Definition A.31 $t|_p$ denotes the *subterm of* $t \in \mathsf{terms}(\Sigma, X)$ *at position* $p \in \textsc{Pos}_t$ and is defined by:

$$t|_p \coloneqq \begin{cases} t & \text{if } p = () \\ t_i|_q & \text{if } t = f(t_1, \ldots, t_n) \text{ and } p = (i) \cdot q \text{ with } i \leq n \end{cases}$$

Note that $t|_p$ is defined only for positions $p \in \textsc{Pos}_t$ of term $t \in \mathsf{terms}(\Sigma, X)$.

Example A.32 For term $t = f(g(x), y)$ with $t \in \mathsf{terms}(\{f, g\}, \{x, y\})$, the following propositions hold:

1. $t|_{()} = f(g(x), y)$
2. $t|_{(1)} = g(x)$
3. $t|_{(1,1)} = x$
4. $t|_{(2)} = y$

In the following, we replace subterms at certain positions by other terms.

Definition A.33 $t[t']_p$ denotes the *term* $t \in \mathsf{terms}(\Sigma, X)$ *after replacing subterm* $t|_p$ *at position* $p \in \textsc{Pos}_t$ *with term* $t' \in \mathsf{terms}(\Sigma, X)$ and is defined by

$$t[t']_p \coloneqq \begin{cases} t' & \text{if } p = () \\ t_i[t']_q & \text{if } t = f(t_1, \ldots, t_n) \text{ and } p = (i) \cdot q \text{ with } i \leq n \end{cases}$$

Example A.34 For the two terms $t = f(g(x), y)$ and $t' = f(x, y)$ with $t, t' \in \mathsf{terms}(\{f, g\}, \{x, y\})$, the following holds:

$$t[t']_{(1,1)} = f(g(f(x, x)), y)$$

In term rewriting, variables are substituted by terms in order to generate new terms.

Definition A.35 A *substitution* $\widehat{\sigma} : \mathsf{terms}(\Sigma, X) \to \mathsf{terms}(\Sigma, X)$ induced by a mapping $\sigma \in \mathsf{terms}(\Sigma, X)^X$ for a term $t \in \mathsf{terms}(\Sigma, X)$ is defined by:

$$\widehat{\sigma}(t) = \begin{cases} \sigma(t) & \text{if } t \in X \\ f(\widehat{\sigma}(t_1), \ldots, \widehat{\sigma}(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Example A.36 For term $t = f(g(x), y)$ with $t \in \mathsf{terms}(\{f, g\}, \{x, y\})$ and a mapping $\sigma = \{(x, x), (y, g(x))\}$, the following holds:

$$\widehat{\sigma}(t) = f(g(x), g(x))$$

Finally, we define term rewriting systems using the previously introduced concepts.

Definition A.37 A *term rewriting system* is a triple (Σ, X, R) containing a signature Σ, a set of variables X, and a set of rules $R \subseteq \mathsf{terms}(\Sigma, X)^2$ so that:

$$\forall (l, r) \in R : l \notin X \land \mathsf{var}(l) \supseteq \mathsf{var}(r)$$

Notation By $l \to r$ we denote the rule $(l, r) \in R$ of a term rewriting system (Σ, X, R).

A term rewriting system induces a rewrite relation.

Definition A.38 The *rewrite relation* $\to_R \subseteq \mathsf{terms}(\Sigma, X)^2$ induced by a term rewriting system (Σ, X, R) is defined by:

$$\forall t \in \mathsf{terms}(\Sigma, X) : \exists (l \to r) \in R : \exists p \in \mathrm{Pos}_t : \exists \sigma \in \mathsf{terms}(\Sigma, X)^{\mathsf{var}(l)} : \\ t|_p = \widehat{\sigma}(l) \implies t \to_R t[\widehat{\sigma}(r)]_p$$

Example A.39 For a term rewriting system $(\{f, g\}, \{x, y\}, R)$ with $R = \{g(y) \to f(y, y)\}$, $f(g(x), y) \to_R f(f(x, x), y)$ because the single rule in R is applicable to subterm $g(x)$ at position 1 under mapping $\{(y, x)\}$.

Appendix B

Propositional Logic

Propositional logic studies formulas built from propositions and logical connectives [66].

Definition B.1 gives the signature over which propositional formulas are defined.

Definition B.1 The *signature of Boolean formulas* $\Sigma_{\mathcal{B}}$ is defined by:

$$\Sigma_{\mathcal{B}} = \{\mathsf{False}, \mathsf{True}, \neg, \vee, \wedge\}$$

with

1. $\mathsf{arity}(\mathsf{False}) = \mathsf{arity}(\mathsf{True}) = 0$
2. $\mathsf{arity}(\neg) = 1$
3. $\mathsf{arity}(\vee) = \mathsf{arity}(\wedge) = 2$

The Boolean algebra given in Definition B.2 is the underlying concept of propositional logic.

Definition B.2 $\mathcal{B} = (\mathbb{B}, [.])$ denotes a *Boolean algebra* with $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$ and:

$$[\mathsf{False}] = \{\mathsf{False}\} \qquad [\mathsf{True}] = \{\mathsf{True}\} \qquad [\neg] = \{(\mathsf{False}, \mathsf{True}), (\mathsf{True}, \mathsf{False})\}$$

$$\begin{aligned}[\vee] = \{&((\mathsf{False}, \mathsf{False}), \mathsf{False}), \\ &((\mathsf{False}, \mathsf{True}), \mathsf{True}), \\ &((\mathsf{True}, \mathsf{False}), \mathsf{True}), \\ &((\mathsf{True}, \mathsf{True}), \mathsf{True})\}\end{aligned} \qquad \begin{aligned}[\wedge] = \{&((\mathsf{False}, \mathsf{False}), \mathsf{False}), \\ &((\mathsf{False}, \mathsf{True}), \mathsf{False}), \\ &((\mathsf{True}, \mathsf{False}), \mathsf{False}), \\ &((\mathsf{True}, \mathsf{True}), \mathsf{True})\}\end{aligned}$$

Note that False and True denote symbols in $\Sigma_{\mathcal{B}}$ as well as the elements of $\mathbb{B}$.

Propositional formulas may contain propositional variables, i.e., variables over the domain $\mathbb{B}$.

Definition B.3 The set V denotes the set of *propositional variables*.

Terms over the signature $\Sigma_{\mathcal{B}}$ and the set of variables V form the set of propositional formulas.

Definition B.4 The set of *propositional formulas* F is defined by:

$$\mathrm{F} \coloneqq \mathsf{terms}(\Sigma_{\mathcal{B}}, \mathrm{V})$$

We specify the semantical equivalence of propositional formulas.

Definition B.5 Two propositional formulas $f, g \in \mathrm{F}$ are *semantically equivalent* if they evaluate to the same value under all assignments that assign all variables of f and g:

$$f \equiv g \Leftrightarrow \left(\forall \sigma \in \mathbb{B}^{\mathsf{var}(f) \cup \mathsf{var}(g)} : \mathsf{eval}_{\mathcal{B}}(\sigma, f) = \mathsf{eval}_{\mathcal{B}}(\sigma, g)\right)$$

We define additional logical connectives by semantical equivalence:

- Implication $\implies$: $\quad \forall (f, g) \in \mathrm{F}^2 : f \implies g \equiv \neg f \vee g$
- Equivalence $\Leftrightarrow$: $\quad \forall (f, g) \in \mathrm{F}^2 : f \Leftrightarrow g \equiv (f \implies g) \wedge (g \implies f)$
- Exclusive disjunction $\oplus$: $\quad \forall (f, g) \in \mathrm{F}^2 : f \oplus g \equiv (f \wedge \neg g) \vee (\neg f \wedge g)$

B.1 SAT solver

In order to reason about the size and complexity of the propositional formulas generated during the compilation proposed in Chapter 4, we glance on some basic concepts of SAT solvers.

Firstly, we define the set of satisfiable formulas.

Definition B.6 The *set of satisfiable formulas* SAT $\subsetneq$ F contains all formulas that have an assignment under which the formula evaluates to True:

$$\mathrm{SAT} \coloneqq \{f \mid f \in \mathrm{F} \ \wedge \exists \sigma \in \mathbb{B}^{\mathsf{var}(f)} : \mathsf{eval}_{\mathcal{B}}(\sigma, f) = \mathsf{True}\}$$

An assignment under which a formula evaluates to True is called a *satisfying assignment*. A function that gives such a satisfying assignment for a formula in SAT is called SAT solver.

Definition B.7 A *SAT solver* is a partial function $\mathsf{solve} : \mathrm{F} \nrightarrow \mathbb{B}^{\mathrm{V}}$ that gives a satisfying assignment for all satisfiable formulas, i.e.,

$$\forall f \in \mathrm{SAT} : f \in \mathrm{dom}(\mathsf{solve}) \wedge \mathsf{eval}_{\mathcal{B}}(\mathsf{solve}(f), f) = \mathsf{True}$$

and is undefined for all non-satisfiable formulas:

$$\forall f \notin \mathrm{SAT} : f \notin \mathrm{dom}(\mathsf{solve})$$

Checking the satisfiability of propositional formulas is an NP-complete problem [20], thus, it is a challenging task to implement a SAT solver with a low runtime and space complexity.

Most SAT solvers deal with propositional formulas in conjunctive normal form (CNF). A formula in CNF consists of conjunctions of clauses where each clause is a disjunction of literals.

Definition B.8 The *set of literals* $\textsc{Literal} \subsetneq \mathrm{F}$ equals the least set L for which the following properties hold:

1. $\forall v \in \mathrm{V} : v \in L$, i.e., every variable is a literal with positive parity, and
2. $\forall v \in \mathrm{V} : \neg v \in L$, i.e., every variable's negation is a literal with negative parity.

We define the set of clauses.

Definition B.9 The *set of clauses* $\textsc{Clause}$ is defined by:

$$\textsc{Clause} \coloneqq \{(l_1 \vee l_2 \vee \cdots \vee l_n) \mid n \in \mathbb{N} \ \wedge \ (l_1, \ldots, l_n) \in \textsc{Literal}^n\}$$

A formula in CNF consists of conjunctions of clauses.

Definition B.10 The *set of propositional formulas in conjunctive normal form* $\mathrm{CNF} \subsetneq \mathrm{F}$ is defined by:

$$\mathrm{CNF} \coloneqq \{(c_1 \wedge c_2 \wedge \cdots \wedge c_n) \mid n \in \mathbb{N} \ \wedge \ (c_1, \ldots, c_n) \in \textsc{Clause}^n\}$$

A formula in CNF may contain an empty clause, i.e., a clause without any literal. Such a formula is always unsatisfiable.

Notation For brevity, formulas in CNF are often noted as subsets of $2^{\textsc{Literal}}$, e.g. $\{\{x_1, x_2\}, \{\neg x_3, x_4\}\}$ instead of $(x_1 \vee x_2) \wedge (\neg x_3 \vee x_4)$.

Theorem B.11 For every formula in F there is a semantically equivalent formula in CNF [66]. ■

Naively transforming a formula into a semantically equivalent formula in CNF often results in a exponential blow-up of the formula's size, which may lead to longer runtimes of the SAT solver.

Tseitin [73] introduced a linear-time method of transforming a formula f to a equisatisfiable formula g in CNF.

Definition B.12 Two formulas $f, g \in \mathrm{F}$ are *equisatisfiable* if there is a satisfying assignment for f whenever there is a satisfying assignment for g and vice versa:

$$\begin{gathered}\exists \sigma_1 \in \mathbb{B}^{\mathsf{var}(f)} : \mathsf{eval}_{\mathcal{B}}(\sigma_1, f) = \mathsf{True} \\ \Leftrightarrow \\ \exists \sigma_2 \in \mathbb{B}^{\mathsf{var}(g)} : \mathsf{eval}_{\mathcal{B}}(\sigma_2, g) = \mathsf{True}\end{gathered}$$

We give a specification for Tseitin's transformation.

Definition B.13 $\mathsf{tseitin} : \mathrm{F} \to \mathrm{CNF}$ maps a formula $f \in \mathrm{F}$ to a formula in CNF and is defined by:

$$\mathsf{tseitin}(f) \coloneqq \{\{\mathsf{fresh}(f)\}\} \cup \bigcup_{g \in \mathsf{subterms}(f)} \mathsf{sub\text{-}tseitin}(g)$$

where

1. $\mathsf{fresh} : \mathrm{F} \to \mathrm{V}$ gives a variable for each subformula of f:

$$\mathsf{fresh}(f) \coloneqq \begin{cases} f & \text{if } f \text{ is a variable, i.e., } f \in \mathrm{V} \\ v_f & \text{otherwise return a fresh variable } v_f \in \mathrm{V} \end{cases}$$

2. $\mathsf{sub\text{-}tseitin} : \mathrm{F} \to \mathrm{CNF}$ maps a subformula of f to its counterpart in CNF and is defined by:

$$\mathsf{sub\text{-}tseitin}(f) \coloneqq$$

$$\begin{cases} \{\} & \text{if } f \text{ is a variable, i.e., } f \in \mathrm{V} \\ \{\{\mathsf{fresh}(g), \mathsf{fresh}(f)\} & \text{if } f = \neg g \\ \ ,\{\neg\mathsf{fresh}(g), \neg\mathsf{fresh}(f)\} & \\ \} & \\ \{\ \{\mathsf{fresh}(g_1), \mathsf{fresh}(g_2), \neg\mathsf{fresh}(f)\} & \\ \ ,\{\neg\mathsf{fresh}(g_1), \mathsf{fresh}(f)\} & \text{if } f = g_1 \vee g_2 \\ \ ,\{\neg\mathsf{fresh}(g_2), \mathsf{fresh}(f)\} & \\ \} & \\ \{\ \{\neg\mathsf{fresh}(g_1), \neg\mathsf{fresh}(g_2), \mathsf{fresh}(f)\} & \\ \ ,\{\mathsf{fresh}(g_1), \neg\mathsf{fresh}(f)\} & \text{if } f = g_1 \wedge g_2 \\ \ ,\{\mathsf{fresh}(g_2), \neg\mathsf{fresh}(f)\} & \\ \} & \end{cases}$$

Figure 6.2 shows an example of mapping a propositional formula to its conjunctive normal form.

Theorem B.14 For all formulas $f \in \mathrm{F}$, f and $\mathsf{tseitin}(f)$ are equisatisfiable [73]. ■

Besides the equisatisfiability given in Theorem B.14, Tseitin's transformation has another important feature:

Theorem B.15 For each formula $f \in \mathrm{F}$, a satisfying assignment for $\mathsf{tseitin}(f)$ is also a satisfying assignment for f:

$$\forall (f, f') \in \mathsf{tseitin} : \ \forall \sigma \in \mathbb{B}^{\mathsf{var}(f')} :$$
$$\mathsf{eval}_{\mathcal{B}}(\sigma, f') = \mathsf{True} \implies \mathsf{eval}_{\mathcal{B}}(\sigma, f) = \mathsf{True}$$ ■

Thus, finding a satisfying assignment for a formula $f \in \mathrm{F}$ can be reduced to finding a satisfying assignment for $\mathsf{tseitin}(f)$. This is useful because there are powerful methods for finding satisfying assignments for formulas in CNF, e.g., the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [23]. The DPLL algorithm is a well-known method to check if a formula in CNF is included in SAT. It is recursively defined so that a variable is assigned to a truth value in each recursion.

Definition B.16 $f\langle v/b \rangle \in \mathrm{CNF}$ denotes a propositional formula after applying the following transformations for a value $b \in \mathbb{B}$ and a formula $f \in \mathrm{CNF}$ that contains a variable $v \in \mathrm{V}$:

1. If $b = \mathsf{True}$, remove every clause in f that contains the literal v.
2. If $b = \mathsf{True}$, remove every literal $\neg v$.
3. If $b = \mathsf{False}$, remove every clause in f that contains the literal $\neg v$.
4. If $b = \mathsf{False}$, remove every literal v.

Note that $f\langle v/b \rangle$ may contain empty clauses.

Example B.17

$$\{\{x_1, x_2\}, \{\neg x_1, x_3\}\}\langle x_1/\,\mathsf{False}\rangle = \{\{x_2\}\}$$
$$\{\{x_1, x_2\}, \{\neg x_3\}\}\langle x_3/\,\mathsf{True}\rangle = \{\{x_1, x_2\}, \varnothing\}$$

We generalize Definition B.16 to compute a formula under a partial assignment of variables.

Definition B.18 $f\langle \sigma \rangle \in \mathrm{CNF}$ assigns the value b to the variable v in a formula $f \in \mathrm{CNF}$ for all $n \in \mathbb{N}_{>0}$ tuples $(v, b) \in \mathrm{V} \times \mathbb{B}$ of a partial assignment $\sigma : \mathrm{V} \nrightarrow \mathbb{B}$:

$$f\langle \{(v_1, b_1), \ldots, (v_n, b_n)\} \rangle = ((f\langle v_1/b_1 \rangle) \ldots)\langle v_n/b_n \rangle$$

While solving a formula f, the DPLL algorithm continuously updates an assignment $\sigma \in \mathbb{B}^X$ for a subset $X \subseteq \mathsf{var}(f)$ of all variables $\mathsf{var}(f)$ in f. There are two ways of assigning a value to a variable: explicitly fixing an assignment (*decision*) or inferring a new assignment from former decisions. An assignment can be inferred for each variable that appears in a unit clause, i.e., a clause that contains only a single literal. The process of assigning a value to each literal of all unit clauses in f is called unit-propagation.

Definition B.19 $\mathsf{unit\text{-}propagation} : \mathrm{CNF} \rightarrow \mathbb{B}^X$ gives an assignment $\sigma \in \mathbb{B}^X$ for a subset of variables $X \subseteq \mathsf{var}(f)$ in a formula $f \in \mathrm{CNF}$:

```
unit-propagation(f) :=
  σ ← ∅
  while (f⟨σ⟩ contains unit-clause c)
    if (c = {v}) then            // c contains positive literal
      σ ← σ[v/ True]
    else if (c = {¬v}) then      // c contains negative literal
      σ ← σ[v/ False]
  return σ
```

Note that $\sigma[v/b]$ for a tuple $(v, b) \in \mathsf{var}(f) \times \mathbb{B}$ denotes the update of σ by (v, b) (cf. Definition A.17).

After applying unit-propagation, there are no more assignments that can be inferred. Thus, an assignment must explicitly be fixed in case there are any unassigned variables.

Definition B.20 The function $\mathsf{decision\text{-}variable} : \mathrm{CNF} \rightarrow \mathrm{V} \times \mathbb{B}$ gives a pair $(v, a) \in \mathrm{V} \times \mathbb{B}$ for a formula in CNF where v denotes the next decision variable in the DPLL algorithm and a is the value that is assigned to v.

$\mathsf{decision\text{-}variable}$ is left undefined here. Realizing a viable implementation for $\mathsf{decision\text{-}variable}$ so that a solver performs optimally for most definitions of *optimal* is far from trivial and a crucial design decision when developing SAT solvers.

Using the previously defined concepts, we introduce the DPLL algorithm.

Definition B.21 $\mathsf{DPLL} : \mathbb{B}^X \times \mathrm{CNF} \rightarrow \mathbb{B}^Y \cup \{\textsc{Unsat}\}$ is a recursively defined algorithm that takes a formula $f \in \mathrm{CNF}$ and a partial assignment $\sigma \in \mathbb{B}^X$ with $X \subseteq \mathsf{var}(f)$ and gives a satisfying assignment $\sigma \in \mathbb{B}^Y$ with $Y = \mathsf{var}(f)$ in case that f is satisfiable under σ:

```
DPLL(σ, f) :=
  σ ← σ[unit-propagation(f)]
  f ← f⟨σ⟩
  if (f contains empty clause) then return UNSAT
  else if (f contains no clauses) then return σ
  else
```

```
  (v, a) ← decision-variable(f)
  r ← DPLL(σ[v/a], f)
  if (r ≠ Unsat) then return r
  else
    return DPLL(σ[v/¬a], f)
```

Note that $\sigma[\mathsf{unit\text{-}propagation}(f)]$ denotes the update of σ by the assignment resulting from evaluating $\mathsf{unit\text{-}propagation}(f)$ (cf. Definition A.18).

Note that DPLL returns Unsat if the given formula is not satisfiable under the given assignment.

DPLL backtracks using the stack of recursive function calls: whenever there is a conflict, i.e., a partial variable assignment for which DPLL returns Unsat, the last decision variable's value is negated. If the conflict remains, the next-to-last decision variable's value is negated, and so on. Thus, backtracking is done in the reversed order as decision variables are chosen. This can be computationally expensive when the conflict results from a decision variable that was chosen very early in the process.

Conflict-driven clause learning (CDCL) [49] improves the DPLL algorithm by learning clauses at run-time and allowing back-jumps to previous decision variables other than the last one. Whenever a conflict occurs under a partial variable assignment, CDCL inspects the conflict clause of that assignment.

Definition B.22 A clause $c \in 2^{\text{Literal}}$ is a *conflict clause* under an assignment $\sigma \in \mathbb{B}^X$ with $X \subseteq \mathrm{V}$ if $\{c\}\langle\sigma\rangle = \{\varnothing\}$.

Example B.23 $c = \{x_1, \neg x_2, x_3\}$ is a conflict clause under the assignment $\sigma_1 = \{(x_1, \mathsf{False}), (x_2, \mathsf{True}), (x_3, \mathsf{False})\}$. Note that c is no conflict clause under $\sigma_2 = \{(x_1, \mathsf{True})\}$ because $\{c\}\langle\sigma_2\rangle = \varnothing$ and $\varnothing \neq \{\varnothing\}$.

In order to analyze a conflict, we introduce a special notation that is explicit about the order and the reason of partial variable assignments.

Definition B.24 The *set of ordered variable assignments* OVA^n of length $n \in \mathbb{N}_{>0}$ equals the greatest set $O \subsetneq (\{\mathsf{Decide}, \mathsf{Propagate}\} \times \mathrm{V} \times \mathbb{B})^n$ for which each vector $((d_1, x_1, b_1), \ldots, (d_n, x_n, b_n)) \in O$ satisfies:

$$\forall (i, j) \in \{1 \ldots n\}^2 : i \neq j \implies x_i \neq x_j$$

For $n \in \mathbb{N}_{>0}$ and $i \in \{1 \ldots n\}$, an ordered variable assignment specifies a sequence of assignments (d_i, x_i, b_i) of Boolean values $b_i \in \mathbb{B}$ to propositional variables $x_i \in \mathrm{V}$ where $d_i \in \{\mathsf{Decide}, \mathsf{Propagate}\}$ denotes if x_i is a decision variable ($d_i = \mathsf{Decide}$) or if it was assigned by unit-propagation ($d_i = \mathsf{Propagate}$).

When a conflict occurs, the conflict clause is analyzed by searching for a reason clause that led the CDCL algorithm to infer an empty clause.

Definition B.25 A clause $d = \{\dots, \neg x_n\} \in 2^{\textsc{Literal}}$ is a *reason clause* for a conflict clause $c = \{\dots, x_n\} \in 2^{\textsc{Literal}}$ under an ordered variable assignment $((d_1, x_1, b_1), \dots, (\mathsf{Propagate}, x_n, b_n)) \in \mathrm{OVA}^n$ of length $n \in \mathbb{N}_{>0}$ if d is satisfied under assignment $\{(x_1, b_1), \dots, (x_n, b_n)\}$:

$$\{d\}\Big\langle\big\{(x_1, b_1), \dots, (x_n, b_n)\big\}\Big\rangle = \varnothing$$

Note that there is always a reason clause if the most recently assigned variable has been assigned by unit-propagation.

Example B.26 Assume $c = \{x_1, \neg x_2, x_3\}$ being a conflict clause under the ordered variable assignment:

$$\big((\mathsf{Propagate}, x_1, \mathsf{False}), (\mathsf{Propagate}, x_2, \mathsf{True}), (\mathsf{Propagate}, x_3, \mathsf{False})\big)$$

where x_3 has been most recently assigned by unit-propagation. Then, $d = \{x_1, \neg x_3\}$ is a reason clause because:

$$\{d\}\Big\langle\big\{(x_1, \mathsf{False}), (x_2, \mathsf{True}), (x_3, \mathsf{False})\big\}\Big\rangle = \varnothing$$

Computing a reason clause $d \in 2^{\textsc{Literal}}$ is not helpful if the satisfaction of d is caused by unit-propagation. Thus, we must repeatedly compute reason clauses until the most recently assigned literal in the current reason clause is a decision variable. To do so, CDCL updates a reason clause using the resolution inference rule.

Definition B.27 The function $\otimes_r : 2^{\textsc{Literal}} \times 2^{\textsc{Literal}} \to 2^{\textsc{Literal}}$ implements the *resolution inference rule* for two clauses

$$\{x_1, \dots, r, \dots, x_m\}, \{y_1, \dots, \neg r, \dots, y_n\} \in 2^{\textsc{Literal}}$$

according to literal $r \in \textsc{Literal}$ and is defined by:

$$\{x_1, \dots, r, \dots, x_m\} \otimes_r \{y_1, \dots, \neg r, \dots, y_n\} = \{x_1, \dots, x_m, y_1, \dots, y_n\}$$

The result of the resolution inference rule is called the *resolvent*.

Theorem B.28 For all formulas $f \in \mathrm{CNF}$ and two clauses $c_{1,2} \in f$ that share a complementary literal r, $f \equiv f \cup \{c_1 \otimes_r c_2\}$ [66]. ■

Note that a conflict clause $c \in 2^{\textsc{Literal}}$ and its reason clause $d \in 2^{\textsc{Literal}}$ always share a complementary literal $r \in \textsc{Literal}$. The resolvent $c \otimes_r d$ denotes an updated conflict clause that does not contain the unit-propagated literal r. That is useful because r itself does not contribute to the initial conflict c as it is the result of unit-propagation.

By using the resolution inference rule repeatedly on reason clauses, we obtain a final reason for a conflict.

Definition B.29 reason-unsat : $\mathrm{OVA}^n \times \mathrm{CNF} \times 2^{\textsc{Literal}} \to 2^{\textsc{Literal}}$ gives the reason clause for a conflict clause $c \in 2^{\textsc{Literal}}$ in a formula $f \in \mathrm{CNF}$ under an ordered variable assignment $o \in \mathrm{OVA}^n$ with $n \in \mathbb{N}_{>0}$:

reason-unsat$(((d_1, x_1, b_1), \ldots, (d_n, x_n, b_n)), f, c) :=$
  if (f contains a reason clause d for c with a complementary literal r)
  then
    return reason-unsat$(((d_1, x_1, b_1), \ldots, (d_{n-1}, x_{n-1}, b_{n-1})), f, c \otimes_r d)$
  else
    return c

Example B.30 Assume a formula

$$\begin{aligned} f = \{k_1, k_2, k_3\} \text{ where } k_1 &= \{x_1, x_4\} \\ k_2 &= \{x_3, \neg x_4, \neg x_5\} \\ k_3 &= \{\neg x_2, \neg x_3, \neg x_4\} \end{aligned}$$

and the following ordered variable assignment:

$$\begin{pmatrix} (\mathsf{Decide}, x_5, \mathsf{True}), (\mathsf{Decide}, x_2, \mathsf{True}), (\mathsf{Decide}, x_1, \mathsf{False}), \\ (\mathsf{Propagate}, x_4, \mathsf{True}), (\mathsf{Propagate}, x_3, \mathsf{True}) \end{pmatrix}$$

After unit-propagation of x_3, k_3 becomes a conflict clause c_0. The reason for the conflict is computed by:

$$\begin{aligned} c_0 &= \{\neg x_2, \neg x_3, \neg x_4\} \text{ with reason clause } k_2 \\ c_1 &= c_0 \otimes_{x_3} k_2 = \{\neg x_2, \neg x_4, \neg x_5\} \text{ with reason clause } k_1 \\ c_2 &= c_1 \otimes_{x_4} k_1 = \{x_1, \neg x_2, \neg x_5\} \end{aligned}$$

The clause c_2 is the final reason because it has no reason clause.

The reason clause of a conflict serves two purposes:

1. It is added as new clause to the formula because it will prevent the algorithm to end up in the same conflict again.
2. The algorithm may back-jump to the assignment of a previous decision variable by omitting decision variables that do not contribute to the conflict.

If $c = \{x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n\} \in 2^{\textsc{Literal}}$ is a final reason clause for a formula $f \in \mathrm{CNF}$ and an ordered variable assignment where x_i is the last decision variable and x_j is the next-to-last decision variable, then CDCL jumps back to the assignment of the decision variable x_j [53]. If c only contains a single decision variable, it restarts the search with an empty assignment. If c does not contain any decision variable at all, f is unsatisfiable.

Example B.31 Assume a formula

$$
\begin{aligned}
f = \{k_1, k_2, k_3, k_4, k_5\} \text{ where } k_1 &= \{x_1, x_4\} \\
k_2 &= \{x_1, \neg x_3, \neg x_6\} \\
k_3 &= \{x_1, x_6, x_9\} \\
k_4 &= \{x_2, x_8\} \\
k_5 &= \{\neg x_3, \neg x_5, x_7\} \\
k_6 &= \{\neg x_5, x_6, \neg x_7\}
\end{aligned}
$$

and the following ordered variable assignment:

$$
\left(\begin{array}{c}
(\mathsf{Decide}, x_1, \mathsf{False}), (\mathsf{Propagate}, x_4, \mathsf{True}), (\mathsf{Decide}, x_3, \mathsf{True}), \\
(\mathsf{Propagate}, x_6, \mathsf{False}), (\mathsf{Propagate}, x_9, \mathsf{True}), (\mathsf{Decide}, x_2, \mathsf{False}), \\
(\mathsf{Propagate}, x_8, \mathsf{True}), (\mathsf{Decide}, x_5, \mathsf{True}), (\mathsf{Propagate}, x_7, \mathsf{True})
\end{array}\right)
$$

After assigning x_7, k_6 becomes a conflict clause c_0. The reason for the conflict is computed by:

$$
\begin{aligned}
c_0 &= \{\neg x_5, x_6, \neg x_7\} \\
c_1 &= c_0 \otimes_{x_7} k_5 = \{\neg x_3, \neg x_5, x_6\}
\end{aligned}
$$

In c_1, x_5 is the last decision variable, and x_3 is the next-to-last decision variable. Thus, a back-jump to the assignment of x_3 is performed. Note that the decision variable x_2 is skipped as its consequences are completely independent of the conflict clause c_0. This example highlights the difference to the DPLL algorithm: it would re-assign x_2 prior to x_3 and therefore end up in the same conflict again.

After back-jumping to x_3, c_1 is added to f as learned clause.

$$f \leftarrow f \cup \{\neg x_3, \neg x_5, x_6\}$$

This leads to the following ordered variable assignment:

$$
\left(\begin{array}{c}
(\mathsf{Decide}, x_1, \mathsf{False}), (\mathsf{Propagate}, x_4, \mathsf{True}), (\mathsf{Decide}, x_3, \mathsf{True}), \\
(\mathsf{Propagate}, x_6, \mathsf{False}), (\mathsf{Propagate}, x_9, \mathsf{True}), (\mathsf{Propagate}, x_5, \mathsf{False}), \ldots
\end{array}\right)
$$

Note that the learned clause c_1 leads to the propagation of (x_5, False): the conflict in k_6 is avoided.

We illustrate a conflict-driven, clause learning SAT solver in pseudo-code.

Definition B.32 $\mathsf{CDCL} : \mathrm{CNF} \to \mathrm{OVA}^n \cup \{\textsc{Unsat}\}$ is an iterative algorithm that returns $\textsc{Unsat}$ for a formula $f \in \mathrm{F}$ if f is not satisfiable, otherwise, it gives an ordered variable assignment $o \in \mathrm{OVA}^n$ of length $n = |\,\mathsf{var}(f)|$:

```
1  CDCL(f) :=
2    o ← ()
3    while (f contains unassigned variables)
4      o ← o · ((Propagate, v, a) | (v, a) ∈ unit-propagation(f))
5      if (f⟨o⟩ contains no clauses) then return o
6      else if (f⟨o⟩ contains empty clause) then
7        c ← get conflict clause for f and o
8        d ← reason-unsat(o, f, c)
9        if (d contains no decision variable)
10         return UNSAT
11       else
12         f ← f ∪ {d}
13         jump back to next−to−last decision variable in d
14     else
15       (v, a) ← decision-variable(f)
16       o ← o · (Decide, v, a)
17   return o
```

Note the following remarks:

1. In Line 4, each assignment returned by unit-propagation is added to the ordered variable assignment o together with the Propagate tag.
2. In Line 5 and 6, the ordered variable assignment o is used for assigning variables in f. Although this is contrary to Definition B.18, we allow it nonetheless because o can be trivially transformed to a mapping from variables to Boolean values.
3. In Line 16, both the decision variable v and the value a returned by decision-variable are added to the ordered variable assignment o together with the Decide tag.

B.1.1 Preprocessing

In order to implement a runtime-efficient SAT solver it is beneficial to perform preprocessing steps on the input formula. Exploiting *pure literals* is one way of simplifying a propositional formula. A literal is denoted as pure if it appears only with a single parity in a formula.

Lemma B.33 A formula $f \in \text{CNF}$ where a literal $x \in \textsc{Literal}$ only occurs positive (resp. negative) is equisatisfiable to $f\langle x/\,\mathsf{True}\rangle$ (resp. $f\langle x/\,\mathsf{False}\rangle$).$\blacksquare$

Due to Lemma B.33, a SAT solver may solve $f\langle x/\,\mathsf{True}\rangle$ (resp. $f\langle x/\,\mathsf{False}\rangle$) instead of the original formula $f \in \text{CNF}$ if $x \in \textsc{Literal}$ is a pure literal. This is beneficial as literal x does neither appear in $f\langle x/\,\mathsf{True}\rangle$ nor $f\langle x/\,\mathsf{False}\rangle$, i.e., the formula might become easier to solve for the SAT solver.

Another simplification considers clauses which are subsumed by other clauses.

Lemma B.34 A formula $f \in \text{CNF}$ that contains two clauses $c_1, c_2 \in 2^{\textsc{Literal}}$ with $c_1 \subseteq c_2$ is equisatisfiable to $f \setminus \{c_2\}$. ■

Recent SAT solvers use several more preprocessing steps to further simplify the input formula [40].

Appendix C

Supplemental Material

This chapter provides supplemental material for examples that have been shown only in extracts.

C.1 Exemplary Abstract Program

To complement Example 3.71, we give the complete abstract program that results from compiling the concrete program in Example 3.9.

```
constraint_𝔸 = \p u ->
  let v_d = u
  in
    valid_v_d
      ( let v_1 = let a = arguments_1 v_d
                      b = arguments_2 v_d
                  in
                    let v_2 = greaterOne a
                        v_3 =
                          let v_4 = greaterOne b
                              v_5 =
                                let v_6 = p
                                    v_7 =
                                      let v_8 = a
                                          v_9 = b
                                      in
                                        times v_8 v_9
                                in
                                  eq v_6 v_7
                          in
                            and2 v_4 v_5
                    in
```

```
                      and2 v_2 v_3
        in
          merge_{v_d} v_1 )

plus = \x y ->
  let v_d = x
  in
    valid_{v_d} ( let v_1 = y
                  v_2 = let x' = arguments_1 v_d
                        in
                          let a_1 =
                                let v_3 = x'
                                    v_4 = y
                                in
                                  plus v_3 v_4
                          in
                            cons_(2,2) a_1
              in
                merge_{v_d} v_1 v_2 )

times = \x y ->
  let v_d = x
  in
    valid_{v_d} ( let v_1 = cons_(1,2)
                  v_2 = let x' = arguments_1 v_d
                        in
                          let v_3 = y
                              v_4 =
                                let v_5 = x'
                                    v_6 = y
                                in
                                  times v_5 v_6
                          in
                            plus v_3 v_4
              in
                merge_{v_d} v_1 v_2 )

eq = \x y ->
  let v_dx = x
  in
    valid_{v_dx}
      ( let v_1 = let v_dy = y
                  in
                    valid_{v_dy}
                      ( let v_11 = cons_(2,2)
                            v_12 = let y' = arguments_1 v_dy
                                   in
                                     cons_(1,2)
                        in
                          merge_{v_dy} v_11 v_12 )
```

```

            v_2 = let x' = arguments_1 v_dx
                  in
                    let v_dy = y
                    in
                      valid_v_dy
                        ( let v_21 = cons_(1,2)
                              v_22 = let y' = arguments_1 v_dy
                                     in
                                       let v_23 = x'
                                           v_24 = y'
                                       in
                                         eq v_23 v_24
                          in
                            merge_v_dy v_21 v_22 )
        in
          merge_v_dx v_1 v_2 )

greaterOne = \x ->
  let v_dx = x
  in
    valid_v_dx
      ( let v_1 = cons_(1,2)
            v_2 = let x' = arguments_1 v_dx
                  in
                    let v_dx' = x'
                    in
                      valid_v_dx'
                        ( let v_21 = cons_(1,2)
                              v_22 = let x'' = arguments_1 v_dx'
                                     in
                                       cons_(2,2)
                          in
                            merge_v_dx' v_21 v_22 )
        in
          merge_v_dx v_1 v_2 )

and2 = \x y ->
  let v_dx = x
  in
    valid_v_dx
      ( let v_1 = cons_(1,2)
            v_2 = y
        in
          merge_v_dx v_1 v_2 )
```

C.2 Explicit Binary Encoding of Natural Numbers

To complement Example 6.17, we give a complete concrete program that implements binary encoded natural numbers.

```
data Bool = False | True
data List a = Nil | Cons a (List a)
data Pair a b = Pair a b

constraint :: List Bool -> Pair (List Bool) (List Bool) -> Bool
constraint = \p u -> case u of
  Pair x y -> case add x y of
    Pair sum carry -> and (eqNat sum p) (not carry)

add :: List Bool -> List Bool -> Pair (List Bool) Bool
add = \x y ->
  let add' pair accu = case pair of
    Pair u v -> case accu of
      Pair bits carry -> case fullAdder u v carry of
        Pair sum carry' -> Pair (Cons sum bits) carry'
  in
    foldr add' (Pair Nil False) (zip x y)

fullAdder :: Bool -> Bool -> Bool -> Pair Bool Bool
fullAdder = \x y carry -> case halfAdder x y of
  Pair sum1 carry1 -> case halfAdder sum1 carry of
    Pair sum2 carry2 -> Pair sum2 (or carry1 carry2)

halfAdder :: Bool -> Bool -> Pair Bool Bool
halfAdder = \x y -> Pair (xor x y) (and x y)

eqNat :: List Bool -> List Bool -> Bool
eqNat = \x y -> case x of
  Nil -> case y of Nil -> True
                   Cons v vs -> False

  Cons u us -> case y of Nil -> False
                         Cons v vs -> and (eq u v)
                                          (eqNat us vs)

foldr :: (a -> b -> b) -> b -> List a -> b
foldr = \f accu xs -> case xs of
  Nil -> accu
  Cons y ys -> f y (foldr f accu ys)

zip :: List a -> List b -> List (Pair a b)
zip = \x y -> case x of
```

```
  Nil -> Nil
  Cons u us -> case y of
    Nil -> Nil
    Cons v vs -> Cons (Pair u v) (zip us vs)

and :: Bool -> Bool -> Bool
and = \x y -> case x of False -> False
                        True -> y

or :: Bool -> Bool -> Bool
or = \x y -> case x of False -> y
                       True -> True

xor :: Bool -> Bool -> Bool
xor = \x y -> not (eq x y)

eq :: Bool -> Bool -> Bool
eq = \x y -> case x of False -> not y
                       True -> y

not :: Bool -> Bool
not = \x -> case x of False -> True
                      True -> False
```

C.3 Specification of Looping Derivations

To complement Listing 7.4, we give a complete concrete program that specifies looping derivations that are compatible with a given term rewriting system.

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

data Term = Var Nat
          | Node Nat (List Term)

data Unary = Z | S Unary

data Step = Step Term
                 (Pair Term Term)
                 (List Unary)
                 (List (Pair Nat Term))
                 Term

data LoopingDerivation = LoopingDerivation (List Step)
                                           (List Unary)
                                           (List (Pair Nat Term))

```

```
constraint :: List (Pair Term Term) -> LoopingDerivation -> Bool
constraint = \trs deriv -> isCompatibleLoopingDerivation
  trs deriv

isCompatibleLoopingDerivation :: List (Pair Term Term)
                              -> LoopingDerivation
                              -> Bool
isCompatibleLoopingDerivation = \trs loopDeriv ->
  case loopDeriv of
    LoopingDerivation deriv lastPos lastSub ->
      case deriv of
        Nil -> False
        Cons step steps -> case step of
          Step t0 rule pos sub t1 ->
            let last    = deriveTerm trs t0 deriv
                subterm = getSubterm lastPos last
                t0'     = applySubstitution lastSub t0
            in
              eqTerm t0' subterm

deriveTerm :: List (Pair Term Term)
           -> Term
           -> (List Step)
           -> Term
deriveTerm = \trs term deriv -> case deriv of
  Nil -> term
  Cons step steps -> case isValidStep trs step of
    False -> undefined
    True  -> case step of
      Step t0 rule pos sub t1 -> case eqTerm term t0 of
        False -> undefined
        True  -> deriveTerm trs t1 steps

isValidStep :: List (Pair Term Term) -> Step -> Bool
isValidStep = \trs step -> case step of
  Step t0 rule pos sub t1 ->
    and2 (isValidRule trs rule)
         (case rule of
            Pair lhs rhs ->
              let subT0  = getSubterm pos t0
                  lhs'   = applySubstitution sub lhs
                  rhs'   = applySubstitution sub rhs
                  result = putSubterm t0 pos rhs'
              in
                and2 (eqTerm subT0 lhs')
                     (eqTerm result t1)
         )

isValidRule :: List (Pair Term Term) -> Pair Term Term -> Bool
isValidRule = \trs rule -> elem eqRule rule trs
```

```

getSubterm :: (List Unary) -> Term -> Term
getSubterm = \pos term -> case pos of
  Nil -> term
  Cons p pos’ -> case term of
    Var v -> undefined
    Node f ts -> getSubterm pos’ (at p ts)

putSubterm :: Term -> (List Unary) -> Term -> Term
putSubterm = \term pos term’ -> case pos of
  Nil -> term’
  Cons p pos’ -> case term of
    Var v -> undefined
    Node f ts ->
      Node f (replace p ts (putSubterm (at p ts) pos’ term’))

applySubstitution :: List (Pair Nat Term) -> Term -> Term
applySubstitution = \subs term -> case term of
  Var v -> applySubstitutionToVar subs v
  Node f ts -> Node f (map (\t -> applySubstitution subs t) ts)

applySubstitutionToVar :: List (Pair Nat Term) -> Nat -> Term
applySubstitutionToVar = \sub v -> case sub of
  Nil -> Var v
  Cons s ss -> case s of
    Pair name term -> case eqNat v name of
      False -> applySubstitutionToVar ss v
      True  -> term

at :: Unary -> List a -> a
at = \u xs -> case xs of
  Nil -> undefined
  Cons y ys -> case u of
    Z -> y
    S u’ -> at u’ ys

replace :: Unary -> List a -> a -> List a
replace = \u xs x -> case xs of
  Nil -> undefined
  Cons y ys -> case u of
    Z -> Cons x ys
    S u’ -> Cons y (replace u’ ys x)

eqRule :: Pair Term Term -> Pair Term Term -> Bool
eqRule = \x y -> case x of
  Pair xLhs xRhs -> case y of
    Pair yLhs yRhs -> and2 (eqTerm xLhs yLhs) (eqTerm xRhs yRhs)

elem :: (a -> a -> Bool) -> a -> List a -> Bool
elem = \eq x xs -> case xs of
```

```
  Nil -> False
  Cons y ys -> or2 (eq x y) (elem eq x ys)

eqTerm :: Term -> Term -> Bool
eqTerm = \x y -> case x of
  Var xV -> case y of
    Var yV -> eqNat xV yV
    Node g ts -> False
  Node f ss -> case y of
    Var yV -> False
    Node g ts -> and2 (eqNat f g) (eqList eqTerm ss ts)

eqList :: (a -> a -> Bool) -> List a -> List a -> Bool
eqList = \eq xs ys -> case xs of
  Nil -> case ys of
    Nil -> True
    Cons y ys' -> False
  Cons x xs' -> case ys of
    Nil -> False
    Cons y ys' -> and2 (eq x y) (eqList eq xs' ys')

or2 :: Bool -> Bool -> Bool
or2 = \x y -> case x of
  True  -> True
  False -> y

and2 :: Bool -> Bool -> Bool
and2 = \x y -> case x of
  False -> False
  True  -> y

map :: (a -> b) -> List a -> List b
map = \f xs -> case xs of
  Nil -> Nil
  Cons y ys -> Cons (f y) (map f ys)
```

C.4 Specification of LPO-inducing Precedences

To complement Listing 7.10, we give a complete concrete program that specifies precedences that induce lexicographic path orders that are compatible with a given term rewriting system.

```
data Bool = False | True
data Pair a b = Pair a b
data List a = Nil | Cons a (List a)

data Term = Var Nat | Node Nat (List Term)
```

```

data Order = Gr | Eq | NGe

data TRS = TRS (List Nat) (List (Pair Term Term))

constraint :: TRS -> List Nat -> Bool
constraint = \trs prec -> case trs of
  TRS symbols rules ->
    and2 (forall rules   (\rule -> ordered rule prec))
         (forall symbols (\sym  -> exists prec sym eqNat))

ordered :: Pair Term Term -> List Nat -> Bool
ordered = \rule prec -> case rule of
  Pair lhs rhs -> eqOrder (lpo prec lhs rhs) Gr

lpo :: List Nat -> Term -> Term -> Order
lpo = \prec s t -> case t of
  Var x -> case eqTerm s t of
    False -> case varOccurs x s of
               False -> NGe
               True  -> Gr
    True  -> Eq

  Node g ts  -> case s of
    Var v     -> NGe
    Node f ss ->
      case forall ss (\si -> eqOrder (lpo prec si t) NGe) of
        False -> Gr
        True  -> case ord prec f g of
          Gr ->
            case forall ts (\ti -> eqOrder (lpo prec s ti) Gr) of
              False -> NGe
              True  -> Gr
          Eq ->
            case forall ts (\ti -> eqOrder (lpo prec s ti) Gr) of
              False -> NGe
              True  -> lex (\xs ys -> lpo prec xs ys) ss ts
          NGe -> NGe

ord :: List Nat -> Nat -> Nat -> Order
ord = \prec a b ->
  let run = \ps -> case ps of
        Nil        -> undefined
        Cons p ps' -> case eqNat p a of
          True  -> Gr
          False -> case eqNat p b of
            True  -> NGe
            False -> run ps'
  in
    case eqNat a b of
```

```
      True  -> Eq
      False -> run prec

varOccurs :: Nat -> Term -> Bool
varOccurs = \var term -> case term of
  Var var'  -> eqNat var var'
  Node f ts -> exists' ts (\t -> varOccurs var t)

lex :: (a -> b -> Order) -> List a -> List b -> Order
lex = \ord xs ys -> case xs of
  Nil -> case ys of Nil -> Eq
                    Cons y ys' -> NGe
  Cons x xs' -> case ys of
    Nil -> Gr
    Cons y ys' -> case ord x y of
      Gr  -> Gr
      Eq  -> lex ord xs' ys'
      NGe -> NGe

eqTerm :: Term -> Term -> Bool
eqTerm = \x y -> case x of
  Var u -> case y of
    Var v -> eqNat u v
    Node v vs -> False

  Node u us -> case y of
    Var v -> False
    Node v vs -> and2 (eqNat u v) (eqList eqTerm us vs)

eqOrder :: Order -> Order -> Bool
eqOrder = \x y -> case x of
  Gr  -> case y of Gr  -> True
                   Eq  -> False
                   NGe -> False
  Eq  -> case y of Gr  -> False
                   Eq  -> True
                   NGe -> False
  NGe -> case y of Gr  -> False
                   Eq  -> False
                   NGe -> True

eqList :: (a -> a -> Bool) -> List a -> List a -> Bool
eqList = \f xs ys -> case xs of
  Nil -> case ys of Nil -> True
                    Cons v vs -> False
  Cons u us -> case ys of Nil -> False
                          Cons v vs -> and2 (f u v)
                                            (eqList f us vs)

forall :: List a -> (a -> Bool) -> Bool
```

```
forall = \xs f -> case xs of
  Nil -> True
  Cons y ys -> and2 (f y) (forall ys f)

exists :: List a -> a -> (a -> a -> Bool) -> Bool
exists = \xs y f -> exists' xs (\x -> f x y)

exists' :: List a -> (a -> Bool) -> Bool
exists' = \xs f -> case xs of
  Nil -> False
  Cons y ys -> or2 (f y) (exists' ys f)

and2 :: Bool -> Bool -> Bool
and2 = \x y -> case x of
  False -> False
  True -> y

or2 :: Bool -> Bool -> Bool
or2 = \x y -> case x of
  True -> True
  False -> y
```

C.5 Profiling Lexicographic Path Orders

We show the complete profiling log of CO^4 when constructing a propositional encoding for the concrete program in Appendix C.4 and the term rewriting system given in Example 7.11.

```
Start producing CNF
Number of shared values: 0
Allocator: #variables: 6, #clauses: 0
Cache hits: 202 (28%), misses: 498 (71%)
Profiling (inner-under):
("constraint", {numCalls = 1, numVariables = 166, numClauses = 415})
("forallHO_2999", {numCalls = 4, numVariables = 152, numClauses = 378})
("ordered", {numCalls = 3, numVariables = 145, numClauses = 359})
("globalLambda_1456", {numCalls = 3, numVariables = 145, numClauses = 359})
("globalLambdaSat_2136", {numCalls = 3, numVariables = 145, numClauses = 359})
("lpo", {numCalls = 41, numVariables = 140, numClauses = 349})
("forallHO_3012", {numCalls = 30, numVariables = 100, numClauses = 250})
("globalLambda_1472", {numCalls = 23, numVariables = 90, numClauses = 226})
("globalLambdaSat_2250", {numCalls = 23, numVariables = 90, numClauses = 226})
("run_39", {numCalls = 20, numVariables = 64, numClauses = 182})
("ord", {numCalls = 8, numVariables = 64, numClauses = 182})
("forallHO_3014", {numCalls = 31, numVariables = 46, numClauses = 107})
("globalLambda_1478", {numCalls = 22, numVariables = 41, numClauses = 97})
("globalLambdaSat_2263", {numCalls = 22, numVariables = 41, numClauses = 97})
("forallHO_3013", {numCalls = 21, numVariables = 35, numClauses = 79})
("and2", {numCalls = 34, numVariables = 32, numClauses = 76})
("globalLambda_1475", {numCalls = 15, numVariables = 30, numClauses = 69})
("globalLambdaSat_2257", {numCalls = 15, numVariables = 30, numClauses = 69})
("eqOrder", {numCalls = 26, numVariables = 26, numClauses = 52})
("forallHO_3000", {numCalls = 4, numVariables = 12, numClauses = 32})
("or2", {numCalls = 14, numVariables = 9, numClauses = 24})
```

```
("globalLambda_1458", {numCalls = 3, numVariables = 9, numClauses = 24})
("globalLambdaSat_2140", {numCalls = 3, numVariables = 9, numClauses = 24})
("existsHO_3007", {numCalls = 3, numVariables = 9, numClauses = 24})
("exists'HO_3010", {numCalls = 12, numVariables = 9, numClauses = 24})
("eqNat", {numCalls = 21, numVariables = 9, numClauses = 27})
("lexHO_3015", {numCalls = 11, numVariables = 2, numClauses = 4})
("varOccurs", {numCalls = 14, numVariables = 0, numClauses = 0})
("lpoSat_2269", {numCalls = 7, numVariables = 0, numClauses = 0})
("globalLambda_1467", {numCalls = 10, numVariables = 0, numClauses = 0})
("globalLambdaSat_2216", {numCalls = 10, numVariables = 0, numClauses = 0})
("globalLambdaSatHO_3008", {numCalls = 9, numVariables = 0, numClauses = 0})
("globalLambdaHO_3009", {numCalls = 9, numVariables = 0, numClauses = 0})
("exists'HO_3006", {numCalls = 13, numVariables = 0, numClauses = 0})
("eqTerm", {numCalls = 11, numVariables = 0, numClauses = 0})

Profiling (inner):
("run_39", {numCalls = 20, numVariables = 55, numClauses = 155})
("lpo", {numCalls = 41, numVariables = 33, numClauses = 77})
("and2", {numCalls = 34, numVariables = 32, numClauses = 76})
("eqOrder", {numCalls = 26, numVariables = 26, numClauses = 52})
("or2", {numCalls = 14, numVariables = 9, numClauses = 24})
("eqNat", {numCalls = 21, numVariables = 9, numClauses = 27})
("lexHO_3015", {numCalls = 11, numVariables = 2, numClauses = 4})
("varOccurs", {numCalls = 14, numVariables = 0, numClauses = 0})
("ordered", {numCalls = 3, numVariables = 0, numClauses = 0})
("ord", {numCalls = 8, numVariables = 0, numClauses = 0})
("lpoSat_2269", {numCalls = 7, numVariables = 0, numClauses = 0})
("globalLambda_1478", {numCalls = 22, numVariables = 0, numClauses = 0})
("globalLambda_1475", {numCalls = 15, numVariables = 0, numClauses = 0})
("globalLambda_1472", {numCalls = 23, numVariables = 0, numClauses = 0})
("globalLambda_1467", {numCalls = 10, numVariables = 0, numClauses = 0})
("globalLambda_1458", {numCalls = 3, numVariables = 0, numClauses = 0})
("globalLambda_1456", {numCalls = 3, numVariables = 0, numClauses = 0})
("globalLambdaSat_2140", {numCalls = 3, numVariables = 0, numClauses = 0})
("globalLambdaSat_2136", {numCalls = 3, numVariables = 0, numClauses = 0})
("globalLambdaSatHO_3008", {numCalls = 9, numVariables = 0, numClauses = 0})
("globalLambdaHO_3009", {numCalls = 9, numVariables = 0, numClauses = 0})
("forallHO_3014", {numCalls = 31, numVariables = 0, numClauses = 0})
("forallHO_3013", {numCalls = 21, numVariables = 0, numClauses = 0})
("forallHO_3012", {numCalls = 30, numVariables = 0, numClauses = 0})
("forallHO_3000", {numCalls = 4, numVariables = 0, numClauses = 0})
("forallHO_2999", {numCalls = 4, numVariables = 0, numClauses = 0})
("existsHO_3007", {numCalls = 3, numVariables = 0, numClauses = 0})
("exists'HO_3010", {numCalls = 12, numVariables = 0, numClauses = 0})
("exists'HO_3006", {numCalls = 13, numVariables = 0, numClauses = 0})
("eqTerm", {numCalls = 11, numVariables = 0, numClauses = 0})
("constraint", {numCalls = 1, numVariables = 0, numClauses = 0})

Cases:
((51,20),CaseProfileData {numEvaluations = 15, numKnown = 0, numUnknown = 15})
((49,24),CaseProfileData {numEvaluations = 15, numKnown = 0, numUnknown = 15})
((34,18),CaseProfileData {numEvaluations = 18, numKnown = 7, numUnknown = 11})
((124,11),CaseProfileData {numEvaluations = 14, numKnown = 5, numUnknown = 9})
((119,12),CaseProfileData {numEvaluations = 34, numKnown = 25, numUnknown = 9})
((86,15),CaseProfileData {numEvaluations = 26, numKnown = 21, numUnknown = 5})
((40,18),CaseProfileData {numEvaluations = 18, numKnown = 16, numUnknown = 2})
((70,20),CaseProfileData {numEvaluations = 8, numKnown = 7, numUnknown = 1})
((114,16),CaseProfileData {numEvaluations = 25, numKnown = 25, numUnknown = 0})
((106,15),CaseProfileData {numEvaluations = 90, numKnown = 90, numUnknown = 0})
((93,10),CaseProfileData {numEvaluations = 20, numKnown = 20, numUnknown = 0})
((90,10),CaseProfileData {numEvaluations = 7, numKnown = 7, numUnknown = 0})
((87,10),CaseProfileData {numEvaluations = 9, numKnown = 9, numUnknown = 0})
((81,16),CaseProfileData {numEvaluations = 10, numKnown = 10, numUnknown = 0})
((77,12),CaseProfileData {numEvaluations = 1, numKnown = 1, numUnknown = 0})
((76,14),CaseProfileData {numEvaluations = 11, numKnown = 11, numUnknown = 0})
((68,18),CaseProfileData {numEvaluations = 10, numKnown = 10, numUnknown = 0})
((66,11),CaseProfileData {numEvaluations = 1, numKnown = 1, numUnknown = 0})
((65,17),CaseProfileData {numEvaluations = 11, numKnown = 11, numUnknown = 0})
```

```
((60,22),CaseProfileData {numEvaluations = 14, numKnown = 14, numUnknown = 0})
((55,5),CaseProfileData {numEvaluations = 8, numKnown = 8, numUnknown = 0})
((47,16),CaseProfileData {numEvaluations = 20, numKnown = 20, numUnknown = 0})
((36,18),CaseProfileData {numEvaluations = 11, numKnown = 11, numUnknown = 0})
((32,7),CaseProfileData {numEvaluations = 20, numKnown = 20, numUnknown = 0})
((29,17),CaseProfileData {numEvaluations = 30, numKnown = 30, numUnknown = 0})
((24,14),CaseProfileData {numEvaluations = 10, numKnown = 10, numUnknown = 0})
((23,12),CaseProfileData {numEvaluations = 11, numKnown = 11, numUnknown = 0})
((22,22),CaseProfileData {numEvaluations = 41, numKnown = 41, numUnknown = 0})
((18,27),CaseProfileData {numEvaluations = 3, numKnown = 3, numUnknown = 0})
((12,29),CaseProfileData {numEvaluations = 1, numKnown = 1, numUnknown = 0})

Toplevel: #variables: 0, #clauses: 2
CNF finished
#variables: 172, #clauses: 417, #literals: 989, clause density: 2.4244
#variables (Minisat): 172, #clauses (Minisat): 415, clause density: 2.4128
#clauses of length 1: 2
#clauses of length 2: 258
#clauses of length 3: 157

Starting solver
Solver finished in 0.0 seconds (result: True)
Starting decoder
Decoder finished
Test: True
Just (Cons (nat 2 0) (Cons (nat 2 2) (Cons (nat 2 1) Nil)))
```

C.6 Specification of LPO-inducing Precedences with Semantic Labelling

To complement Listing 7.24, we give a complete concrete program that combines semantic labelling with the specification of precedences that induce lexicographic path orders that are compatible with a given term rewriting system. Note that the following listing contains type aliases; a feature of CO^4's constraint specification language that has not been introduced in the present thesis. A type alias of the form `type` $T_1 = T_2$ for $T_{1,2} \in \textsc{Type}$ introduces a the type alias T_1 for the type T_2 such that both types can be used interchangeably in a concrete program.

```
data Pair a b         = Pair a b
data Triple a b c     = Triple a b c
data List a           = Nil | Cons a (List a)

type Symbol           = Nat
type Map k v          = List (Pair k v)
type Function         = Map (List Nat) Nat
type Interpretation a = Map a Function
type Sigma            = Map Symbol Nat
type Label            = List Nat
type Labelled a       = Pair a Label
type Precedence a     = List a

```

```
data Term a           = Var Symbol | Node a (List (Term a))
type Rule a           = Pair (Term a) (Term a)
type TRS a            = Pair (List a) (List (Rule a))

data Order            = Gr | Eq | NGe

constraint :: Triple (TRS Symbol)
                     (List (Labelled Symbol))
                     (List Sigma)
           -> Pair (Precedence (Labelled Symbol))
                   (Interpretation Symbol)
           -> Bool
constraint = \p u ->
  let eqSymbol         = eqNat
      eqLabelledSymbol = eqLabelled eqNat
  in
    case p of Triple trs lsymbols assigns ->
      case u of Pair prec interp ->
        case trs of Pair symbols rules ->
          let lrules = labelledRules eqNat interp assigns rules
              ltrs   = Pair lsymbols lrules
          in
            and2 (lpoConstraint eqLabelledSymbol ltrs prec)
                 (isModel eqNat interp assigns trs)

lpoConstraint :: (a -> a -> Bool) -> TRS a -> Precedence a
              -> Bool
lpoConstraint = \eq trs prec -> case trs of
  Pair symbols rules ->
    and2 (forall rules   (\rule -> ordered eq rule prec))
         (forall symbols (\sym  -> exists prec sym eq))

ordered :: (a -> a -> Bool) -> Rule a -> Precedence a -> Bool
ordered = \eq rule prec -> case rule of
  Pair lhs rhs -> eqOrder (lpo eq prec lhs rhs) Gr

lpo :: (a -> a -> Bool) -> Precedence a -> Term a -> Term a
    -> Order
lpo = \eq prec s t -> case t of
  Var x -> case eqTerm eq s t of
    False -> case varOccurs x s of
               False -> NGe
               True  -> Gr
    True  -> Eq

  Node g ts  -> case s of
    Var _     -> NGe
    Node f ss ->
      case forall ss (\si -> eqOrder (lpo eq prec si t) NGe) of
        False -> Gr
```

```
        True  -> case ord eq prec f g of
          Gr  -> case forall ts
                      (\ti -> eqOrder (lpo eq prec s ti) Gr) of
                   False -> NGe
                   True  -> Gr
          Eq  -> case forall ts
                      (\ti -> eqOrder (lpo eq prec s ti) Gr) of
                   False -> NGe
                   True  -> lex (lpo eq prec) ss ts
          NGe -> NGe

ord :: (a -> a -> Bool) -> Precedence a -> a -> a -> Order
ord = \eq prec a b ->
  let run = \ps -> case ps of
        Nil        -> undefined
        Cons p ps' -> case eq p a of
          True  -> Gr
          False -> case eq p b of
            True  -> NGe
            False -> run ps'
  in
    case eq a b of
      True  -> Eq
      False -> run prec

varOccurs :: Symbol -> Term a -> Bool
varOccurs = \var term -> case term of
  Var var' -> eqNat var var'
  Node _ ts -> exists' ts (\t -> varOccurs var t)

lex :: (a -> b -> Order) -> List a -> List b -> Order
lex = \ord xs ys -> case xs of
  Nil -> case ys of Nil -> Eq
                    Cons y ys' -> NGe
  Cons x xs' -> case ys of
    Nil -> Gr
    Cons y ys' -> case ord x y of
      Gr  -> Gr
      Eq  -> lex ord xs' ys'
      NGe -> NGe

labelledRules :: (a -> a -> Bool) -> Interpretation a
              -> List Sigma -> List (Rule a)
              -> List (Rule (Labelled a))
labelledRules = \eq interp assigns rules ->
    concat' (map' (\rule -> case rule of
      Pair lhs rhs ->
        map' (\sigma -> Pair (labelledTerm eq interp sigma lhs)
                             (labelledTerm eq interp sigma rhs)
             ) assigns) rules)
```

```

labelledTerm :: (a -> a -> Bool) -> Interpretation a -> Sigma
             -> Term a -> Term (Labelled a)
labelledTerm = \eq interp sigma t -> case t of
  Var v     -> Var v
  Node f ts -> let as  = map' (eval eq interp sigma) ts
                   ts' = map' (labelledTerm eq interp sigma) ts
               in
                 Node (Pair f as) ts'

isModel :: (a -> a -> Bool) -> Interpretation a -> List Sigma
        -> TRS a -> Bool
isModel = \eq interp assigns trs -> case trs of
  Pair symbols rules ->
    forall assigns (\sigma ->
      forall rules (\(Pair lhs rhs) ->
        eqNat (eval eq interp sigma lhs)
              (eval eq interp sigma rhs)))

eval :: (a -> a -> Bool) -> Interpretation a -> Sigma -> Term a
     -> Nat
eval = \eq interp sigma t ->
  let lookup = \f k map -> case map of
        Nil -> undefined
        Cons m ms -> case m of
          Pair k' v -> case f k k' of
            False -> lookup f k ms
            True  -> v
  in case t of
    Var  v    -> lookup eqNat v sigma
    Node f ts -> let i  = lookup eq f interp
                     as = map' (\t -> eval eq interp sigma t) ts
                 in
                   lookup (eqList eqNat) as i

eqTerm :: (a -> a -> Bool) -> Term a -> Term a -> Bool
eqTerm = \eq x y -> case x of
  Var u -> case y of
    Var v -> eqNat u v
    Node v vs -> False

  Node u us -> case y of
    Var v -> False
    Node v vs -> and2 (eq u v) (eqList (eqTerm eq) us vs)

eqOrder :: Order -> Order -> Bool
eqOrder = \x y -> case x of
  Gr  -> case y of Gr  -> True
                   Eq  -> False
                   NGe -> False
```

```
  Eq  -> case y of Gr  -> False
                   Eq  -> True
                   NGe -> False
  NGe -> case y of Gr  -> False
                   Eq  -> False
                   NGe -> True

eqLabelled :: (a -> a -> Bool) -> Labelled a -> Labelled a
           -> Bool
eqLabelled = \eq (Pair a aLabel) (Pair b bLabel) ->
  and2 (eq a b) (eqList eqNat aLabel bLabel)

eqList :: (a -> a -> Bool) -> List a -> List a -> Bool
eqList = \f xs ys -> case xs of
  Nil -> case ys of Nil -> True
                     Cons v vs -> False
  Cons u us ->
    case ys of Nil -> False
               Cons v vs -> and2 (f u v) (eqList f us vs)

forall :: List a -> (a -> Bool) -> Bool
forall = \xs f -> case xs of
  Nil -> True
  Cons y ys -> and2 (f y) (forall ys f)

exists :: List a -> a -> (a -> a -> Bool) -> Bool
exists = \xs y f -> exists' xs (\x -> f x y)

exists' :: List a -> (a -> Bool) -> Bool
exists' \xs f -> case xs of
  Nil -> False
  Cons y ys -> or2 (f y) (exists' ys f)

map' :: (a -> b) -> List a -> List b
map' \f xs -> case xs of
  Nil -> Nil
  Cons y ys -> Cons (f y) (map' f ys)

concat' :: List (List a) -> List a
concat' = \xs -> foldr' append' Nil xs

append' :: List a -> List a -> List a
append' = \a b -> foldr' Cons b a

foldr' :: (a -> b -> b) -> b -> List a -> b
foldr' = \n c xs -> case xs of
  Nil -> c
  Cons y ys -> n y (foldr' n c ys)

and2 :: Bool -> Bool -> Bool
```

```
and2 = \x y -> case x of
  False -> False
  True -> y

or2 :: Bool -> Bool -> Bool
or2 = \x y -> case x of
  True -> True
  False -> y
```

C.7 Specification of the RNA Design Problem

To complement Listing 7.37, we give a complete concrete program that specifies the RNA design problem.

```
data Bool = False | True
data List a = Nil | Cons a (List a)
data Pair a b = Pair a b
data N = Z | S N
data Base = A | C | G | U
data Paren = Open | Close | Blank
data Energy = MinusInfinity | Finite Nat

constraint :: List Paren
           -> Pair (List Base) (List (List Energy))
           -> Bool
constraint = \secondary u -> case u of
  Pair primary e ->
    let c1 = geEnergy (boundEnergy primary secondary) (upright e)
        c2 = matrixAll eqEnergy e (energyM primary e)
        c3 = matrixAll eqEnergy e (gap (S Z) MinusInfinity e)
    in
      and2 c1 (and2 c2 c3)

energyM :: List Base -> List (List Energy) -> List (List Energy)
energyM = \p m ->
  let mInfty = MinusInfinity
  in sum
    (Cons (item mInfty zeroE p)
    (Cons (product (Cons m (Cons m Nil)))
    (Cons (pointwise timesE
            (costM MinusInfinity p)
            (matrixShift mInfty (gap (S (S (S Z))) mInfty m)))
     Nil)))

upright :: List (List a) -> a
upright = \m -> last (head m)

```

```
vectorGet :: List a -> N -> b -> (a -> b) -> b
vectorGet = \xs i nothing just -> case xs of
  Nil -> nothing
  Cons y ys -> case i of
    Z -> just y
    S j -> vectorGet ys j nothing just

matrixGet :: List (List a) -> N -> N -> b -> (a -> b) -> b
matrixGet = \m i j nothing just ->
  vectorGet m i nothing (\row ->
    vectorGet row j nothing (\x -> just x))

matrixMap :: (N -> N -> a -> b) -> List (List a) -> List (List b)
matrixMap = \f m -> for (zipNats m) (\zippedRow ->
  case zippedRow of
    Pair i row -> for (zipNats row) (\zippedElement ->
      case zippedElement of
        Pair j x -> f i j x ))

matrixTimes :: (a -> a -> a) -> (a -> a -> a)
            -> List (List a) -> List (List a)
            -> List (List a)
matrixTimes = \plus times a b ->
  let b' = matrixTranspose b
      dot row col =
        let zs = zipWith times row col
        in
          foldr plus (head zs) (tail zs)
  in
    for a (\row -> for b' (\col -> dot row col))

matrixTranspose :: List (List a) -> List (List a)
matrixTranspose = \xss -> case xss of
  Nil -> Nil
  Cons row rows -> case rows of
    Nil -> map (\x -> Cons x Nil) row
    Cons x xs -> zipWith Cons row (matrixTranspose rows)

pointwise :: (a -> b -> c) -> List (List a)
          -> List (List b) -> List (List c)
pointwise = \f a b -> zipWith (\row1 row2 ->
  zipWith f row1 row2) a b

matrixAll :: (a -> b -> Bool) -> List (List a)
          -> List (List b) -> Bool
matrixAll = \f a b -> and (map and (pointwise f a b))

matrixShift :: a -> List (List a) -> List (List a)
matrixShift = \zero m -> matrixMap (\i j x -> case j of
  Z -> zero
```

```
  S j' -> matrixGet m (S i) j' zero id) m

sum :: List (List (List Energy)) -> List (List Energy)
sum = \ms -> foldr (\x y -> pointwise plusE x y)
                   (head ms) (tail ms)

product :: List (List (List Energy)) -> List (List Energy)
product = \ms -> foldr (\x y -> matrixTimes plusE timesE x y)
                       (head ms) (tail ms)

costM :: Energy -> List Base -> List (List Energy)
costM = \zero p ->
  let addX  = \m -> append m
                    (Cons (map (\x -> zero) (head m)) Nil)
      dropY = \m -> map (\row -> Cons zero row) m
  in
    gap Z zero (dropY (addX
      (for (zipNats p) (\zip1 -> case zip1 of
         Pair i x -> for (zipNats p) (\zip2 -> case zip2 of
           Pair j y -> case ltN i j of
             False -> zero
             True  -> energyBase x y)))))

gap :: N -> a -> List (List a) -> List (List a)
gap = \delta zero m ->
  for (zipNats m) (\zippedRow -> case zippedRow of
    Pair i row -> for (zipNats row) (\zippedElement ->
      case zippedElement of
        Pair j x -> case leN (plusN i delta) j of
          True -> x
          False -> zero))

item :: a -> a -> List Base -> List (List a)
item = \zero one p ->
  let p' = Cons (head p) p
  in
    for (zipNats p') (\zippedRow -> case zippedRow of
      Pair i row -> for (zipNats p') (\zippedElement ->
        case zippedElement of
          Pair j x -> case eqN (S i) j of
            False -> zero
            True  -> one))

plusN :: N -> N -> N
plusN = \x y -> case x of
  Z -> y
  S x' -> S (plusN x' y)

leN :: N -> N -> Bool
leN = \x y -> case x of
```

```
  Z -> True
  S x' -> case y of
    Z -> False
    S y' -> leN x' y'

gtN :: N -> N -> Bool
gtN = \x y -> not (leN x y)

ltN :: N -> N -> Bool
ltN = \x y -> gtN y x

eqN = \x y -> case x of
  Z -> case y of
    Z -> True
    S y' -> False
  S x' -> case y of
    Z -> False
    S y' -> eqN x' y'

energyBase :: Base -> Base -> Energy
energyBase = \b1 b2 -> case b1 of
  A -> case b2 of { U -> twoE  ;             _ -> MinusInfinity }
  C -> case b2 of { G -> threeE;             _ -> MinusInfinity }
  G -> case b2 of { C -> threeE; U -> oneE; _ -> MinusInfinity }
  U -> case b2 of { A -> twoE  ; G -> oneE; _ -> MinusInfinity }

zeroE  = Finite (nat 8 0)
oneE   = Finite (nat 8 1)
twoE   = Finite (nat 8 2)
threeE = Finite (nat 8 3)

eqEnergy :: Energy -> Energy -> Bool
eqEnergy = \a b -> case a of
  MinusInfinity -> case b of
    MinusInfinity -> True
    Finite g -> False
  Finite f -> case b of
    MinusInfinity -> False
    Finite g -> eqNat f g

geEnergy :: Energy -> Energy -> Bool
geEnergy = \a b -> case b of
  MinusInfinity -> True
  Finite b' -> case a of
    MinusInfinity -> False
    Finite a' -> geNat a' b'

plusE :: Energy -> Energy -> Energy
plusE = \e f -> case e of
  Finite x -> case f of
```

```
    Finite y -> Finite (maxNat x y)
    MinusInfinity -> e
  MinusInfinity -> f

timesE :: Energy -> Energy -> Energy
timesE = \e f -> case e of
  Finite x -> case f of
    Finite y -> Finite (plusNat x y)
    MinusInfinity -> f
  MinusInfinity -> e

boundEnergy :: List Base -> List Paren -> Energy
boundEnergy = \p s -> parse Nil p s

parse :: List Base -> List Base -> List Paren -> Energy
parse = \stack p s -> case s of
  Nil -> case stack of
    Nil -> zeroE
    Cons z zs -> MinusInfinity
  Cons y ys -> case p of
    Nil -> MinusInfinity
    Cons x xs ->
      let stack' = case y of
            Blank -> stack
            Open  -> Cons x stack
            Close -> tail stack
          here = case y of
            Blank -> zeroE
            Open  -> zeroE
            Close -> energyBase (head stack) x
      in
        timesE here (parse stack' xs ys)

append :: List a -> List a -> List a
append = \xs ys -> foldr Cons ys xs

zipNats :: List a -> List (Pair N a)
zipNats = \xs ->
  let f = \n xs -> case xs of
        Nil -> Nil
        Cons y ys -> Cons (Pair n y) (f (S n) ys)
  in
    f Z xs

zipWith :: (a -> b -> c) -> List a -> List b -> List c
zipWith = \f xs ys -> case xs of
  Nil -> Nil
  Cons u us -> case ys of
    Nil -> Nil
    Cons v vs -> Cons (f u v) (zipWith f us vs)
```

```
for :: List a -> (a -> b) -> List b
for = \xs f -> map f xs

map :: (a -> b) -> List a -> List b
map = \f xs -> case xs of
  Nil -> Nil
  Cons y ys -> Cons (f y) (map f ys)

last :: List a -> a
last = \xs -> case xs of
  Nil -> undefined
  Cons y ys -> case ys of
    Nil -> y
    Cons z zs -> last ys

head :: List a -> a
head = \xs -> case xs of
  Nil -> undefined
  Cons y ys -> y

tail :: List a -> List a
tail = \xs -> case xs of
  Nil -> Nil
  Cons y ys -> ys

and :: List Bool -> Bool
and = \xs -> foldr and2 True xs

or :: List Bool -> Bool
or = \xs -> foldr or2 False xs

foldr :: (a -> b -> b) -> b -> List a -> b
foldr = \n c xs -> case xs of
  Nil -> c
  Cons y ys -> n y (foldr n c ys)

or2 :: Bool -> Bool -> Bool
or2 = \x y -> case x of
  True  -> True
  False -> y

and2 :: Bool -> Bool -> Bool
and2 = \x y -> case x of
  False -> False
  True  -> y

not :: Bool -> Bool
not = \x -> case x of
  False -> True
```

```
  True  -> False

id :: a -> a
id = \x -> x
```

C.8 Exemplary Energy Matrix

To complement Example 7.38, we give the complete energy matrix.

$$E = \begin{bmatrix}
-\infty & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 & 3 & 4 & 4 & 4 & 4 & 5 & 5 & 5 & 5 & 6 & 6 & 7 & 8 & 9 & 9 & 10 & 11 & 14 & 14 & 14 & 15 \\
-\infty & -\infty & 0 & 0 & 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 & 4 & 4 & 4 & 4 & 5 & 5 & 5 & 6 & 7 & 8 & 9 & 9 & 10 & 13 & 13 & 14 & 14 \\
-\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 & 6 & 6 & 7 & 8 & 9 & 9 & 12 & 13 & 13 & 13 \\
-\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 5 & 7 & 8 & 8 & 9 & 12 & 12 & 12 & 12 \\
-\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 5 & 7 & 8 & 8 & 9 & 11 & 11 & 11 & 11 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 5 & 6 & 7 & 7 & 8 & 11 & 11 & 11 & 11 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 5 & 6 & 7 & 7 & 8 & 11 & 11 & 11 & 11 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 5 & 6 & 7 & 7 & 8 & 11 & 11 & 11 & 11 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 & 3 & 5 & 6 & 7 & 7 & 8 & 11 & 11 & 11 & 11 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 & 3 & 5 & 6 & 6 & 6 & 8 & 11 & 11 & 11 & 11 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 & 3 & 5 & 5 & 6 & 6 & 8 & 8 & 8 & 8 & 8 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 3 & 3 & 4 & 5 & 6 & 6 & 7 & 8 & 8 & 8 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 5 & 5 & 7 & 7 & 7 & 7 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 4 & 4 & 4 & 4 & 7 & 7 & 7 & 7 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 3 & 3 & 3 & 4 & 6 & 6 & 6 & 6 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 & 3 & 5 & 6 & 6 & 6 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 2 & 2 & 3 & 5 & 5 & 5 & 5 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 1 & 1 & 2 & 2 & 2 & 5 & 5 & 5 & 5 \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 2 & 5 & 5 & 5 & 5 \\
-\infty & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 3 \\
-\infty & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 2 & 3 \\
-\infty & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 \\
-\infty & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 2 \\
-\infty & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
-\infty & 0 & 0 & 0 & 0 & 0 & 1 \\
-\infty & 0 & 0 & 0 & 0 & 1 \\
-\infty & 0 & 0 & 0 & 0 \\
-\infty & 0 & 0 & 0 \\
-\infty & 0 & 0 \\
-\infty & 0 \\
-\infty & -\infty
\end{bmatrix}$$

Index

Bibliography

[1] Termination Problems Data Base 8.0.7. `http://termcomp.uibk.ac.at/status/downloads/tpdb-8.0.7_by_category.tgz`, 2013. [accessed 06. October 2016].

[2] Rosalia Aguirre-Hernandez. *Computational RNA Secondary Structure Design: Empirical Complexity and Improved Methods*. PhD thesis, University of British Columbia, 2007.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Johan Ankner and Josef Svenningsson. An EDSL Approach to High Performance Haskell Programming. In *SIGPLAN Symposium on Haskell*, 2013.

[5] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[8] Alexander Bau. SAT Compilation for Constraints over Finite Structured Domains. In *Cross-Fertilization Between CSP and SAT*, 2014.

[9] Alexander Bau, Jörg Endrullis, and Johannes Waldmann. SAT Compilation for Termination Proofs via Semantic Labelling. In *Workshop on Termination*, 2013.

[10] Alexander Bau, René Thiemann, and Johannes Waldmann. SAT Compilation for Termination Proofs via Semantic Labelling and Unlabelling. In *Workshop on Termination*, 2014.

[11] Alexander Bau, Johannes Waldmann, and Sebastian Will. RNA Design by Program Inversion via SAT Solving. In *Workshop on Constraint-Based-Methods for Bioinformatics*, 2013.

[12] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[13] Piero A. Bonatti. Reasoning with Infinite Stable Models. *Artificial Intelligence*, 2004.

[14] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional Satisfiability and Constraint Programming: A Comparative Survey. *ACM Computing Surveys*, 2006.

[15] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer Set Programming at a Glance. *Communications of the ACM*, 2011.

[16] Koen Claessen, Niklas Eén, Mary Sheeran, Niklas Sörensson, Alexey Voronov, and Knut Akesson. SAT-Solving in Practice, with a Tutorial Example from Supervisory Control. *Discrete Event Dynamic Systems*, 2009.

[17] Michael Codish, Yoav Fekete, Carsten Fuhs, Jürgen Giesl, and Johannes Waldmann. Exotic Semi-Ring Constraints. In *Workshop on Satisfiability Modulo Theories*, 2013.

[18] Michael Codish, Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. SAT Solving for Termination Proofs with Recursive Path Orders and Dependency Pairs. *Journal of Automated Reasoning*, 2012.

[19] Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Solving Partial Order Constraints for LPO Termination. *Journal on Satisfiability, Boolean Modeling and Computation*, 2008.

[20] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Theory of Computing*, 1971.

[21] Tom Crick, Martin Brain, Marina De Vos, and John P. Fitch. Generating Optimal Code Using Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning*, 2009.

[22] Luís Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *Principles of Programming Languages*, 1982.

[23] Martin Davis, George Logemann, and Donald W. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 1962.

[24] Iavor S. Diatchki. Improving Haskell Types With SMT. In *SIGPLAN Symposium on Haskell*, 2015.

[25] Niklas Eén and Niklas Sörensson. An Extensible SAT-Solver. In *Theory and Applications of Satisfiability Testing*, 2003.

[26] Niklas Eén and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Theoretical Computer Science*, 2003.

[27] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems*, 2009.

[28] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *Journal of Automated Reasoning*, 2008.

[29] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *Rewriting Techniques and Applications*, 2011.

[30] International Organization for Standardization. Prolog (ISO/IEC 13211-1:1995), 1995.

[31] Free Software Foundation. GNU General Public License, Version 3. `http://www.gnu.org/licenses/gpl.html`, 2007.

[32] Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.

[33] Carsten Fuhs. *SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis*. PhD thesis, RWTH Aachen University, 2012.

[34] Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the Input Language of ASP Grounder Gringo. In *Logic Programming and Nonmonotonic Reasoning*, 2009.

[35] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming*, 1988.

[36] Robert Giegerich. A Systematic Approach to Dynamic Programming in Bioinformatics. *Bioinformatics*, 2000.

[37] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving Termination of Programs Automatically with AProVE. In *Automated Reasoning*, 2014.

[38] Jürgen Giesl, Frédéric Mesnard, Albert Rubio, René Thiemann, and Johannes Waldmann. Termination Competition (termCOMP 2015). In *Automated Deduction - CADE-25*, 2015.

[39] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 1996.

[40] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause Elimination Procedures for CNF Formulas. In *Logic for Programming, Artificial Intelligence, and Reasoning*, 2010.

[41] Nao Hirokawa and Aart Middeldorp. Tyrolean Termination Tool: Techniques and Features. *Information and Computation*, 2007.

[42] Petra Hofstedt and Armin Wolf. *Einführung in die Constraint-Programmierung - Grundlagen, Methoden, Sprachen, Anwendungen*. eXamen.press. Springer, 2007.

[43] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *History of Programming Languages*, 2007.

[44] John Hughes. Why Functional Programming Matters. *Computer Journal*, 1989.

[45] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture*, 1985.

[46] Simon L. Peyton Jones. Composing Contracts: An Adventure in Financial Engineering. In *International Symposium of Formal Methods*, 2001.

[47] Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[48] Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the Chase - Solving Linear Integer Arithmetic. *Journal of Automated Reasoning*, 2013.

[49] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Artificial Intelligence and Innovative Applications of Artificial Intelligence*, 1997.

[50] Randy H. Katz. *Contemporary Logic Design*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[51] Edward Kmett. Ersatz. `https://github.com/ekmett/ersatz`, 2010. [accessed 21. July 2015].

[52] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In *Rewriting Techniques and Applications*, 2009.

[53] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[54] Frédéric Lafitte, Jorge Nakahara Jr., and Dirk Van Heule. Applications of SAT Solvers in Cryptanalysis: Finding Weak Keys and Preimages. *Journal on Satisfiability*, 2014.

[55] Hai Liu, Neal Glew, Leaf Petersen, and Todd A. Anderson. The Intel Labs Haskell Research Compiler. In *SIGPLAN Symposium on Haskell*, 2013.

[56] Joao Marques-Silva. Practical Applications of Boolean Satisfiability. In *Discrete Event Systems*, 2008.

[57] Donald Michie. “Memo” Functions and Machine Learning. *Nature*, 1968.

[58] David Mitchell, Bart Selman, and Hector Levesque. Hard and Easy Distributions of SAT Problems. In *Artificial Intelligence*, 1992.

[59] Neil Mitchell and Colin Runciman. Losing Functions without Gaining Data: Another Look at Defunctionalisation. In *SIGPLAN Symposium on Haskell*, 2009.

[60] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming*, 2007.

[61] Antoni Niederliński. *A Gentle Guide to Constraint Logic Programming via ECLiPSe*. 2014.

[62] Martin Odersky. The Scala Language Specification Version 2.9. `www.scala-lang.org/docu/files/ScalaReference.pdf`, 2014. [accessed 31. July 2016].

[63] Étienne Payet. Loop Detection in Term Rewriting Using the Eliminating Unfoldings. *Theoretical Computer Science*, 2008.

[64] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[65] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[66] Uwe Schöning. *Logik für Informatiker*. Spektrum Akademischer Verlag, 1995.

[67] Uwe Schöning and Jacobo Torán. *Das Erfüllbarkeitsproblem SAT - Algorithmen und Analysen*. Lehmann, 2012.

[68] Tim Sheard and Simon L. Peyton Jones. Template Meta-Programming for Haskell. *SIGPLAN Notices*, 2002.

[69] Michael J. Spivey. A Functional Theory of Exceptions. *Science of Computer Programming*, 1990.

[70] Tommi Syrjänen. Lparse 1.0 User’s Manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps`, 2000. [accessed 22. September 2016].

[71] Simon J. Thompson. *Haskell - The Craft of Functional Programming, 3rd Edition*. Addison-Wesley, 2011.

[72] Yoshihito Toyama. Counterexamples to Termination for the Direct Sum of Term Rewriting Systems. *Information Processing Letters*, 1987.

[73] Grigorii Samuilovich Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Automation of Reasoning*. 1983.

[74] Johannes Waldmann. Matchbox: A Tool for Match-Bounded String Rewriting. In *Rewriting Techniques and Applications*, 2004.

[75] Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Nagoya Termination Tool. In *Rewriting and Typed Lambda Calculi*, 2014.

[76] Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and Certifying Loops. In *Current Trends in Theory and Practice of Computer Science*, 2010.

[77] Hans Zantema. Termination of Term Rewriting by Semantic Labelling. *Fundamenta Informaticae*, 1995.