Master Thesis
Master of Science Informatik

Implementing Abstract Dependent Classes with SMT Solving

Matthias Krebs

Technische Universität Darmstadt
Fachbereich Informatik
Software Technology Group

Prüfer: Prof. Dr. Mira Mezini
Betreuer: Oliver Bracevac, M.Sc.

Abgabetermin: 13.08.2019

Erklärung

Hiermit versichere ich, Matthias Krebs, die vorliegende Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Ort, Datum	(Matthias Krebs)

Abstract

Dependent Classes support typical object oriented techniques as inheritance and subtype polymorphism at the scope of class groups. The DC_C calculus extends the semantics of Dependent Classes with abstract classes and methods and expresses inheritance via a constraint system.

We implement the DC_C calculus in Scala and utilize SMT solving to solve its constraint system. We propose *context enrichment* as a process to transfer information from compile time to runtime to decrease the time to solve constraint entailments emerging during runtime.

Contents

Chapter 1

Introduction

1.1 Motivation

Dynamic dispatch is the process to determine the implementation of a polymorphic method to call at runtime. In contrast to static dispatch, which determines this during compile time. Safety describes the ability of a language to prevent errors. In object oriented programming related classes can be grouped together, but inheritance between those groups can not be expressed.

Dependent Classes [GMO07] combine dynamic dispatch à la Smalltalk [GR83] with safety à la Scala [OMM+04], as well as supporting inheritance between groups of related classes. The support for abstract methods is yet to be fully explored. The DC_C calculus by Vaidas Gasiūnas [Gas10] captures the semantics of Dependent Classes and extends it with the support for abstract classes and methods. Subtyping in the DC_C calculus is expressed as constraint entailment.

The goal of this thesis is to implement the DC_C calculus and to explore the usage of SMT to solve the constraint system of the calculus.

1.2 Contributions

The main contributions of the thesis are:

- An implementation of the DC_C calculus using SMT solving to resolve the constraint system.
- A first-order model of the DC_C constraint system, which translates the rules of a sequent calculus to universal quantified formulae.
- Optimizations to the rules of the first-order model, such that a SMT solver can make better use of them.
- An implementation of the DC_C relation for type assignments to expressions.
- A way to use type information of an expression in the interpretation of that expression to reduce the time to solve the constraint system.

1.3 Structure

Chapter 2 presents preliminaries. We introduce Satisfiability Modulo Theories, Dependent Classes and the DC_C calculus.

Chapter 3 gives an implementation of the DC_C calculus. We implement the constraint system in Section 3.1, the operational semantics in Section 3.2 and the type relation in Section 3.3.

Chapter 4 gives applications of the interpreter to a DC_C program and evaluates which expressions the interpreter can successfully reduce.

Chapter 5 presents an approach to use information gained from compile time (type checker) during runtime (interpreter).

Related work is discussed in Chapter 6 and we conclude the thesis in Chapter 7.

Chapter 2

Preliminaries

2.1 Satisfiability modulo theories

This section shortly introduces Satisfiability modulo theories (SMT) and the SMT problem. This section is a write-up of the Handbook of Model Checking [BT18], the technical report on The SMT-Lib Standard [BFT17] and Z3: An efficient SMT solver [dMB08].

Many problems like formal verification of hard- and software can be reduced to checking the satisfiability of a formula in some logic. Some of these problems can be easily described as a satisfiability problem in propositional logic and solved using a propositional SAT solver. Other problems can be described more naturally in other traditional logics like first-order logic. These logics support more expressive language features including non-boolean variables, function and predicate-symbols and quantifiers.

The defining problem of Satisfiability modulo theories is checking if a logical formula is satisfiable in the context of some background theory. Checking validity can be achieved using SMT for formulas closed under logical negation [BFT17], since such a formula is valid in a theory if its negation is not satisfiable in the theory.

There is a trade-off between the expressiveness of a logic and the ability to automatically check satisfiability of formulae. A practical compromise is the usage of fragments of first-order logic, where the used fragment is restricted syntactically or semantically. Such restrictions achieve the decidability of the satisfiability problem and allow the development of procedures that exploit properties of the fragment for efficiency.
A SMT solver is any software implementing a procedure for determining satisfiability modulo a given theory. SMT solvers can be distinguished based on the underlying logic (first-order, modal, temporal, ...), the background theory, the accepted input formulas and the interface provided by the solver. Z3 is a SMT solver from Microsoft Research [dMB08], which is used in this thesis.

2.1.1 SMT-Lib

SMT-Lib is an international initiative with the goal to easing research and development in SMT. SMT-Lib's main motivation is the availability of common

$$\forall x : List[Int].\exists y : List[Int].$$
$$\textbf{let } h := head(x), t := tail(x)$$
$$\textbf{in } insert(h, t) = y$$

(a) First-order Formula

```
(forall ((x (List Int))) (exists ((y (List Int)))
  (let ((h (head x)) (t (tail x)))
    (= (insert h t) y))))
```

(b) SMTLib Representation

Figure 2.1: SMT-Lib Format Example

standards in SMT with the focus on

- providing a standard description of background theories.
- developing a common in- and output language for SMT solvers.
- establishing a library of benchmarks for SMT solvers.

to advance the state of the art in the field. The SMT-Lib Standard: Version 2.6 [BFT17] defines a language for writing terms and formulas in sorted first-order logic, specifying background theories, specifying logics, as well as a command language for interacting with SMT solvers. The SMT-Lib format is accepted by the majority of current SMT solvers. An example of the SMT-Lib format is shown in Figure 2.1. The example is showing a formula in sorted first-order logic in Figure 2.1a and the SMT-Lib representation of the formula in Figure 2.1b. The formula is quantifying over two integer lists, using a let binding to extract the head and the tail from the first list to construct a identical list out of these bindings and checks for equality to the second list. The example is purely for demonstrating the syntax of the SMT-Lib format.

2.2 Dependent Classes

Classes tend to be a too isolated unit to achieve modularity. Desired functionality involves groups of related classes. Grouping mechanisms exist, like namespaces in C++ and packages in Java, but these mechanisms do not cover inheritance and polymorphism for expressing variability. Virtual Classes [EOC06, Gas10] provide a solution for inheritance and polymorphism. They introduce classes as a kind of object members and treat these as virtual methods, which allows for overriding in subclasses and are late-bound. Virtual Classes are inner classes refinable in the subclasses of the enclosing class.

Figure 2.2 shows an example of the nesting of Virtual Classes in an informal Java-like syntax. The example is a shortened version from [EOC06] to show the nested behavior of Virtual Classes. The class Base contains two virtual classes: class Exp representing expressions and its subclass Lit representing numeric literals. The class WithNeg extends the collection of virtual classes defined in

```
class Base {
  class Exp {}
  class Lit extends Exp {
    int value;
  }
}

class WithNeg extends Base {
  class Neg extends Exp {
    Neg(out.Exp e) { this.e = e; }
    field out.Exp e;
  }
}
```

Figure 2.2: Nesting Of Virtual Classes

Base to include a class Neg representing negation expressions. Class Neg has an immutable field of type **out**.Exp and the keyword **out** is used to refer to the enclosing object of Neg.

The downside of Virtual Classes is that they must be nested within other classes. This requires to cluster classes depending on instances of some class together, introducing maybe unwanted coupling between these classes. Introducing a new class depending on the instances of an existing class requires the modification this class and its subclasses, limiting extensibility. This Nesting does also limit the expression of variability: The interface and the implementation of a virtual class can only depend on its single enclosing object.

Dependent Classes [GMO07, Gas10] are generalizations of Virtual Classes. The structure of a dependent class depends on arbitrarily many objects and dependency is expressed with class parameters. Dependent Classes can be seen as a combination of Virtual Classes with multi-dispatch.

The semantics of dependent classes involve non-trivial aspects, such as method calls and class instantiation expressions, which rely on dynamic dispatch over multiple parameters and the types of their fields. As well as the complexity of the type system, which integrates static dispatch with dependent typing.

The vc^n calculus captures the core semantics of dependent classes, such as static and dynamic dispatch and dependent typing based on paths. The calculus ensures that the static dispatch and static normalization of terms in types define a proper abstraction over dynamic dispatch and evaluation of expressions. It is defined in an algorithmic style. This algorithmic style has the advantage of being constructive and through this outlining a possible implementation. Due to the algorithmic nature of the calculus its difficult to extend the semantics with new expressive power, such as the support for abstract declarations.

2.2.1 DC_C Calculus

The vc^n calculus does not support abstract classes and methods. The DC_C calculus [Gas10] extends vc^n with support for abstract classes and methods and symmetric method dispatch. DC_C encodes dependent classes with a constraint

$$
\begin{aligned}
\mathit{Program} &::= \overline{\mathit{Decl}} \\
\mathit{Decl} &::= C(x.\ \overline{\mathit{Constr}}) \mid \forall x.\ \overline{\mathit{Constr}} \Rightarrow \mathit{Constr} \\
&\quad \mid m(x.\ \overline{\mathit{Constr}}) : \mathit{Type} \mid m(x.\ \overline{\mathit{Constr}}) : \mathit{Type} := \mathit{Expr} \\
\mathit{Type} &::= [x.\ \overline{\mathit{Constr}}] \\
\mathit{Constr} &::= \mathit{Path} \equiv \mathit{Path} \mid \mathit{Path} :: C \mid \mathit{Path}.\mathbf{cls} \equiv C \\
\mathit{Path} &::= x \mid \mathit{Path}.f \\
\mathit{Expr} &::= x \mid \mathit{Expr}.f \mid \mathbf{new}\ C(\overline{f} \equiv \overline{\mathit{Expr}}) \mid m(\mathit{Expr})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{MType}(m, x, y) &= \{\langle \overline{a}, \overline{b} \rangle | (m(x.\overline{a}) : [y.\ \overline{b}]...) \in P\} \\
\mathit{MImpl}(m, x) &= \{\langle \overline{a}, e \rangle | (m(x.\overline{a}) : [y.\ \overline{b}] := e) \in P\}
\end{aligned}
$$

$x, y \in$ variable names
$f \in$ field names
$C \in$ class names
$m \in$ method names

Figure 2.3: Syntax

system, which is used in both the static and dynamic semantics. The main relation of the DC_C calculus is constraint entailment.

The runtime structure is heap-based, with explicit object identities and relationships between objects based on these identities. Expressions evaluate to an identifier pointing to an object in the heap. Heaps preserve object identity and enable shared references to objects. The heap provides a direct interpretation for equivalent paths, two paths are equivalent if they point to the same object at runtime. Heaps can be easily translated to a set of constraints describing its objects and their relations, which enables the usage of the constraint system for dynamic dispatch and expression typing.

Here we show the calculus by Vaidas Gasiūnas.

Syntax

The syntax of DC_C is given in Figure 2.3. Types are lists of constraints to be satisfied by their instances. Types have the form $[x.\ \overline{a}]$, where x is a bound variable and $\overline{a}$ is a list of constraints on x. An object belongs to a type if it fulfills its constraints.

Constraints of the form $p \equiv q$ express that two paths p and q are equivalent and paths are considered to be equivalent if they refer to the same object at runtime. $p :: C$ specifies that path p refers to an instance of class C. The stronger form $p.\mathbf{cls} \equiv C$ denotes that path p refers to an object instantiated by a constructor of class C, excluding indirect instances of C inferred through inheritance rules.

A Program P consists of a list of declarations D. Possible declarations are constructor declarations, abstract method declarations, method implementations and constraint entailment rules.

$$\frac{}{a \vdash a}\ \text{(C-Ident)}$$

$$\frac{}{\epsilon \vdash p \equiv p}\ \text{(C-Refl)}$$

$$\frac{\overline{a} \vdash p.\mathbf{cls} \equiv C}{\overline{a} \vdash p :: C}\ \text{(C-Class)}$$

$$\frac{\overline{a} \vdash c \qquad \overline{a'}, c \vdash b}{\overline{a}, \overline{a'} \vdash b}\ \text{(C-Cut)}$$

$$\frac{\overline{a} \vdash a_{\{\!\{x \mapsto p\}\!\}} \qquad \overline{a} \vdash p' \equiv p}{\overline{a} \vdash a_{\{\!\{x \mapsto p'\}\!\}}}\ \text{(C-Subst)}$$

$$\frac{(\forall x.\ \overline{a} \Rightarrow a) \in P \qquad \overline{b} \vdash \overline{a}_{\{\!\{x \mapsto p\}\!\}}}{\overline{b} \vdash a_{\{\!\{x \mapsto p\}\!\}}}\ \text{(C-Prog)}$$

Figure 2.4: Constraint Entailment

A Path expression can be a variable x or navigation over fields starting from a variable e.g. $x.f$.

Expressions can be variables, field access, object construction and method invocation. Field assignments are not supported since the calculus is functional.

Example 2.1 (Natural Numbers)**.**

$$\texttt{Zero}(x.\ \epsilon)$$
$$\forall x.\ x :: \texttt{Zero} \Rightarrow x :: \texttt{Nat}$$
$$\texttt{Succ}(x.\ x.p :: \texttt{Nat})$$
$$\forall x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat} \Rightarrow x :: \texttt{Nat}$$
$$\texttt{prev}(x.\ x :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}]$$
$$\texttt{prev}(x.\ x :: \texttt{Zero}) : [y.\ y :: \texttt{Nat}] := \textbf{new}\ \texttt{Zero}()$$
$$\texttt{prev}(x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}] := x.p$$

Example 2.1 shows a program defining natural numbers. The program has constructors `Zero` representing zero and `Succ` representing the successor of its field p. The inheritance of `Zero` and `Succ` to `Nat` is expressed through the two constraint entailment rules. The method `prev` is abstractly declared to be a function from `Nat` to `Nat`. It has two implementations, one for `Zero` and one for `Succ`.

Constraint System

The constraint system is given in the style of the sequent calculus and the rules for the constraint system are specified in Figure 2.4. The sequent $\overline{a} \vdash a$ is interpreted as constraint entailment: constraints $\overline{a}$ entail constraint a. The constraints on the left-hand side are referred to as the context and the constraint

on the right-hand side as the constraint entailed by the context. The notion $\overline{a} \vdash \overline{b}$ is used differently than in the sequent calculus. For DC_C it is used as a shortcut for a list of judgments $\overline{a} \vdash b_i$ for each $b_i \in \overline{b}$, meaning that all b_i are entailed by $\overline{a}$.

Rules C-Ident and C-Cut are standard rules of the sequent calculus, the remainder of the rules are specific to the programming language. The standard structural rules of the sequent calculus allowing permutation, weakening and contraction of the context are implicitly assumed to be specified.

The properties of path equivalence are specified with rules C-Refl and C-Subst. Rule C-Refl establishes reflexivity of path equivalence. Rule C-Subst specifies that paths can be substituted with equivalent paths at any position of any other constraint. Other typical rules of equivalence as symmetry and transitivity can be derived from these rules.

Rule C-Class specifies that a direct instance of a class is an instance of that class, describing that $p :: C$ is a weaker relationship than $p.\mathbf{cls} \equiv C$.

With Rule C-Prog it is possible to specify new axioms for the constraint system in programs. This is used to express inheritance declarations between dependent classes: the constraint at the right-hand side of the implication must be $x :: p$, where x is the bound variable of the rule. The rule is restricted to avoid an undecidable constraint system and the restrictions are specified by the well-formedness rule WF-RD in Figure 2.6.

Operational Semantics

The operational semantics is given in Figure 2.7. It is defined as a small-step reduction relation of a heap and an expression. A heap is a list of mappings from variables to objects. Each object is specified by a class and a list of the values of its fields, which are again references in the heap. For heaps h and variables x, $h(x)$ denotes the object referenced by x in h.

The function HC takes a heap and gives the constraints satisfied by all variables of the heap. The function OC converts an object to a list of constraints on a given variable. A Constraint a is satisfied by a heap h, if $HC(h) \vdash a$ holds.

Evaluation in DC_C is a process of moving information from the expression to the heap and the values of DC_C are references in the heap. Evaluation must yield a heap and a variable. During reduction only new objects can be added to the heap, while existing objects remain unchanged.

The congruence rules RC-Field, RC-New and RC-Call propagate reduction to subexpressions. The computation rules R-Field, R-New and R-Call can only be applied when all subexpressions of an expression are reduced to normal forms (variables).

Field access $x.f$ is reducible to the value of $x.f$ in the heap if the heap constraints include $x.f \equiv y$, where y represents the value retrieved from the heap. x does not have field f in the heap if the constraint is not included.

Object construction **new** $C(\overline{f} \equiv \overline{x})$ is reduced to a fresh variable x representing the new object, as well as extending the heap with an object $o = \langle C; \overline{f} \equiv \overline{x} \rangle$ of class C with $\overline{x}$ as the values of the fields of that object. It is checked that a constructor of class C exists in the program and that the constraints $\overline{b}$ specified by the constructor are satisfied by the new object $HC(h), OC(x, o) \vdash \overline{b}$.

A method call $m(x)$ is reduced to the body of the most specific applicable method implementation. Applicability of the implementations and selection of

$$\frac{\overline{c} \vdash e : [x.\ \overline{a}] \qquad \overline{c}, \overline{a} \vdash x.f :: C \qquad \overline{c}, \overline{a}, x.f \equiv y \vdash \overline{b} \qquad x \notin FV(\overline{b})}{\overline{c} \vdash e.f : [y.\ \overline{b}]}\ \text{T-Field}$$

$$\frac{\overline{c} \vdash x :: C}{\overline{c} \vdash x : [y.\ y \equiv x]}\ \text{T-Var}$$

$$\frac{\overline{c} \vdash e : [x.\ \overline{a}] \qquad \langle \overline{a'}; \overline{b} \rangle \in MType(m, x, y) \qquad \overline{c}, \overline{a} \vdash \overline{a'} \qquad \overline{c}, \overline{a}, \overline{b} \vdash \overline{b'} \qquad x \notin FV(\overline{b'})}{\overline{c} \vdash m(e) : [y.\ \overline{b'}]}\ \text{T-Call}$$

$$\frac{\forall i.\ \overline{c} \vdash e_i : [x_i.\ \overline{a_i}] \qquad C(x.\ \overline{b'}) \in P \qquad \overline{b} = (x.\mathbf{cls} \equiv C), \bigcup_i \overline{a_i}\{\!\{x_i \mapsto x.f_i\}\!\} \qquad \overline{c}, \overline{b} \vdash \overline{b'}}{\overline{c} \vdash \mathbf{new}\ C(\overline{f} \equiv \overline{e}) : [x.\ \overline{b}]}\ \text{T-New}$$

$$\frac{\overline{c} \vdash e : [x.\ \overline{a'}] \qquad \overline{c}, \overline{a'} \vdash \overline{a}}{\overline{c} \vdash e : [x.\ \overline{a}]}\ \text{T-Sub}$$

Figure 2.5: Type Assignment

the most specific one are determined by constraints. A method declaration is applicable if its arguments are entailed by the context. Set S contains the applicable methods. A method declaration with argument constraints $\overline{a}$ is more specific than a method declaration with argument constraints $\overline{a'}$ if $\overline{a'} \vdash \overline{a}$, but not the other way around $\neg\overline{a} \vdash \overline{a'}$.

Type Checking

Type assignment for expressions is defined in Figure 2.5. The context of type assignment is a list of constraints providing information about variables that occur in the expression. The typing rules ensure that all free variables of the assigned types appear in the context. An expression type $[x.\ \overline{a}]$ is a collection of constraints $\overline{a}$ with a bound variable x that will hold for all possible values of the expression at runtime, if the runtime environment satisfies the constraints of the context. The relation does not guarantee unique type assignments, since it is possible to have different constraints satisfied by an expression. The rule T-Sub explicitly allows weakening the type of an expression.

The type of a variable x is specified by rule T-Var. The type of x asserts equivalence of the bound variable y of the type to x. Further it is checked that $x :: C$ can be satisfied by the context for some class C.

Rule T-Field specifies type assignment of a field access $e.f$. The constraints of such a field access are the constraints on $x.f$ entailed by the type of e , where x is the bound variable of the type of e. The elimination of x is done via the usage of constraints free of x entailed by the type of e and $y \equiv x.f$, where y is used as the bound variable of the type of $e.f$ and $\overline{c}, \overline{a} \vdash x.f :: C$ is used to check if field f of e is available at runtime.

Rule T-Call specifies type assignment of method calls $m(e)$. The type of e

$$\frac{FV(\overline{a}) = \{x\}}{\text{wf } C(x.\ \overline{a})}\ \text{WF-CD}$$

$$\frac{FV(\overline{a}) = \{x\} \qquad FV(\overline{b}) = \{x, y\}}{\text{wf } (m(x.\ \overline{a}) : [y.\ \overline{b}])}\ \text{WF-MS}$$

$$\frac{FV(\overline{a}) = \{x\} \qquad x :: C' \in \overline{a}}{\text{wf } (\forall x.\ \overline{a} \Rightarrow x :: C)}\ \text{WF-RD}$$

$$\frac{FV(\overline{a}) = \{x\} \qquad FV(\overline{b}) = \{x, y\} \qquad \overline{a} \vdash e : [y.\ \overline{b}]}{\text{wf } (m(x.\ \overline{a}) : [y.\ \overline{b}] := e)}\ \text{WF-MI}$$

$$\frac{\begin{array}{c}\forall D \in P.\ \text{wf } D \\ \forall m.\ \forall \langle \overline{a}; \overline{b} \rangle, \langle \overline{a'}; \overline{b'} \rangle \in MType(m, x, y).\ \overline{b} = \overline{b'} \\ \forall m.\ unique(m) \\ \forall m.\ \forall \langle \overline{a}; \overline{b} \rangle \in MType(m, x, y).\ complete(m, [x.\ \overline{a}])\end{array}}{\text{wf } P}\ \text{WF-Prog}$$

Figure 2.6: Type Checking

is $[x.\ \overline{a}]$. The rule checks the existence of a declaration of m, whose parameter constrains are $\overline{a'}$ and return type constraints are $\overline{b}$. It is checked with $\overline{c}, \overline{a} \vdash \overline{a'}$ if the parameter constraints are entailed by the type of e. The type constraints of $m(e)$ are derived from the declared return type of m and the type of the argument e. The argument variable x is eliminated, because it does not appear in the context and the entailed constraints of the return types and the argument types free of x are taken.

Rule T-New specifies type assignment of object constructions. The type $[x.\ \overline{b}]$ of an object construction **new** $C(\overline{f} \equiv \overline{e})$ consists of a constraint stating that C is the class of the object and the constraints of its fields. The field constraints are taken from the types of the expressions e_i assigned to the fields f_i. Additionally $\overline{c}, \overline{b} \vdash \overline{b'}$ checks that the new object satisfies the constraints of at least one constructor $C(x.\ \overline{b'})$ of class C.

Typechecking a program is defined in Figure 2.6. A program is well-formed if all its declarations are well-formed, method implementations need to be unique and complete and all declarations of a method need to have the same return type. Completeness and uniqueness of method implementations are properties of well-formed heaps. A declaration is well-formed if its used variables are bound. Rule WF-MI additionally checks if the body of a method declaration respects the declared return type. Rule WF-RD restricts constraint derivation rules: constraints of the form $x :: C$ can only be derived if some other class of x is known. Such derivation rules are used to encode inheritance between dependent classes.

$$o ::= \langle C; \overline{f} \equiv \overline{x} \rangle \qquad \text{(objects)}$$
$$h ::= \overline{x} \mapsto \overline{o} \quad (x_i \text{ distinct}) \qquad \text{(heaps)}$$

$$OC(x, o) = (x.\mathbf{cls} \equiv C, x.\overline{f} \equiv \overline{x}) \qquad \text{where } o = \langle C; \overline{f} \equiv \overline{x} \rangle$$
$$HC(h) = \bigcup_i OC(x_i, o_i) \qquad \text{where } h = \overline{x} \mapsto \overline{o}$$

$$\frac{x \notin dom(h) \qquad \text{o} = \langle C; \overline{f} \equiv \overline{x} \rangle \qquad C(x.\ \overline{b}) \in P \qquad HC(h), OC(x, o) \vdash \overline{b}}{\langle h; \mathbf{new}\ C(\overline{f} \equiv \overline{x}) \rangle \to \langle h, x \mapsto o; x \rangle}\ \text{R-New}$$

$$\frac{(x.f \equiv y) \in HC(h)}{\langle h; x.f \rangle \to \langle h; y \rangle}\ \text{R-Field}$$

$$S = \{\langle \overline{a}; e \rangle \mid \langle \overline{a}; e \rangle \in MImpl(m, x) \wedge HC(h) \vdash \overline{a}\} \qquad \langle \overline{a}; e \rangle \in S$$
$$\frac{\forall \langle \overline{a'}; e' \rangle \in S.\ (e' \neq e) \longrightarrow (\overline{a'} \vdash \overline{a}) \wedge \neg(\overline{a} \vdash \overline{a'})}{\langle h; m(x) \rangle \to \langle h; e \rangle}\ \text{R-Call}$$

$$\frac{\langle h; e \rangle \to \langle h'; e' \rangle}{\langle h; e.f \rangle \to \langle h'; e'.f \rangle}\ \text{RC-Field}$$

$$\frac{\langle h; e \rangle \to \langle h'; e' \rangle}{\langle h; m(e) \rangle \to \langle j'; m(e') \rangle}\ \text{RC-Call}$$

$$\frac{\langle h; e \rangle \to \langle h'; e' \rangle}{\langle h; \mathbf{new}\ C(\overline{f} \equiv \overline{x}, f \equiv e, \overline{f'} \equiv \overline{e'}) \rangle \to \langle h'; \mathbf{new}\ C(\overline{f} \equiv \overline{x}, f \equiv e', \overline{f'} \equiv \overline{e'}) \rangle}\ \text{RC-New}$$

Figure 2.7: Operational Semantics

Chapter 3

DC_C Implementation

In this chapter we show our implementation of the DC_C Calculus presented in Section 2.2.1. We use Scala [OMM+04] for the implementation. The constraint system will be solved using SMT with Z3 prover [dMB08].

3.1 Constraint System

The DC_C constraint system as presented in Figure 2.4 is specified in the style of the sequent calculus. In order to use a SMT solver on the constraint system we model the sequent calculus in sorted first-order logic. The integration of the SMT solving with the Scala implementation is done via calling the SMT solver as a sub-process and using the SMT-Lib format for communicating information between the Scala implementation and the solver process.

In sorted first order logic variables are not limited to a single domain. Variable bindings (e.g. in quantifiers) have an assigned sort as seen in Figure 2.1a, where the all-quantified variable x has the sort $List[Int]$. The notion for such a sorted variable is $x : S$ for variables x and sorts S. We also use let-bindings, if statements, pattern matching and primitive recursive functions. Let bindings are written with the keywords **let** and **in**. If statements can be written inline as $b\ ?\ x\ :\ y$ or using the keywords **if** , **then** and **else**. Pattern matching is

$elem(c : Constraint, cs : List[Constraint]) : Bool = cs$ **match**
  $nil \Rightarrow false$
  $hd :: tl \Rightarrow c = hd\ ?\ true\ :\ elem(c, tl)$

$concat(l_1 : List[Constraint], l_2 : List[Constraint]) : List[Constraint] = l_1$ **match**
  $nil \Rightarrow l_2$
  $hd :: tl \Rightarrow hd :: concat(tl, l_2)$

Figure 3.1: List Functions

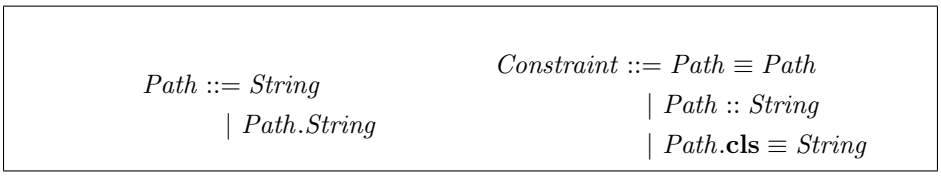


Figure 3.2: Sorts

written using the notion x **match** to start matching some x and patterns are specified using $y \Rightarrow z$, where variables occurring in y are bound in z. Pattern matching can be written as a combination of if statements and let bindings and is therefore used for better readability.

The parametrized sort *List* models lists and is predefined in Z3. The constructor *nil* represents the empty list and the constructor *insert* is used for list constructions and $insert(x, l)$ will also be written as $x :: l$ for elements x and lists l with matching sorts. Selectors *head* and *tail* are used to extract the head and tail from a list. We write $[x_1, ..., x_n]$ as a shortcut for $x_1 :: ... :: x_n :: nil$ for lists with elements $\{x_i | i > 0 \wedge i \leq n\}$. Figure 3.1 defines functions for list concatenation as well as element-of checking. We write $l_1 +\!\!+ l_2$ for concatenating two lists l_1 and l_2 and $x \in l$ for checking if element x is in list l.

3.1.1 Naïve approach

The goal of this model of the constraint system is to be close to the given sequent calculus and to preserve the structure of the calculus rules in the first-order formulae.

First we define a set of sorts, predicates and functions. These definitions form the general structure of the model and provide basic functionality needed in the modeling of the calculus rules.

Sorts for paths and constraints are defined in Figure 3.2. The definitions are similar to the syntax specification in Figure 2.3. A path is either a variable name or a field path consiting of another path followed by a field name, where variable and field names are modeled as strings. The three types of DC_C constraints are translated as follows in our model:

- For two paths p, q the constraint $p \equiv q$ requires p and q to be equivalent.
- For a path p and a class name C modeled as a string, $p :: C$ requires path p to be an instance of class C.
- For a path p and a class name C modeled as a string, $p.\mathbf{cls} \equiv C$ requires path p to be directly instantiated by class C.

Functions defining path substitution are given in Figure 3.3. Path substitution is the process of substituting the variable name of a path with another path. Therefore substitution can only occur on the innermost part of a path and substitution only happens if the variable name of the path equals the variable name to be substituted. Substitution for constraints propagates path substitution to each path contained within a constraint.

$$
\begin{array}{l}
\mathit{subst\text{-}path}(p_1 : \mathit{Path}, x : \mathit{String}, p_2 : \mathit{Path}) : \mathit{Path} = p_1 \ \textbf{match} \\
\quad y \Rightarrow x = y \ ?\ p_2 \ :\ p_1 \\
\quad q.f \Rightarrow \mathit{subst\text{-}path}(q, x, p_2).f \\
\\
\mathit{subst\text{-}constraint}(c : \mathit{Constraint}, x : \mathit{String}, p : \mathit{Path}) \colon \mathit{Constraint} = c \ \textbf{match} \\
\quad q_1 \equiv q_2 \Rightarrow \mathit{subst\text{-}path}(q_1, x, p) \equiv \mathit{subst\text{-}path}(q_2, x, p) \\
\quad q :: C \Rightarrow \mathit{subst\text{-}path}(q, x, p) :: C \\
\quad q.\textbf{cls} \equiv C \Rightarrow \mathit{subst\text{-}path}(q, x, p).\textbf{cls} \equiv C \\
\\
\mathit{subst\text{-}constraints}(cs : \mathit{List}[\mathit{Constraint}], x : \mathit{String}, p : \mathit{Path}) \colon \mathit{List}[\mathit{Constraint}] = \\
\quad cs \ \textbf{match} \\
\qquad \epsilon \Rightarrow \epsilon \\
\qquad \mathit{hd} :: \mathit{tl} \Rightarrow \mathit{subst\text{-}constraint}(\mathit{hd}, x, p) :: \mathit{subst\text{-}constraints}(\mathit{tl}, x, p)
\end{array}
$$

Figure 3.3: Path Substitution Functions

E.g. for path equivalence constraints $p_1 \equiv p_2$ a substitution from x to q substitutes both p_1 and p_2 with the substitution from x to q. Substitution for lists of constraints applies the substitution to each constraint contained in the list. Substitution for strings x and paths p will be written for constraints a as $a_{x \mapsto p}$ and for constraint lists $\overline{a}$ as $\overline{a}_{x \mapsto p}$, function application is distinguishable based on the sort of the first argument.

$$
\begin{array}{l}
\mathit{class}(\mathit{String}) \\
\mathit{variable}(\mathit{String}) \\
\mathit{in\text{-}program}(\mathit{String}, \mathit{List}[\mathit{Constraint}], \mathit{Constraint}) \\
\mathit{entails}(\mathit{List}[\mathit{Constraint}], \mathit{Constraint}) \\
\\
\mathit{Entails}(cs_1 : \mathit{List}[\mathit{Constraint}], cs_2 : \mathit{List}[\mathit{Constraint}]) = cs_2 \ \textbf{match} \\
\quad \epsilon \Rightarrow \mathit{true} \\
\quad \mathit{hd} :: \mathit{tl} \Rightarrow \mathit{entails}(cs_1, \mathit{hd}) \wedge \mathit{Entails}(cs_1, \mathit{tl}) \\
\\
\mathit{subst}(c_1 : \mathit{Constraint}, x : \mathit{String}, p : \mathit{Path}, c_2 : \mathit{Constraint}) = \\
\quad \mathit{subst\text{-}constraint}(c_1, x, p) = c_2
\end{array}
$$

Figure 3.4: Predicates

Predicates are declared in Figure 3.4. The predicate $\mathit{class}(C)$ ensures that C is a valid class name and $\mathit{variable}(x)$ ensures that x is a valid variable name of the program. Class- and variable names are modeled as strings. The predi-

cate *in-program* models the existence check of program entailments as seen in rule C-Prog in Figure 2.4. For strings x, constraint lists $\overline{a}$ and constraints a, *in-program*$(x, \overline{a}, a)$ ensures that a declaration $\forall x.\ \overline{a} \Rightarrow a$ exists in the program. The predicate *entails* models the entailment judgement of the sequent calculus. The predicate *Entails* models the entailment judgement for multiple constraints, which is defined as $\overline{a} \vdash \overline{b} = \bigwedge_{b \in \overline{b}} \overline{a} \vdash b$. For constraint lists $\overline{a}$ and constraints a *entails*$(\overline{a}, a)$ ensures that $\overline{a} \vdash a$ holds. For constraint lists $\overline{a}$ and constraint lists $\overline{b}$, *Entails*$(\overline{a}, \overline{b})$ holds if *entails*$(\overline{a}, b)$ holds for each $b \in \overline{b}$. For better readability in the rules, we write *entails*$(\overline{a}, a)$ as $\overline{a} \vdash a$ and *Entails*$(\overline{a}, \overline{b})$ as $\overline{a} \vdash \overline{b}$. The writing styles of *entails* and *Entails* can be distinguished based on the sort of the right-hand side argument. The predicate *subst* models constraint equality after substitution. For constraints a, b, strings x, paths p *subst*(a, x, p, b) holds if x substituted with p in a equals b.

$$\forall c : \mathit{Constraint}.\ [c] \vdash c \qquad \text{(C-Ident)}$$

$$\forall p : \mathit{Path}.\ \mathit{nil} \vdash p \equiv p \qquad \text{(C-Refl)}$$

$$\forall \overline{a} : \mathit{List}[\mathit{Constraint}], p : \mathit{Path}, C : \mathit{String}. \qquad \text{(C-Class)}$$
$$\mathit{class}(C) \wedge \overline{a} \vdash p.\mathbf{cls} \equiv C \rightarrow \overline{a} \vdash p :: C$$

$$\forall \overline{a}, \overline{a'} : \mathit{List}[\mathit{Constraint}], b, c : \mathit{Constraint}. \qquad \text{(C-Cut)}$$
$$\overline{a} \vdash c \wedge c :: \overline{a'} \vdash b \rightarrow \overline{a} +\!\!+ \overline{a'} \vdash b$$

$$\forall \overline{a} : \mathit{List}[\mathit{Constraint}], x : \mathit{String}, p_1, p_2 : \mathit{Path}, a, a_1, a_2 : \mathit{Constraint}. \qquad \text{(C-Subst)}$$
$$\overline{a} \vdash p_2 \equiv p_1 \wedge \mathit{variable}(x) \wedge \mathit{subst}(a, x, p_1, a_1) \wedge \mathit{subst}(a, x, p_2, a_2)$$
$$\wedge\, \overline{a} \vdash a_1 \rightarrow \overline{a} \vdash a_2$$

$$\forall \overline{a}, \overline{b} : \mathit{List}[\mathit{Constraint}], a : \mathit{Constraint}, x : \mathit{String}, p : \mathit{Path}. \qquad \text{(C-Prog)}$$
$$\mathit{in\text{-}program}(x, \overline{a}, a) \wedge \overline{b} \vdash \overline{a}_{x \mapsto p} \rightarrow \overline{b} \vdash a_{x \mapsto p}$$

$$\forall \overline{a} : \mathit{List}[\mathit{Constraint}], a, b : \mathit{Constraint}. \qquad \text{(C-Weak)}$$
$$\overline{a} \vdash b \rightarrow a :: \overline{a} \vdash b$$

$$\forall \overline{a}, \overline{b} : \mathit{List}[\mathit{Constraint}], a : \mathit{Constraint}. \qquad \text{(C-Perm)}$$
$$\forall b : \mathit{Constraint}.\ b \in \overline{a} \rightarrow b \in \overline{b} \wedge \neg b \in \overline{a} \rightarrow \neg b \in \overline{b}$$
$$\wedge\, \overline{a} \vdash a \rightarrow \overline{b} \vdash a$$

Figure 3.5: First-order Model Of Constraint Entailment

With the previous sort, predicate and function definitions we can model the calculus rules. A first-order representation of the calculus rules is given in Figure 3.5

We start with the structural rules for weakening, contraction and permutation of the sequent calculus that are implicitly assumed in the DC_C constraint system. Since the entailment judgement of the DC_C constraint system has only a single constraint on its right-hand side we only need to consider structural rules on the left-hand side.

Weakening allows the addition of arbitrary elements to a sequence. The weakening rule in the DC_C sequent calculus would look as follows:

$$\frac{\overline{a} \vdash b}{a, \overline{a} \vdash b}\ \text{C-Weak}$$

The rule states that the context of an entailment can always be restricted. The first-order representation of this rule models this as follows: For constraint lists $\overline{a}$ and constraints a, b, if $\overline{a} \vdash b$ holds then $a :: \overline{a} \vdash b$ holds as well. Here we restrict the context $\overline{a}$ by prepending an additional constraint a.

Contraction and permutation assure that the ordering of a sequent and the existence of multiple occurences of the same element in a sequent do not matter. Rule C-Perm covers both cases. For constraint lists $\overline{a}$, $\overline{b}$ and constraints a: if $\overline{a} \vdash a$ holds then $\overline{b} \vdash a$ holds as well if both lists $\overline{a}$ and $\overline{b}$ consist of the same elements. This is checked with an additional quantifier ranging over constraints b. If b is an element of $\overline{a}$ then it must also be an element of $\overline{b}$ and if b is not in $\overline{a}$ then it must not be in $\overline{b}$. This ensures that the ordering of the context does not matter and since we do not require the length of $\overline{a}$ and $\overline{b}$ to be equal it does also allow for the removal of multiple occurences of the same element as long as one remains.

The modeling of the remaining rules follow the structure of the rules specified in the DC_C constraint system in Figure 2.4.

Rules C-Ident and C-Refl are closure rules. Rule C-Ident states that each constraint entails itself. For all constraints c, c is entailed by a context consisting of the identical constraint $[c]$. Rule C-Refl states that path equivalence is reflexive, each path is equivalent to itself without any prerequisites. For all constraints p, $p \equiv p$ is entailed by the empty context *nil*.

Rule C-Class establishes the fact that an object created with a constructor of a class is an instance of that class. For all constraints $\overline{a}$, paths p and strings C: $p :: c$ is entailed by context $\overline{a}$ if $p.\mathbf{cls} \equiv C$ is entailed by context $\overline{a}$. We make sure that C is a classname by requiring that $\mathit{class}(C)$ holds.

Rule C-Cut is a standard rule of the sequent calculus. It states that if a constraint c is entailed by a context $\overline{a}$ and a constraint b is entailed by a context $\overline{b}$ containing c then b is also entailed by $\overline{a}$ and $\overline{b}$ without c, meaning that we can replace c in $\overline{b}$ with constraints $\overline{a}$ entailing c. In our model we use the list structure of our context to make this replacement. In the model the context of b is a list $\overline{b} = c :: \overline{a'}$ and the context of c is $\overline{a}$. We then concatenate $\overline{a}$ with $\overline{a'}$, which is the remainder of b without c to eliminate c from the context of b.

Rule C-Subst establishes that equivalent paths can be substituted within constraints. A constraint a_2 is entailed by $\overline{a}$ if a constraint a_1 is entailed by $\overline{a}$ and $\overline{a}$ entails the equivalence of paths p_2 and p_1. For this a_2 needs to be equal to the substitution of a from x to p_2 and a_1 needs to be equal to the substitution of a from x to p_1. $\mathit{variable}(x)$ ensures that x is a valid variable name.

Rule C-Prog applies program entailments expressing inheritance to the calculus. For constraint lists $\overline{a}$, $\overline{b}$, constraints a, strings x and paths p: if a program entailment $\forall x.\ \overline{a} \Rightarrow a$ exists in the program and $\overline{a}$ is entailed by $\overline{b}$ then a is entailed by $\overline{b}$ as well. x is eliminated in $\overline{a}$ and a through substitution, since x is the bound variable name of the program entailment. The existence check of the program entailment is modeled with the predicate $\mathit{in\text{-}program}(x, \overline{a}, a)$.

Example 3.1 (Constraint entailment)**.** In this example we will show the application of the sequent calculus for constraint entailment and compare it to the application of our model of the calculus. We will show the resolution of the

entailment $x \equiv y \vdash y \equiv x$ stating the symmetry of path equivalence.

We begin with the application of the sequent calculus:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon \vdash y \equiv y}\,\text{C-Refl}}{x \equiv y \vdash y \equiv y}\,\text{C-Weak}}{x \equiv y \vdash y \equiv x_{x \mapsto y}} \qquad \dfrac{}{x \equiv y \vdash x \equiv y}\,\text{C-Ident}}{x \equiv y \vdash y \equiv x_{x \mapsto x}}\,\text{C-Subst}$$

For showing that $x \equiv y \vdash y \equiv x$ holds we apply rule C-Subst. We choose $x \equiv y$ to be our equivalent paths used for substitution. To show that $x \equiv y$ is entailed by $x \equiv y$ we use rule C-Ident to close this branch. What is left to show is that $x \equiv y$ entails $y \equiv x_{x \mapsto y}$. We apply the substitution to gain the constraint $y \equiv y$. Finally we apply rule C-Weak to remove the information $x \equiv y$ from the context and apply rule C-Refl to show the entailment $\epsilon \vdash y \equiv y$ and close the branch. With this derivation we showed $x \equiv y \vdash y \equiv x_{x \mapsto x}$. We apply the substitution to obtain our original goal $x \equiv y \vdash y \equiv x$.

We now try to find a solution for the same entailment in our model. The entailment has two valid variable names x and y which we need to add in our model. We do so by requiring $\mathit{variable}(\texttt{"x"})$ and $\mathit{variable}(\texttt{"y"})$. Since there are no valid classnames we do not have to add any requirements for the *class* predicate.

To show $x \equiv y \vdash y \equiv x$ we instantiate rule C-Subst. Since we want to show the entailment we need it to appear on the right-hand side of the implication, for this we instantiate $\overline{a}$ as $[x \equiv y]$ and a_2 as $y \equiv x$. We then choose $x :=$ `"x"`, $p_1 := y$ and $p_2 := x$ for usage in the substitutions. The instantiation of x with `"x"` ensures that $\mathit{variable}(x)$ holds. With the instantiations of x, p_1, p_2 and a_2 we get $\mathit{subst}(a, \texttt{"x"}, y, a_1)$ and $\mathit{subst}(a, \texttt{"x"}, x, y \equiv x)$. We now need to choose a and a_2 such that both *subst* requirements are fulfilled. We instantiate $a := y \equiv x$ and can resolve $\mathit{subst}(y \equiv x, \texttt{"x"}, x, y \equiv x)$ requiring $y \equiv x_{\texttt{"x"} \mapsto x} = y \equiv x$. Applying the substitution shows that $y \equiv x$ is equal to $y \equiv x$. From $\mathit{subst}(y \equiv x, \texttt{"x"}, y, a_1)$ we know that a_1 needs to be equal to $y \equiv x_{\texttt{"x"} \mapsto y} = y \equiv y$ and we choose $a_1 := y \equiv y$. This leaves us with two new subgoals:

1. $[x \equiv y] \vdash x \equiv y$ and
2. $[x \equiv y] \vdash y \equiv y$.

To show (1) we instantiate rule C-Ident with $c := x \equiv y$.
To show (2) we instantiate rule C-Weak with $a := x \equiv y$ and $\overline{a} := \mathit{nil}$ to match the context and $b := y \equiv y$ to match the constraint to be entailed by this context. This leads to the new subgoal $\mathit{nil} \vdash y \equiv y$ which can be resolved using rule C-Refl and instantiating $p := y$.

By comparing the application of the sequent calculus with the application of our model, we can see that both derivations are similar. Both derivations use the same rule applications in the same order to obtain the result that the entailment holds. We can also observe that we chose the variable instantiations of rule C-Subst in the application of our model to match the path equivalence and substitution used in the application of the sequent calculus.

3.1.2 Rule Refinement

In Example 3.1 we have seen the application of our model of the constraint entailment. The structure of the derivation followed the application of the sequent calculus. In the example we used rule C-Subst and chose the quantified variables to fulfill the various requirements of the rule. These requirements included *variable* and *subst* as well as the relationship between variables a, a_1 and a_2. We found it difficult for the solver to make these variable instantiations and we refine the modeled rules to improve usage with the solver.

Rule C-Subst

We start with refining rule C-Subst. The identified problems of this rule are

1. the amount of quantified variables and
2. the non-algorithmic nature of the rule.

As observed in (1) the rule quantifies over seven variables. This leads to the explosion of the search space for this rule and the quantifier instantiation routines of the solver have a problem to instantiate the quantified variables in a fitting way. To solve this problem we try to ground at least some of the quantified variables of the rule. We notice that all variable names and paths are known at runtime and none of the rules introduces new ones. We can ground variables x, p_1 and p_2 of rule C-Subst by enumerating each possible combination of those variables and by creating a specialized subst rule for each of these combinations.

Following this we can transform the rule C-Subst given in Figure 3.5 by removing the quantified variables x, p_1 and p_2 and since we enumerate over each variable name we do also remove the $variable(x)$ requirement. With this we reduced the number of quantified variables of the rule to four. We note that these variables are now free in the rule and would need to be supplied from the outside. Since we do not want a single rule with free variables, but rather a template for generating subst rules, we treat the now free variables as placeholders for a routine that replaces these for each possible combination. For better visibility we write the placeholders in bold:

$$\begin{aligned} &\forall \overline{a} : List[Constraint], a, a_1, a_2 : Constraint. \qquad \text{(C-Subst)} \\ &\quad \overline{a} \vdash \mathbf{p_2} \equiv \mathbf{p_1} \wedge subst(a, \mathbf{x}, \mathbf{p_1}, a_1) \wedge subst(a, \mathbf{x}, \mathbf{p_2}, a_2) \wedge \overline{a} \vdash a_1 \\ &\quad \rightarrow \overline{a} \vdash a_2 \end{aligned}$$

For (2) we take a look the instantiations of a, a_1 and a_2 made in Example 3.1. In the example we started with instantiating a_2 to match the entailment to be shown. Afterwards we chose a such that the substitution of x with p_2 in a would equal a_2. We see that to instantiate a we solved a substitution where a is used as the input and a_2 as the output of the substitution. Finally we instantiated a_1 with x substituted with p_1 in a. We observe that this resembles a linear instantiation scheme starting with a_2, but we note that we had to make an educated guess for choosing a. So with the current rule, the solver needs to make random instantiations for a to afterwards check if the requirements on a are met instead of following the observed instantiation pattern we did manually. To resolve this problem we need to change the rule that a can be instantiated by

$$
\begin{array}{l}
\mathit{gen\text{-}path}(p_1 : \mathit{Path}, p_2 : \mathit{Path}, x : \mathit{String}) : \mathit{Path} = \\
\quad \textbf{if } p_1 = p_2 \\
\quad\quad \textbf{then } x \\
\quad\quad \textbf{else } p_1 \textbf{ match} \\
\quad\quad\quad y \Rightarrow y \\
\quad\quad\quad q.f \Rightarrow \mathit{gen\text{-}path}(q, p_2, x).f \\
\\
\mathit{gen\text{-}constraint}(c : \mathit{Constraint}, p : \mathit{Path}, x : \mathit{String}) : \mathit{Constraint} = c \textbf{ match} \\
\quad q_1 \equiv q_2 \Rightarrow \mathit{gen\text{-}path}(q_1, p, x) \equiv \mathit{gen\text{-}path}(q_2, p, x) \\
\quad q :: C \Rightarrow \mathit{gen\text{-}path}(q, p, x) :: C \\
\quad q.\textbf{cls} \equiv C \Rightarrow \mathit{gen\text{-}path}(q, p, x).\textbf{cls} \equiv C
\end{array}
$$

Figure 3.6: Path Generalization Functions

direct derivation from a_2. As previously stated we used a on an input position and a_2 on the output position of a function call. To reverse these positions, we introduce *generalization* as the inverse of substitution.
Functions for generalizing paths and constraints are defined in Figure 3.6. Generalization for paths functions reverse of substitution for paths. Instead of substituting a variable with a path we generalize a subpath with a variable. This generalization of a subpath cannot happen at random places inside a path, but only starting on the innermost/leftmost variable of the path (the root of the abstract syntax tree (AST) of that path). For example we can generalize $x.f$ with y in $x.f.g$ to obtain $y.g$, but we cannot generalize $f.g$ with h in $x.f.g$ since f is not the innermost occurrence in $x.f.g$. So we can only start generalization from the variable of a path and not its fields. For paths p, q and strings x path generalization is defined such that $\mathit{gen}(\mathit{subst}(p, x, q), q, x) = p$.
Generalization for constraints does, as substitution for constraints did, propagate path generalization to the paths contained in the constraint to be generalized.

For the notation of generalization we use $c_{p \mapsto x}$ for generalizing paths p with variable names x in constraints c. This notion is similar to the one used for substitution. We can distinguish both notations based on the swapped positions of p and x.
We can now solve the problem of choosing an instantiation for a. With the concept of generalization as previously presented we can choose a to be a generalization of a_2, where a_2 is used in an input position and the output is used as the instantiation of a. This brings the instantiations of a, a_1 and a_2 into a derivation chain starting from a_2 over a to a_1. Since we can derive a and a_1 from a_2 through a defined function, we can remove a and a_1 from the list of quantified variables.

$$
\begin{array}{l}
\forall \overline{a} : \mathit{List}[\mathit{Constraint}], a_2 : \mathit{Constraint}. \qquad \text{(C-Subst)} \\
\quad \textbf{let } a := a_{2\,\mathbf{p_2} \mapsto \mathbf{x}} \textbf{ in} \\
\quad\quad \textbf{let } a_1 := a_{\mathbf{x} \mapsto \mathbf{p_1}} \\
\quad\quad \textbf{in } \overline{a} \vdash \mathbf{p_2} \equiv \mathbf{p_1} \wedge \overline{a} \vdash a_1 \\
\rightarrow \overline{a} \vdash a_2
\end{array}
$$

Figure 3.7: C-Subst Template

We incorporate these changes into the template presented in step (1) to obtain the new template for rule C-Subst as shown in Figure 3.7. In this template we reduced the amount of quantified variables to two. These two quantified variables can be used to match the right-hand side of the implication. We used let bindings to derive a from a_2 and a_1 from a.

We can combine the observations made in steps (1) and (2) to improve the rule generation out of the template. A procedure to generate subst rules out of the template needs to enumerate over all possible combinations of variable names and path pairs.

The first observation to make is that in order to be complete we do not need all possible combinations. We can safely remove combinations where $x = p_1 = p_2$ for strings x and paths p_1, p_2, since there is no new information to be gained out of such a combination. For a we derive $a_{2\,p_2 \mapsto x}$ and since $p_2 = x$ we obtain $a := a_2$. The following derivation would then be $a_{x \mapsto p_1}$ and since $x = p_1$ and $a = a_2$ we obtain $a_1 := a_2$. This results in the implication:

$$\overline{a} \vdash p_2 \equiv p_1 \wedge \overline{a} \vdash a_1 \rightarrow \overline{a} \vdash a_2$$

Since $p_1 = p_2$ and path equivalence is reflexive $\overline{a} \vdash p_2 \equiv p_1$ holds for any $\overline{a}$ and since $a_2 = a = a_1$ we remain with

$$\overline{a} \vdash a_2 \rightarrow \overline{a} \vdash a_2$$

from where we can not gain any information. More so this could lead for the solver to repetitively try to use this rule and loop.

The second observation follows from the first. We can omit

1. the generalization $a_{2\,p_2 \mapsto x}$ if $x = p_2$, since $a_{2\,x \mapsto x} = a_2$.
2. the substitution $a_{x \mapsto p_1}$ if $x = p_1$, since $a_{x \mapsto x} = a$.
3. the entailment check $\overline{a} \vdash p_2 \equiv p_1$ if $p_1 = p_2$, since $\overline{a} \vdash p \equiv p$ is derivable through using rule C-Weak until we obtain $\mathit{nil} \vdash p \equiv p$ followed by using rule C-Refl to close the proof.

By omitting the generalization or the substitution we can remove the respective let binding and replace the now free variable in its subexpression. Example 3.2 shows this for the case $x = p_1$.

Example 3.2 (Optimized subst rule with $x = p_1$)**.**

$$\begin{aligned}&\forall \overline{a} : \mathit{List}[\mathit{Constraint}], a_2 : \mathit{Constraint}.\\ &\quad \textbf{let } a := a_{2\,\mathbf{p_2 \mapsto x}}\\ &\quad \textbf{in } \overline{a} \vdash \mathbf{p_2} \equiv \mathbf{p_1} \wedge \overline{a} \vdash a\\ &\rightarrow \overline{a} \vdash a_2\end{aligned}$$

Rule C-Prog

Rule C-Prog has similar problems as the ones observed for rule C-Subst. The rule is again defined in a non-algorithmic way.
The check if a program entailment exists in the program is done via the predicate *in-program*(x, $\overline{a}$, a) for strings x, constraint lists $\overline{a}$ and constraints a, where x denotes a variable binding that can occur in $\overline{a}$ and a. The predicate models the existence of a declaration $\forall x.\ \overline{a} \Rightarrow a$ in the program. This is aggravated by the fact that the rule requires us to find a substitution from x to some path p to eliminate the bound variable of the program entailment x, since it would be otherwise free in the constraint entailment.

To solve this problem we apply the same technique as we previously did with rule C-Subst.
We can enumerate over all possible combinations of variable names and paths to eliminate the need to find a proper substitution. As well as finding a way to express the existence check of program entailments in a derivable way, where the quantified constraint a can be used as an input and would result in an output to be usable as an instantiation for $\overline{a}$.

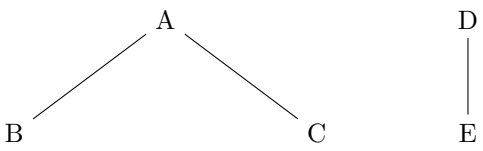
With this requirement specification we see that we want to have some sort of lookup function ranging from constraint a to constraints $\overline{a}$. Such a lookup function would need to be specifically generated for a program and we cannot give an universal function definition.

We develop the generation process of a lookup function which models the existence check from rule C-Prog through a series of examples. Since such a lookup function needs to be specific to a given program, we start by defining a set of program entailments in Example 3.3.

Example 3.3 (Program entailment class hierarchy)**.**
The program for which we want to generate a lookup function contains the following entailment declarations.

$$\begin{aligned}&\forall x.\ x :: B \Rightarrow x :: A\\ &\forall x.\ x :: C, x.f :: D \Rightarrow x :: A\\ &\forall x.\ x :: E \Rightarrow x :: D\end{aligned}$$

Since program entailments are used to express inheritance relations, the given entailments form the following class hierarchy.



We recall from Figure 2.6 that a program entailment $\forall y.\ \overline{b} \Rightarrow b$ is only well-formed if constraint b is of the form $q :: Cls$ and $q :: Cls' \in \overline{b}$.
With this in mind, rule C-Prog states that to show that path q is an instance of class Cls it suffices to show that q is an instance of subclass Cls' and that the field requirements of Cls' are fulfilled.

To translate this into a lookup function, we use b as an argument to the function and $\overline{b}$ as our return value. Since a function needs to be total we need to combine all existing program entailments into one function, as well as the need for a result in the case where no match exists.

Since the well-formedness requires $\overline{b}$ to contain at least one element, $\overline{b}$ cannot be empty and we use the empty list *nil* as the return value for the non matching case.
To differentiate between program entailments we use if statements to check if the input matches one of the program entailments.
Such a lookup function is shown in Example 3.4.

Example 3.4 (Lookup function using if statements)**.**

$$
\begin{aligned}
&\mathit{lookup}(a : \mathit{Constraint}) : \mathit{List}[\mathit{Constraint}] = \\
&\textbf{if } a = x :: A \\
&\textbf{then } [x :: B] \\
&\textbf{else if } a = x :: A \\
&\quad \textbf{then } [x :: C, x.f :: D] \\
&\quad \textbf{else if } a = x :: D \\
&\quad\quad \textbf{then } [x :: E] \\
&\quad\quad \textbf{else } \mathit{nil}
\end{aligned}
$$

We can observe in Example 3.4 that we created one if statement per program entailment. This practice can lead to the shadowing of program entailments as seen in the example, where the first and the second if statement share the same guard. This is the case because there can be more than one entailment with the same right-hand side constraint, as seen in the given program entailments in Example 3.3. This leads to shadowing, since we use the right-hand side as the guard for the if statements.

To avoid this shadowing of entailments, we change the return type of the lookup function from returning a list of constraints ($\mathit{List}[\mathit{Constraint}]$) to a list of constraint lists ($\mathit{List}[\mathit{List}[\mathit{Constraint}]]$).
This enables us to merge program entailments that share the same right-hand side together. With this we can give the function definition shown in Example 3.5 that avoids shadowing.

Example 3.5 (Avoid shadowing in lookup)**.**

$$
\begin{aligned}
&\mathit{lookup}(a : \mathit{Constraint}) : \mathit{List}[\mathit{List}[\mathit{Constraint}]] = \\
&\textbf{if } a = x :: A \\
&\textbf{then } [[x :: B], [x :: C, x.f :: D]] \\
&\textbf{else if } a = x :: D \\
&\quad \textbf{then } [[x :: E]] \\
&\quad \textbf{else } \mathit{nil}
\end{aligned}
$$

We can see in Example 3.5 that we merged the two entailments whose right-hand side is $x :: A$ and return both possible constraint lists as the result for a match to these entailments.

To translate program entailments $\forall y.\ \overline{b} \Rightarrow b$ we used b to check for equality to the argument to the function and $\overline{b}$ as the return value. As of now we neglected the bound variable y of the program entailment.
So we only solved the non-algorithmic nature of the rule and we still need to find a suitable substitution of the quantified variable a from rule C-Prog to input into the lookup function, such that the substitution of a matches b. We can incorporate the enumeration of combinations of variable names and paths into the lookup function generation to resolve this.

To do this we look at rule C-Prog and see that the bound variable of the program entailment is used for the substitution. This is because, according to the well-formedness, for program entailments $\forall y.\ \overline{b} \Rightarrow b$ only the bound variable y can occur in $\overline{b}$ and b needs to be of the form $y :: \mathit{Cls}$ for class names Cls. So for finding a valid substitution, we can fix the variable name to be substituted to y.
Since we fixed the variable name to be substituted, what remains is the need to instantiate each program entailment with each possible path. Previously we only needed to generate a lookup function per program. Now we need to generate a new lookup function for each constraint entailment we want to show, since we enumerate over all possible paths.

With this in mind we generate a lookup function for the program entailments shown in Example 3.3 with valid paths y and $y.g$. The resulting lookup function is shown in Example 3.6.

Example 3.6 (Program entailment lookup function)**.**

$$
\begin{aligned}
&\mathit{lookup}(a : \mathit{Constraint}) : \mathit{List}[\mathit{List}[\mathit{Constraint}]] = \\
&\textbf{if } a = y :: A \\
&\textbf{then } [[y :: B], [y :: C, y.f :: D]] \\
&\textbf{else if } a = y :: D \\
&\quad \textbf{then } [[y :: E]] \\
&\quad \textbf{else if } a = y.g :: A \\
&\qquad \textbf{then } [[y.g :: B], [y.g :: C, y.g.f :: D]] \\
&\qquad \textbf{else if } a = y.g :: D \\
&\qquad\quad \textbf{then } [[y.g :: E]] \\
&\qquad\quad \textbf{else } \mathit{nil}
\end{aligned}
$$

If we compare Example 3.5 and Example 3.6, we can observe that the function in Example 3.6 uses twice as much if statements as the function in Example 3.5. This is because we instantiated each of the entailments with two possible paths, doubling the amount of cases to consider.

With the final lookup function definition from Example 3.6 we can refine rule C-Prog. We do this by replacing the declarative requirement of the *in-program* predicate with the algorithmic call to the lookup function.

$$
\begin{aligned}
&\forall \overline{a} : \mathit{List}[\mathit{Constraint}], a : \mathit{Constraint}. && \text{(C-Prog)} \\
&\textbf{let } \overline{\overline{a}} := \mathit{lookup}(a) \\
&\textbf{in } \neg(\overline{\overline{a}} = \mathit{nil}) \wedge \bigvee_{\overline{b} \in \overline{\overline{a}}} \overline{a} \vdash \overline{b} \rightarrow \overline{a} \vdash a
\end{aligned}
$$

Figure 3.8: Rule C-Prog Using Lookup

The refined rule C-Prog is defined in Figure 3.8.
The rule calls function *lookup* to check if an entailment declaration matching a exists in the program. The result is a list containing possible constraint lists to evaluate for entailment to a.
Since we modeled the lookup function to return the empty list *nil* if no matching entailment declaration exists, we require the lookup result $\overline{\overline{a}}$ to be non empty.
Since we merged entailment declarations with matching right-hand sides to avoid shadowing, it suffices to show that at least one of the constraint lists contained in $\overline{\overline{a}}$ is entailed by $\overline{a}$ to show that a is entailed by $\overline{a}$.

The refined rule C-Prog does not contain any free variables and is therefore not a template like the refined C-Subst, but a general rule that does not need any preprocessing. This is the case because we moved the enumeration process of possible paths into the generation of the lookup function.

Rule C-Perm

The permutation rule C-Perm is defined in a non-structural way. The SMT solver can use the rule without a problem to check for two known entailments $\overline{a} \vdash a$ and $\overline{b} \vdash a$ if $\overline{a}$ is a permutation of $\overline{b}$. The solver struggles with using this rule for generating new entailments $\overline{b} \vdash a$ from a single known entailment $\overline{a} \vdash a$, such that $\overline{b}$ is a permutation of $\overline{a}$.

The permutation generating aspect is, in our model, used for closures with rule C-Ident in combination with rule C-Weak since our model is list based. Rule C-Ident requires a list containing only one element for its context and rule C-Weak can only eliminate the first element from the context. Therefore to close a proof for $\overline{c} \vdash c$ with C-Ident where $c \in \overline{c}$ and $\neg(last(\overline{c}) = c)$, we need to find a permutation $\overline{c'}$ of $\overline{c}$ where $last(\overline{c'}) = c$.

$$\forall \overline{c} : List[Constraint], c : Constraint. \quad \text{C-DirectIdent}$$
$$c \in \overline{c} \rightarrow \overline{c} \vdash c$$

Figure 3.9: Rule C-DirectIdent

To avoid the generation of permutations by the solver, we define a rule for direct closure for such entailments using rule C-Ident.
We introduce rule C-DirectIdent in Figure 3.9. Rule C-DirectIdent can be used to directly close entailments of the form $\overline{c} \vdash c$ if $c \in \overline{c}$.
Rule C-DirectIdent introduces a new quantified variable, potentially leading to an increased search space for the solver. This is not the case for the rule. The newly introduced quantified variable $\overline{c}$ is contained in the entailment we want to show and therefore used in an input position which can be easily matched by the solver. This is also true for the the quantified variable c in combination with the element of check. Both quantified variables are used in input positions. Therefore the solver does not need to generate instantiations that match the property $c \in \overline{c}$, but can rather check if the property is fulfilled for the known values.

Rule C-DirectIdent does not contradict any of the other rules. It can solely be used as a "shortcut" for rule C-Ident. We show that rule C-DirectIdent and rule C-Ident are interchangeable.

Each entailment $\overline{c} \vdash c$ closable with rule C-DirectIdent can also be closed with the usage of rule C-Ident. We produce a derivation for $\overline{c} \vdash c$ without the usage of C-DirectIdent. Since the entailment is closable by C-DirectIdent, we know that $c \in \overline{c}$. We generate a permutation $\overline{c'}$ of $\overline{c}$ with the property $last(\overline{c'}) = c$. We do this by appending c at the end of $\overline{c}$ and obtain $\overline{c'} := \overline{c} +\!\!+ [c]$. $\overline{c'}$ is a valid permutation in our model, since rule C-Perm does not require both lists to be of equal length and we know that $c \in \overline{c}$. From there we repeatedly apply rule C-Weak until we are left with the subgoal $[c] \vdash c$. We show this entailment by applying rule C-Ident.

Each entailment $\bar{c} \vdash c$ closable with rule C-Ident can also be closed with the usage of rule C-DirectIdent. The entailment is closable with rule C-Ident, we therefore know that $\bar{c}$ is of the form $[c]$. Since $c \in [c]$, we show the entailment $[c] \vdash c$ by directly applying rule C-Ident.

We showed that each entailment closable by C-DirectIdent can also be closed by C-Ident and vice versa. We replace rule C-Ident with rule C-DirectIdent.

3.1.3 Scala Integration

To integrate the model of the sequent calculus into our Scala implementation of the DC_C calculus, we need to translate the first-order model into the SMT-Lib format. We also need a procedure for preprocessing, since we enumerated over variable names and paths in the refined rules C-Subst and C-Prog. We can then check an entailment by calling the SMT solver with the translated and preprocessed model.

SMTLib Representation

First we give a SMTLib representation for the sort, predicate and functions definitions and the non-preprocessed rules.

We translate the sorts defined in Figure 3.4. The parametrized list datatype used is predefined in Z3. We define sorts *CList* and *CsList*, to increase compatibility with other solvers and to avoid ambiguities with the usage of *List*[*Constraint*]and *List*[*List*[*Constraint*]]. The SMTLib representation is defined in Figure 3.10. All datatypes are defined with unique constructor- and selector names.

```
(declare-datatype Path (
  (var (id String))
  (pth (obj Path) (field String))))
(declare-datatype Constraint (
  (path-eq (p-left Path) (p-right Path))
  (instance-of (instance Path) (cls String))
  (instantiated-by (object Path) (clsname String))))
(declare-datatype CList (
  (empty)
  (construct (first Constraint) (rest CList))))
(declare-datatype CsList (
  (nan)
  (cons (hd CList) (tl CsList))))
```

Figure 3.10: SMTLib Datatype Declarations

Figure 3.11 shows a SMTLib translation of the list functions defined in Figure 3.1.
We can define recursive functions in the SMTLib format with the keyword **define-fun-rec**. The functions in the first-order model used pattern matching for better readability. While SMTLib supports pattern matching, it is not

```
(define-fun-rec conc ((l1 CList) (l2 CList)) CList
  (ite (is-construct l1)
    (construct (first l1) (conc (rest l1) l2))
     l2))
(define-fun-rec elem ((c Constraint) (cs CList)) Bool
  (ite (is-construct cs)
    (ite (= c (first cs))
       true
      (elem c (rest cs)))
     false))
```

Figure 3.11: SMTLib List Functions

fully supported by the latest version of Z3. We translated the pattern matching using if statements, where the guard checks for specific constructors of a datatype.
This can be observed in the translation of the list concatenating *conc*. We used an if statement to check wheter the first argument $l1$ is a list construction with *is-construct*($l1$). These constructor checking functions get automatically generated for each constructor of a datatype declaration. Since the *CList* datatype has only two constructors, the else part of the if statement must match the empty list and we do not need to further check for its constrcutor.

We avoided the introduction of new local variable bindings through let where the new binding would be used only once. We directly applied the selectors for those occurences.

```
(define-fun-rec generalize-path
  ((p1 Path) (p2 Path) (x String)) Path
  (ite (= p1 p2)
    (var x)
    (ite (is-var p1)
       p1
      (pth (generalize-path (obj p1) p2 x)
           (field p1)))))
(define-fun generalize-constraint
  ((c Constraint) (p Path) (x String)) Constraint
  (ite (is-path-eq c)
    (path-eq (generalize-path (p-left c) p x)
             (generalize-path (p-right c) p x))
    (ite (is-instance-of c)
      (instance-of (generalize-path (instance c) p x) (cls c))
      (instantiated-by (generalize-path (object c) p x)
                       (clsname c))))))
```

Figure 3.12: SMTLib Representation Of Generalization

A SMTLib translation for substitution defined in Figure 3.3 is given in Figure 3.13 and the SMTLib representation for generalization defined in Figure 3.6

```
(define-fun-rec subst-path
  ((p1 Path) (x String) (p2 Path)) Path
  (ite (is-var p1)
    (ite (= x (id p1))
      p2
      p1)
    (pth (subst-path (obj p1) x p2) (field p1))))
(define-fun subst-constraint
  ((c Constraint) (x String) (p Path)) Constraint
  (ite (is-path-eq c)
    (path-eq (subst-path (p-left c) x p)
             (subst-path (p-right c) x p))
    (ite (is-instance-of c)
      (instance-of (subst-path (instance c) x p) (cls c))
      (instantiated-by (subst-path (object c) x p) (clsname c)
     ))))
(define-fun-rec subst-constraints
  ((cs CList) (x String) (p Path)) CList
  (ite (is-construct cs)
    (construct (subst-constraint
                 (first cs) x p)
                 (subst-constraints (rest cs) x p))
    empty))
```

Figure 3.13: SMTLib Representation Of Substitution

can be found in Figure 3.12.
The translations for substitution and generalization follow the same pattern regarding pattern matching as the translation of the list functions shown previously. We define non-recursive functions using the keywords **define−fun**, so we strictly distinguish syntactically between recursive and non-recursive functions in the SMTLib format.

```
// Base Properties
(declare-fun class (String) Bool)
// DCC Properties
(declare-fun entails (CList Constraint) Bool)
(define-fun-rec Entails ((cs1 CList) (cs2 CList)) Bool
  (ite (is-construct cs2)
    (and (entails cs1 (first cs2))
         (Entails cs1 (rest cs2)))
    true))
```

Figure 3.14: SMTLib Predicate Definitions

A SMTLib representation of the predicates defined in Figure 3.4 is given in Figure 3.14. We can declare undefined functions with the keywords **declare−fun**. Those function declarations are specified, like the predicates in the first-order model, through requiring that specific instances of the function hold.
We kept only the predicate *class*, since the refined rules C-Subst and C-Prog

```
(assert (!
  (forall ((p Path)) (entails empty (path-eq p p)))
  :named C-Refl))
(assert (!
  (forall ((c Constraint) (cs CList))
    (=> (elem c cs) (entails cs c)))
  :named C-Ident))
(assert (!
  (forall ((cs CList) (p Path) (c String))
    (=> (and (class c)
             (entails cs (instantiated-by p c)))
        (entails cs (instance-of p c))))
  :named C-Class))
(assert (!
  (forall ((cs1 CList) (cs2 CList)
           (b Constraint) (c Constraint))
    (=> (and (entails cs1 c)
             (entails (construct c cs2) b))
        (entails (conc cs1 cs2) b)))
  :named C-Cut))
```

Figure 3.15: SMTLib DC_C Rules

enumerate over valid variable names and paths, and the entailment predicates.

A SMTLib representation for the DC_C rules C-Refl, C-Ident, C-Class and C-Cut, that do not require enumeration, is given in Figure 3.15. The structural rules of the sequent calculus C-Weak and C-Perm are defined in Figure 3.16. The keyword **assert** is used to add an assertion to the solver and the keyword **!** can be used to annotate a formula. Annotation is done via adding a key-value pair to a formula, where the key specifies the name of the annotated property. We *named* the rules according to their names in the calculus. This enables the solver to produce an unsatisfiable core for an unsatisfiable entailment, by returning the names of the violated rules.

Figure 3.17 shows a SMTLib representation of the refined rule C-Prog defined in Figure 3.8.

The function big−or−Entails models $\bigvee_{\bar{b}\in ccs} cs \vdash \bar{b}$ as used in the refined model of rule C-Prog. The function lookup−program−entailment is a lookup function of the form seen in Example 3.6.

SMTLib Format in Scala

We modeled the SMTLib format in Scala as an AST. Each node of the AST knows how to format itself and formatting will be propagated to each sub-node. Figure 3.18 shows **trait** SMTLibFormatter, as well as **object** SMTLibFormatter. The trait provides the function format and each node inherits from this trait. The singleton **object** SMTLibFormatter provides a function for formatting a SMTLib formattable sequence using a specifiable separator.

Example 3.7 shows the implementation of the universal quantifier.

The **trait** Term models the category *term* of the SMTLib format. The universal quantifier is a member of this category and therfore **case class** Forall inherits

```
(assert (!
  (forall ((cs CList) (a Constraint) (b Constraint))
    (=> (entails cs b)
        (entails (construct a cs) b)))
  :named C-Weak))
(assert (!
  (forall ((cs1 CList) (cs2 CList) (c Constraint))
    (=> (and (forall ((a Constraint))
               (and (=> (elem a cs1) (elem a cs2))
                    (=> (not (elem a cs1)) (not (elem a cs2)))
             ))
             (entails cs1 c))
        (entails cs2 c)))
  :named C-Perm))
```

Figure 3.16: SMTLib Structural Rules

```
(define-fun-rec big-or-Entails ((ccs CsList) (cs CList)) Bool
  (ite (is-cons ccs)
    (or (Entails cs (hd ccs))
        (big-or-Entails (tl ccs) cs))
     false))
(assert (!
  (forall ((cs CList) (c Constraint))
    (let ((ccs (lookup-program-entailment c)))
      (=> (and (not (= ccs nan))
               (big-or-Entails ccs cs))
          (entails cs c))))
  :named C-Prog))
```

Figure 3.17: SMTLib C-Prog And Helper Function

from **trait** Term.
The format of an universal quantifier is defined in **case class** Forall and starts with the keyword **forall**. The keyword is followed by the format of the quantified variables enclosed in parens and by the format of the body. Finally the formatted components are enclosed in parens to obtain the format of the quantifier.

Example 3.7 (SMTLib Implementation: Universal quantification)**.**

```
trait Term extends SMTLibFormatter

case class Forall(vars: Seq[SortedVar], body: Term) extends Term {
  override def format(): String =
    s"(forall␣(${SMTLibFormatter.format(vars)})␣${body.format()})"
}
```

```
trait SMTLibFormatter {
  def format(): String
}

object SMTLibFormatter {
  def format(seq: Seq[SMTLibFormatter]
           , separator: String = "␣"): String =
    seq.foldRight(""){
    (x, xs) ⇒ s"${x.format()}$separator$xs"}.dropRight(1)
}
```

Figure 3.18: SMTLib Formatting In Scala

Preprocessing Variable Names and Paths

The previously shown SMTLib representation is missing the refined components of the model, which enumerate over variable names and paths.

Lookup Function generation is the process described in Section 3.1.2, with the goal to generate a function modeling the existence check of program entailments combined with the search for a substitution matching the right-hand side from the entailment to be shown, to an entailment declaration of the program.

```
def makeProgramEntailmentLookupFunction(
        p: Program, paths: List[Path]): SMTLibCommand = {
  val x = SimpleSymbol("c")
  val body = makeProgramEntailmentLookupFunctionBody(
        paths.flatMap(instantiateProgramEntailments(p, _)), x)

  DefineFun(FunctionDef(
    SimpleSymbol("lookup-program-entailment"),
    Seq(SortedVar(x, SimpleSymbol("Constraint"))),
    SimpleSymbol("CsList"),
    body
  ))
}

private def instantiateProgramEntailments(p: Program, path: Path
  , entailments: Map[Constraint, List[List[Constraint]]] = Map())
              : Map[Constraint, List[List[Constraint]]] = p match {
  case Nil ⇒ entailments
  case ConstraintEntailment(x, as, a) :: rst ⇒
    val cs: List[Constraint] = substitute(x, path, as)
    val c: Constraint = substitute(x, path, a)

    entailments.get(c) match {
      case None ⇒ instantiateProgramEntailments(
                    rst, path, entailments + (c -> List(cs)))
      case Some(ccs) ⇒ instantiateProgramEntailments(
                    rst, path, entailments + (c -> (cs :: ccs)))
    }
  case _ :: rst ⇒ instantiateProgramEntailments(
```

```
                        rst, path, entailments)
}

private def makeProgramEntailmentLookupFunctionBody(
    entailments: List[(Constraint, List[List[Constraint]])], x: Term)
                            : Term = entailments match {
  case Nil => SimpleSymbol("nan")
  case (c, ccs) :: rst =>
    Ite(
      Eq(x, convertConstraint(c)),
      Axioms.makeCsList(
        ccs.map(cs => Axioms.makeList(cs.map(convertConstraint)))
      ),
      makeProgramEntailmentLookupFunctionBody(rst, x)
    )
}
```

Listing 3.1: Lookup Function Generation

Listing 3.1 shows the Scala implementation of the lookup function generation.

The function instantiateProgramEntailments implements the instantiation of entailment declarations for a program p with a given path *path*. The already instantiated entailments are accumulated in *entailments* and returned if no uninstantiated declaration remains.
We match entailment declarations $\forall x.\ as \Rightarrow a$ and substitute the bound variable x with *path* in as and a. We declare $cs := as_{x \mapsto path}$ and $c := a_{x \mapsto path}$. We check if we previously instantiated a constraint matching c. If not, we create an entry mapping from c to cs. If we did, we add the constraints cs to the already accumulated constraints for c. We repeat this until no declaration remains and return the accumulated mappings.

The function makeProgramEntailmentLookupFunctionBody is responsible for creating the body of the lookup function. It implements the process of generating a series of if statements out of mappings from *Constraint* to $List[List[Constraint]]$. We match $c \mapsto ccs :: rst$ and create an if statement with the guard $x = c$, where x is the argument of the lookup function. In the then branch, we put the SMTLib representation of *ccs*. The else branch then contains a recursive call with rst. We repeat this until all mappings are processed.

The function makeProgamEntailmentLookupFunction generates a lookup function for entailment declarations in programs. It takes a program and a list of paths as argument and instantiates each path with each entailment declaration in the program. The function returns a SMTLib representation usable by the solver.
We set x to be the argument of the function to be created. We map function instantiateProgramEntailments over the provided paths and obtain a list of mappings m. We pass m into makeProgramEntailmentLookupFunctionBody to obtain a series of if statements *body* as a SMTLib representation. We then create the lookup function with the signature $Constraint \rightarrow List[List[Constraint]]$ and the body *body*.

C-Subst preprocessing is the process of enumeration, to ground the quantification over variable names and paths in rule C-Subst as shown in Section 3.1.2. An implementation of the C-Subst template from Figure 3.7 incorporating the optimizations demonstrated with Example 3.2 is given in Listing 3.3. The process of generating instances of this template is implemented in Listing 3.2.

```
def generateSubstRules(
    vars: List[Id], paths: List[Path]): Seq[SMTLibCommand] = {
  var rules: Seq[SMTLibCommand] = Seq()
  val pathPairs = makePathPairs(paths)

  vars.foreach(x => pathPairs.foreach{
    case (p, q) if x == p && p == q => () // skip
    case (p, q) => rules = rules
                          :+ instantiateSubstRule(x, p, q)
  })
  rules
}

private def instantiateSubstRule(
    variable: Id, p: Path, q: Path): SMTLibCommand =
  Assert(
    Annotate(
      substRuleTemplate(variable, p, q),
      Seq(KeyValueAttribute(Keyword("named")
        , SimpleSymbol("C-Subst")))))
```

Listing 3.2: Rule Generation For C-Subst

The function generateSubstRules takes a list of variables and a list of paths as argument and returns a sequence of C-Subst rules. In the function, we iterate over all variables x and for each x we iterate over all paths p and q. We call instantiateSubstRule with x, p and q to generate a rule C-Subst and append it to the sequence of already generated rules. We skip cases where $x = p = q$. Finally, we return the collected rules.

The function instantiateSubstRule creates the surrounding frame of rule C-Subst and calls substRuleTemplate to instantiate the template of rule C-Subst.

The function substRuleTemplate takes a variable and two paths as argument and returns a SMTLib representation of rule C-Subst. In the function, we convert the given variable and paths into their counterpart x, p and q in the SMTLib format. The function defines three subroutines. These subroutines implement the optimizations discussed in Section 3.1.2.
We check in the subroutine gen if variable x equals path p. If not, we generalize constraint $a2$ and call subroutine subst with the generalized constraint. Otherwise we skip the generalization and proceed with subst directly.
Subroutine subst takes a term a representing a constraint as argument. We check if variable x equals path q. If not, we substitute a and call subroutine conjunction on the result of the substitution. Otherwise we call conjunction, without substituting the argument.

```
private def substRuleTemplate(
    variable: Id, p1: Path, p2: Path): Term = {
  val x: Term = convertId(variable)
  val p: Term = convertPath(p1)
  val q: Term = convertPath(p2)

  def conjunction(a1: Term) =
    if (p1 == p2)
      Apply(SimpleSymbol("entails"),
        Seq(SimpleSymbol("cs"), a1))
    else
      And(
        Apply(SimpleSymbol("entails"), Seq(SimpleSymbol("cs"),
          Apply(SimpleSymbol("path-eq"), Seq(p, q)))),
        Apply(SimpleSymbol("entails"),
          Seq(SimpleSymbol("cs"), a1))
      )
  def subst(a: Term) =
    if(variable == p2)
      conjunction(a)
    else
      Let(
        Seq(VarBinding(SimpleSymbol("a1"),
          Apply(SimpleSymbol("subst-constraint"), Seq(a, x, q))
        )),
        conjunction(SimpleSymbol("a1"))
      )
  val gen =
    if (variable == p1)
      subst(SimpleSymbol("a2"))
    else
      Let(
        Seq(VarBinding(SimpleSymbol("a"),
          Apply(SimpleSymbol("generalize-constraint"),
            Seq(SimpleSymbol("a2"), p, x))
        )),
        subst(SimpleSymbol("a"))
      )

  Forall(
    Seq(
      SortedVar(SimpleSymbol("a2"), SimpleSymbol("Constraint")),
      SortedVar(SimpleSymbol("cs"), SimpleSymbol("CList"))
    ),
    Implies(
      gen,
      Apply(SimpleSymbol("entails"),
        Seq(SimpleSymbol("cs"), SimpleSymbol("a2"))))
    )
  )
}
```

Listing 3.3: C-Subst Template

Subroutine conjunction takes a term $a1$ representing a constraint as argument. We check if path p equals path q. If not, we build the conjunction of the entailments $cs \vdash p \equiv q$ and $cs \vdash a1$. Otherwise we build the entailment $cs \vdash a1$ and skip the check for equivalent paths.
In substRuleTemplate, we create the SMTLib representation of

$$\forall a_2 : \mathit{Constraint}, cs : \mathit{Constraint}.\ \mathbf{gen} \rightarrow cs \vdash a_2$$

where **gen** is a call to subroutine gen as our return value.

Calling the Solver

With the SMTLib representation and preprocessing of the model defined, we can call the solver to evaluate if entailments $\overline{a} \vdash a$ hold in the model.

For this we define **trait** SMTSolver in Listing 3.4 as an interface for SMT solvers and function entails to check entailments in Listing 3.5. The conversion from the DC_C syntax to the SMTLib format is done via **object** SMTLibConverter. The object defines functions for converting variables, paths and constraints to their respective representation in the SMTLib format. The functions for preprocessing, e.g. generateSubstRules and makeProgramEntailmentLookupFunction, are also defined in SMTLibConverter.

```
trait SMTSolver {
  val axioms: SMTLibScript

  def addCommand(command: SMTLibCommand): Boolean
  def addCommands(commands: Seq[SMTLibCommand]): Boolean
  def addScript(script: SMTLibScript): Boolean
  def flush(): Unit

  /**
    * Executes the SMTSolver with the currently held commands.
    * @param timeout The timeout for each query
             to the SMTSolver in milliseconds.
    * @return The return code of the SMTSolver
               or -1 in case of a timeout
    *          and the raw output of the solver.
    */
  def execute(timeout: Int = 1000): (Int, Seq[String])

  /**
    * Executes the SMTSolver with the currently held commands
    * and checks them for satisfiability.
    * @param timeout The timeout for each query
             to the SMTSolver in milliseconds.
    * @return 'Sat' if the input is satisfiable
    *         'Unsat' if the infput is unsatisfiable
    *         'Unknown' if the solver can't decide.
    */
  def checksat(timeout: Int = 1000): CheckSatResponse
}
```

Listing 3.4: SMTSolver Interface

The trait SMTSolver defines functionality for calling a SMT solver. The trait defines a set of axioms which are true for each query to the solver, as well as functions addCommand, addCommands and addScript for adding additional commands to be passed to the solver. The function flush removes all added commands and keeps only the axioms.
Function execute calls and uses the axioms as well as all added commands as input for the solver. The function returns the exit code and the raw output of the solver.
Function checksat does call the solver like execute, with the additional smtlib input (**check−sat**). It then parses the output of the solver and returns either sat if the input was satisfiable, unsat if the input was unsatisfiable or unknown if the solver could not decide whether the input was satisfiable or unsatisfiable. The result unknown does not imply that the input formula is undecidable. There can be other reasons resulting in the output unknown, as timeouts.

```
def entails(
    context: List[Constraint], c: Constraint): Boolean = {
  val (vars, paths, classes) = [...]

  // SMTLib representation
  val entailment =
      SMTLibConverter.convertEntailment(ctx, c)
  val classPredicates = classes.map(
      cls => Apply(SimpleSymbol("class"),
                   Seq(SMTLibString(cls))))

  val solver: SMTSolver = new Z3Solver(Axioms.all)
  solver.addCommands(
    SMTLibConverter.generateSubstRules(vars, paths))
  solver.addCommand(SMTLibConverter
      .makeProgramEntailmentLookupFunction(P, paths))
  solver.addCommand(Axioms.cProg)
  solver.addCommands(
    SMTLibConverter.makeAsserts(classPredicates))
  solver.addCommand(Assert(Not(entailment)))

  solver.checksat() match {
    case Sat => false
    case Unsat => true
    case Unknown => false
  }
}

def entails(
    ctx: List[Constraint], cs: List[Constraint]): Boolean =
  cs.forall(c => entails(ctx, c))
```

Listing 3.5: Entailment Function

The function entails can be used to check constraint entailments. It has two implementations. The first implementation models $context \vdash c$ for constraint

lists *context* and constraints c. The second implementation models $ctx \vdash cs$ for constraint lists ctx and cs. It requires that entails (ctx, c) holds for all $c \in cs$.

The function is responsible for invoking the preprocessing process. For this, we implemented the function as follows: We first extract the variable names, paths and class names from the function arguments. We then convert the arguments and create a SMTLib representation $context \vdash c$ as well as generating $class(cls)$ for each class name $cls \in classes$.

We create an instance of SMTSolver with Axioms.all, where Axioms.all contains the SMTLib representations of all declarations, definitions and rules. The axioms exclude rule C-Subst, rule C-Prog and a definition of a lookup function. We add the generated class predicates to the solver. Afterwards we call generateSubstRules to enumerate over variable names and paths to generate rules C-Subst and makeProgramEntailmentLookupFunction to generate a lookup function from the program and paths. We add the generated C-Subst rules as well, as the lookup function and rule C-Prog to the solver. Finally, we add the negation of the generated entailment $context \vdash c$ to the solver.

We call solver.checksat to check for satisfiability with the SMT solver. We return true if the call returns unsat, because the entailment $context \vdash c$ is valid if its negation is unsatisfiable.

3.1.4 Pruning

We found that the solver sometimes times out given a valid entailment, if we supply rules that are unused in the proof of the entailment. We observed this in combination with the enumeration of variables and paths of rule C-Subst.

While all the generated rules C-Subst are valid, the solver does sometimes focus on one substitution and loop instantiations of the same rule building an implication chain of the same rule. This happens if the substitution produces implications of the form $\overline{a} \vdash a \rightarrow \overline{a} \vdash a$ for the input. In our observations, this happened with reflexive path pairs $p \equiv p$. To counter this behavior we introduce *pruning* into function generateSubstRules, through skipping the generation of rules C-Subst for reflexive paths.
The incorporation of pruning into function generateSubstRules is given in Figure 3.19. We extended the arguments of the function with a boolean parameter *pruning* with the default value *false*. In the implementation, we skip rule generations if *pruning* is enabled and $p = q$.

Pruning removes potentially needed rules. Therefore, we incorporate pruning into function entails as a two-pass process. Figure 3.20 shows the updated implementation of function entails defined in Listing 3.5. In the updated implementation, we enable the *pruning* parameter for the first call to generateSubstRules. We check for the result of the solver. In case its unknown, meaning that either the input is undecidable or the solver timed out, we start a second run of the solver without pruning. For this, we flush the added commands to the solver. Afterwards we add the flushed rules again, but this time we call generateSubstRules with disabled pruning.

```
def generateSubstRules(
  vars: List[Id], paths: List[Path], pruning: Boolean = false)
    : Seq[SMTLibCommand] = {
  var rules: Seq[SMTLibCommand] = Seq()
  val pathPairs = makePathPairs(paths)

  vars.foreach(x => pathPairs.foreach{
    case (p, q) if x == p && p == q => () // skip
    case (p, q) if pruning && p == q => () // skip
    case (p, q) => rules = rules
                        :+ instantiateSubstRule(x, p, q)
  })
  rules
}
```

Figure 3.19: Pruning

3.2 Interpreter

In this section we implement the operational semantics of the DC_C calculus presented in Figure 2.7. For this we define an interpreter, which will evaluate expressions relative to a heap. The operational semantics is a small-step semantics over the structure of expressions.

```
def interp(heap: Heap, expr: Expression)
          : (Heap, Expression) = expr match {
  case FieldAccess(X@Id(_), F@Id(_)) =>    // R-Field
    HC(heap).filter{
        case PathEquivalence(FieldPath(X, F), Id(_)) => true
        case PathEquivalence(Id(_), FieldPath(X, F)) => true
        case _ => false
        } match {
      case PathEquivalence(FieldPath(X, F), y@Id(_)) :: _ =>
        (heap, y)
      case PathEquivalence(y@Id(_), FieldPath(X, F)) :: _ =>
        (heap, y)
      case _ => (heap, expr) // f no field of x
    }
  case MethodCall(m, x@Id(_)) =>    // R-Call
    // Applicable methods
    val S: List[(List[Constraint], Expression)] =
        mImplSubst(m, x).filter{
          case (as, _) => entails(HC(heap), as)}
    if (S.isEmpty) // m not in program
      return (heap, expr)

    var (a, e) = S.head // Most specific method
    S.foreach{
      case (a1, e1) if e != e1 =>
        if (entails(a1, a) && !entails(a, a1)) {
          a = a1; e = e1
        }
```

```
def entails(
    context: List[Constraint], c: Constraint): Boolean = {
  [...]
  solver.addCommands( // pruning enabled
    SMTLibConverter.generateSubstRules(vars, paths, true))
  [...]

  solver.checksat() match {
    case Sat ⇒ false
    case Unsat ⇒ true
    case Unknown ⇒
      solver.flush() // Second pass without pruning
      solver.addCommands(
        SMTLibConverter.generateSubstRules(vars, pths, false))
      [...]

      solver.checksat() match {
        case Unsat ⇒ true
        case _ ⇒ false
      }
  }
}
```

Figure 3.20: Function Entails With Pruning

```
    }
    interp(heap, e)
  // R–New
  case ObjectConstruction(cls, args)
    if args.forall{ // if args are values (Id)
      case (_, Id(_)) ⇒ true
      case _ ⇒ false
    } ⇒
    val x: Id  = freshvar()
    val o: Obj = (cls, args.asInstanceOf[List[(Id, Id)]])
    // cls in Program: alpha renaming of y to x in b
    val (y: Id, b: List[Constraint]) =
      classInProgram(cls, P).getOrElse(return (heap, expr))
    val b1 = substitute(y, x, b)
    // heap constraints entail cls constraints
    if (entails(HC(heap) ++ OC(x, o), b1))
      (heap + (x -> o), x)
    else
      (heap, expr) // stuck

  // RC–Field
  case FieldAccess(e, f) ⇒
    val (h1, e1) = interp(heap, e)

    if(h1 == heap && e1 == e) {
      (heap, expr) // stuck
    } else {
```

```
      interp(h1, FieldAccess(e1, f))
    }

  // RC-Call
  case MethodCall(m, e) =>
    val (h1, e1) = interp(heap, e)

    if(h1 == heap && e1 == e) {
      (heap, expr) // stuck
    } else {
      interp(h1, MethodCall(m, e1))
    }

  // RC-New
  case ObjectConstruction(cls, args) =>
    val (h1, args1) = objArgsInterp(heap, args)

    if(h1 == heap && args1 == args) {
      (heap, expr) // stuck
    } else {
      interp(h1, ObjectConstruction(cls, args1))
    }

    case Id(_) => (heap, expr) // variables are values
}
```

Listing 3.6: Interpreter

Figure 3.21 implements functions HC and OC defined in Figure 2.7.
The constraints of a heap are the constraints on the objects contained in the heap. The implementation of $HC(heap)$ maps OC over all elements contained in *heap*.

The constraints of an object state the class of the object and the values of its fields. In our implementation, an object consists of a class name and a list of pairs of field names and variable names. In the implementation of $OC(x, o)$ with $o := (cls, fields)$, we create a constraint $x.\mathbf{cls} \equiv cls$ stating the class of o. Further, we create for each field-value pair $(f, v) \in fields$ a constraint $x.f \equiv v$ stating that field f of x has value v.
We define function interp in Listing 3.6. The function takes a heap *heap* and an expression *expr* as arguments and returns a heap and an expression. We do not return intermediate steps in the function, but fully evaluate the argument expression to a value. The implementation follows the structure of the operational semantics. This is done via pattern matching on the argument expression and allows to recreate each rule from Figure 2.7 as a separate case in the implementation.

The first case implements rule R-Field, it is marked in the Scala code with the comment R-Field. We match for field access $e.f$, where the subexpression e is evaluated to a variable. Variables are the values of the DC_C calculus. We then build the heap constraints $HC(heap)$ and filter them to check if they contain $x.f \equiv y$ or $y \equiv x.f$ for some variable y. If such a path equivalence exists, we reduce $x.f$ to y and return the pair $(heap, y)$. Otherwise we can not reduce $x.f$, because field f is not a member of the object corresponding to x in the heap.

```
// Heap Constraints
private def HC(heap: Heap): List[Constraint] =
  heap.flatMap{case (x, o) => OC(x, o)}.toList

// Object Constraints
private def OC(x: Id, o: Obj): List[Constraint] = o match {
  case (cls, fields) =>
    val init = InstantiatedBy(x, cls)
    val fieldCs = fields.map{case (f, v) =>
      PathEquivalence(FieldPath(x, f), v)}

    init :: fieldCs
}
```

Figure 3.21: Heap And Object Constraints

We return the arguments $(heap, expr)$ as there are no subsequent transitions.

For rule R-Call, we match for method calls $m(x)$, where x is a value. First, we generate the list of applicable methods S. For this we call $mImplSubst(m, x)$ to search for all implementations of method m and filter its result to check if the argument constraints are entailed by the heap. Function *mImplSubst* implements *mImpl* defined in Figure 2.3. In *mImplSubst* we explicitly incorporated the unification of the formal argument of method implementations with x, since the unification is implicit in the operational semantics. If S is empty, there is no implementation for m in the program and we can not reduce $m(x)$. Afterwards, we determine the most specific method implementation. With $(a, e) := S.head$, we set the first applicable method as the currently most specific method. We then try for each applicable method (a_1, e_1) where $e \neq e_1$, if $a_1 \vdash a$ and not $a \vdash a_1$. If so we update the most specific method $(a, e) := (a_1, e_1)$. After all applicable methods have been checked, we reduce $m(x)$ to the body e of the most specific implementation of m. Since we do not want to produce intermediate results and e is not guaranteed to be a value we recursively call interp(heap, e).

```
private def mImplSubst(
    m: Id, x: Id): List[(List[Constraint], Expression)] =
  P.foldRight(Nil: List[(List[Constraint], Expression)]){
    case (MethodImplementation('m', y, a, _, e), rst) =>
      (substitute(y, x, a),
       alphaRename(y, x, e)) :: rst
    case (_, rst) => rst
  }
```

Figure 3.22: Function mImplSubst

The implementation of $mImplSubst(m, x)$ is given in Figure 3.22. In the implementation we fold over declarations of program P. In the case of a method implementation $m(y.\ a) : \_ := e$, we substitute the formal argument y with x

in constraints a and apply α-renaming of y to x in the method body e.
For rule R-New we match for object creations **new** *cls*(*args*), where *args* has the type List [(Id, Expression)] and the argument expressions are reduced to values.
We generate a fresh variable x for the object to be constructed and set object $o := (cls, args)$, where *args* are cast to List [(Id, Id)]. This is a safe typecast, since we ensured that the argument expressions are values.
We search for a constructor *cls*(y. b) in the program. If there is no constructor for class *cls*, we can not reduce **new** *cls*(*args*). If we found a constructor, we substitute the formal argument of the constructor y with the variable of the object to be created x in the constructor constraints b. We obtain $b_1 := b_{y \mapsto x}$. We check if the heap constraints combined with the constraints of o entail b_1. If so, we reduce **new** *cls*(*args*) to x and extend the heap with the mapping from x to o. Otherwise, we can not reduce **new** *cls*(*args*).

We implemented the rules RC-Field, RC-Call and RC-New by matching for their respective expression structure and the rules have no requirements on the subexpressions.

For field access expressions $e.f$, we interpret subexpression e in the heap *heap* to obtain an updated heap h_1 and evaluated subexpression e_1. We check if $e = e_1$ and $heap = h_1$. If not, we proceed with interpreting $e_1.f$ in the heap h_1. Otherwise, the subexpression can not be reduced and we return (*heap*, *expr*) to avoid endless recursion.

The implementation for method calls $m(e)$ is the same as the previously shown implementation for field access expressions $e.f$. We evaluate subexpression e to e_1 and proceed interpreting $m(e_1)$ if we are not stuck.

```
private def objArgsInterp(
    heap: Heap, args: List[(Id, Expression)])
      : (Heap, List[(Id, Expression)]) = args match {
  case Nil => (heap, Nil)
  case (f, x@Id(_)) :: rst =>
    val (h1, args1) = objArgsInterp(heap, rst)
    (h1, (f, x) :: args1)
  case (f, e) :: rst =>
    val (h1, e1) = interp(heap, e)
    val (h2, args1) = objArgsInterp(h1, rst)
    (h2, (f, e1) :: args1)
}
```

Figure 3.23: Object Argument Interpretation

The implementation for object constructions **new** *cls*(*args*) is similar to the implementations for field access expressions $e.f$ and method calls $m(e)$, but differs from the rule RC-New presented in Figure 2.7. Rule RC-New is defined to only reduce one argument at a time. Our implementation interprets all the argument expressions in the same step. This argument interpretation is done via function objArgsInterp. We obtain a new heap h_1 and evaluated arguments $args_1$ from

the call to objArgsInterp. We check if $args = args_1$ and $heap = h_1$. If not, we proceed with interpreting **new** $cls(args_1)$ in heap h_1. Otherwise, we can not reduce **new** $cls(args)$.

The function objArgsInterp is defined in Figure 3.23. It iterates over each argument, evaluates it and passes the updated heap to the evaluation of the subsequent arguments to ensure that the side-effects occurring during evaluation of the argument expressions are respected. The function takes a heap *heap* and a list of tuples of identifiers and expressions *args* as arguments. It returns a heap and a list of tuples of identifiers and expressions. In the implementation, we match for the structure of the $args$ list.

If the list is empty, we are done and return the input heap *heap* and the empty list *nil*.

If the first element is of the form (f, x) where x is a value, we continue with interpreting the rest of the list to obtain a new heap h_1 and evaluated arguments $args_1$. We return the updated heap h_1 and evaluated arguments $(f, x) :: args_1$.

If the first element is of the form (f, e), we interpret e and obtain heap h_1 and the evaluated expression e_1. We proceed to interpret the rest of the list in heap h_1 to obtain an updated heap h_2 and evaluated arguments $args_1$. We return the latest heap h_2 and $(f, e_1) :: args_1$.

The **case** Id(_) =>(heap, expr) does not implement a rule from the operational semantics presented in Figure 2.7. We added the case in the implementation, to avoid that the function has undefined inputs. Since variables are the values of the language, we do not need to further evaluate them.

Example 3.8 (Interpreting a method call)**.**
In this example we demonstrate the process of interpreting expressions. For this, we take the program describing natural numbers from Example 2.1. We set heap $h := [x \mapsto o]$ with $o := (\texttt{Zero}, [])$, containing a mapping from variable x to an instance of class `Zero`. Our expression to interpret is $\texttt{prev}(x)$. We call interp with arguments h and $\texttt{prev}(x)$.

The expression $\texttt{prev}(x)$ is a method call and x is a value. We match the implementation of R-Call. First we build the list of applicable methods. We call mImplSubst to retrieve the argument constraints and the bodies of implementations of method `prev` and substitute the formal parameter of the implementations with x. Method `prev` has two implementations and we obtain

$$[([x :: \texttt{Zero}], \textbf{new } \texttt{Zero}()), ([x :: \texttt{Succ}, x.p :: \texttt{Nat}], x.p)]$$

from mImplSubst. We check for each method implementation $(\overline{a}, e)$ if $HC(h) \vdash \overline{a}$. We call $HC(h)$ to generate the constraints of the heap. Since h contains only one object mapping from x to $(\texttt{Zero}, [])$, we call $OC(x, o)$. From the call to OC we obtain $x.\textbf{cls} \equiv \texttt{Zero}$ constraining x to be a direct instance of `Zero`. We do not generate further constraints, since object o has no fields.

We call function entails to solve the entailment $x.\textbf{cls} \equiv \texttt{Zero} \vdash x :: \texttt{Zero}$ with the result *true* as well as the entailment $x.\textbf{cls} \equiv \texttt{Zero} \vdash x :: \texttt{Succ}$ with the result *false*. With this, we set the list of applicable methods

$$S := [([x :: \texttt{Zero}], \textbf{new } \texttt{Zero}())].$$

We set $(a, e) := S.head = ([x :: \texttt{Zero}], \textbf{new } \texttt{Zero}())$ to be the most specific method and since S has only one element we do not need to compare the method against the elements of S and e is the most specific method body.

We proceed with interpreting **new** `Zero`() in h. We match the implementation of rule R-New. We generate a fresh variable y and set $o := (\texttt{Zero}, [])$. We search for a constructor of class `Zero` in the program and retrieve $\texttt{Zero}(x.\ \epsilon)$. Since the constraints required by the constructor are empty, we do not need to check if the heap entails the constructor constraints. We extend heap h with the new object binding to obtain

$$h_1 := [x \mapsto (\texttt{Zero}, []), y \mapsto (\texttt{Zero}, [])]$$

and return (h_1, y) as the result of the evaluation of $\texttt{prev}(x)$ in h. We observe that we created a new object of class `Zero` in h_1, because the body of method `prev` for `Zero` is an object creation instead of a variable access.

3.3 Type Relation

In this section we implement the type relations presented in Section 2.2.1. The DC_C calculus defines type assignments for expressions shown in Figure 2.5 and well-formedness of programs shown in Figure 2.6.

```
private def mTypeSubst(m: Id, x: Id, y: Id)
    : List[(List[Constraint], List[Constraint])] =
  P.foldRight(Nil: List[(List[Constraint], List[Constraint])]){
    case (AbstractMethodDeclaration(
            'm', x1, a, Type(y1, b)), rst) =>
      (substitute(x1, x, a),
       substitute(y1, y, b)) :: rst
    case (MethodImplementation(
            'm', x1, a, Type(y1, b), _), rst) =>
      (substitute(x1, x, a),
       substitute(y1, y, b)) :: rst
    case (_, rst) => rst
  }
```

Figure 3.24: Function mTypeSubst

We define function mTypeSubst in Figure 3.24. The function implements function *mType* presented in Figure 2.3. The function takes a method name m and variables x and y as arguments. In mTypeSubst, we explicitly unify the formal argument with x and the bound variable of the declared type with y. In the implementation we fold over the declarations of the program. We match abstract declarations $m(x_1.\ a) : [y_1.\ b]$, as well as implementations $m(x_1.\ a) : [y_1.\ b] := e$ of method m. For each match, we substitute x_1 with x in a and y_1 with y in b.

3.3.1 Type Assignments for Expressions

```
def typeassignment(context: List[Constraint], expr:Expression)
    : List[Type] = expr match {
  case x@Id(_) => [...] // T-Var
  case FieldAccess(e, f) => [...] // T-Field
  case MethodCall(m, e) => [...] // T-Call
  case ObjectConstruction(cls, args) => [...] // T-New
}
```

Figure 3.25: Type Assignment

In this section, we define a function typeassignment implementing type assignments for expressions presented in Figure 2.5. The type relation of the DC_C calculus is not immediately suitable for implementation, as the rules are not syntax directed. The main obstacle is the rule of subsumption T-Sub, as the rule has a bare meta-variable e in its conclusion. The other rules apply only to a specific form of expressions. A process of transforming such a declarative

system into an algorithmic one is shown in [Pie02] for the simply typed lambda-calculus. This process includes the transformation of the declarative subtyping relation into an algorithmic subtyping. Since inheritance in the DC_C calculus is expressed through constraint entailment declarations, we can use the SMT solver to check for subtyping and incorporate subtyping into the type rules.

Type assignments in the DC_C calculus are not unique. For this, we define the function typeassignment to return a list of types, instead of a single type.

The implementation of function typeassignment is given in Figure 3.25. The four cases of the pattern matching are given in Listings 3.7 to 3.10. The function takes a list of constraints as the type context and an expression to assign a type for as arguments. It returns as list of types. The function matches the structure of the expression.

```
case x@Id(_) => // T-Var
  classes.foldRight(Nil: List[Type]) {
    case (cls, _)
    if entails(context, InstanceOf(x, cls)) =>
      val y = freshvar()
      Type(y, List(PathEquivalence(y, x))) :: Nil
    case (_, clss) => clss
  }
```

Listing 3.7: Case T-Var

For rule T-Var, we match for variable access. To assign a type to variable x, we need to ensure that the variable is an instance of some class. We check for each class *cls* if $context \vdash x :: cls$. If so, we generate a fresh variable y and return $[y.\ y \equiv x]$. If x is not an instance of any class, we return the empty list. For this case it is sufficient to return after the first match, because every further type assignment would result in the same type after unification of the bound variable of the types. E.g. if $context \vdash x :: C$, we would generate a fresh variable z and obtain type $[z.\ z \equiv x]$ and $[z.\ z \equiv x]_{z \mapsto y} = [y.\ y \equiv x]$.

```
case FieldAccess(e, f) => // T-Field
  val types = typeassignment(context, e)
  var ts: List[Type] = Nil
  types.foreach {
    case Type(x, a) =>
      val y = freshvar()
      // instance of relations for type constraints
      classes.foldRight(Nil: List[Constraint]) {
        case (cls, clss)
        if entails(context ++ a,
                   InstanceOf(FieldPath(x, f), cls)) =>
          val c = InstanceOf(y, cls)
          ts = Type(y, List(c)) :: ts
          c :: clss
        case (_, clss) => clss
      }
  }
  ts
```

Listing 3.8: Case T-Field

For rule T-Field, we match for field access $e.f$. First, we call typeassignment on subexpression e to obtain the type assignments of e and we create a list *ts* to collect all the possible types for $e.f$. We iterate over each type $[x.\ a]$ of e and generate a fresh variable y. We fold over all classes *cls* and check if $x.f :: cls$ is entailed by the context concatenated with a. If so, we add $[y.\ y :: cls]$ to the collected types *ts*. The entailment check of $x :: cls$ does consider subtyping, since we check for all classes and also add the subtype constraints a to the context.

```
// T-Call
case MethodCall(m, e) ⇒
  val eTypes = typeassignment(context, e)
  val y = freshvar()

  var types: List[Type] = Nil
  // for all possible argument types
  for (Type(x, a) <- eTypes) {
    // for all method declarations
    for ((a1, b) <- mTypeSubst(m, x, y)) {
      val entailsArgs = entails(context ++ a, a1)
      val b1 = (a1 ++ b).foldRight(Nil: List[Constraint]) {
        case (c, cs) if !FV(c).contains(x) ⇒ c :: cs
        case (_, cs) ⇒ cs
      }
      if (entailsArgs && entails(context ++ a ++ b, b1))
        types = Type(y, b1) :: types
    }
  }
  types
```

Listing 3.9: Case T-Call

We implement rule T-Call by matching for method calls $m(e)$. We create a fresh variable y and call typeassignment on the parameter e. We iterate over each parameter type $[x.\ a]$ and call mTypeSubst to obtain the types of method m, where the parameter constraints are substituted with x and the return type constraints are substituted with y. For each (a_1, b) obtained from mTypeSubst, we set b_1 to be the constraints free of x from the parameter constraints a_1 and the type constraints b. The provided parameter is applicable to the method, if the supplied parameter entails the formal argument of the method. We check this through the entailment $context +\!\!+ a \vdash a1$. Finally, if $context +\!\!+ a +\!\!+ b \vdash b_1$ holds we set $[y.\ b_1]$ as one valid type of $m(e)$.

For rule T-New, we check for object constructions **new** *cls*(*args*) and generate a fresh variable x. We also extract the field names from the arguments and set *argsTypes* as the mapping of typeassignment over the values of the arguments. We create a list *types* to collect the types of **new** *cls*(*args*). There are two cases to consider.

1. The arguments are empty.
2. The arguments are not empty.

```
// T-New
case ObjectConstruction(cls, args) ⇒
  val fields: List[Id] = args.map(_._1)
  val argsTypes: List[List[Type]] =
    args.map(arg ⇒ typeassignment(context, arg._2))

  val x = freshvar()
  var types: List[Type] = Nil

  argsTypes match {
    case Nil ⇒
      val b = List(InstantiatedBy(x, cls))
      val (x1, b1) = classInProgram(cls, P)
                       .getOrElse(return Nil)

      if (entails(context ++ b, b1))
        types = Type(x, b) :: types

      classes.foreach{
        c ⇒
          if (entails(context ++ b, InstanceOf(x, c)))
            types = Type(x, List(InstanceOf(x, c))) :: types
      }
    case _ ⇒ combinations(argsTypes).foreach {
      argsType ⇒
        val argsPairs: List[(Id, Type)] = fields.zip(argsType)
        val argsConstraints: List[Constraint] =
        argsPairs.flatMap{
          case (fi, Type(xi, ai)) ⇒
            substitute(xi, FieldPath(x, fi), ai)
        }

        val b: List[Constraint] =
          InstantiatedBy(x, cls) :: argsConstraints

        val (x1, b1) = classInProgram(cls, P)
                         .getOrElse(return Nil)

        if (entails(context ++ b, substitute(x1, x, b1)))
          types = Type(x, b) :: types

        classes.foreach{
          c ⇒
            if (entails(context ++ b, InstanceOf(x, c)))
              types = Type(x, List(InstanceOf(x, c))) :: types
        }
    }
  }
  types
```

Listing 3.10: Case T-New

For (1), if there are no arguments supplied to the constructor we set $b := x.\mathbf{cls} \equiv cls$. We check if the program has a constructor $cls(x_1.\ b_1)$. If so, we check $context +\!\!+ b \vdash b_{1\,x_1 \mapsto x}$ and add $[x.\ b]$ to *types*. Otherwise we return the empty list. Additionally, we examine subtyping through checking $context +\!\!+ b \vdash x :: C$ for all classes C. If the entailment holds, we add $[x.\ x :: C]$ to *types*.

For (2), we generate all possible combinations of the types of the arguments *argsTypes*. E.g. the possible combinations of lists $[[1,2],[3,4],[5]]$ are the lists $[1,3,5]$, $[1,4,5]$, $[2,3,5]$ and $[2,4,5]$. For each combination of types, we zip the extracted field names with the types of the arguments to obtain a mapping from field name to type. For each of the zipped pairs $(f_i, [x_i.\ a_i])$, we instantiate the type of field f_i with the field access $x.f_i$, where x is the variable of the new object. To do this, we substitute x_i with $x.f_i$ in a_i. We set *argsConstraints* to be the mapping of substitution $a_{i\,x_i \mapsto x.f_i}$ over all pairs. We set $b := x.\mathbf{cls} \equiv cls :: argsConstraints$. We check if a constructor $cls(x_1.\ b_1)$ exists in the program. If not, we can not create a new object of a non-existing class and return the empty list. Otherwise, if $context +\!\!+ b \vdash b_{1\,x_1 \mapsto x}$ holds we add $[x.\ b]$ to *types*. We examine subtyping through checking $context +\!\!+ b \vdash x :: C$ for all classes C. If the entailment holds, we add $[x.\ x :: C]$ to *types*.

3.3.2 Well-formedness of Programs

We define a function typecheck, which implements well-formedness of programs. The function has two implementations, one taking a program and the other taking a declaration as argument.

```
def typecheck(P: Program): Boolean = {
  val x = freshvar()
  val y = freshvar()

  methods.forall { m ⇒
    val mTypes = mTypeSubst(m, x, y)

    mTypes.forall {
      case (_, b) ⇒
        mTypes.forall {
          case (_, b1) ⇒
            b.size == b1.size &&
            b.forall(c ⇒ b1.contains(c))
        }
    }
  } && P.forall(typecheck)
}
```

Figure 3.26: Well-formedness Of Programs

Figure 3.26 shows the implementation of function typecheck for program inputs. We generate fresh variables x and y and for all methods m and variables x and y, we call *mTypeSubst* to obtain the argument- and return types for method m.

```
def typecheck(D: Declaration): Boolean = D match {
  // WF-CD
  case ConstructorDeclaration(cls, x, a) ⇒
    FV(a) == List(x) || FV(a).isEmpty

  // WF-RD
  case ConstraintEntailment(x, a, InstanceOf(y, _))
   if x == y ⇒
    FV(a) == List(x) && a.exists {
      case InstanceOf(`x`, _) ⇒ true
      case _ ⇒ false
    }

  // WF-MS
  case AbstractMethodDeclaration(_, x, a, Type(y, b)) ⇒
    val vars = FV(b)
    FV(a) == List(x) && vars.nonEmpty &&
      vars.forall(v ⇒ v == x || v == y)

  // WF-MI
  case MethodImplementation(_, x, a, Type(y, b), e) ⇒
    val vars = FV(b)
    FV(a) == List(x) && vars.nonEmpty &&
      vars.forall(v ⇒ v == x || v == y) &&
    typeassignment(a, e).exists {
      case Type(z, c) ⇒
        c.size == b.size &&
        substitute(z, y, c).forall(b.contains(_))
    }
}
```

Figure 3.27: Well-formedness Of Declarations

We check for each combination of specified return types b and b_1, if b and b_1 have the same number of elements and that each element from b is also contained in b_1. This ensures that each implementation of method m have the same specified return type. Additionally, we check if each declaration is well-formed, by passing function typecheck to each declaration with P. forall (typecheck).

The implementation of well-formedness for declarations is given in Figure 3.27. The rules are syntax-directed and we match for the structure of the declarations in the implementation.

For constructor declarations $cls(x.\ a)$ we require the free variables of a to be the bound variable of the constructor x or to be empty. We allow the free variables to be empty, because the constructor `Zero`$(x.\ \epsilon)$ as seen in Example 2.1 is well-formed.

In the implementation of rule WF-RD, we match for constraint entailment declarations $\forall x.\ a \Rightarrow y :: C$ with $x = y$. This pattern ensures that the entailed

constraint expresses that x is the instance of some class. Further we require that x is the only variable free in a and that at least one constraint of the form $x :: C'$ exists in a.

In the implementation of rule WF-MS, we match for abstract method declarations $m(x.\ a) : [y.\ b]$. We require the free variables of a to be x and the free variables of b to be either x or y.

In the implementation of rule WF-MI, we match for method implementation declarations $m(x.\ a) : [y.\ b] := e$. We require the free variables of a to be x and the free variables of b to be either x or y. We further check if the type of the method body e matches the annotated type $[y.\ b]$. For this, we call function typeassignment with the context a and expression e to obtain a list of types of e. For each type $[z.\ c]$ of e, we substitute z with y in c to unify the type variable and check that each element of c is also contained in b, as well as requiring that the size of c equals the size of b.

Example 3.9 (Typechecking the Natural Numbers Program)**.**
In this example we check if the natural numbers program, given in Example 2.1, is well-formed. For this, we apply *typecheck* to the program.

We need to check if all the declarations of the same method have the same return type. The program declares one method `prev`, with two implementations. We generate fresh variables x and y. We apply $mTypeSubst(\texttt{prev}, x, y)$ and obtain

$$\begin{aligned}&[([x :: Nat], && [y :: Nat]),\\ &\ ([x :: Zero], && [y :: Nat]),\\ &\ ([x :: Succ, x.p :: Nat], && [y :: Nat])]\end{aligned}$$

We see that all return type constraints of `prev` are $y :: Nat$.

Now we need to check if each declaration of the program is well-formed. We apply *typecheck* to each declaration.

A constructor declaration $C(x.\ a)$ is well-formed, if x is the only variable free in a. The constructor $\texttt{Zero}(x.\ \epsilon)$ specifies no constraints and does therefore contain no free variables. For constructor $\texttt{Succ}(x.\ x.p :: \texttt{Nat})$, we check $FV(x.p :: \texttt{Nat}) = [x]$.

We check the constraint entailment $\forall x.\ x :: \texttt{Zero} \Rightarrow x :: \texttt{Nat}$. The entailment is well-formed, since the right-hand side is $x :: \texttt{Nat}$ and the left-hand side contains $x :: \texttt{Zero}$. Both constraints have no free variable besides x.

We check the constraint entailment $\forall x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat} \Rightarrow x :: \texttt{Nat}$. The entailment is well-formed, since the right-hand side is $x :: \texttt{Nat}$ and the left-hand side contains $x :: \texttt{Succ}$. All constraints have no free variable besides x.

We check the abstract method declaration $\texttt{prev}(x.\ x :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}]$. The declaration is well-formed, since x is the only variable free in $x :: \texttt{Nat}$ and y is the only variable free in $y :: \texttt{Nat}$.

We review the method implementation

$$\texttt{prev}(x.\ x :: \texttt{Zero}) : [y.\ y :: \texttt{Nat}] := \textbf{new}\ \texttt{Zero}().$$

We check the free variables $FV(x :: \texttt{Zero}) = [x]$ and $FV(y :: \texttt{Nat}) = [y]$.

We apply *typeassignment*$([x :: \texttt{Zero}], \textbf{new}\ \texttt{Zero}())$ and match case T-New. We generate a fresh variable x_{Obj}. Since **new** `Zero`() has no arguments, we set $b := [x_{Obj}.\textbf{cls} \equiv \texttt{Zero}]$. The constructor $\texttt{Zero}(x.\ \epsilon)$ exists in the program. Since the constructor for `Zero` does not specify any constraints, we memorize the type $[x_{Obj}.\ x_{Obj}.\textbf{cls} \equiv \texttt{Zero}]$.

We explore subtyping by checking the entailment $x :: \texttt{Zero}, x_{Obj}.\textbf{cls} \equiv \texttt{Zero} \vdash x_{Obj} :: C$ for each class C in the program. We successfully solve the entailment for classes `Zero` and `Nat`. We can not solve the entailment for class `Succ`. We return types

$$[[x_{Obj}.\ x_{Obj}.\textbf{cls} \equiv \texttt{Zero}], [x_{Obj}.\ x_{Obj} :: \texttt{Zero}], [x_{Obj}.\ x_{Obj} :: \texttt{Nat}]]$$

as the types of **new** `Zero`() We check if the types of the object construction contains the return type of the method $[y.\ y :: \texttt{Nat}]$. We see that

$$[x_{Obj}.\ x_{Obj} :: \texttt{Nat}]_{x_{Obj} \mapsto y} = [y.\ y :: \texttt{Nat}].$$

The method implementation is well-formed, since all requirements are fulfilled.

We check the method implementation

$$\texttt{prev}(x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}] := x.p$$

and see that $FV(x :: \texttt{Succ}) = [x]$, $FV(x.p :: \texttt{Nat}) = [x]$, and $FV(y :: \texttt{Nat}) = [y]$.

We apply *typeassignment*$([x :: \texttt{Succ}, x.p :: \texttt{Nat}], x.p)$ and match case T-Field. We apply *typeassignment* on the subexpression x. Since the variable access x can be related to a class in the context, we obtain $types := [[x_{var}.\ x_{var} \equiv x]]$. We create a fresh variable y_p and check if $x :: \texttt{Succ}, x.p :: \texttt{Nat}, x_{var} \equiv x \vdash x_{var}.p :: C$ for classes C. We can only successfully solve the entailment for class `Nat` and return type $[y_p.\ y_p :: \texttt{Nat}]$. We unify the type variable y_p with y and obtain

$$[y_p.\ y_p :: \texttt{Nat}]_{y_p \mapsto y} = [y.\ y :: \texttt{Nat}].$$

The method implementation is well-formed, since all requirements are fulfilled.

With the last declaration being well-formed, we conclude that the program is well-formed.

Chapter 4

Runtime Evaluation

In this chapter we evaluate the implementation of the DC_C calculus. To do this, we extend the natural numbers program with the support of abstract syntax of expressions.

4.1 Program Definition

4.1.1 Natural Numbers

First, we recap the implementation of the natural numbers:

$$\texttt{Zero}(x.\ \epsilon)$$
$$\forall x.\ x :: \texttt{Zero} \Rightarrow x :: \texttt{Nat}$$
$$\texttt{Succ}(x.\ x.p :: \texttt{Nat})$$
$$\forall x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat} \Rightarrow x :: \texttt{Nat}$$
$$\texttt{prev}(x.\ x :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}]$$
$$\texttt{prev}(x.\ x :: \texttt{Zero}) : [y.\ y :: \texttt{Nat}] := x$$
$$\texttt{prev}(x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}] := x.p$$

A natural number (`Nat`) is either the number zero (`Zero`) or a successor of a natural number (`Succ`). `Succ` has a field p of class `Nat`. The function `prev` computes the previous element of a natural number. The previous number of zero is defined as zero. We changed the implementation for this case. The program defined in Example 2.1 created a new instance of class `Zero`. The new implementation returns the parameter, as it is constrained to be an instance of `Zero`.

4.1.2 Numeric Expressions

We define a program for the abstract syntax of numeric expressions:

$$\texttt{Lit}(x.\ x.\mathit{value} :: \texttt{Nat})$$
$$\forall x.\ x :: \texttt{Lit}, x.\mathit{value} :: \texttt{Nat} \Rightarrow x :: \texttt{Exp}$$
$$\texttt{Plus}(x.\ x.l :: \texttt{Exp}, x.r :: \texttt{Exp})$$
$$\forall x.\ x :: \texttt{Plus}, x.l :: \texttt{Exp}, x.r :: \texttt{Exp} \Rightarrow x :: \texttt{Exp}$$

The constructor `Lit` defines a field *value*, representing numeric literals with the value *value*. The constructor `Plus` defines fields l and r, representing the addition of r and l. The two constraint entailment declarations express the inheritance relation of `Lit` and `Plus` to `Exp`.

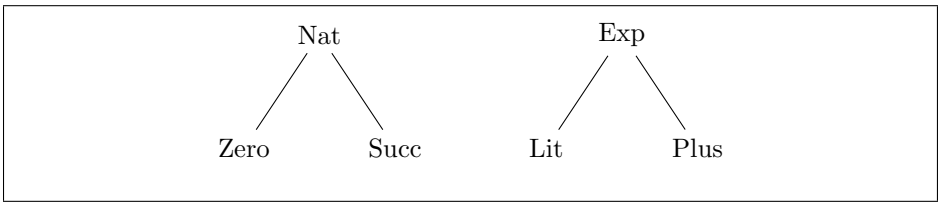


Figure 4.1: Inheritance Relation

Figure 4.1 shows the inheritance relation of the natural numbers and the numeric expressions.

4.1.3 Numeric Expression Evaluation

We extend the abstract syntax of numeric expressions with a method to evaluate expressions:

$\texttt{eval}(x.\ x :: \texttt{Exp}) : [y.\ y :: \texttt{Exp}]$

$\texttt{eval}(x.\ x :: \texttt{Lit}, x.\mathit{value} :: \texttt{Nat}) : [y.\ y :: \texttt{Exp}] := x$

$\texttt{eval}(x.x :: \texttt{Plus}, x.l :: \texttt{Lit}, x.r :: \texttt{Lit}, x.l.\mathit{value} :: \texttt{Nat},$
$\qquad x.r.\mathit{value} :: \texttt{Zero}) : [y.\ y :: \texttt{Exp}] :=$
$\quad x.l$

$\texttt{eval}(x.x :: \texttt{Plus}, x.l :: \texttt{Lit}, x.r :: \texttt{Lit}, x.l.\mathit{value} :: \texttt{Nat},$
$\qquad x.r.\mathit{value} :: \texttt{Succ}, x.r.\mathit{value}.p :: \texttt{Nat}) : [y.\ y :: \texttt{Exp}] :=$
$\quad \texttt{eval}(\textbf{new}\ \texttt{Plus}($
$\qquad l \equiv \textbf{new}\ \texttt{Lit}(\mathit{value} \equiv \textbf{new}\ \texttt{Succ}(p \equiv x.l.\mathit{value})),$
$\qquad r \equiv \textbf{new}\ \texttt{Lit}(\mathit{value} \equiv \texttt{prev}(x.r.\mathit{value}))))$

$\texttt{eval}(x.\ x :: \texttt{Plus}, x.l :: \texttt{Exp}, x.r :: \texttt{Plus}, x.r.l :: \texttt{Exp}, x.r.r :: \texttt{Exp}) : [y.\ y :: \texttt{Exp}] :=$
$\quad \texttt{eval}(\textbf{new}\ \texttt{Plus}(l \equiv x.l, r \equiv \texttt{eval(x.r)}))$

$\texttt{eval}(x.\ x :: \texttt{Plus}, x.l :: \texttt{Plus}, x.r :: \texttt{Exp}, x.l.l :: \texttt{Exp}, x.l.r :: \texttt{Exp}) : [y.\ y :: \texttt{Exp}] :=$
$\quad \texttt{eval}(\textbf{new}\ \texttt{Plus}(l \equiv \texttt{eval(x.l)}, r \equiv x.r))$

We abstractly declare method `eval` to evaluate an expression to an expression, where literals are the values of the expression language. We use the constraints on the parameter to model a behavior similar to pattern matching. We overload the method for each case we want to match and set the requirements on the parameter accordingly.

Literals

Since literals are considered to be values, we evaluate an input x conforming to the requirements $x :: \texttt{Lit}$ and $x.\mathit{value} :: \texttt{Nat}$ to itself. The requirements make sure that the input is a proper literal.

Addition

We implement the evaluation of `Plus` expressions through evaluation on the right-hand side. We consider four cases:

1. Fields l and r are literals and r contains the *value* `Zero`.
2. Fields l and r are literals and r contains a *value* `Succ`.
3. Field r is a `Plus` expression.
4. Field l is a `Plus` expression.

The implementation of these cases is as follows:

1. The parameter is a `Plus` expression, both fields are literals and $r.value$ is `Zero`. The result of the evaluation is l.

2. The parameter x is a `Plus` expression, both fields are literals and $x.r.value$ contains a `Succ`. We create a new `Plus`. We set l to be the successor of $x.l$ and r to be the previous number of $x.r$.

3. and 4. The parameter x is a `Plus` expression and $x.l$ or $x.r$ are `Plus` expressions. We propagate evaluation to $x.l$ or $x.r$ respectively.

4.2 Object Construction

We interpret expressions of the defined program. We set the time out of the solver to 2 seconds.

We construct literals capturing 0 and 1.

$$\begin{aligned}(h_0, \mathit{zero}) &:= \mathit{interp}(\epsilon, \textbf{new } \texttt{Zero}())\\ (h_{l0}, \mathit{litZero}) &:= \mathit{interp}(h_0, \textbf{new } \texttt{Lit}(\mathit{value} \equiv \mathit{zero}))\\ (h_1, \mathit{one}) &:= \mathit{interp}(h_{l0}, \textbf{new } \texttt{Succ}(p \equiv \mathit{zero}))\\ (h_{l1}, \mathit{litOne}) &:= \mathit{interp}(h_1, \textbf{new } \texttt{Lit}(\mathit{value} \equiv \mathit{one}))\end{aligned}$$

We obtain

$$\begin{aligned}h_{l1} = &x_1 \mapsto (\texttt{Zero}, \epsilon)\\ &x_2 \mapsto (\texttt{Lit}, [\mathit{value} \equiv x_1])\\ &x_3 \mapsto (\texttt{Succ}, [p \equiv x_1])\end{aligned}$$

and $\mathit{litOne} = \textbf{new } \texttt{Lit}(\mathit{value} \equiv x_3)$. We observe that the last call to the interpreter returned prematurely. By inspecting the steps taken by the interpreter, we find that the entailment

$$x_1.\textbf{cls} \equiv \texttt{Zero}, x_3.\textbf{cls} \equiv \texttt{Succ}, x_3.p \equiv x_1, x_4.\textbf{cls} \equiv \texttt{Lit}, x_4.\mathit{value} \equiv x_3 \vdash x_4.\mathit{value} :: \texttt{Nat}$$

could not be shown by the solver. We know that

$$\begin{aligned}x_3 :: \texttt{Succ}, x_3.p :: \texttt{Nat} &\vdash x_3 :: \texttt{Nat}\\ x_1 :: \texttt{Zero} &\vdash x_1 :: \texttt{Nat}\\ x_4.\mathit{value} \equiv x_3, x_3 :: \texttt{Nat} &\vdash x_4.\mathit{value} :: \texttt{Nat}\end{aligned}$$

can be shown by the solver. These are sub-goals of the timed out derivation.

4.3 Optimization

We observed in Section 4.2 that the solver could not show an entailment, despite being able to show the sub-goals of the entailment in isolation. We can pre-optimize the entailment to be shown, to help the solver find a derivation. We can

1. pre apply path equivalence.
2. explore inheritance.

4.3.1 Apply Path-Equivalence

We apply path equivalence constraints contained in the context over the context. This can be done prior to the call to the solver. For path equivalence of the form $x \equiv p$, where x is a variable and p is a path. We substitute x with p in each constraint contained in the context. After all substitutions are applied, we remove reflexive path equivalence constraints from the context.

```
private def preprocEquiv(
              cs: List[Constraint]): List[Constraint] = {
  val eqs = cs.filter {
    case PathEquivalence(_, _) => true
    case _ => false
  }
  var res = cs
  eqs.foreach {
    case PathEquivalence(x@Id(_), p) =>
      res = res.map(c => substitute(x, p, c))
    case PathEquivalence(p, x@Id(_)) =>
      res = res.map(c => substitute(x, p, c))
    case _ => ()
  }
  res.filter {
    case PathEquivalence(p, q) if p == q => false
    case _ => true
  }
}
```

Figure 4.2: Preprocessing Of Path Equivalence

Figure 4.2 shows the Scala implementation of this process. The implementation matches for $x \equiv p$ as well as for the symmetric case $p \equiv x$. This preprocessing of the path equivalence constraints does not involve a call to the solver. The overhead produced by this process is small, with the complexity of the function being linear to the size of the context.

4.3.2 Explore Inheritance

We explore the inheritance relations of constraints in the context. Inheritance is expressed through constraint entailment declarations and these declarations are restricted to be of the form $\forall x.\ \overline{a} \Rightarrow p :: C$.

For each constraint of the form $p :: C$ in *context*: We check if $context \vdash p :: C'$ holds for each class C' declared in the program. If so, we add $p :: C'$ to *context*.

```
private def preprocInheritance(
        cs: List[Constraint]): List[Constraint] = {
  val insts = cs.filter {
    case InstanceOf(_, _) => true
    case _ => false
  }
  var res = cs
  insts.foreach {
    case InstanceOf(x, _) =>
      classes.foreach {
        case cls1
        if entails(cs, InstanceOf(x, cls1), List()) =>
          res = InstanceOf(x, cls1) :: res
        case _ => ()
      }
    case _ => ()
  }
  res
}
```

Figure 4.3: Preprocessing Of Inheritance

Figure 4.3 shows the Scala implementation of this process. The exploration of the inheritance relations in the context requires multiple calls to the solver, thus adding a significant overhead to the execution time.

4.4 Reapply Interpreter

With the optimizations from Section 4.3 we can now reapply the interpretations from Section 4.2. The interpreter can, with the optimizations to the context, reduce the expressions to values. Now we obtain

$$\begin{aligned} h_{l1} = & x_1 \mapsto (\texttt{Zero}, \epsilon) \\ & x_2 \mapsto (\texttt{Lit}, [value \equiv x_1]) \\ & x_3 \mapsto (\texttt{Succ}, [p \equiv x_1]) \\ & x_4 \mapsto (\texttt{Lit}, [value \equiv x_3]) \end{aligned}$$

and $litOne = x_4$.

We further interpret $\texttt{eval}(litOne)$ via

$$(h_{e1}, evl_1) := interp(h_{l1}, \texttt{eval}(litOne))$$

and obtain $h_{e1} = h_{l1}$ and $evl_1 = x_4$.

We construct $\texttt{Plus}(0, 1)$ and call `eval` on the new instance.

$$\begin{aligned}(h_{p01}, plus_{01}) &:= interp(h_{l1}, \textbf{new } \texttt{Plus}(l \equiv x_2, r \equiv x_4)) \\ (h_{ep}, evl_p) &:= interp(h_{p01}, \texttt{eval}(plus_{01}))\end{aligned}$$

We obtain $evl_p = \texttt{eval}(x_5)$

$$\begin{aligned}h_{ep} = h_{p01} = &x_1 \mapsto (\texttt{Zero}, \epsilon) \\ &x_2 \mapsto (\texttt{Lit}, [value \equiv x_1]) \\ &x_3 \mapsto (\texttt{Succ}, [p \equiv x_1]) \\ &x_4 \mapsto (\texttt{Lit}, [value \equiv x_3]) \\ &x_5 \mapsto (\texttt{Plus}, [l \equiv x_2, r \equiv x_4]).\end{aligned}$$

The evaluation of evl_p would require to find the most applicable method implementation, which times out. This shows that the optimizations have a limit.

The optimizations are applied during runtime and the application increased the amount of expressions reducible to values by the interpreter. Simultaneously, this increases the execution time of the interpreter, since we make additional calls to the solver.

Chapter 5

Improving Interpreter With Type Information

In the DC_C calculus we need to solve a constraint system when interpreting an expression, as well when type checking a program. The need to solve a constraint system produces overhead, as solving the constraint system takes some time. We are not concerned with the overhead during type checking (compile time), but only with the overhead during interpreting (runtime).

For example, we interpreted a method call in Example 3.8. During evaluation, we needed to find the most specific method that was applicable to the provided parameter. For this we needed to solve a constraint entailment for each method implementation. Another example is the type checking of the natural numbers program as seen in Example 3.9. There, we again needed to call the SMT solver.

In both examples, we needed to solve constraint entailments during execution. There are two cases to consider: The solver

1. does find a derivation.

2. does not find a derivation.

For (1): If solver finds a derivation, the constraint entailment holds. The amount of time it takes the solver to find this derivation is between 0 and *limit*, where *limit* is the maximum amount of time before the solver returns unknown.

For (2): If the solver finds a contradiction of the constraint entailment to the rules, the execution time of the solver is between 0 and *limit*. If the SMT solver does not find a contradiction for the constraint entailment, the time it takes for the solver to return is *limit*, as in those cases the solver needs to evaluate all possible rule instantiations to find either a derivation or a contradiction. In both examples, we try to solve constraint entailments that have no derivation.

While the provided implementation of the DC_C calculus can surely be further optimized without touching the model of the constraint system, this is an inherent problem of the DC_C calculus.

5.1 Formulating An Optimization Goal

We have seen that solving a constraint system, even in the best case scenario, adds overhead to the execution time. Therefore a goal for optimization could be to remove the need to solve a constraint system.

The complete elimination of the constraint system might be too optimistic. An alternative goal is to minimize the time needed for the solver to find a solution. The time it takes the solver, to find a derivation for a solvable constraint entailment, depends on the size of the search space. The size of the search space is affected by

- the amount of variables in quantifiers. Each additional quantified variables leads to an exponential growth in possible instantiations.
- the amount of rules supplied to the solver. Each additional rule leads to new possible rule applications.
- the amount of constraints in the context of a constraint entailment. Each additional constraint adds a restriction to the constraint entailment.

In Section 3.1.2, we explored the reduction of quantified variables. For rule C-Subst, we removed quantification for variables and paths through grounding. Additionally, we introduced generalization to find variables a and a_1 algorithmically. In the process to ground the quantified variables we introduced new rules to the solver. We reduced the amount of possible instantiations of the rule, but at the same time added new rules.

In Section 3.1.4 we explored the removal of rules, through the introduction of pruning. Pruning removed valid rules, in order to solve constraint entailments where the solver would otherwise instantiate the same rule in a loop. We introduced pruning as an additional call to the solver to preserve completeness. We reduced the amount of rules, but at the same time added the need to solve the same constraint entailment twice, if the first try times out.

Both, grounding and pruning, had positive effects as well as negative effects on the search space. Technically pruning did not increase the search space, but it has nonetheless negative effects on the execution time for a constraint entailment. Only the introduction of generalization removed quantified variables without introducing new rules or other means to increase execution time.

For a constraint entailment $ce := \overline{a} \vdash a$, the addition of new constraints to $\overline{a}$ does not require the introduction of additional rules or quantified variables into the model. It only affects the possible derivations of ce. It specifies information about the constraint entailment that was previously unknown.

5.2 Context Enrichment

In this section we explore, how to enrich the context $\overline{a}$ of a constraint entailment $ce := \overline{a} \vdash a$ with new information to reduce the time it takes the solver to find a derivation for ce.

To add new information to the context of ce, we can concatenate $\overline{a}$ with new constraints $\overline{b}$ and solve $\overline{a} +\!\!+ \overline{b} \vdash a$. The SMT solver can, after the addition of $\overline{b}$, make new rule instantiations that can lead to new derivations for ce.

The addition of new constraints $\overline{b}$ to the context can be dangerous. A misuse is possible, where the added constraints result in the SMT solver being able to prove an otherwise unsolvable constraint entailment.

Example 5.1 (Faulty derivation)**.**
The constraint entailment $x :: C \vdash y :: C$ is not solvable, as we have no way to derive the class of y from the class of x.

We enrich the context with $x \equiv y$ and obtain $x :: C, x \equiv y \vdash y :: C$. The enriched constraint entailment is solvable, because we added the information that x and y are equivalent.

Example 5.2 (Contradicting Information)**.**

$$x :: \texttt{Succ}, x.p :: \texttt{Nat} \vdash c$$

Taking the natural numbers program as a base, the context of the constraint entailment specifies x to be an instance of class `Succ`.

$$x :: \texttt{Succ}, x.p :: \texttt{Nat}, x :: \texttt{Zero} \vdash c$$

We added the information $x :: \texttt{Zero}$ to the context. This contradicts the inheritance relation of the program, as a class can not be an instance of `Zero` and `Succ` simultaneously.

In Example 5.1 we created $x \equiv y$ out of thin air and added it to the context of the constraint entailment. This enabled the solver to find a derivation and to solve the entailment. In Example 5.2 we added contradicting information to the context.

Hence, we need to make sure that the information we want to add holds for the constraint entailment we want to show.

5.3 Using Type Information

In this section we discuss how to find valid information to enrich the context of a constraint entailment.

Since we are only interested in the optimization of the runtime, we explore the information to gain we take a look at what information we can obtain from compile time. Type checking is defined in Section 2.2.1 for the DC_C calculus. We presented an implementation of the type checking in Section 3.3. We have a type relation that assigns a list of types ts to an expression e.

A type t is of the form $[x.\ \overline{a}]$. If t is a type of e, then $\overline{a}$ will hold for all possible values of e at runtime. We propose enriching the context of constraint entailments originating from e during runtime with $\overline{a}$. Since t introduces a variable binding x that is valid in $\overline{a}$, we need to eliminate x in $\overline{a}$. If y is the reduction value of e, we can eliminate x through substituting x with y in $\overline{a}$.

Example 5.3 (Context enrichment)**.**
We want to solve the constraint entailment

$$x :: \texttt{Succ}, x.p :: \texttt{Nat}, y :: \texttt{Zero}, x.p \equiv y \vdash x :: \texttt{Nat}$$

originating from the evaluation of an expression e with value x.

To solve this entailment, we use rule C-Prog with $\forall x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat} \Rightarrow x :: \texttt{Nat}$. The entailment is solved, after checking if $x :: \texttt{Succ}$ and $x.p :: \texttt{Nat}$ are also entailed by the context.

We enrich the context with types of e $[y.\ y :: \texttt{Succ}]$ and $[y.\ y :: \texttt{Nat}]$. We substitute y with x in the types and obtain $x :: \texttt{Succ}$ and $x :: \texttt{Nat}$. We add the constraints to the context and obtain the constraint entailment

$$x :: \texttt{Succ}, x.p :: \texttt{Nat}, y :: \texttt{Zero}, x.p \equiv y, x :: \texttt{Succ}, x :: \texttt{Nat} \vdash x :: \texttt{Nat}$$

We can close the entailment by using C-Ident, since $x :: \texttt{Nat}$ is in the enriched context.

We need to solve constraint entailments of the form as seen in Example 5.3 when interpreting expressions with the natural numbers program. In the example we see that, by augmenting the context of the entailment, the complexity of the derivation decreases. Instead of using rule C-Prog and checking its sub requirements, we can directly apply rule C-Ident. This is because a similar entailment needed to be solved in the type checking phase and we reuse the information instead of computing it again.

Chapter 6

Related Work

Logically Qualified Data Types (Liquid Types)

Liquid Types [RKJ08] are a system combining Hindley-Milner type inference with Predicate Abstraction to infer dependent types. The inferred dependent types are precise enough to prove various safety properties. Liquid Types adopt the benefits of static verification of critical properties and the elimination of runtime checks from dependent types, without the need for manual annotations.

Liquid Haskell: Haskell as a Theorem Prover

Liquid Haskell [Vaz16] is an usable program verifier, integrating the specification of correctness properties as logical refinements of Haskell's types. It uses the abstract interpretation framework of Liquid Types, to check correctness of specifications via SMT solving, requiring no explicit proofs or annotations. The specification language for Liquid Haskell is arbitrary expressive to allow the writing of general correctness properties, thus turning Haskell into a theorem prover.

Dependent Object Types (DOT)

DOT [AMO12, AGO$^+$16] is a calculus modeling Scala's path-dependent types and abstract type members. It uses refinement types to model the mixture of nominal and structural typing in Scala. DOT normalizes the type system of Scala through the unification of the constructs for type members. It provides intersection and union types to simplify the computations for greatest lower-bounds and least upper-bounds. DOT is at the core of *dotty*, a compiler for Scala under development since 2013 considered for inclusion in Scala 3.

Veritas

VeriTaS [GERM16, GPM18] is an ongoing project for the specification and verification of domain-specific languages (DSL). VeriTaS is aimed at automatically proving type soundness using first-order theorem proving. It allows for specifying the syntax and semantics of DSL and proof goals and axioms. It includes a compiler to translate these language specifications and the proof goals on them and proofs can be structured via proof graphs.

Multiple Dispatch as Dispatch on Tuples

Single dispatch selects a method using the dynamic class of the message's receiver. Multiple dispatch is a generalization of single dispatch. It selects a method based on the dynamic class of any subset of the message's arguments. Multiple Dispatch as Dispatch on Tuples [LM98] proposes a way to add multiple dispatch to existing languages with single dispatch, without affecting existing code. This is achieved through the addition of tuples as primitive expressions and the ability of messages to be sent to tuples. Methods are selected based on the dynamic classes of the elements of the tuple.

Chapter 7

Discussion

In this chapter we conclude the thesis. We discuss the contributions of the thesis, as well as possible future work.

7.1 Constraint System

We implemented the constraint system of the DC_C calculus in Section 3.1. The DC_C constraint system is given as a sequent calculus. We defined a model of the constraint system in first-order logic. Our aim for the model was to be structurally similar to the sequent calculus, as we want to use the same structural reasoning as in the rules of the sequent calculus. Since we employ a SMT solver for resolution of the constraint system, we defined a representation of the first-order model in the SMTLib format.

In Section 3.1.2 we refined the first-order model, by improving the usability of some rules with the SMT solver. We enumerated over variables and paths in rule C-Subst to reduce the amount of quantified variables. This reduction in quantified variables through enumeration came with the disadvantage of creating additional rules, where we need to create one rule for each variable and path combination.

We expressed declarative elements in rules C-Subst and C-Prog in an algorithmic way.

For rule C-Subst we introduced generalization as the inverse function of substitution. This allowed us to express the relation from the input a_2 to a via generalization and from a to a_1 via substitution.

For rule C-Prog we developed a process to convert the existence check for entailment declarations in the program to a lookup function. This allowed us to call the lookup function in rule C-Prog with the input a.

We observed that the SMT solver can loop and time out, if we supply unnecessary rules to find a derivation for the given input. We introduced pruning in Section 3.1.4 to remove rules from the model. In the generation process for rules C-Subst with enumeration, we multiplied the amount of available rules. We plugged the pruning process into this process to skip the generation of some

rules. Since the removal of valid rules can lead to incompleteness we used a two-pass call to the SMT solver, one with enabled pruning and one with disabled pruning.

Despite these improvements to the model, the SMT solver still does time out for some valid inputs.

7.2 Interpreter

We implemented the operational semantics with an interpreter in Section 3.2. The implementation of the interpreter is straightforward. The rules of the operational semantics are syntax oriented and can be "read from bottom to top" to yield an algorithm [Pie02].

The performance of the interpreter is bound by the performance of the SMT solver. This includes the time needed to evaluate an expression to a value as well as the question if we can fully evaluate an expression, given that the expression is reducible to a value.

7.3 Type Checker

We implemented the type relation of the DC_C calculus in Section 3.3.

The definition of type assignments for expressions in the DC_C calculus is declarative. The rules are not syntax oriented, as rule T-Sub can be applied to any expression. Type assignment is also not unique.

The implementation of the type assignment relation is not as straightforward as the implementation of the operational semantics.

For the implementation of the subsumption rule T-Sub, we integrated subtyping checks into the remaining type rules. The inheritance relation is expressed via the constraint system and we can therefore use the SMT solver to check for subtyping.

Since type assignments are not unique, our implementation returns a list of types for an expression.

The rules for the well-formedness of programs are again syntax oriented and the rules form a structure for the implementation. The rules of well-formedness check mostly the free variables of declarations. The outlier being the rule WF-Mi for method implementations. The rule requires that the return type of a method equals the type of the method body. For this, the rule uses type assignment on the body and compares the result to the annotated return type. Since the type assignment implementation returns a list of types instead of one type, our implementation requires that the annotated return type is contained in the types returned by the type assignment.

The performance of the type checker is, as the performance of the interpreter, bound by the performance of the SMT solver.

7.4 Information Gain From Type Checker

The SMT solver plays a role in the time it takes to interpret an expression. The time it takes the SMT solver to find a solution for a solvable input, depends on the amount of rule instantiations needed to close the derivation.

We proposed *context enrichment* in Chapter 5, as a process to reduce the number of rule applications needed to solve a constraint entailment. It is possible to reduce the amount of steps it takes to solve a constraint entailment ce by adding constraints t to the context of ce. Adding constraints to the context can result in the solver being able to find a solution for an "invalid" constraint entailment. Thus, we need to make sure that the added constraints do not contradict the information available in the context.

In the DC_C calculus, a type of an expression e is a list of constraints that hold for each possible value of that expression during runtime. So it is safe to add t to the context of ce, if ce originated from the interpretation of e.

7.5 Future Work

The used SMT solver, as already stated, is unable to successfully show all solvable constraint entailments. There are multiple possibilities to increase the performance of the solver.

- Exploring if Cut-elimination [Sch77] can be applied to the defined sequent calculus of the DC_C calculus.
- Further optimizations to the presented first-order model.
  - Applying the enumeration process used for rule C-Subst onto more variables and rules.
  - There are still declarative rules in the model. Searching for patterns that can be expressed in an algorithmic way in those rules can help the solver to successfully instantiate those rules.
  - The presence of unnecessary rules is a problem for the SMT solver. Developing a heuristic to determine which rules will not be used, would prevent the solver from looping in such a rule and time out less as a result.
  - SMT solvers in general and Z3 in particular are highly configurable. The amount of different available options and their possible side-effects on each other make it difficult to find the best configuration for the problem under consideration.
- Developing a new model of the constraint system. It might yield better result in combination with a SMT solver, if a model does not mimic the structural reasoning of the sequent calculus.
- There is a wide range of SMT solvers available, all with unique strengths and weaknesses. Other SMT solvers than Z3 could be more suitable for our model of the constraint system.

- There are automated theorem provers available apart from SMT solvers. As with the previous point about alternative SMT solvers, one could explore the capabilities of different proving systems on the constraint system.

Bibliography

[AGO+16] Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272, 2016. URL: `http://infoscience.epfl.ch/record/215280`, `doi:10.1007/978-3-319-30936-1_14`.

[AMO12] Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. 2012. URL: `http://infoscience.epfl.ch/record/183030`.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[BT18] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. URL: `https://doi.org/10.1007/978-3-319-10575-8_11`, `doi:10.1007/978-3-319-10575-8_11`.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM. URL: `http://doi.acm.org/10.1145/1111037.1111062`, `doi:10.1145/1111037.1111062`.

[Gas10] Vaidas Gasiūnas. *Advanced Object-Oriented Language Mechanisms for Variability Management*. PhD thesis, Technische Universität, Darmstadt, December 2010. URL: `http://tuprints.ulb.tu-darmstadt.de/2353/`.

[GERM16] Sylvia Grewe, Sebastian Erdweg, Michael Raulf, and Mira Mezini. Exploration of language specifications by compilation to first-order logic. In *Proceedings of the 18th International Symposium*

on Principles and Practice of Declarative Programming, PPDP '16, pages 104–117, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2967973.2968606, doi:10.1145/2967973.2968606.

[GMO07] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 133–152, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1297027.1297038, doi:10.1145/1297027.1297038.

[GPM18] Sylvia Grewe, André Pacak, and Mira Mezini. Using vampire with support for algebraic datatypes in type soundness proofs. In Laura Kovács and Andrei Voronkov, editors, *Vampire 2017. Proceedings of the 4th Vampire Workshop*, volume 53 of *EPiC Series in Computing*, pages 42–51. EasyChair, 2018. URL: https://easychair.org/publications/paper/9gkr, doi:10.29007/pmmz.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[LM98] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 374–387, New York, NY, USA, 1998. ACM. URL: http://doi.acm.org/10.1145/286936.286977, doi:10.1145/286936.286977.

[OMM+04] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, 2004.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1375581.1375602, doi:10.1145/1375581.1375602.

[Sch77] Helmut Schwichtenberg. Proof theory: Some applications of cut-elimination. In Jon Barwise, editor, *HANDBOOK OF MATHEMATICAL LOGIC*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 867 – 895. Elsevier, 1977. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08711248, doi:https://doi.org/10.1016/S0049-237X(08)71124-8.

[Vaz16] Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, UC San Diego, 2016. URL: `https://escholarship.org/uc/item/8dm057ws`.