Master Thesis
Master of Science Informatik

???

Matthias Krebs

Technische Universität Darmstadt
Fachbereich Informatik
Software Technology Group

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

Abgabetermin: ???

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

Contents

1 Introduction **3**

1.1 Motivation . . . 3

1.2 Contributions . . . 3

1.3 Structure . . . 3

2 Preliminaries **4**

2.1 Satisfiability modulo theories . . . 4

2.1.1 SMT-Lib . . . 4

2.2 Dependent Classes . . . 5

2.2.1 DC_C Calculus . . . 6

3 DC_C Implementation **9**

3.1 Constraint System . . . 9

3.1.1 Axioms / FO . . . 9

3.2 Interpreter . . . 9

3.3 Type Relation . . . 9

3.3.1 Well formedness of Programs . . . 9

3.3.2 Type assignments for Expressions . . . 9

4 Improving Interpreter / Learning from Type checking **10**

5 Case Study: ??? **11**

6 Related Work **12**

Bibliography **13**

Chapter 1

Introduction

1.1 Motivation

1.2 Contributions

1.3 Structure

Chapter 2

Preliminaries

2.1 Satisfiability modulo theories

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 ptroblem in propositional logic and solved using a propositional SAT solver. Other problems can be described more naturally in other classical logics like first-order logic. These logics support more epressive language features including non-boolean variables, function and predicate-symbols and quantifiers.

The defining problem of Satisfiability modulo theories (SMT) is checking if a logical formula is satisfiable in the context of some background theory. Checking validity can be achieved using SMT if the formula is closed under logical negation, since a formula is valid in a theory if it's 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 propoerties 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, temportal, ...), the background theory, the accepted input formulas and the interface provided by the solver.[BT18, BFT17] Z3 is a SMT solver from Microsoft Research[dMB08], which is used in this thesis.

2.1.1 SMT-Lib

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

- providing a standard description of background theories
- develping a common in- and output language for SMT solvers

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

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

Figure 2.1: SMT-Lib format example

- 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 and the SMT-Lib representation of the formula. 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 modulatiry. Desired functionality involes groups of related classes. Grouping mechanisms exist, like namespaces in C++ and packages in Java, but these mechanisms do not cover inhertiance 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.

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 modifcation 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 generalization 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 apossible implementation. The algorithmic nature of the calculus comes also with the disadvantage of being difficult to extend semantics with new expressive power, such as extending the calculus with 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 system, which is used in both the static and dynamic semantics. The main relation of the DC_C calculus is constraint entailment, replacing vc^n's relations for type equivalence, subtyping, static and dynamic dispatch.

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.

Syntax

The syntax of DC_C is given in figure 2.2. 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.

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.

$\langle \mathit{Program} \rangle$ ::= $\overline{\mathit{Decl}}$

$\langle \mathit{Decl} \rangle$ ::= $C(x.\ \overline{\mathit{Constr}})$ | $\forall x.\ \overline{\mathit{Constr}} \Rightarrow \mathit{Constr}$
| $m(x.\ \overline{\mathit{Constr}}) : \mathit{Type}$ | $m(x.\ \overline{\mathit{Constr}}) : \mathit{Type} := \mathit{Expr}$

$\langle \mathit{Type} \rangle$ ::= [x. $\overline{\mathit{Constr}}$]

$\langle \mathit{Constr} \rangle$ ::= $\mathit{Path} \equiv \mathit{Path}$ | $\mathit{Path} :: C$ | $\mathit{Path}.\mathbf{cls} \equiv C$

$\langle \mathit{Path} \rangle$ ::= x | Path.f

$\langle \mathit{Expr} \rangle$ ::= x | Expr.f | **new** $C(\overline{f} \equiv \overline{\mathit{Expr}})$ | m(Expr)

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

x, y ∈ variable names
f ∈ field names
C ∈ class names
m ∈ method names

Figure 2.2: Syntax

Constraint System

Operational Semantics

Type Checking

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

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

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

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

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

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

Figure 2.3: Constraint Entailment

$$\begin{aligned} o &::= \langle C; \overline{f} \equiv \overline{x} \rangle \\ h &::= \overline{x} \mapsto \overline{o} \ \ (x_i \text{ distinct}) \end{aligned}$$

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

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

$$\frac{A \vee B \qquad \begin{matrix}[A] \\ C\end{matrix} \qquad \begin{matrix}[B] \\ C\end{matrix}}{C}$$

Figure 2.4: Operational semantics

TODO

Figure 2.5: Type assignment

TODO

Figure 2.6: Type checking

Chapter 3

DC_C Implementation

3.1 Constraint System

3.1.1 Axioms / FO

3.2 Interpreter

3.3 Type Relation

3.3.1 Well formedness of Programs

3.3.2 Type assignments for Expressions

Chapter 4

Improving Interpreter / Learning from Type checking

Chapter 5

Case Study: ???

Chapter 6

Related Work

Bibliography

[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/`.

[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`.