Theory theory (and an attempt to orient objections to object orientation)

by

Ioannis T. Kassios

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

© Copyright by Ioannis T. Kassios 2001
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

The subject of this thesis lies in two different areas: formal theories and object oriented programming. First, a “theory theory” is presented, that is a mathematical formalism that expresses formal theories in a structured way. Several new concepts related to this formalism are explored and its potential advantages in comparison to similar attempts are pinpointed. Following that, we engage in a conversation about the popular object oriented programming paradigm, we argue about what it tries to achieve and we present various important frameworks developed in the past that formalize its ideas. We go on to “build” our own formal understanding of object orientation based on theory theory, and we expand the model in various interesting ways. Finally, we use the insight acquired by marrying formal theories and object orientation to sketch the design of a programming language.
Theory theory (and an attempt to orient objections to object orientation)

Ioannis T. Kassios
Master of Science
Department of Computer Science
University of Toronto 2001

October 3, 2001
Acknowledgments

Lots of people are to be thanked for their direct or indirect contribution towards the creation of this work. First of all, those to whom I dedicate it, my family and beloved ones, because they gave me both life and a reason to live. I will never be able to offer back what all these people have offered me.

I owe many thanks to my supervisor, Eric Hehner, both for his technical contributions and his general support. The technical part of his help consists of both his “bunch theory” upon which my whole theory is founded and the various valuable ideas and directions he gave me for this work. Besides that, he has been as enthusiastic as I was about this project and this attitude generally helped me finish it a lot earlier than expected, with what I believe to be a satisfactory result.

Many thanks to the Graduate Computer Science Department of the University of Toronto, for providing me with the academic atmosphere needed to conduct my research and for making it very enjoyable. I should especially mention the people at the Formal Methods group of the Department. Professors Derek Corneil, Tony Pittassi, monica schraefel, who gave me the opportunity to work with them. I don’t have words to describe the excellence of the aca-
demic environment provided by the faculty and the quality (both scientific and personal) of the people I had the chance to work with. Of course I should not forget the financial help I received by the Department, without which my study in Toronto and this work would have been impossible.

I also recognize the help given to me by fellow graduate students in the department, who not only made my stay in Toronto enjoyable, but with whom I also had valuable discussions that influenced this work. I found it very interesting to talk about my research with people from other fields of Computer Science. Especially I would like to thank my officemates Travis Gagie, Ben Vitale and Rozalia Christodoulopoulou.

Of course, I should also thank all the referees in the bibliography (even though I don't know personally most of them), because their work was very important to me and very inspiring. Most of all, I thank Rustan Leino of Compaq, with whom I also discussed the project and who gave me interesting references and ideas.

Thanks to all the people that facilitated my stay in Toronto and my study at the University. That would include Irene Stubos, Marinos Georgatos, Eftychis Psimoulis, the Xenos travel agency, the Graduate Studies Office, the Greek Computer Science students, the people who work at restaurants and coffee shops near my office and the list goes on and on.

Thanks to people in the Software Engineering Laboratory (SoftLab) of the Electrical and Computer Engineering Department of the National Technical University of Athens, especially my former supervisor Manolis Skordalakis for
helping me and working with me and Nikos Papaspyrou who also helped me with the typesetting. Part of this thesis is written using the resources of this laboratory, where I was also working the year before I was admitted to the University of Toronto.

Last but not least, I thank all future readers of this thesis for being willing to suffer their way through it. Thanks and I apologize for any inconvenience my eccentric formalism may cause, but I do believe it's all for a good cause!

Yannis Kassios
Dedicated to my family.
to Georgia.
to my friends
and to all the other people that I love
# Contents

1 Preface

2 Mathematical basis
   2.1 Elements .................................................. 5
   2.2 Bunches ...................................................... 7
   2.3 Strings ..................................................... 10
   2.4 Functions ................................................... 11
      2.4.1 Operators ................................................ 11
      2.4.2 Distribution over bunch union .................. 13
      2.4.3 Function inclusion .................................... 14
      2.4.4 Prefix and infix operators ..................... 16
      2.4.5 Restricted domain functions .................... 16
   2.5 Structure ................................................... 19
      2.5.1 Sets .................................................... 19
      2.5.2 Lists ................................................... 20
   2.6 Formalizing elementhood ................................. 22
   2.7 Quantifiers ................................................ 24
2.7.1 Generic definition of quantifiers ........................................ 24
2.7.2 Set-operation quantifiers .................................................. 26
2.7.3 Bunch-wise quantifiers ..................................................... 27
2.7.4 The solution quantifier ..................................................... 27
2.8 Inversion of functions .......................................................... 29

3 Theory theory .......................................................................... 30
3.1 An introduction to formal theories ........................................... 31
  3.1.1 Defining theories ............................................................. 31
  3.1.2 Operations on theories ..................................................... 34
3.2 Implementation as implication .................................................. 38
3.3 Basics of theory theory ............................................................ 39
  3.3.1 Taking care of sorts and functions ..................................... 40
  3.3.2 Theory axioms ................................................................. 40
  3.3.3 Implementation as bunch containment ............................... 41
  3.3.4 Term abstraction .............................................................. 42
  3.3.5 The $\Theta$ quantifier ....................................................... 43
  3.3.6 The $\rightarrow$ notation ....................................................... 44
  3.3.7 Theory laws and operations ............................................. 45
  3.3.8 Syntactic sugar for abs axiom ......................................... 49
3.4 Examples ............................................................................... 52
  3.4.1 A theory is one of its sorts: a number theory ..................... 52
  3.4.2 A theory as not one of its sorts: a group theory ................ 54
  3.4.3 Why elementhood doesn't matter: a set theory .................. 56
3.4.4 Combining theories: a sorting function theory .......... 59
3.4.5 Gradual implementation: a mergesort implementation .. 63
3.4.6 Transforming theories: a point theory .................. 68
3.5 Advanced theory theory .................................. 72
  3.5.1 High level elementhood ................................ 72
  3.5.2 Elevating .............................................. 76
  3.5.3 Accessing various levels of elementhood ............... 77
  3.5.4 Metatheories .......................................... 78
3.6 Related work: Z and B .................................... 80

4 The object oriented paradigm ................................. 83
  4.1 Overview of object orientation ............................ 84
    4.1.1 Objects and encapsulation ............................ 85
    4.1.2 Classes ............................................... 87
    4.1.3 Inheritance .......................................... 89
    4.1.4 Polymorphism ......................................... 92
    4.1.5 Overriding ........................................... 93
    4.1.6 "Object invariant" ................................... 95
    4.1.7 Generalization ....................................... 96
    4.1.8 Interaction with functional programming ............. 97
    4.1.9 Summary ............................................... 101
  4.2 Formal semantics for object orientation .................. 102
    4.2.1 Quantification of types ............................... 103
    4.2.2 $F$-bounded polymorphism ............................. 109
B  Theory laws and proofs  160

C  High level elementhood theory  171
Chapter 1

Preface

This thesis is about the marriage of two seemingly different areas of research in Computer Science, namely *formal theories* and *object orientation*. Formal theories exist to express specifications in an unambiguous way. Object orientation exists to modularize the effort of software development. The two of them together may be used to provide modularity and precision to both software design and development. To be merged together, both worlds need to be extended accordingly. This thesis therefore first presents an object-oriented-like view of formal theories and then a formal-theory-like view of objects to finally conclude that those two can be unified into one single concept keeping the virtues of both.

The first contribution of the thesis is the creation of "theory theory", a way of viewing whole theories as a special kind of mathematical objects, instead of collections of axioms. Theory theory is a theory that speaks about theories, its objects of interest are theories. Operations on theories and relations between theories are defined mathematically. A significant degree of unification is built in this formalism: everything is based on *bunch theory*, everything is a bunch.
A function is a special kind of bunch and a theory is defined as a special kind of function. Everything else we want to express mathematically can be expressed in terms of theories. In that sense, theory theory is intended to be a "universal mathematical language". It keeps the uniformity, simplicity and flexibility of bunch theory but it also adds modularity to it.

The second contribution is the formalization of object oriented programming using theory theory. This is not very straightforward, because object orientation is not mathematically defined and not everybody agrees on exactly what it is. To deal with this problem, we go on to define what we think object orientation's purpose is. We argue that this goal is not yet met by the model (that's why we "orient objections" to object orientation) and that the model needs to be significantly extended. We do that in a formal way using theory theory. The result is a novel view of object orientation that goes further away in the direction of meeting what we think it was created for in the first place. In support of our argument, we sketch the design of a programming language based on our idea of object oriented programming. We believe this language to be a very useful basis for software design, implementation and automated verification.

In chapter 2, we present a (slightly modified) version of Eric Hehner's bunch theory [16] and unified algebra [17] that we will use throughout the whole thesis as the mathematical basis. This theory is referred to in the thesis as bunch theory. Bunch theory emphasizes unification of similar mathematical ideas (e.g. everything is a bunch, functions are also bunches) and simplicity of the used mathematical structures (e.g. bunches are used instead of sets when structure
is not relevant).

In chapter 3 we present theory theory. First, the notion of formal theories as collections of formal axioms is treated [6], and then theory theory is proposed as a universal formalism to reason about them. Several examples are given, before going on to more advanced material. The framework is related to Z and B specification languages [1], [30]. A short comment comparing Z and B with theory theory is included at the end of chapter 3.

Chapter 4 is divided in three major sections. The first speaks informally about the object oriented programming paradigm, presents its basic ideas, its strong points, its semantic gaps and shortfalls and our own view on them. After that, the second section refers to various attempts by others to formalize the model into semantics, emphasizing the most influential ones for this work. Based on all this background, the last section presents a theory theory semantics of object orientation that unifies many concepts, thus simplifying things as much as possible, but, more importantly, formally creates solutions to the problems that our previous analysis of object orientation revealed.

Finally, in chapter 5 we offer the "formal" conclusion of this study, creating in an abstract level a programming language based on theory theory and the related object orientation. The language is named Alpha. Chapter 6 contains the "informal" conclusion of the thesis.
Chapter 2

Mathematical basis

This chapter presents the mathematical background of the thesis, namely bunch theory and unified algebra [16], [17]. Eric Hehner's bunch theory is a formalism designed to unify as much as possible concepts that are similar with one another. For example, the notion of one element and of one collection of elements (a bunch) are in this theory indistinguishable. The ultimate goal is to avoid redundancy as much as possible, with the hope that this simplicity reflects to the formal semantics of programs and therefore facilitates their automated verification. The bunch theory is included in unified algebra, a further attempt by the same author to unify all mathematical elements, such as boolean values, numbers and operators that apply to them. Some of the unifications seem somewhat far-fetched and surprising, but they turn out to be useful later on.

In this thesis, we utilize ideas from both theories, but our framework will be somewhat modified to suit the material better. These modifications will be pointed out in this chapter, together with the reasons that led to them. From now on, we will refer to this mathematical background as the "bunch theory".
2.1 Elements

The unification of other mathematical theories begins from the simplest items in bunch theory, that is the elements. This is primarily an idea seen in unified algebra [17]. There are two kinds of unstructured elements in bunch theory, characters and numbers. By numbers we mean of course complex numbers, which include real numbers, which include rational numbers etc. but we also incorporate the boolean algebra. We do that by defining true to be \( -\infty \) and false to be \( \infty \). Boolean functions are now special cases of numerical functions:

- Logical conjunction is unified with the maximum function, and are both denoted by \( \wedge \).
- Logical disjunction is unified with the minimum function, and are both denoted by \( \vee \).
- Logical negation is just \( \neg \).
- Implication is unified with the "greater than or equal to" relation, and they're both denoted by \( \geq \). Reverse implication is also unified with \( \leq \).
- Logical equivalence is unified with \( = \).
- Exclusive or can be represented as either \( \neq \) or \( \times \) (multiplication).

The rest of the number theory is left as it is, but now we may perform operations between booleans and numbers, for example \((\text{true} \land 3) = 3\) and \((\text{true} \lor 3) = \text{true}\).
The value of this unification is not immediately seen if one does not think about our goal to eliminate redundancy from our symbolism. But keeping that goal in mind shows why unified algebra is important: for example by unifying $\Leftrightarrow$ with $=$ allows us to state the very important equivalence axioms (that hold for both) only once. Such economy may prove very useful in our try to reduce the complexity of a proving system. This concept of unification will be popping up in various places in this thesis.

In the original unified algebra proposal \[17\], the convention is different, that is $T$ (which represents true) is unified with $\infty$ and $\bot$ (false) with $-\infty$. Here it's the other way around, and we don't use the top and bottom symbols at all. The reason for that is that in this way the unification of operators makes more sense: (for example $\land$ looks a lot like a maximum symbol, or $\leq$ looks a lot like $\Leftarrow$ and both can be typed $\Leftarrow$).

Another difference with unified algebra is the use of the ternary operator if then else, which will remain here as is in \[16\].

The axioms of unified algebra include all number theory and all classic logic and won't be mentioned here. \[17\] contains a complete list of such axioms.

Characters are also elements we will incorporate in our theory. To denote any character $a$ we precede it with $'$, so that we may distinguish it from variable $a$. 

6
2.2 Bunches

The notion of bunch is central to the mathematical theory presented here, because everything else is expressed using bunches. Intuitively bunch is a set without a structure, or, in other words, the contents of a set. So for example, \{2,3\} is a set, whose content is the bunch 2, 3. The main difference between sets and bunches is that a bunch of one element is the element (unlike sets, where \{x\} \neq x) and that bunches lack set structure (they cannot nest). Operator \(\sim\) transforms a set into the bunch it contains, that is for all bunches \(A\):

\[ \sim \{A\} = A \]

We present bunch containment (\(\cdot\)), bunch union (\(\cdot\)) and bunch intersection (\(\cdot\)) with respect to their corresponding set theoretic operators (for all bunches \(A, B\)):

\[ A \cdot B = \sim (\{A\} \cup \{B\}) \]

\[ A \cdot B = \sim (\{A\} \cap \{B\}) \]

\[ A : B = A \in \{B\} \]

Also:
\[ A = B = \{A\} = \{B\} \]

(Note that \( \equiv \) is used as a version of \( = \) with smaller precedence, look at Appendix A for a list of all operators and their precedence).

[16] contains the formal definition of bunches. Several important axioms are listed there. Of them, the distribution of functions over \( , \) is worth mentioning. This allows us to create bunches from bunches in a very natural and compact way, for example

\[ (0, 1) + (2, 3) = 2, 3, 4 \]

says that 2, 3, 4 is the bunch of elements that we take if we add any of 0, 1 to any of 2, 3.

Bunches are a convenient way to denote collections. In this thesis several constant bunches will be used:

- \( \text{nat} \) is the bunch of natural numbers \( \text{nat} \) is defined by \( \text{nat} = 0, \text{nat} + 1 \) and the axiom that for all bunches \( B = 0, B + 1 \geq \text{nat} : B \)

- \( \text{bool} = \infty, -\infty \) (also \( \text{bool} = \text{false, true} \))

- \( x\text{nat} = \text{nat}, \infty \)

- \( \text{int} = -\text{nat}, \text{nat} \) is the bunch of integers

- \( \text{rat} = \text{int}/(\text{nat}+1) \) is the bunch of rational numbers (notice the convenient way to denote positive integers: \( \text{nat} + 1 \))
• real is the bunch of real numbers

• xreal = real, bool and xrat = rat, bool and xint = int, bool

• complex = real + \mathcal{J} \times real, where \mathcal{J} is the imaginary square root of -1

• xcomplex = complex, bool

• char is the bunch of characters

• null =\emptyset is the empty bunch

Another bunch operation we will introduce is .. For \(x, y : int\) and \(x \leq y\),
\(x,..y\) is the bunch of all integers \(z\) such that \(x \leq z \leq y\).

The elem predicate returns true iff its bunch parameter is an element:
\(elem\ 1 = true\) while \(elem\ (0, 1) = false\).

The if then else operator distributes over bunch union, with respect to its boolean condition, so for any bunches \(A\) and \(B\) we have:

\[
\text{if} \ true \ \text{then} \ A \ \text{else} \ B \ = \ A
\]

\[
\text{if} \ false \ \text{then} \ A \ \text{else} \ B \ = \ B
\]

\[
\text{if} \ bool \ \text{then} \ A \ \text{else} \ B \ = \ A, B
\]

\[
\text{if} \ null \ \text{then} \ A \ \text{else} \ B \ = \ null
\]
The \texttt{let in} construct allows us to name expressions within an expression. The naming part consists of assignments of the form \texttt{name = expression} separated by dots. For example,

\begin{verbatim}
let x = 1. y = 2 in x + y  =  3
\end{verbatim}

In this thesis we use bunches instead of sets, unless we need set structure.

\section{2.3 Strings}

\textit{Strings} are sequences of bunches. Our string theory is identical to that of [16], before the introduction of lists. Axioms are found there.

The string notation we heavily use in this thesis is:

- \texttt{nil} is the empty list
- \texttt{;} is the catenation operator
- \texttt{*} is the repetition operator. For any \texttt{x : nat}, \texttt{x * s} means \texttt{x} copies of string \texttt{s}. Note that \texttt{x} may be a (non-elementary) bunch, which gives a (non-elementary) bunch of strings as a result of \texttt{x * s}
- Unary \texttt{*} abbreviates \texttt{nat*}. Thus a way to denote the Kleene star closure of string \texttt{s} is \texttt{*s}
- \texttt{..} is an operator similar to \texttt{...}. For elements \texttt{x, y : int} and \texttt{x \leq y}, \texttt{x;..y} is the sequence of integers \texttt{x; x + 1,...} etc. up to (but not including) \texttt{y}.
String catenation distributes over bunch union. An elementary string is one whose items are all elements. A non-elementary string is a bunch, for example $0; (1, 2) = (0; 1), (0; 2)$. A string of bunches is a bunch of strings. A bunch of strings however does not always resolve into a string of bunches.

Strings have natural length. $\textit{nil}$ has length 0. A string of one bunch has length 1. The length of the catenation of two strings is the sum of their lengths.

We also wish to introduce $\textit{text} = \textit{char}$ as the bunch of all character strings. Such strings may be written using the natural double-quotes notation: "hello" = 'h'; 'e'; 'l'; 'l'; 'o'.

2.4 Functions

A function maps sub-bunches of a bunch called the domain to other bunches. If $f$ is a function, then $\mathcal{D}f$ denotes its domain. If $x : \mathcal{D}f$, then $f x$ is the bunch to which $x$ is mapped by $f$. $f x$ is called the application of $f$ to $x$.

2.4.1 Operators

We heavily use the symbolism of [17] for functions. There are three main operators that create functions.

Given bunches $A$ and $B$ with $B \neq \textit{null}$, $A \rightarrow B$ is the function $f$ that maps each sub-bunch of $A$ to $B$, that is:

$$\mathcal{D}(A \rightarrow B) = A$$

and for all $x : A$
Operator \(\langle \rangle\) is meant to substitute \(\lambda\)-notation. Let \(D\) be a bunch and \(E\) be a bunch expression that includes \(z\). Then \(\langle z : D \to E \rangle\) denotes the function that applied to an element \(y : D\) returns \(E^y_x\), that is the value of \(E\) if we substitute (according to the proper \(\lambda\)-calculus substitution rules) \(x\) with \(y\) within \(E\). This function distributes over bunch union. If it is applied to a non-elementary bunch \(X : D\), its result will be the union of the application to all elements in \(X\). All this stated formally:

\[D\langle z : D \to E \rangle = D\]

\[
elem y \land y : D \geq \langle z : D \to E \rangle y = E^y_x
\]

\[X, Y : D \geq \langle z : D \to E \rangle (X, Y) = \langle z : D \to E \rangle X, \langle z : D \to E \rangle Y\]

Note that \(E\) should not evaluate to null for any value of \(x\) in \(D\).

Finally, operator \(|\) is used to create a new function out of two other functions. If \(f\) and \(g\) are functions that distribute over bunch union, then \(f|g\) also distributes over bunch union and:

\[D(f|g) = Df, Dg\]
For $x$ an element of $\mathcal{D}(f|g)$:

$$(f|g)x = \text{if } x : \mathcal{D}f \text{ then } f \ x \text{ else } g \ x$$

Notice that we only allow for one parameter in our functions. That's general enough, because we can always return a function as a result (that's called "currying", in honor of Haskell Curry).

2.4.2 Distribution over bunch union

With the operators introduced so far we may create functions that distribute over bunch union, so that when applied to non-elementary bunches their result is the union of their application to each element of the parameter. This is a very interesting and useful property, but we would wish to be able to define functions that do not distribute over bunch union too. For example the cardinality of a bunch should have that property. However we will not introduce any notation to do that. We prefer to only define distributing functions (also see [27]) and let set structuring achieve the non-distributing effect for us. A set is considered an elementary bunch as will be discussed later on.

Let $f$ be a function distributing over bunch union. Then the following important properties hold:

$f \text{ null } = \text{ null}$

$f(A, B) = f \ A, f \ B$
\[ A : B : \mathcal{D}f = f \quad A : B : f(\mathcal{D}f) \]

Distribution over bunch union is an interesting feature but it must be used with great care. For example take a look at the function \( \langle x : X \rightarrow Y \rangle \). \( x \) within \( Y \) is always considered an element, which means that non-bunch union distributing functions may be used in a non-proper way. In particular \( \langle x : X \rightarrow \text{elem } x \rangle \) will always return \textit{true} and it is not equal to the \textit{elem} function. The same way, equality may be abused (the parameter equated with a non-element results in \textit{false}) and also the \( \mathcal{D} \) operator. Suppose that we want to check whether a given function is the identity \( \text{id} = \langle x : \text{nat} \rightarrow x \rangle \) over \text{nat}. Then:

\[ \text{isid} = \langle f : (\text{nat} \rightarrow \text{nat}) \rightarrow f = \text{id} \rangle \]

doesn't work, because \( f = \text{id} \) will always resolve to \textit{false} for any element \( f \) (here in fact we \textit{imagine} the existence of an elementary \( f \) since functions never really get to be elementary). As mentioned earlier, set structure will be used to avoid bunch union distribution whenever needed.

### 2.4.3 Function inclusion

A function is also a bunch and a very interesting one too. We want to define a convenient : relation for functions. A function \( f \) should be \textit{included} in function \( g \) if it can be used in its place. \( g \) serves as a type and \( f \) as a value here, but \( f \) can serve as a type for other functions. Suppose \( g = 2 \rightarrow \text{nat} \). Then \( g \) can be thought of as the function that maps 2 to \text{nat}. It can also be thought of as
the bunch of all the functions that map 2 to nat. If $f : g$ then $f$ can be either a restriction of $g$ such as $2 \rightarrow nat \times 2$ or even $2 \rightarrow 3$ or an extension such as $2 \rightarrow nat|3 \rightarrow 0$.

In general the function inclusion axiom says that a function is included in another function if both of these bunch containments happen. So function inclusion is antimonotonic with respect to the domain and monotonic with respect to the return value. Formally:

$$f : g = Dg : Df \wedge \{x : Dg \rightarrow f x : g x\}$$

The $\wedge$ quantifier will be introduced later on, but informally it's a universal quantification over all elements within the domain of the quantified function.

The net effect is that all functions are included in any function with domain $null$, so we denote $allf = null \rightarrow 0$ the bunch of all functions. This sounds a bit dangerous, since it can produce Russell paradox-like situation such as:

$$strange = \{f : allf \rightarrow if f f then false else true\}$$

where it seems that

$$strange\ strange = true = strange\ strange = false$$

but that only goes to say that $strange\ strange$ never gets to be an elementary bunch. This doesn't seem to be much of a problem.
A form of the inclusion axiom for "function types", that is general functions of the form \( A \rightarrow B \) follows:

\[
A \rightarrow B : C \rightarrow D \equiv C = \text{null} \lor C : A \land B : D
\]

Function equality is consistent with bunch equality:

\[
f = g \equiv f : g \land g : f
\]

### 2.4.4 Prefix and infix operators

In this thesis various unary prefix and binary infix operators are used, listed in Appendix A. They are really bunch distributing functions, so to facilitate their use we introduce the \textit{operator} function. This function takes a character as a parameter and returns the appropriate function that it denotes. For example, for any \( x \) and \( y \):

\[
\text{operator ' + } x \ y = x + y
\]

### 2.4.5 Restricted domain functions

The function inclusion axiom is very powerful. Its rationale is to provide infinite extensibility to functions (a function is \textit{never} an elementary bunch). Sometimes we know exactly how much we want a function to be extended, and then the unrestricted extension of the function inclusion axiom poses problems. This makes for example the unification of lists with functions (in [16]) have some really undesirable properties.
To remedy the problem, we introduce a new notion, the *restricted domain function*. A restricted domain function $f$ can be extended up to a specified bunch, which is called its universal domain and denoted $\mathcal{U}f$. If it's defined throughout all its domain and its values are elements, then it is also an element.

To denote a restricted domain function we use normal function notation together with the universal domain prefixed and in subscript:

$$\mathcal{U}v_f = U$$

$$\mathcal{D}v_f = \mathcal{D}f\cdot U$$

$$x : U \geq (v_f)x = f x$$

The inclusion axiom for such functions is more restrictive:

$$v_f : wg = U = W \land f_1 : g_1$$

where $f_1$ and $g_1$ are respectively the restrictions of $f$ and $g$ to $U$ and $W$:

$$f_1 = \langle x : \mathcal{D}f\cdot U \rightarrow f x \rangle$$

$$g_1 = \langle x : \mathcal{D}g\cdot W \rightarrow g x \rangle$$
We choose not to relate any non-restricted domain functions with restricted domain functions, in terms of bunch containment. The following is an axiom for a normal (non restricted domain) function $g$:

$$-(\forall f : g \lor g : \forall f \lor g = \forall f)$$

Note that generally a bunch of functions is a function too and we may apply it to values. But here, a bunch of restricted domain functions is not necessarily a restricted domain function and when not, we may not apply it to values. For example let

$$f = 0.1\langle x : 0,1 \rightarrow x \rangle$$

and

$$g = 0.1\langle x : 0,1 \rightarrow 1-x \rangle$$

If $h = f, g$ was a function we would have

$$h = 0.1((0,1) \rightarrow (0,1))$$

with the unfortunate effect that $0.1((0,1) \rightarrow 0)$ is also a function in $h$ and this does not make $f$ and $g$ elements.

However a restricted domain function that returns non-elementary bunches can safely be regarded as a bunch of restricted domain functions.
To construct a restricted domain function, we may also use the following syntax (universal domain $U$ and domain $D$):

$$\langle x : D : U \to E \rangle$$

When $D = U$ we may abbreviate to:

$$\langle x : D \to E \rangle$$

### 2.5 Structure

#### 2.5.1 Sets

To introduce nesting capabilities into bunches we use sets. Given a bunch $A$, the set containing it, $\{A\}$, is an element. Bunch union does not penetrate through set brackets, so any set obeys the fundamental element law for bunches, that is for elements $x$ and $y$:

$$x : y = x = y$$

and therefore

$$\{A\} : \{B\} = \{A\} = \{B\} = A = B$$

The powerset of a set $A$, denoted $2^A$ is the set containing all subsets of $A$. Likewise, the powerbunch of a bunch $A$, denoted $setof A$ is the bunch of all subsets of $\{A\}$, that is
So an interesting law that can be proven is:

\[
A : B = \{ A \} : \text{setof } B
\]

\text{setof} gives us the opportunity to compactly express collections of sets that contain elements within a given bunch, for example \text{setof int} is a bunch that contains all sets that contain only integers.

This is a way to produce the non-union distributing effect that we want some functions to have. We wrap the bunches to pass in sets. The following function tests if a given bunch of numbers is the \text{null} bunch. The bunch is passed wrapped within a set:

\[
isnull = \{ x : \text{setof complex} \rightarrow x = \text{null} \}
\]

### 2.5.2 Lists

Strings also lack structure. \textit{Lists} are a way to incorporate structure to strings. If \( s \) is a string, or a bunch of strings, then \([s]\) is a list. A list is considered one item, that is a string of one bunch. Catenating it to a string does not allow its contents to merge, so for example \( 1; (2; 3) = 1; 2; 3 \) while \( 1; [2; 3] \neq 1; 2; 3 \), thus adding nesting to strings.

The \textit{length} of a list \( L \), (denoted \( \#L \)) is the length of the contained string. If \( x \) is an item and \( s \) and \( t \) are strings then:
Catenation (denoted $L+M$) catenates two lists together:

$$[s]+[t] = [s; t]$$

$head$ and $tail$ are functions that return the first item and the list of all but the first item in a list respectively. These axioms define their behaviour:

$$#[\text{head } L] = 1$$

$$[\text{head } L] + \text{tail } L = L$$

In [16] lists were unified with functions, which made them have some pretty bad properties such as the fact that $[0; 1]: [0]$ because of the function inclusion axiom. Here, this problem is overcome by the use of restricted domain functions. A list $L = [s]$, where $s$ is a string, is a function restricted to the universal domain $0, ..\#L$ (which happens to be null for $[nil]$). Its value at $x: 0, ..\#L$ is the value of the item $x$ within the list (starting to count from 0) that is:
\[ [s] = 0, \ldots, \#s \langle x : 0, \ldots, \#s \rightarrow \text{if } x = 0 \text{ then head } [s] \text{ else tail } [s] (x - 1) \]}

For a list to be treated as a function, it must have been defined in terms of a string of bunches. Not all bunches of strings are single bunches. For example \([1; 0; 0; 1]\) should not be considered as a function, because its behavior would then be the same as \([0, 1); (0, 1)]\).

Such a definition of lists leaves lists of different lengths completely unrelated. For any two strings \(s\) and \(t\):

\[ [s] : [t] = \#s = \#t \land \langle x : 0, \ldots, \#s \rightarrow [s] x : [t] x \rangle \]

Meanwhile bunch union distributes over lists. For strings \(s\) and \(t\):

\[ [s, t] = [s], [t] \]

A list is an element if its content is an elementary string. A string is an element if its items are elements.

### 2.6 Formalizing elementhood

Elementhood is a universal equivalence partition in bunch theory. For any two elements \(a\) and \(b\) we know that:

\[ a \neq b \Rightarrow a \cup b = \text{null} \]
We use the predicate \textit{elem} to denote elementhood. To define which entities in bunch theory are elements, we introduce elementhood axioms. The restricted domain function elementhood axiom asserts:

\[ x : U^D f \wedge \text{elem} x \wedge \text{elem}(\mu f) \geq \text{elem}(f x) \]

We can now define exactly the elements in \textit{xcomplex}:

\textit{elem} 0

\textit{elem} 1

\textit{elem} \textit{true}

\textit{elem} \textit{false}

\[ \wedge \{ x : '+', '-', '\times', '/', '\wedge', '\vee' \rightarrow \text{elem}(y :: \textit{xcomplex} \rightarrow \textit{xcomplex} (\text{operator } z y)) \} \]

\textit{elem}(z :: \textit{xcomplex} \rightarrow - z)

A string is an element if it consists of elements. Alternatively:
\( \text{elem}[s] = \text{elem } s \)

A set is always an element:

\( \text{elem}\{B\} \)

A character is always an element.

2.7 Quantifiers

2.7.1 Generic definition of quantifiers

Let \( f \) be a binary, total, bunch union distributing, commutative and associative operator on \( x\text{complex} \) that is for any \( x, y, z: x\text{complex} \):

\[
\begin{align*}
  f &: x\text{complex} \to x\text{complex} \to x\text{complex} \\
  \land (\text{elem } x \land \text{elem } y &\geq \text{elem } (f x y)) \\
  \land f \text{null } = f x \text{null } = \text{null} \\
  \land f(x, y) = f x , f y \\
  \land f x (y, z) = f x y , f x z \\
  \land f x y = f y x \\
  \land f(f x y) z = f x (f y z)
\end{align*}
\]

Let now \( i_f : x\text{complex} \) be a unique identity element for \( f \), that is an element with the property that for any \( x: x\text{complex} \):

\[
f i_f x = f x i_f = x
\]

that no other element in \( x\text{complex} \) has.

Then it is possible to define a quantifier corresponding to that function, denoted \( Q_f \). We apply this quantifier to functions that return \( x\text{complex} \) elements
according to the following axioms. For any bunch \( A \), bunch union distributing function \( g \) that returns \( \mathbb{C} \) elements, element \( x : \mathcal{D}g \) and \( B, C : \mathcal{D}g \):

\[
Q_f(null \to A) = i_f
\]

\[
Q_f(x \to A) = A
\]

\[
Q_f(y : (B.C) \to g y) = f(Q_f(y : B \to g y))(Q_f(y : C \to g y))
\]

A quantifier is usually denoted by a bigger version of the operator symbol that denotes \( f \). Thus we have the following quantifiers:

- \( \land = Q_\land \) is the maximum quantifier. It is precisely a universal quantification (\( V \) effect) if used with predicates, that is with \textit{bool}-returning functions.
  \[ i_\land = -\infty \]

- \( \lor = Q_\lor \) is the minimum quantifier. It is precisely an existential quantification (\( \exists \) effect) if used with predicates. \( i_\lor = \infty \)

- \( + = Q_+ \) is the summation quantifier (elsewhere denoted \( \Sigma \)). \( i_+ = 0 \)

- \( \times = Q_\times \) is the product quantifier (elsewhere denoted \( \Pi \)). \( i_\times = 1 \)

The universal and existential quantifications have certain very interesting domain-restriction properties. Suppose that \( p \) is a predicate, and \( D, E : \mathcal{D}p \). Then the domain restriction laws assert:
\[ \land (x : D \rightarrow z : E \geq p z) = \land (x : (D\cdot E) \rightarrow p z) \]

\[ \lor (x : D \rightarrow z : E \land p z) = \lor (x : (D\cdot E) \rightarrow p z) \]

which become the one-point laws when \( E = e \) is an element:

\[ \land (x : D \rightarrow z = e \geq p z) = p e \]

\[ \lor (x : D \rightarrow z = e \land p z) = p e \]

### 2.7.2 Set-operation quantifiers

The same way we defined quantifiers for \textit{complex} binary operators, we can define quantifiers for binary set operators \( \cup \) and \( \cap \). If \( f \) returns sets, \( x \) is an element in \( D \) and \( B, C : D \), the same quantifier axioms apply:

\[ Q_{\cup}(null \rightarrow A) = \emptyset = \{null\} \]

\[ Q_f(x \rightarrow A) = A \]

\[ Q_f(y : (B,C) \rightarrow g y) = f(Q_f(y : B \rightarrow g y))(Q_f(y : C \rightarrow g y)) \]

We name:
2.7.3 Bunch-wise quantifiers

Any of the quantifiers can be applied to a bunch instead of a function, using its bunch-wise version. The bunch-wise version of a quantifier \( Q_f \) is denoted \( Q_f^b \) and it is defined for any bunch \( B \) by:

\[
Q_f^b B = Q_f(x : B \rightarrow x)
\]

For example:

\[
\bigwedge^b (2, 3) = 3
\]

2.7.4 The solution quantifier

Let \( g \) be a bunch union distributing predicate:

\[
g(Dg) : bool
\]

The bunch of all elements \( x \) in \( Dg \) such that \( g x \) is denoted \( \§ g \). \( \§ \) is named the solution quantifier. \( \§ \) can be defined in terms of \( \bigwedge \). For any predicate \( g \):

\[
\bigwedge (x : Dg \rightarrow x : \§ g = g x) \land \§ g : Dg
\]
Some useful axioms can be found in [16]. In particular, for any bunch $A$, bunch union distributing predicate $g$, element $x : Dg$ and $B, C : Dg$:

$$\$ (null \to A) \ = \ null$$

$$\$ (y : x \to g \ x) \ = \ \text{if } g \ x \ \text{then } g \ x \ \text{else } null$$

$$\$ (y : (B, C) \to g \ y) \ = \ \$ (y : B \to g \ y), \$ (y : C \to g \ y)$$

If $g$ is a bunch union distributing predicate, then the following laws relate $\$ to $\land$ and $\lor$:

$$\land \ g \ \equiv \ \$ g = Dg$$

$$\lor \ g \ \equiv \ \$ g \neq null$$

Also

$$g(\$ g) = \text{if } \lor \ g \ \text{then } true \ \text{else } null$$

The solution of containment property states that for any bunches $A$ and $B$:

$$\$ (x : A \to x : B) \ = \ A \cdot B$$
elem is a non-bunch union distributing predicate that can be expressed using \$ by the following axiom. For any bunch \( X \):

\[
\$\langle x : X \rightarrow x = X \rangle = X = elem \ X
\]

because within \( \langle \rangle \) the parameter is always an element.

### 2.8 Inversion of functions

In bunch theory, all functions and restriction domain functions that distribute over bunch union are invertible. The inverse of a function \( f \), denoted \( inv \ f \) is the function:

\[
inv \ f = \langle y : f(Df) \rightarrow \$\langle x : Df \rightarrow y : f \ x \rangle \rangle
\]

\[
inv \ uf = \langle y :: f(Df \ U) \rightarrow \$\langle x : Df \ U \rightarrow y : f \ x \rangle \rangle
\]

For example \( inv([1: l]) = \ (l \rightarrow (0, l)) \)

Notice that it is generally not true that \( x = inv \ f \ (f \ x) \). The following laws apply:

\[
inv(inv \ f) = f
\]

\[
D(inv \ f) = f(Df)
\]
Chapter 3

Theory theory

This chapter deals with the central notion of formal theories. First, the concept is presented, which is based mainly on an early paper from Burstall [6], but several unifications are also proposed, to simplify things. An implementation of a theory is defined as a stronger theory. This is an approach taken in [16], with the great advantage that an implementation can be further strengthened to produce a more refined implementation and so on. We will see that this idea is basic to most of the contributions of this thesis.

In an attempt to introduce some kind of modularity within theories, we proceed to define them as functions from names to bunches that satisfy the axioms of the theory. This is a novel approach that achieves a certain encapsulation of axioms, so that one may reason about multiple theories at the same time, without having their axioms leak consequences from one theory to another. This will give us the opportunity in chapter 4 to relate theories to objects and classes. The representation of theories as bunches requires a new representation for implementations, and that is the bunch containment relation. The same way, we
come up with corresponding bunch operations for all theory operations that we want to perform.

3.1 An introduction to formal theories

3.1.1 Defining theories

A formal theory is basically a collection of axioms that specify the behavior of certain objects we are interested in. For example Euclidean geometry specifies the behavior of points and lines in the plane, or number theory specifies the behavior of natural numbers and of some operations on them. One comes up with custom small theories to specify the behavior of programs or modules of a program.

To define a theory we need two sets of things: first some names, and second some axioms about the names. For example we cannot say anything about natural numbers without introducing names such as zero, or succ (for successor), or we cannot define Euclidean geometry without introducing the notion of point.

[6] defines two kinds of names for a theory, sorts and operators. Informally sorts are the collections of things we are dealing with, such as point or natural while operators are constants of a certain sort (0 is a natural constant or successor is a natural -> natural constant). Here's some natural number theory in Burstall's notation (variables are implicitly universally quantified within the axioms):

- Sorts: natural boolean

- Operations
zero :→ natural
succ : natural → natural
iszero : natural → boolean
T :→ boolean
F :→ boolean
not : boolean → boolean

- Variables n : natural

- Axioms

\[ iszero(zero) = T \]
\[ iszero(succ(n)) = F \]
\[ not(T) = F \]
\[ not(F) = T \]

Understandably using bunches permits us to avoid this distinction, for example now both natural and zero are constant bunches. So in this thesis, a theory is totally defined if we are given a set of names and a set of axioms with these names. The information that a certain name n is of type X may be included in the axioms as \( n : X \).

The fact that a certain name denotes an element can be included in the axiom system as \( elem \ n \). It might also prove irrelevant, because we are not really interested whether a particular implementation of an operator is an element or not. We do not make any effort to define “elementhood” in our theory.
Also note that we are only interested in theories that incorporate classic logic inference rules as axioms (that we'll always imply). That spares us the effort of introducing classic logic related axioms such as \( \text{not}(T) = F \) seen above.

Finally, all axioms may be conjoined together into a single axiom. Our version of Burstall's natural number theory can thus be more compact:

Bunch of names: "zero", "natural", "succ", "iszero"

Axiom:

\[
\begin{align*}
\text{zero} : \text{natural} \\
\land \text{succ} : \text{natural} \rightarrow \text{natural} \\
\land \text{iszero} : \text{natural} \rightarrow \text{bool} \\
\land \text{iszero} \text{zero} \\
\land \bigwedge \langle n : \text{natural} \rightarrow \neg \text{iszero} (\text{succ} \ n) \rangle
\end{align*}
\]

Notice that Burstall's notion of theories contains only equations. A theory is considered closed under the reflexive, symmetric and transitive properties of equation, therefore axioms that can be inferred by these properties, although explicitly not mentioned, are considered part of the theory as well. In our version, this restriction is discarded. The properties of \( = \) are included in classical logic as axioms. To make it an equational theory, we just have to include \( = \text{true} \) in all the axioms we conjoin.

A theory is defined in terms of an axiom. Everything that can be inferred by this axiom is called a theorem and is also considered part of the theory. A theorem in our natural number theory is for example \(-\text{iszero} (\text{succ} (\text{succ} (0)))\).
3.1.2 Operations on theories

[6] defines four operations on theories: combination, enrichment, induction and derivation. We revisit all of them and argue that the three first are in reality different ways to say the same one interesting operation, which is conjunction. while derivation may be described by abstracting away details with the $\lor$ quantifier. The punch line is that there are two interesting operations we can perform on theories (other than implementing them) and they are conjunction and abstraction.

Combination and enrichment

The presentation of theory operations in [6] starts with combination which is exactly the logical conjunction of two theories. Let $T$ and $U$ be two theories with name bunches $n_T$ and $n_U$ and axioms $a_T$ and $a_U$ respectively. Then their combination is the theory over $n_T, n_U$ and axiom $a_T \land a_U$.

The enrichment of a theory is the conjunction of some more axioms and/or the inclusion of some more names. This is exactly the same as the combination with an unnamed theory. The resulting theory is stronger than the original.

Here's an enrichment of number theory proposed in [6] (in our notation), that introduces the equality operator ($eq$) and the less-or-equal-to operator ($leq$)

Let $T_0$ be the previous number theory axiom and $n_0$ be the previous bunch of names.

Bunch of names: $n_0$, "eq", "leq"

Axiom:
\[
T_0 \wedge \forall n : \text{natural} \\
\quad \quad \quad \rightarrow \wedge \forall m : \text{natural} \\
\quad \quad \quad \rightarrow \text{leq : natural} \rightarrow \text{natural} \rightarrow \text{bool} \\
\quad \quad \quad \wedge \text{eq : natural} \rightarrow \text{natural} \rightarrow \text{bool} \\
\quad \quad \quad \wedge \text{leq zero n} \\
\quad \quad \quad \wedge \neg \text{leq(succ n) zero} \\
\quad \quad \quad \wedge \text{leq(succ m)(succ n) = leq m n} \\
\quad \quad \quad \wedge \text{eq m n = (leq m n \wedge leq n m)}
\]

Let’s name this theory \( T_1 \).

**Induction**

[6] says that \( T_1 \) is unable to prove \( \forall n : \text{natural} \rightarrow \text{eq n n} \), even though taking any natural \( n \) defined by this theory (building upon \( \text{zero} \) with \( \text{succ} \)) we can prove \( \text{eq n n} \). Burstall proposes induction as a theory operator that will create a new theory in which this expression is a provable theorem. To do that we use induction on all natural objects created in the theory. The induction procedure somewhat “discovers” (in fact: further specifies) that there is no exception to the rule \( \text{eq n n} \) and creates theory \( T_2 \) which contains it as an axiom.

Notice that \( \forall n : \text{natural} \rightarrow \text{eq n n} \) is not inferable in the first theory. We can easily imagine for example a model like that:

\[
\text{natural} = \text{znat} \\
\wedge \text{zero} = 0 \\
\wedge \text{iszero} = \langle n : \text{znat} \rightarrow n = 0 \rangle \\
\wedge \text{succ} = \langle n : \text{nat} \rightarrow n + 1 \rangle|_{\infty \rightarrow \infty} \\
\wedge \text{leq} = \langle n : \text{znat} \rightarrow \langle m : \text{znat} \rightarrow (\text{if } n = m = \infty \text{ then } \text{false} \text{ else } n \leq m) \rangle \rangle \\
\wedge \text{eq} = \langle n : \text{natural} \rightarrow \langle m : \text{natural} \rightarrow \text{leq n m} \wedge \text{leq m n} \rangle \rangle
\]
In this model, the axiom of the theory is true, but there is a point \( n \) in \textit{natural} where \( eq n n \) is not satisfied. This is \( \infty \):

\[
eq \infty \infty = leq \infty \infty \land leq \infty \infty = false
\]

So, what is missing is a further specification that \textit{natural} is a bunch \textit{generated} by the \textit{zero} and \textit{succ} constructors. This is not the axiom

\[
\land \langle n : \text{natural} \rightarrow n = \text{zero} \lor n : \text{succ natural} \rangle
\]

because our model also satisfies it: \( succ \infty = \infty \), and \( \infty : \text{natural} \). What we want is an axiom that asserts that we can \textit{induce} that a property \( P \) holds for all \textit{natural} if it holds for \textit{zero} and we can infer \( P(succ \ n) \) from \( P \ n \). This axiom will give a precise definition of what is allowed within the \textit{natural} bunch and its conjunction will create a stronger theory in which proof by induction is possible:

\[
\land \langle P : (\text{natural} \rightarrow \text{bool}) \rightarrow \land \langle n : \text{natural} \rightarrow P \ n \rangle \land P(\text{"zero"}) \land \land \langle n : \text{natural} \rightarrow P \ n \geq P(succ \ n) \rangle \rangle
\]

If we now conjoin this axiom to the theory we can prove

\[
\land \langle n : \text{natural} \rightarrow eq n n \rangle
\]

specializing the axiom for

\[
P = \langle n : \text{natural} \rightarrow eq n n \rangle
\]
Admittedly, to write this complicated formal expression for something as intuitive as this is rather annoying. The Larch Prover [13] specification language allows for a simpler syntax for induction rules, which would read in this case

```
assert natural generated by zero.succ
```

In any case, induction is a specialized way to strengthen a theory by conjunction of a special form of axiom.

**Derivation**

Suppose that we have our theory $T_1$ as defined and that we are really interested in some of the operators it defines. We may derive a weaker theory $T_3$ that abstracts away all the other details. In [6] this is called derivation. We will demonstrate derivation by abstracting away zero. Abstraction is performed using the $\forall$ quantifier. Here’s $T_3$ theory defined on all names of $T_1$ except for zero:

```
\forall (zero : natural \rightarrow T_1)
```

This theory is strictly weaker than $T_1$ in that it does not specify the behavior of the name “zero”. However the influence of the existence of zero is obvious: we cannot simply say that $T_3$ is a theory that contains none of the axioms containing zero. In $T_3$ for example

```
\land (n : natural \rightarrow \land (m : natural \rightarrow iszero n \geq leq n m))
```
is still a theorem, which wouldn't be the case in a theory without zero.

By using the one-point law for existential quantification it is possible to derive a theory similar to its origin with a certain operator renamed:

\[ \forall (\text{zero} : \text{natural} \rightarrow T_1 \land z = \text{zero}) \]

[6] lists as a derivation the introduction of a new operator in terms of old ones. We feel that this is again a special case of conjunction, so we won't say more on this here.

### 3.2 Implementation as implication

In [6], a theory exists to be interpreted. An interpretation of a theory is the assignment of concrete set values to sorts and concrete operator values to the operators of the theory, such that the axiom of the theory is satisfied. Such a concrete representation is also called an algebra or a model of the theory. We've already seen a model for theory \( T_1 \). An obvious model for theories \( T_1 \) and \( T_2 \) and \( T_3 \) is of course:

\[
\begin{align*}
\text{natural} &= \text{nat} \\
\land \text{zero} &= 0 \\
\land \text{iszero} &= \langle n : \text{nat} \rightarrow n = 0 \rangle \\
\land \text{succ} &= \langle n : \text{nat} \rightarrow n + 1 \rangle \\
\land \text{leq} &= \text{operator}' \leq \\
\land \text{eq} &= \text{operator}' =
\end{align*}
\]

Notice that the model is also good for \( T_3 \) where \( \text{zero} \) is not defined, because its assignment does no harm in satisfying the axiom. An interpretation may be regarded as a conjunction of equations that implies the axiom of the theory.
This definition of interpretation is somewhat restricted and it doesn't help if we want to gradually implement a theory or to further refine an implementation. That's why [16] generalizes the notion of implementation of a theory $T$ to be a stronger theory $T'$ such that $T' \geq T$. An interpretation is now a special kind of theory. This schema permits a theory to be implemented in a gradual fashion. What's more, the implementation is still a theory, which means that it is possible to be further implemented.

Following this approach, we don't use models or algebras in our formalism, but just bare theories. The most general theory is the axiom true implemented by any theory. The most specific theory is the inconsistent theory false, which implements all theories. We are interested in consistent theories, that is theories that are not equal to the inconsistent theory (false is not a theorem).

### 3.3 Basics of theory theory

The formalism introduced so far isn't very modular and does not allow one to reason about many theories and subtheories at one example. In this section we present a formalism that allows us to express multiple theories within bunch theory. The idea is that a theory is really the bunch of all objects that satisfy a certain axiom. These objects are functions from names. We use the solution quantifier to create these functions out of functions from names to bool. These functions express the axioms. Names are elements of the text bunch. Term abstraction is also introduced and a handy new notation is invented, upon which we build all our theory and implementation paradigm.
3.3.1 Taking care of sorts and functions

There are some points to consider before delving into the function notation approach. First there's a problem with bunch union distribution and sorts. To express a function from say "natural" to the sort natural, we better make it non-distributing, because otherwise natural will become an element within the function. This is not a good approach however, because we'll have to give up some important properties of bunch union distribution. So we will use set structure here to avoid this pitfall and that's what is going to be typical treatment of sorts from now on.

Second, the same bunch union distributing effect has a side effect on functions too: equating the functional parameter within the body of a function will always return false (see section 2.4.2). Instead what we'll do is wrap it in set structure too. Revisiting the example of section 2.4.2, this is the proper way to check if the parameter is the id function when applied on naturals:

\[ \text{isid} = (F: \text{setof(nat} \to \text{nat}) \to \neg F = \text{id}) \]

3.3.2 Theory axioms

A theory is a sub-bunch of the bunch

\[ \theta = \text{extallf} \]

which is the bunch of all functions restricted on domain text. To describe a theory, we frequently use the $ quantifier applied to a function $ that belongs
in the bunch

\[ \text{axiom} = s(\theta \rightarrow \text{bool}) \]

and is called an \textit{axiom}. This theory is the bunch of all theories \( s \) for which \( f s \).

For example, theory \( T_0 \) is written:

\[ T_0 = \{ s : \theta \rightarrow \text{true} \wedge s(\text{zero}) \rightarrow (s(\text{natural})) \wedge s(\text{succ}) \rightarrow (s(\text{natural})) \wedge s(\text{iszero}) \rightarrow (s(\text{natural})) \rightarrow \text{bool} \wedge s(\text{iszero}) (s(\text{zero})) \wedge \forall n (s(\text{natural}) \rightarrow \neg s(\text{iszero})(s(s(\text{succ}) n))) \} \]

where of course \( s(\text{natural}) \) is passed as a set.

3.3.3 Implementation as bunch containment

A theory was defined as the bunch of all theories that satisfy a certain axiom.

A sub-bunch of such a theory is then an implementation, because each member of the sub-bunch is also a member of the theory. Let \( f, g : \text{axiom} \) be axiom functions, distributing over bunch union. Then the following law relates implementation of theories as implication and implementation of theories as bunch containment:

\[ \forall f : \forall g \equiv \forall \langle x : \theta \rightarrow f \geq g \rangle \]

Notice the importance of bunch union distribution: all our axiom functions will have that property.
A theory always implements itself:

$\text{theory } f : \text{theory } f$

An intersection of two theories is always an implementation of both, and corresponds to the conjunction of their axioms:

$\text{theory } f \cap \text{theory } g = \text{theory } \{ x : \theta \rightarrow f \land g \land x \}$

A union of two theories is implemented by both and corresponds to the disjunction of their axioms:

$\text{theory } f \cup \text{theory } g = \text{theory } \{ x : \theta \rightarrow f \lor g \lor x \}$

The weakest theory is $\theta$ which corresponds to the axiom $\theta(\theta \rightarrow \text{true})$:

$\text{theory } \theta \rightarrow \text{true} = \theta$

Unsatisfiable axioms create the *inconsistent* theory, the *null* bunch, which is the strongest of all theories:

inconsistent = $\text{theory } \theta \rightarrow \text{false} = \text{null}$

### 3.3.4 Term abstraction

To introduce a layer of abstraction, we give our axioms another theory parameter over which we'll quantify existentially. *Abs axiom* will be the bunch of axioms
that permit abstraction:

\[ \text{absaxiom} = \mathfrak{g}(\theta \rightarrow \text{axiom}) \]

Let \( g : \text{absaxiom} \). Then

\[ G = \mathfrak{s}(\text{pub} : \theta \rightarrow \mathfrak{v}(\text{hid} : \theta \rightarrow g\text{ pub hid})) \]

is a theory for which there are two sets of names and one of them is abstracted away. For a theory \( \text{pub} \) to be in \( F \) there must exist a theory \( \text{hid} \) such that \( g\text{ pub hid} \). The axiom that corresponds to \( g \) will be:

\[ f = \langle s : \theta \rightarrow \mathfrak{v}(g\ s) \rangle \]

### 3.3.5 The \( \Theta \) quantifier

The combination of those two quantifiers \( \mathfrak{s} \mathfrak{v} \) is noteworthy, so we’re giving it a new name \( \Theta \). \( \Theta \) stands for \( \theta\epsilon\omega\pi\alpha \) which is the Greek spelling for “theory”. \( \Theta \) quantifies elements of the absaxiom bunch to return the corresponding theory.

The fundamental law that defines \( \Theta \) is (for \( g : \text{absaxiom} \)):

\[ \Theta g = \mathfrak{s}(s : \theta \rightarrow \mathfrak{v}(g\ s)) \]

Also:

\[ \Theta g = \mathfrak{s}(s : \theta \rightarrow \mathfrak{s}(g\ s) \neq \text{null}) \]
The *create* function acts as a *constructor*:

\[
create = \{ g : absaxiom \rightarrow \{ p : \theta \rightarrow \Theta \{ p : \theta \rightarrow \{ h : \theta \rightarrow g \ p \ h \ \land \ h : T \} \} \} \}
\]

To intuitively understand *create*, suppose we have an *absaxiom* member \( g \).

Then \( \Theta g \) represents the theories \( p \) for which there exists \( h : \theta \) such that \( g \ p \ h \).

Now *create* \( g \ T \) restricts \( h \) to \( T \), giving a stronger theory.

The *create*-law asserts:

\[
create \ g \ T : \Theta g
\]

An other useful law states:

\[
x : \Theta g = \bigvee \{ H : setof \ \theta \rightarrow x = create \ g \ (\sim H) \}
\]

### 3.3.6 The \( \rightarrow \) notation

Let \( t : text \) and \( v \) some value. The notation \( t \rightarrow v \) will be an abbreviation for \( text(t \rightarrow v) \). So \( t \rightarrow v \) is a theory. We also overload the \( | \) operator to deal with domain-restricted functions, and therefore with theories. If \( f \) and \( g \) are functions and \( U \) a bunch:

\[
v f \mid u g = u(f \mid g)
\]

So now if \( T \) and \( U \) are theories, \( T \mid U \) is also a theory.
3.3.7 Theory laws and operations

Let \( f, f_1, f_2 : \text{axiom} \land g, y_1, y_2 : \text{axiom} \)

We have introduced so far theory conjunction, disjunction, containment and the \( \Theta \) operator:

\[
\theta f_1 \cdot \theta f_2 = \theta(s : \theta \rightarrow f_1 \land f_2 s)
\]

\[
\theta f_1 \cdot \theta f_2 = \theta(s : \theta \rightarrow f_1 \lor f_2 s)
\]

\[
\theta f_1 : \theta f_2 = \theta(s : \theta \rightarrow f_1 s \geq f_2 s)
\]

\[
\Theta g = \theta(s : \theta \rightarrow \theta(g s) \neq \text{null})
\]

\[
\Theta g = \theta(s : \theta \rightarrow \lor(g s))
\]

In this section we also introduce some other basic laws and operators. In Appendix B we gather all the laws together and we give their formal proofs.

**Obtaining the axiom out of a theory**

From theory \( F \) we can always get the axiom \( f \) such that \( \theta f = F \):

\[
\text{axiomof } F = \langle s : \theta \rightarrow s : F \rangle
\]
Then of course

$$\$ (\text{axiom of } F) = F$$

and

$$\text{axiom of } \$ f = f$$

Name abstraction

*Name abstraction* of name bunch *hidden* within axiom *f* is denoted by:

$$\text{abstract}(\$ f)\text{hidden} = \Theta \{ \begin{array}{c} p :: \theta \\ \rightarrow \{ h :: \theta \\ \rightarrow f(\{t : \text{hidden} : \text{text} \rightarrow h t\}|p) \\ \} \}$$

The abstraction law states that:

$$\$ f : \text{abstract}(\$ f)\text{hidden}$$

Deep conjunction

*Deep conjunction* is conjunction within the existential quantification and produces a stronger theory than the conjunction of the axioms. Deep conjunction is denoted by $\&$ as an infix operator that is applied to two members of *abs axiom*:

$$g_1 \& g_2 = \Theta \{ p :: \theta \rightarrow \{ h :: \theta \rightarrow g_1 p h \& g_2 p h \} \}$$
Deep conjunction of 3 or more *absaxiom* members works like this:

\[ g_0 \land g_2 \land \ldots \land g_{(n-1)} = \Theta(p : \theta \rightarrow \langle h : \theta \rightarrow \land \langle i : 0 \ldots n \rightarrow g, p h \rangle \rangle) \]

The deep-conjunction law states that:

\[ g_1 \land g_2 : \Theta g_1 \Theta g_2 \]

*create* and \( \land \) are associated by the following law:

\[ create g H = g \land \theta \rightarrow \langle h : \theta \rightarrow h : H \rangle \]

**Transformation**

Let \( T, U : \theta \) and \( t : T \rightarrow U \), bunch union distributing function such that \( tt T = U \). \( t \) is called *transformation* from theory \( T \) to theory \( U \).

Transformations preserve bunch containment, because they are bunch union distributing:

\[ A : B \equiv tt A : tt B \]

Let \( eq \) be an equivalence relation on \( T \), and also bunch union distributing with respect to both its parameters. Then a theory transformation \( tt \) preserves \( eq \) if

\[ \land \langle a : T \rightarrow eq a (inv tt (tt a)) \rangle \]
High level equivalence

Let $g$ be an *absaxiom* member. Then $g$ defines an equivalence relation on $\theta$ by the following formula:

$$eq_g = \langle a : \theta \rightarrow \langle b : \theta \rightarrow create \ g \ a = create \ g \ b \rangle \rangle$$

$eq_g$ is called *high level equivalence of* $g$. The law of deep theory transformation says that if a theory transformation $tt$ preserves $eq_g$, then for all theories $a : Dtt$:

$$createg \ a = create(\langle p : \theta \rightarrow \langle h : \theta \rightarrow g \ p \ (inv \ tt \ h) \rangle \rangle)(tt \ a)$$

The high level equivalence relation reveals the exact information in the hidden theory that is of value to the public theory. Preserving it through transformation, preserves the whole theory. A good example for this is found in section 3.4.6.

Projection

Let $t : text$. The projection of $F$ on $t$ is the theory

$$project \ F \ t = \langle x : t'DF : text \rightarrow F \ x \rangle$$

Always

$$F : project \ F \ t$$
and

\[ F = \text{project } F \text{ text} \]

The mutual projection law asserts that

\[ F : G \geq \text{project } F \ t : \text{project } G \ t \]

**Syntax**

All the boldface operators defined in this section:

- abstract axiom of project

have the same syntax (and therefore the same precedence and associativity) as function application.

### 3.3.8 Syntactic sugar for *absaxiom*

The notation introduced so far is complicated and it would be hard to present advanced examples for the use of this framework without simplifying it. This section presents a notation that we will hence use for *absaxiom* functions to make things easier for the reader to understand. Having a convenient way to write up an *absaxiom* immediately gives us a way to write the corresponding theory, applying it to \( \Theta \).
Brackets

An *abs axiom* is created by using this special syntax (where *sans serif* fonts are used for the various parts of the definition clarified later):

\[
\langle \ -- \\
   \text{pub} \quad \text{public\_names} \\
   \text{hid} \quad \text{hidden\_names} \\
   \text{forall} \quad \text{universally\_quantified\_names} \\
   \text{axiom} \quad \text{axiom} \\
   \ -- \rangle
\]

The axiom is the place where the actual *abs axiom* is described. Within the axiom, the special names *public* and *hidden* represent the public and the hidden theory respectively, so our notation is equivalent to

\[
\langle \text{public} :: \theta \rightarrow \langle \text{hidden} :: \theta \rightarrow \text{definition} \rangle \rangle
\]

Member names

Within the axiom, we refer to a lot of values such as *public*"a" or *hidden*"b" which we'll call *members* of our theories *public* and *hidden*. To omit the tedious notation, we introduce all the names used for each theory in the *public\_names* and the *hidden\_names* section of the definition respectively. For example the use of an identifier \(a\) declared in the *public\_names* part, will automatically mean *public*"a". The names are arranged one after the other with spaces between them.

Formally the inclusion of an identifier \(a\) in the *public\_names* part will mean
and same for the hidden part.

**Hidden assignments**

Sometimes we need to use a complex expression a lot, so we assign it to a hidden value in the hidden part. To do that, instead of a normal name declaration in the hidden names part, we write:

\[
\text{\( \text{name = expression} \)}
\]

Use of this name afterwards will invoke the whole expression. We usually do this to extract bunches out of sets and use them:

\[
\text{\( B \approx S \)}
\]

**Universal quantification**

Our axiom is very likely to be a universal quantification over some variables, which we name in the universally_quantified_names section. Each of them is declared in parentheses like that:

\[
\text{\( \text{variable : bunch} \)}
\]

So for example, if we write
we mean that the axiom is quantified like that:

\[
\land \langle a : A \rightarrow \land \langle b : B \rightarrow \text{axiom} \rangle \rangle
\]

Special names

We've seen the use of public and hidden within the axiom. Another very important name we will use a lot is construct that triggers a recursive call on the absaxiom under definition. If we name g our absaxiom under definition, then construct is an abbreviation of create g. construct h creates the theory which is defined augmented with the deep conjunction hidden = h. The theory defined is construct θ. An abbreviation for constructθ is the special name theory.

3.4 Examples

3.4.1 A theory is one of its sorts: a number theory

We have defined a theory as a bunch of objects that satisfy certain axioms. That permits us to unify a theory with one of its sorts. So for example \( T_0 \) could be defined as the bunch natural, whose members have certain properties. Any operator applied on natural can now be a member of the natural in an object oriented fashion. A constant such as zero is a member of all members of natural that is constant and this can be asserted by the axiom (for all s : natural) \( s"\text{zero}" = \text{natural}"\text{zero}" \). This is how our theory looks like now:
\( \Theta(--
\text{pub}~\text{zero succ iszero} \\
\text{hid} \\
\text{forall} \\
\text{axiom} \quad \text{zero} = \text{theory"zero"} \\
\quad \wedge \text{zero} : \text{theory} \\
\quad \wedge \text{suc}: \text{theory} \\
\quad \wedge \text{zero"iszero"} \\
\quad \wedge \neg \text{suc"iszero"} \\
-- \}

Here's an implementation of it:

\[
\text{natural} \\
= \Theta(--
\text{pub} \quad \text{zero succ iszero} \\
\text{hid} \quad \text{imp} \\
\text{forall} \\
\text{axiom} \quad \text{imp} : \text{nat} \\
\quad \wedge \text{zero} : \text{construct("imp" → 0)} \\
\quad \wedge \text{suc} : \text{construct("imp" → imp + 1)} \\
\quad \wedge \text{iszero} = (\text{imp} = 0) \\
-- \}
\]

This is the obvious implementation, that maps \text{nat} to \text{natural}. The hidden theory contains a hidden \text{nat} variable \text{hidden"imp"} which is the implementation variable. To see how we use \text{construct} look at the example: giving it a theory \text{h} where \text{h"imp"} is defined, it creates the appropriate sub-theory of \text{natural}.

For example, to create the subtheory of \text{natural} where \text{h"imp"} = 0 we call \text{construct("imp" → 0)}. Now it should be clear how all members of our implementation are instantiated. The fact that \text{nat} is the "hidden" implementation is recorded by

\[
\text{natural} = \text{create} \ g \ ("imp" \rightarrow \text{nat})
\]
or given in our abs axiom notation:

\[
\text{theory} = \text{construct}('\text{imp} \rightarrow \text{nat})
\]

Any \( N : \text{natural} \) may be created by its \( n : \text{nat} \) counterpart like that:

\[
N = \text{create } g ('\text{imp} \rightarrow n)
\]

### 3.4.2 A theory as not one of its sorts: a group theory

This is to demonstrate how to use our framework more conventionally. We will constructively build a group theory based upon two other theories, \( \text{Id} \) and \( \text{Monoid} \).

Given a binary operator \( \text{prod} \) on a bunch \( M \) an identity element \( i \) satisfies

\[
\text{prod } i \ x = \text{prod } x \ i = x
\]

for all \( x : M \). So our theory \( \text{Id} \) looks like that:

\[
\text{Id} = \Theta \langle - - \\
\text{pub } \text{prod } M \ i \\
\text{hid } (B = \sim M) \\
\text{forall } (x : B) \\
\text{axiom } \text{prod } x \ i = \text{prod } i \ x = x
\]

For example

\[
('M' \rightarrow \{\text{nat}\})|'|\text{prod} \rightarrow \text{operator }' + |'i' \rightarrow 0) : \text{Id}
\]
The following theory describes a monoid. A monoid is a set $M$ and a binary operation on that set $\text{prod}$ such that certain axioms hold:

\[
\text{Monoid} = \Theta\langle \ldots
\begin{array}{ll}
\text{pub} & \text{prod } M \\
\text{hid} & i \ (B \equiv M) \\
\text{forall} & (a : B) \ (b : B) \ (c : B) \\
\text{axiom} & \text{(project hidden "i" | public) : Id} \\
& \land \text{prod} : B \to B \\
& \land \text{prod } a \ (\text{prod } b \ c) = \text{prod(\text{prod } b \ c)}
\end{array}\ldots \rangle
\]

For example,

\[
(\text{"M" \to \{\text{int}\}|\text{"prod" \to operator '+')} : \text{Monoid}
\]

The hidden theory contains the identity element hidden "i". This remains hidden, because we are not interested in exposing it as part of the monoid. We are only interested in its existence.

A group is a monoid whose elements are always invertible. Based upon our previous definition:

\[
\text{Group} = \text{Monoid} \cdot \Theta\langle \ldots
\begin{array}{ll}
\text{pub} & \text{prod } M \\
\text{hid} & (B \equiv M) \\
\text{forall} & (x : B) \\
\text{axiom} & \langle \begin{array}{l}
\{ \ y : B \\
\to \ (\text{"i" \to \text{prod } x \ y|public}, (\text{"i" \to \text{prod } x \ y|public} \\
\quad : \text{Id}
\end{array} \\
\ldots \rangle
\end{array}\ldots \rangle
\]
3.4.3 Why elementhood doesn’t matter: a set theory

A set theory is already incorporated in bunch theory, but suppose we wanted to supply our axioms to write set theory from scratch. Suppose that we also want to create a Holder super-theory where a Set will be a specific kind of Holder [24]. A Holder is a generic sort of collection with operators \( \text{ins} \) (insert a new member to create a new Holder) and \( \text{mem} \) (check membership in a specified Holder). The theory is more general than the Set theory, in that \( \text{ins} \) is not required to be idempotent. Later we will add this requirement in our Set theory.

Moreover we want the Holder theory to be generated by \( \text{new} \) (an empty holder) and \( \text{ins} \). To keep things simple, we will just assert that any Holder is just \( \text{new} \) or the result of applying \( \text{ins} \) to a Holder. (As we've seen above this is not enough for induction, but it doesn’t matter at this point). Moreover, for simplicity, we only care about inserting elements of \( \text{nat} \):

\[
\land (h : \text{Holder} \rightarrow h : \text{new} \lor h : \text{Holder ins nat})
\]

We also want to delete (\( \text{del} \)) members and to have a count of how many members our holder has (\( \text{measure} \)).

Here's our Holder theory, directly “translated” from [24]:

56
Here's a Set theory:

\[ \text{Set} \]

\[ = \quad \text{Holder} \]

\[ \times \quad \Theta \quad \text{ins mem measure} \]

\[ \text{pub} \]

\[ \text{hid} \]

\[ \text{forall} \]

\[ \text{axiom} \]

\[ \text{new} : \text{Holder} \]

\[ \land \quad \text{new} = \text{ins} \text{nat} \text{ "new"} \]

\[ \land \quad \text{ins} : \text{nat} \rightarrow \text{Holder} \]

\[ \land \quad \text{mem} : \text{nat} \rightarrow \text{bool} \]

\[ \land \quad \text{del} : \text{nat} \rightarrow \text{Holder} \]

\[ \land \quad \text{measure} : \text{nat} \]

\[ \land \quad \text{ins} \text{d0} \text{ "ins"d1 = ins d1 \text{ "ins"d0} } \]

\[ \land \quad \text{new} \text{"mem"d} \]

\[ \land \quad \text{ins} \text{d0} \text{ "mem"d1 = (d0 = d1 \lor \text{mem d1} )} \]

\[ \land \quad \text{mem d} \geq (\text{del d \text{ "ins" d = public} )} \]

\[ \land \quad \text{mem d} \geq (\text{measure > del d \text{ "measure} )} \]

\[ \land \quad \text{public : new, Holder ins nat} \]

\[ \lor \]

Notice that we didn’t specify that \text{Holder} “new” has to be an element, while we could do that conjoining \text{new} = \text{Holder} “new”. In fact if we did, we wouldn’t be able to create our Set subtheory. If \text{Holder} “new” is an element, then either
Holder\textquotedblright new\textquotedblright : Set or Holder\textquotedblright new\textquotedblright Set = null. So either the empty holder is a set or there is no empty set. Both versions are highly undesirable, because we want to have empty sets and multisets, and we want to retain the information in our formalism that multisets and sets come from the same basic notion of Holder.

It turns out that bunches give us the way out: Holder\textquotedblright new\textquotedblright may be safely regarded as an element within the Holder theory but when sub-theories are created this abstraction no longer applies and Holder\textquotedblright new\textquotedblright is treated as a non-elementary bunch. As long as we want to extend our theories we don't specify elementhood. The concept of high-level elementhood elaborated later is a way to view many possible layers of elementhood.

What happens now is that Holder is the biggest theory in this example. It contains both Holder\textquotedblright new\textquotedblright and Holder\textquotedblright ins\textquotedblright nat as subtheories, representing respectively the empty holders and the non-empty holders. When we define Set we create a subtheory which is completely orthogonal to the new vs. ins nat theory distinction:

\[
\text{Set} \text{\textquotedblright new\textquotedblright} = \text{Set} \cdot \text{Holder} \text{\textquotedblright new\textquotedblright}
\]

\[
\text{Set} \text{\textquotedblright ins\textquotedblright nat} = \text{Set} \cdot \text{Holder} \text{\textquotedblright ins\textquotedblright nat}
\]

Our new type definitions do not contradict the ones in Holder. For example,

\[
\text{Set} \text{\textquotedblright new\textquotedblright} : \text{Set} \land \text{Set} \text{\textquotedblright new\textquotedblright} : \text{Holder}
= \text{Set} \text{\textquotedblright new\textquotedblright} : \text{Set}
\]
(because $Set : Holder$).

3.4.4 Combining theories: a sorting function theory

We will now gather together various theories to create a specification for a polymorphic sorting function that creates a sorted permutation of a list of elements in a bunch $B$ according to some ordering relation on the bunch.

The ordering relation is described by the following theory:

\[
\text{Ord} = \text{\_\_}
\begin{align*}
\text{pub} & \quad \text{set } r \\
\text{hid} & \quad (B =\sim set) \\
\text{forall} & \quad (x : B) (y : B) (z : B) \\
\text{axiom} & \quad r : B \rightarrow B \rightarrow \text{bool} \\
& \quad (r \ x \ x) \\
& \quad (r \ x \ y \land r \ y \ z) \geq r \ x \ z \\
& \quad (r \ x \ y \land r \ y \ z) = (x = y)
\end{align*}
\]

For example, let

\[NatOrd = \text{"set"} \rightarrow \{\text{nat}\} \text{"r"} \rightarrow \text{operator } \leq\]

then

\[NatOrd : \text{Ord}\]

The following theory uses $\text{Ord}$ and decides whether a list $L$ is sorted according to the ordering relation given:
Sorted
= Ord
  Θ(---
pub L set r
hid (B =~ set)
forall (i : 0,..,#L-1)
axiom L : [*B] \land r(L i)(L(i + 1))
  --)

For example

("set" → {nat}|"r" → operator ' ≤ |"L" → [0:1]) : Sorted

A predicate sorted asserts a list is sorted according to some ordering:

sorted = \langle o : Ord \rightarrow \langle L : [* o "set"] \rightarrow ("L" → L|o) : Sorted \rangle \rangle

For example

sorted NatOrd [0;1]

The following theory decides if list M : [*B] is a permutation of list L : [*B]:


We'll define `permutation`, a predicate that decides whether one list is a permutation of another. We'll give it a theory \( t \) as a first parameter and \( t \text{"set"} \) will be the set containing bunch \( B \):

\[
\text{permutation} = \langle \begin{array}{l}
\quad t : \emptyset \\
\quad \rightarrow \langle \begin{array}{l}
\quad L : [\ast \text{"set"}] \\
\quad \rightarrow \langle \begin{array}{l}
\quad M : [\ast \text{"set"}] \\
\quad \rightarrow (\text{"L"} \rightarrow L) (\text{"M"} \rightarrow M | t) : \text{Permutation}
\end{array}\rangle
\end{array}\rangle
\end{array}\rangle
\]

For example:

\[
\text{permutation NatOrder [0; 1] [1; 0]}
\]

So a theory that decides if a list \( L \) is a sorted version of a list \( M \) is:
A predicate that does the same is

\[ s_v = \langle \ o : \text{Ord} \\
\quad \rightarrow \ \langle \ L : [\ast \sim o\text{"set"}] \\
\quad \quad \rightarrow \ \langle \ M : [\ast \sim o\text{"set"}] \\
\quad \quad \quad \rightarrow \ \text{sorted} \ o \ L \land \text{permutation} \ o \ L \ M \\
\quad \rangle \rangle \]

It is also

\[ s : SV = s_v \ s \ (s\text{"L"})(s\text{'M'}) \]

To describe sorting functions here's our final theory:

\[
\text{SortingFunction} \\
= \text{Ord} \\
\cdot \ \Theta(\ --) \\
\quad \text{pub} \ f \ \text{set} \\
\quad \text{hid} \ (B \equiv \sim \text{set}) \\
\quad \forall \ M : [\ast B] \\
\quad \text{axiom} \ f : [\ast B] \rightarrow [\ast B] \land \text{sv public} \ (f(M)) \ M \\
\quad \quad \quad \quad (- - )
\]

And our predicate

\[
\text{sortingfunction} = \langle \ o : \text{Ord} \\
\quad \rightarrow \ \langle \ f : ([\ast \sim o\text{'set"}] \rightarrow [\ast \sim o\text{'set"}]) \\
\quad \quad \rightarrow \ (\text{"f"} \rightarrow f[o] : \text{SortingFunction} \\
\quad \rangle \rangle 
\]
3.4.5 Gradual implementation: a mergesort implementation

Let

\[ \text{Merge} = \text{Ord} \]
\[ \Theta(-,-) \]
\[ \text{pub} \ mrg \ set \]
\[ \text{hid} \ (B \sim \text{set}) \]
\[ \text{forall} \ (L : [*B]) (M : [*B]) \]
\[ \text{axiom} \ mrg : [*B] \rightarrow [*B] \rightarrow [*B] \]
\[ \land \ (\text{sorted public } L \land \text{sorted public } M) \]
\[ \geq \ (\text{sv public } (mrg L M)(L + M)) \]
\[ \land \ (\text{sorted public } L \land \text{sorted public } M) \]
\[ \geq \ (\text{sv public } (mrg L M)(L + M)) \]

A function \( mrg \) is a merging function, if given two sorted lists \( L \) and \( M \) it returns the sorted version of \( L + M \). A predicate that asserts that a function is a merging function is:

\[ \text{merge} = \langle \text{o : Ord} \]
\[ \rightarrow \langle \ mrg : (\lceil \sim \text{o"set"} \rceil \rightarrow (\lceil \sim \text{o"set"} \rceil \rightarrow (\lceil \sim \text{o"set"} \rceil)) \]
\[ \rightarrow \ (\text{"mrg"} \rightarrow mrg|o) : \text{merge} \]
\[ \rangle \]

A merging function can be used within \text{MergeSort} (note that \text{ceil} returns the smallest integer that is greater than or equal to its real parameter):
What we see here is an interesting abstraction: \textit{MergeSort} is somewhat more general than the standard merge-sort procedure in that it is not necessarily recursive. Any sorting function \textit{somesort} can be used in place of the nested sorting call. The standard merge-sort algorithm is a specific instance of \textit{MergeSort} that uses merge-sort again as its sorting function:

\[
\text{SpecificMergeSort} = g \\
\text{forall axiom } \text{somesort} = f
\]
Of course, to demonstrate correctness, we will also have to show that the function eventually terminates. We won't get into details of that matter here.

To demonstrate the correctness of our merge-sorting method, we need to prove

$$\text{MergeSort} : \text{SortingFunction}$$

Here's a formal proof:

$$\begin{align*}
\text{MergeSort} : \text{SortingFunction} \\
\{ \text{Bunch containment} \} \\
= \land (x : \emptyset \rightarrow x : \text{MergeSort} \supseteq x : \text{SortingFunction}) \\
\{ \text{Domain property of universal quantification} \} \\
= \land (x : \text{MergeSort} \rightarrow x : \text{SortingFunction})
\end{align*}$$

Proving this will prove the original expression, because of transitivity of equivalence. To prove such universal quantification, by the law of generalization it suffices to assume $$x : \text{MergeSort}$$ and to prove $$x : \text{SortingFunction}$$.
\[ x : \text{MergeSort} \]

\[
\begin{align*}
\{ \text{Definition} \} \\
\forall h : \theta \\
\to \text{let } ss = h \text{"somesort"} \\
\quad f = x \text{"f"} \\
\quad LB = [x \sim x \text{"set"}] \\
\quad mrg = h \text{"mrg"} \\
\quad A = (\text{"f"} \rightarrow ss[x] : \text{SortingFunction}) \\
\quad B = (\text{"mrg"} \rightarrow mrg[x] : \text{Merge}) \\
\quad mid = \langle L : LB \rightarrow \text{ceil}(\#L/2) \rangle
\end{align*}
\]

But notice that (with the names defined within the \text{let in} construct):

\[
A \land B \land L : LB \land M : LB \\
\{ \text{Definition of } A \text{ and } \text{SortingFunction and specialization} \} \\
\forall \text{sv} x (ss L) L \land \text{sv} x (ss M) M \land B \\
\{ \text{Definition of } \text{sv}, \text{then } B \text{ and definition of } \text{Merge} \} \\
\{ \text{Specialization and definition of } \text{sv} \} \\
\forall \text{sorted } x (\text{mrg}(ss L)(ss M)) \\
\land \text{permutation } x (\text{mrg}(ss L)(ss M))(ss L^*)(ss M) \\
\{ \text{Permutation theorems} \} \\
\forall \text{sorted } x (\text{mrg}(ss L)(ss M)) \\
\land \text{permutation } x (\text{mrg}(ss L)(ss M))(L^+M) \\
\{ \text{Definition of } \text{sv} \} \\
= \text{sv} x (\text{mrg}(ss L)(ss M))(L^+M)
\]

By transitivity of \( \geq \) generalizing and then applying the generalization to the definition of \( f \) that gives:
\[ x : \text{MergeSort} \]
\[ f : LB \to LB \]
\[ \wedge \left( L : LB \to (M : LB \to sv x (mrg(L)(ss M))(L^+ M)) \right) \]
\[ \wedge \left( M : LB \to \{ \text{if } \# M = 0 \text{ then } f M = [\text{nil}] \text{ else } sv x (f M) M \} \right) \]
\[ f : LB \to LB \wedge \left( M : LB \to sv x (f M) M \right) \]

Also notice that

\[ x : \text{MergeSort} \]
\[ \{ \text{Definition of } \text{MergeSort} \text{ and specialization} \} \]
\[ ("mrg" \to mrg[x]) : \text{Megre} \]
\[ \{ \text{Bunch containment laws} \} \]
\[ ("mrg" \to mrg[x]) : \text{Ord} \]
\[ \{ \text{By mutual projection} \} \]
\[ \text{project}("mrg" \to mrg[x])("set", "r") : \text{project Ord("set", "r")} \]
\[ \{ \text{By projection law, bunch containment laws} \} \]
\[ \{ \text{and the fact that } \text{project Ord("set", "r")} = \text{Ord} \} \]
\[ x : \text{Ord} \]

Generalizing and taking into account the \text{SortingFunction} definition, it is proven:

\[ \forall (x : \text{MergeSort} \to x : \text{SortingFunction}) \]
\[ = \text{MergeSort} : \text{SortingFunction} \]

QED.

In the above formal proof, theorems from the \text{Permutation} theory were utilized without being formally proven. Specifically we used the fact that permutation is transitive, and that for all \( x : \text{Ord} \) and lists \( L \ L' M M' \) within \([* \sim x "set"]:\)

67
permutation $x \ L \ L' \wedge$ permutation $x \ M \ M'$

$\geq$

permutation $x \ (L + L') \ (M + M')$

These theorems can be easily proven using induction on the size of one of the participating lists. We won't do this here.

3.4.6 Transforming theories: a point theory

There are two standard ways to describe points in plane. One theory could utilize Cartesian coordinates:

$$Point_1 = \{s :: \theta \rightarrow s^{"x"}, s^{"y"} : real\}$$

Another theory could utilize polar coordinates:

$$Point_2 = \{s :: \theta \rightarrow s^{"r"} : real \wedge s^{"r"} \geq 0 \\
\wedge (s^{"r"} > 0 \geq s^{"th"} : real \wedge 0 \leq s^{"th"} \leq 2 \times \pi)\}$$

(of course here $\pi = 3.14159...$)

There is a known theory transformation $tt$ that generally takes a $Point_2$ subtheory and transforms it into a $Point_1$ subtheory, in such a way that both subtheories "describe the same point":

$$tt = \langle s : Point_2 \\
\rightarrow ("x" \rightarrow s^{"r"} \times \cos(s^{"th"}) || "y" \rightarrow s^{"r"} \times \sin(s^{"th"}))\rangle$$

68
(cos and \( \sin \) are obviously the cosine and sine functions respectively. Later we will also use the arc-of-tangent trigonometry function too, which we denote \( \text{arctan} \) and the tangent function \( \tan \)).

What do we formally mean by “describing the same point”? In reality those two “data structures” have a semantic mapping to the abstract notion of planar point. There is an equivalence relation among points (what we mean by “same point”) that must be also mapped. The equivalence relation for this semantics for \( \text{Point}_2 \) is for example:

\[
eq_1 = \{ \begin{array}{l}
\forall a : \text{Point}_2 \\
\quad \exists b : \text{Point}_2 \\
\quad \text{project } a \ (\text{"r"}, \text{"th"}) = \text{project } b \ (\text{"r"}, \text{"th"}) \\
\quad \lor \quad a\text{"r"} = b\text{"r"} = 0 \\
\end{array} \}
\]

(which is weaker than theory equivalence). \( \text{tt} \) can be shown to preserve this equivalence:

\[
eq_1 \text{ a } (\text{inv } \text{tt} \ (\text{tt } a))
\]

To prove this, let \( r = a\text{"r"} \) and \( th = a\text{"th"} \) and \( x = \text{tt } a\text{"x"} \) and \( y = \text{tt } a\text{"y"} \).

Then it is known by elementary mathematics:
\[ \text{inv \ tt \ (tt \ a) "r"} = \sqrt{x^2 + y^2} = r \]
\[ \land (r > 0) \]
\[ \text{inv \ tt \ (tt \ a) "th"} = \tan \]
\[ (2 \times \text{nat} \times \pi + (\text{if } x > 0 \text{ then } \arctan(y/x) \text{ else } \text{if } x < 0 \text{ then } \arctan(\pi - y/x) \text{ else } 0)) \]
\[ = \text{th} \]

which proves that the equivalence is preserved. (Notation: superscript denotes exponentiation and \(\sqrt{}\) is the real square root of a positive number).

Now suppose that we have a theory that uses a \(\text{Point}_2\) internally:

\[ g_2 = \langle - - \rangle \]
\[ \langle \text{pub } getR \rangle \]
\[ \langle \text{hid } r \rangle \]
\[ \langle \text{forall } \text{axiom } \text{hidden} : \text{Point}_2 \land getR = r \rangle \]

As the deep transformation law says, we can change this internal representation without affecting the theory or the interface. Indeed, let
\[ g_1 = \langle \text{pub} \text{ getR} \text{ hid} \forall \text{ axiom } inv \ tt \ hidden : \text{Point}_2 \land \text{getR} = inv \ tt \ hidden \ "r" \rangle \]

\[ = \langle \text{pub} \text{ getR} \text{ hid} \forall \text{ axiom } \text{hidden} : \text{Point}_1 \land \text{getR} = \sqrt{x^2 + y^2} \rangle \]

For any \( a : \text{Point}_2 \), it is

\[ \text{create } g_2 \ a = \text{create } g_1 \ (tt \ a) \]

where \( g_1 \) is as \( g_2 \) with an extra application of \( inv \ tt \) every time the \text{hidden} parameter is invoked. For example, let

\[ \text{Circle}_1 = \text{Point}_1 \{ s :: \theta \rightarrow (r"x")^2 + (r"y")^2 = 1 \} \]

\[ \text{Circle}_2 = \text{Point}_2 ("r" \rightarrow 1) \]

It is

\[ \text{create } g_1 \text{ Circle}_1 = \text{create } g_2 \text{ Circle}_2 \]

The concept of high equivalence here plays an important role: For \( \Theta g_2 \) it would be enough to retain partial information for the point, since only the radius
is required. Therefore, the high equivalence in this case is even weaker and a transformation that only preserved the information for \( r \) (like projection on \( r \)) could also be used.

### 3.5 Advanced theory theory

#### 3.5.1 High level elementhood

Suppose that we have a theory \( T : \emptyset \). \( T \) defines axiomatically how certain entities behave, and we give names to these entities \( T"a" \) and \( T"b" \) etc. The conventional approach separates names to sorts and operators. We don’t do that, instead we emphasize that we don’t want to specify elementhood, on the basis that we might want to further extend the theory.

However, sometimes it’s useful for a theory to specify that certain things can be treated as elements within the theory. We can’t always do that, in fact that seems to be the faulty assumption we mentioned before in the \( Holder \) example (\( Holder \) is not term-generated, or if it were, its generators cannot count as elements). To do that we must add further axioms, and the inability to do so reveals a design fault or a misunderstanding of the modelled problem.

Here we investigate those further axioms that need to be added. As we’ve seen, elementhood is an equivalence partition. To make non-elements behave like high level elements, we must make sure that this partitioning is feasible. That is we must make sure that for any two \( a \) and \( b \) candidates for high-level elementhood (denoted \( hle \)):

\[
\text{hle } a \land \text{hle } b \geq (a \neq b \geq a \cdot b = \text{null})
\]
Furthermore, we must specify that any of the functions $f$ that are high-level-elementary must preserve high-level elementhood:

$$\text{hle } f \land \text{hle } x \land x : Df \supseteq \text{hle}(f \ x)$$

This is exactly where the design of $Holder$ fails, because $ins$ doesn't have that property:

$Holder^{new} "ins" 2 "ins" 2$

contains some but not all of the members of

$Holder^{new} "ins" 2$

It contains the sets but not the multisets. (We are content here to assume that high-level elementhood for numbers coincides with elementhood).

To create a high-level equivalence partitioning $HLEp$, we first axiomatize:

$$\{x\}, \{y\} : HLEp \supseteq x = y \lor x \ast y = \text{null}$$

$$\{f\}, \{x\} : HLEp \land \exists x : Df \supseteq \{f \ x\} : HLEp$$

$$\{x\}, \{y\} : HLEp \supseteq \{x; y\}, \{[x]\} : HLEp$$
and then we give the base cases:

\{T"a"}, \{T"b"\} : HLE_p

etc.

Usually,

\(\bigwedge (t : \text{char}, \text{xcomplex} \rightarrow \{t\} : HLE_p)\)

\(\bigwedge (x : +, -, \times, /, \&, \forall \rightarrow \{\text{operator } x\} : HLE_p)\)

\(\{ (x : \text{xcomplex} \rightarrow -x) \} : HLE_p\)

This can be formed into a formal theory:

\[
HLE_1 \equiv \Theta(- - \\
\text{pub } HLE_P \\
\text{hid } (HLE_P =\sim HLE_P) \\
\text{forall } (X : HLE_P) (Y : HLE_P) \\
\text{axiom } let \ x =\sim X \cdot y =\sim Y \\
\text{in } (x = y \lor x \cdot y = \text{null}) \\
\quad \land (x : T y \supset \{y x\} : HLE_P) \\
\quad \land (z ; y, [[z]] : HLE_P)
\]

And the "usual" case we mentioned before:

\[
HLE_2 = HLE_1 \\
\begin{array}{l}
\xi(s :: \theta \\
\rightarrow \bigwedge (t : \text{char}, \text{xcomplex}, \text{simpleop} \rightarrow \{t\} \in s^{"HLE_P"})
\end{array}
\]
where

\[
\text{simpleop} = \text{operator simpleopchar, } (x : \text{xcomplex} \rightarrow -x) \\
\]

and

\[
\text{simpleopchar} = ' + ', ' - ', ' \times ', '/ ', ' \wedge ', ' \vee ' \\
\]

So suppose we have a number theory, whose members zero and succ we want to consider elements:

\[
\begin{align*}
\text{Number} \\
= & \quad \Theta(\quad - - \\
& \quad \text{pub} \quad \text{zero succ} \\
& \quad \text{hid} \quad H \\
& \quad \text{forall} \\
& \quad \text{axiom} \quad \text{zero} = \text{Number }\text{"zero"} \\
& \quad \quad \quad \& \quad \sim \text{zero} : \text{Number} \\
& \quad \quad \quad \& \quad \text{succ} : \text{Number} \\
& \quad \quad \quad \& \quad A \\
& \quad \quad \quad \& \quad H : H\text{LE}_2 \\
& \quad \quad \quad \& \quad \text{zero }\in H \text{"HLEP"} \\
& \quad - - \\
\end{align*}
\]

(where \(A\) denotes some other axioms).

Unless it's in \(A\), this theory doesn't say that for any \(n : \text{Number}\), \(n \neq n\text{"succ"}\). But it does say that if we get two \(\text{Number}\) elements starting by zero and applying succ several times, then these members are either equal or completely disjoint (they have element structure). So, for example it is provable that \(\text{Number }\text{"zero" }\in H \text{"HLEP"}\) and that for any \(n : \text{Number}\),
3.5.2 Elevating

Given an $HLE_1$ member, $H$, we can have a function $\text{elevate}$ that will take any element $e$ and return the exact partition within $H^{"HLEP"}$ where $e$ is contained. In fact, we will create a new theory $HLE$ (which will be our final high-level- elementhood theory), containing $HLE_2$ and the $\text{elevate}$ function. $\text{elevate}$ should be able to classify all elements that are within any of the sets in $HLE_p$ (where $HLE_p \equiv \sim HLEP$), therefore its domain will be the union of those sets, formally denoted $\bigcup^b HLE_p$.

\[
HLE_3 = HLE_1 \Theta(\sim HLEP) pub \text{Elevate}\ HLEP (\text{elevate} \equiv \sim \text{Elevate}) (HLE_p \equiv \sim HLEP)\]
\[
hid \text{forall} \text{axiom} \text{elevate} = \langle z : \bigcup^b HLE_p \rightarrow \sim \exists (S : HLE_p \rightarrow z \in S) \rangle
\]

\[
HLE = HLE_3 \cdot HLE_2
\]

Let $H : HLE$ and $el \equiv \sim H^{"Elevate"}$ and $x, y : \bigcup^b (\sim H^{"HLEP"})$. Then the following laws hold for elevation:

\[
el(el x) = el x
\]
el \( z : D(\text{el} \ y) \trianglerighteq \text{el} \ y \ (\text{el} \ z) = \text{el}(y \ z) \)

\[ \text{el} \ x ; \text{el} \ y = \text{el}(x ; y) \]

\[ [\text{el} \ x] = \text{el}[x] \]

All high-level elementhood laws are listed and formally proved in Appendix C.

### 3.5.3 Accessing various levels of elementhood

Notice that if we have a partition \( H : HLE \) we can further break it into smaller sets, and derive a new more detailed partition compatible with the original, say \( G : HLE \). The fact that \( G \) is compatible but more detailed than \( H \) is formally captured by elevation inclusion:

\[ \sim (G \ "\text{Elevate}\") \sim (H \ "\text{Elevate}\") \]

So supposing that we have a theory \( C \) and a certain high level elementhood \( H \) that applies to it, we can expose the \textit{elevate} function, so that

\[ C \text{"elevate"} = \sim (H \ "\text{Elevate}\") \]

Now the high level equality is stored within the theory. Whenever we have \( c : C \) we can elevate \( c \) to the partition representing \( C \) by:

77
If we later make a subtheory $D : C$ we might also choose to expose the `elevate` function the same way, but now offering a more detailed partition. Because $D : C$, it is also assured that $D^{\text{elevate}} : C^{\text{elevate}}$, that is the two equivalence partitions are compatible. Now $c : D$ can be elevated two ways:

$C^{\text{elevate}}c$

$D^{\text{elevate}}c$

to be compared with other $C$ and $D$ theories.

Note that not all subtheories of $C$ expose a valid `elevate` function, due to function inclusion. For example $c^{\text{elevate}}$ is not necessarily an elevation function. The only thing we know is that it is a subfunction of an elevating function. What we do, is that we pick certain theories among the theory hierarchy that do expose a valid elevating function, we assert the fact and we use those theories only. This has to do with the concept of metatheories, presented below.

### 3.5.4 Metatheories

Sometimes we want to reason about a theory alone as a whole, without saying things about its subtheories. The examples we presented so far cannot achieve that because of function inclusion. So when we wrote a theory up until now, we
were obliged to mention properties that would be valid all over its subtheories, which would be clearly undesirable sometimes. An example we just saw was the elevating function exposure: there is no way that all subtheories of \( C \) expose a valid elevation function, so we would like to say about \( C \) alone that it exposes a valid elevation function. The following example means "all subtheories of \( C \) have the property" so it doesn't work:

\[
C = \exists \{s :: \theta \rightarrow \forall \langle H : HLE \rightarrow~ (H"Elevate") = s"elevate" \rangle
\]

To reason about a theory "alone as a whole" we utilize *metatheories*. A metatheory is a theory that axiomatizes the behavior of another theory. The theory in question is wrapped in set structure to prevent function inclusion from accessing its sub-theories. A metatheory that would describe theory \( C \) follows. Member *Class* of the metatheory is going to be the theory described:

\[
\text{Meta} = \Theta\{ \text{---} \\
\text{pub} \text{ Class} \\
\text{hid} \ H \ (c =~ \text{Class}) \\
\text{forall} \text{ axiom} \ H : HLE \\
\vee \ c"elevate" =~ (H"Elevate") \\
\vee \text{ other axioms} \\
\text{---} \}
\]

To create \( C \) now we use *MetaC*:

\[
g_1 = \text{some abs axiom for } C
\]
If \( g_1 \) cannot satisfy \( Meta \), then both \( MetaC \) and \( C \) will be reduced to inconsistent.

### 3.6 Related work: \( Z \) and \( B \)

Our "theory theory" is basically an effort to create a specification language in which structuring, encapsulating and reasoning about many theories is possible. This is also the goal of specification languages such as \( Z \) [30] and \( B \) [1], which are now used in software engineering industry with success. These languages are very similar in their approach of the problem.

Compared with \( Z \) or \( B \), our framework seems to have an important advantage, which is the unification of many concepts into the absolute necessary mathematical formalism. Indeed, while in \( Z \) and in \( B \), theories (named schemas) consist of various parts such as invariants, typing of members etc. and themselves are not considered first-class citizens (e.g. passed as parameters to functions), in our framework there is a considerable economy in mathematical entities: a theory is just a special function (therefore first-class) and everything we can say about the theory (invariants, typing etc.) is just included into the axiom (which is also a special function). Our framework boils down to bunches and functions to express everything there is to express about specifications.
For another example, Z and B are particularly strict in their handling of specifications and refinement. They have specific layers at which different languages apply (specification and programming language being a different thing). There is a restriction that a certain refinement called *implementation* (that is expression in a deterministic programming language) is done only once, and further refinement is prohibited. Our framework is much more unified. One language fits all, and there is no distinction between implementation and other kinds of refinement. Therefore something implemented may still be refined, as we’ll see in chapter 5, where a programming language based on theory theory is presented.

Also, “theory theory” defines high-level elementhood in various levels, offering a recursive way to distinguish what is an element and what is a collection of elements. This turns out to be useful in modelling problems with theory hierarchies, where what could be considered an element in a parent theory, must break into further elements in a sub-theory. We believe that high-level elementhood theory is an important contribution of our framework to the effort of structuring specifications.

Of course, it is not feasible to compare in terms of usability our “theory theory”, which has never been used in industrial practice before, with successful methods such as Z and B. The point of this section is to argue about the contribution of our framework in relation to these methods. We believe based on our discussion above, that there are crucial points where “theory theory” is
promising to bring improvements to the standard Z and B methodologies.
Chapter 4

The object oriented paradigm

In this chapter an informal overview of the object oriented programming (OOP) paradigm is attempted that introduces the key concepts of its philosophy and its relation to other paradigms, especially functional programming. We comment on strong and weak points of OOP, and also study some of the efforts to create formal semantics for it, which give us additional insight on the theoretical problems that come with it. After that, we develop our own semantics for object orientation based upon formal theories. Our main idea is the unification of classes, objects and formal theories and the introduction of specifications into objects and this is further extended to encompass all important aspects of OOP. It also offers the opportunity for many useful generalizations of the OOP ideas and we argue that many of the problems pinpointed in our informal analysis are adequately treated by these generalizations.
4.1 Overview of object orientation

Object oriented programming has been a great success in the software development industry especially since 1990. It has been empirically proven to be a superior way to design a software system, providing a more natural way of modularization than the traditional structural paradigm. It is also argued that this modularity facilitates software maintenance, code reuse and debugging. OOP is based on certain important ideas that build on top of each other: objects and encapsulation, classes, inheritance and finally a certain type of polymorphism called inclusion polymorphism. Different languages added different new ideas to the model.

The most important languages with OOP features used today are C++ [32], Ada 95 [33] and Java [15], although they are not considered “pure” object oriented languages, based largely on their imperative part. (SmallTalk [14] is considered to be a very “pure” object oriented language in that sense). Almost all recent languages seem to follow the trend including some OOP characteristics. Most of the OOP-enriched languages are based on the imperative programming paradigm, but object orientation is really orthogonal to the imperative / functional aspects of a language, which means that functional languages can also be enriched by OOP. Such languages exist either as stand-alone [4] or as libraries for more complex imperative-style languages [26].

In the following, we take a closer look at OOP characteristics with some references to more general programming languages issues (e.g. polymorphism) and the functional approach to OOP. We also pinpoint a number of weaknesses
and unclear points of the OOP paradigm as it is understood today, which are going to be referred to later on.

4.1.1 Objects and encapsulation

An object is the main unit of modularity within OOP as opposed to a subroutine, which is the unit of modularity in structured programming. The difference is one of perspective: in structured programming we have data and subroutines that operate on data, while in OOP we have objects that contain both data and the available operations on them. In structured programming, if we want to perform an operation, we run a certain subroutine on certain data. In OOP we send a message to an appropriate object that is thought responsible of modifying its state according to it.

In structured programming we may have data waiting to be used by procedure \( p \), but nothing prevents an erroneous call of procedure \( q \) on them. The data might mean a completely different thing for \( q \) in the raw data level (a problem that is avoided but not totally eliminated by using sophisticated type systems in almost all languages) and that might have disastrous effects.

In OOP, what we do instead is send the request to an object that has both the data and the legal operations on it. It is much harder now to perform an illegal operation, since the object is our only gate to the data. This is the concept of encapsulation. In OOP, programmers are encouraged to see objects as black boxes, that exercise a certain behavior. Their inner implementation and raw state data are not accessible from the outside of the object. Only indirectly, through legal operations may an outsider influence the state of the
object. Encapsulation is not simply recommended, but it is rather enforced by the language. An object has private and public members, where, typically but not always, public members are subroutines (called methods) and private members are raw data.

As an example of OO modelling we introduce the Stack object. A Stack is a data structure that supports two operations, push and pop. push inserts another item in the stack and pop deletes the last pushed from the stack item and returns it. A special stack is the emptystack which doesn't contain any item yet.

A stack may be implemented as an array, linked list or even as a more complicated structure of items. From a structured programming point of view this implementation detail matters, because we define subroutines independently of the data, so we want to know the exact type of data we are operating on. From an OOP perspective the implementation is hidden and all we care about is the existence of the emptystack and push and pop methods, because only through them is the object accessible and usable.

This brings us to the notion of behavioral equivalence which is a notion of type equivalence met in OOP clearly superior to any of the type equivalences met in structured languages. Two objects are equivalent in a behavioral sense if their responses to the same message are equivalent. So if we have a stack implemented as an array and a stack implemented as a linked list, in OOP these two completely different implementations can be considered equivalent for all purposes, a great benefit of the OOP paradigm if correctly and fully utilized.
Unfortunately behavioral equivalence is undecidable in general. Common
OOP languages utilize stronger built-in notions of equivalence (shallow and
deep equivalence [11]) that do have to do with implementation details: two
objects are equivalent if they have the same implementation and equivalent
state. Two states are equivalent in the shallow sense if their raw data are
exactly equal. They are equivalent in the deep sense if the non-pointer raw
data are the same and the pointers point to deeply equivalent objects. Shallow
equivalence implies deep equivalence and deep equivalence implies behavioral
equivalence. Decidable equivalence relations are only approximations of what
we want to achieve, so an OOP programmer usually keeps in mind the need to
create methods that check for behavioral equivalence whenever possible.

4.1.2 Classes

It would be tedious and counter-productive to define the public interface and
private implementation of each object we are going to use in an OO program.
For example if we want to use 4 stacks in a program, one general definition of
what is a stack suffices. Then all we have to say is that we have 4 objects of this
type. That's where the notion of class comes: a class is to an object what a type
is to a value: A class is a set of objects that have a common implementation
and different state. In most OOP languages, classes are the main components
that constitute a program.

A C++ [32] definition of a class Stack looks like that:

class Stack {

int x;
Stack *tail;

public:
Stack (): x (0), tail (0L) {}

static Stack emptystack;

virtual void push (int x) {
    Stack s = new Stack;
s->tail = tail;
s->x = x;
tail = s;
}

virtual int pop () {
    Stack s = tail;
    int y = s->x;
tail = s->tail;
delete s;
return y;
}
};
Any object \textit{s} belonging to this class will now be declared by:

\texttt{Stack s;}

which automatically calls the constructor method \texttt{Stack}, which initializes the object into a state defined by the programmer. After this declaration, \texttt{s} possesses methods \texttt{push} and \texttt{pop} that \textit{transform it to another object of the class Stack}. A method is called by referring to it as a member in the C-\texttt{struct} sense:

\texttt{Stack s;}
\texttt{s.push (1);}

A \textit{static} object \texttt{emptyStack} is also defined, not as a member of any particular object but as a unique special object known to the class. It happens to be of the same type (\texttt{Stack}).

We see here that the concept of encapsulation is preserved through classes. Non-public members of an object are only accessible within members of the class. The following is not legitimate in a C++ program:

\texttt{Stack a;}
\texttt{a.x = 5;}

4.1.3 Inheritance

Some classes are clearly more specific than others and they provide the same functionality plus some extra features. In order to be able to reuse the same definition, and also for reasons of proper modelling of the problem the notion of \textit{inheritance} is introduced.
When a class \( A \) inherits a class \( B \) (we say that \( A \) is a sub-class or child of \( B \) and that \( B \) is a super-class or parent of \( A \)), it has all members of \( B \) along with some additional ones and any object of \( A \) can be used where an object of \( B \) is required.

The conventional structural programming paradigm fails to provide inheritance in its type systems. This has the unfortunate effect for example that values 1.0 and 1 are distinguished in such languages (real and integer are two different types, and there's no way to say that integer is a sub-type of real). To fix this problem, these languages invoke "type coercion", that is ad-hoc exceptions that specify for example that 1.0+2 is equivalent to 1.0+2.0 etc. However this should be the rule and not the exception. The problem with this view of typing is so widespread that even OOP languages of today have it in their simple types (C++, Java etc.), either by tradition or to keep backward compatibility (in the case of C++).

To demonstrate inheritance, here's how we augment our C++ Stack class with a counter:

```cpp
class CounterStack : public Stack {
    int counter;

public:
    CounterStack : Stack (), counter (0) {}

    virtual void push (int x) {
```

90
Stack::push (x);
counter++;
}

virtual int pop () {
counter--;
return Stack::pop ()
}

virtual int getCount () { return counter; }
};

Notice how we are free to use previous definitions of the methods. Were we not to add specific statements to the methods, we could have completely omitted them: the default is to inherit the method as it is in the super-class. What we did is usual in practice: we augment our methods calling the original version and adding a few statements related to the sub-class.

All OOP languages support inheritance, but when it comes to multiple inheritance they take different approaches. Multiple inheritance is the ability to define a class as a subclass of more than one (not necessarily related) classes, as opposed to simple inheritance, where the class hierarchy is a tree. Multiple inheritance presents various implementation and theoretical problems and so languages that emphasize simplicity avoid it altogether (Java provides the interface alternative) while languages that emphasize completeness include it.
with various bells and whistles (C++).

4.1.4 Polymorphism

All modern programming languages incorporate some kind of polymorphism in their design, that is they allow in one way or another certain values to acquire more than one types, so they may be used in different contexts. [8] distinguishes four important kinds of polymorphism found in programming languages:

- Parametric polymorphism is met especially in functional languages such as Haskell [19]: An expression takes its type according to some implicit or explicit type parameter, allowing the easy creation of functions that perform the same operation to different types. For example a Haskell user may define simultaneously a function \texttt{sec} that takes the second item of a list for all list types:

  \texttt{sec :: [a] \rightarrow a}
  \texttt{sec = \lambda l \rightarrow head (tail l)}

  Type \texttt{a} is a type parameter which can be instantiated to any type at all, producing infinite types for the \texttt{sec} function. For each type \texttt{a}, \texttt{sec} can be regarded as a function from a list of a values to a single a value.

- Inclusion polymorphism is the main kind of polymorphism supported by OOP. As we've seen in an OOP hierarchy a value may belong to more than one class, since its class may be a sub-class of another class and so on. That effectively means that a method defined for a super-class is generally available to all the subclasses. The fact that a sub-class may \textit{override} a method is an interesting OOP idea which leads to the concept of \textit{late binding} (in Java always,
in C++ enforced by the virtual keyword). For example, in our C++ stack implementation say that we have a Stack object \( s \) and we call:

\[
s\.push(3);\]

Which method is invoked is actually decided at run-time and it depends on the more specific class whose member is \( s \). If \( s \) is of type CounterStack, then \texttt{CounterStack::push} is going to be executed.

Overloading of an identifier \( a \) is the existence of explicit multiple declarations for it. The language compiler decides on the correct interpretation of any occurrences by looking at the context of the interpretation. Coercion is a special kind of overloading for the predefined symbols of a language. These kinds of polymorphism are not very sophisticated and are of limited interest. In [8], they are labeled “ad hoc polymorphism”.

As already mentioned, parametric polymorphism is generally compatible with the functional paradigm, while inclusion polymorphism is essential to OOP. OOP languages always implement inclusion polymorphism. They sometimes try to include some parametric polymorphism capability, but the approaches seen so far (C++ templates, Ada generic entities) are a lot less satisfactory than their ML-like counterparts.

4.1.5 Overriding

Overriding a method to better suit a sub-class is common practice and the practice itself sometimes is referred to as polymorphism (which is an abuse of the term). Abstract classes and methods are based on overriding. An abstract
method is a method with interface but without implementation. An abstract class is a class with at least one abstract method. An object cannot belong directly to an abstract class, so to use abstract classes, we must provide subclasses that override their abstract methods and thus provide an implementation. Abstract classes are good modelling practice. For example imagine the class “mammals”. No creature is an instance of the class “mammals” without belonging to a more specific class in the hierarchy (for example “humans”). However to have the notion of “mammal” and to attribute to it the common mammal characteristics is intuitive and useful.

Overriding is also useful when we want the new method to be more efficient than the original would be. It is possible to achieve a more efficient implementation in general, because a sub-class is more specific, therefore the problem is more specific and potentially easier.

The problem with overriding lies in its power: overriding a method means that we are allowed to introduce mistakes in an otherwise bugless class. Suppose that we have a class C and a method m that works fine. Suppose that a subroutine f somewhere takes a C parameter and relies on the fact that m works fine. Now a programmer sub-classes C to D and overrides m introducing a bug. Now, not only m doesn’t work, but f doesn’t work either, even though we didn’t make any changes to it. Given that overriding is essential in OOP as we understand it now, it is natural to ask if there is a way out of the problem.
4.1.6 "Object invariant"

OOP is famous for modelling reality better than other programming styles, because it provides a formal way to define the entities of a problem, that would otherwise be in the programmer's mind but not in the formalism. But, this modelling only seems to go half the way in capturing the idea that a programmer has about the entities. For example, let's take the class we've just programmed. We know that our C++ program defines a class `Stack` that we can use in certain legal ways and not others. But is that all a stack is? Then, the following is a stack:

```cpp
class Stack {
public:
    virtual void push (int x) { }

    virtual int pop () { return 0; }
};
```

To emphasize our point, imagine that we want to create an abstract type `Stack` that specifies the interface that a stack should implement. This is the way to do it in C++:

```cpp
class Stack {
public:
    virtual void push (int x) = 0;
```
virtual int pop () = 0;
};

We now have a class that can be used as a super-class for both our previous implementations. Something goes wrong here: one of the implementations is correct, while the other is definitely not what we want a stack to be, whereas both are accepted by OOP. When we create and use our super-type, we expect the sub-types to be created in a way that reflects the idea that we have about what a stack is. But the formalism does not reflect this idea, and it does not impose it to further sub-classing.

It would be nice if OOP allowed the designer the freedom to formalize this idea of the stack to the finest detail, by providing axioms on the object use. For example, here we want to enforce our implementations to respect the rule that \texttt{s.push(x); s.pop();} leaves \texttt{s} unchanged. Such axioms we would intuitively call "object invariants".

4.1.7 Generalization

One of the main powers of OOP is that it gives the programmer a lot more opportunities for code / design reuse via the mechanisms of inheritance. Through inheritance a programmer can specialize code written (or even only specified in abstract classes) for a more general case, perhaps by somebody else, perhaps a lot earlier. The new specialization is an augmentation of the system and typically the super-class is designed without knowledge of all the specializations that will occur. So, OOP has successfully addressed the issue of augmenting
specification.

What is at the moment impossible in OOP is the concept of augmenting generalization, for example, given a class create a super-class that satisfies certain (more general) properties. That could open the way for example to create a class previously not thought of, or previously neglected as unimportant and to do that late in the development cycle. Being able to do that seems to have the potential to reduce the cost of undetected design flaws and this is a major issue in software engineering.

We further wish to note that the need to generalize / abstract an idea appears very often. For example, in the history of mathematics, first natural numbers were invented, then integers, then real numbers and finally complex numbers. Each one of these categories of numbers, was invented after a need to model something was not met by what was previously there, so each of them was actually a design decision that included generalization. Were we not capable of abstracting our idea of “number” to introduce these new concepts, we would have to deal with four or more different categories of entities, and we would have to reinvent the same operators and axioms for them over and over again.

4.1.8 Interaction with functional programming

The current trend in OOP involves enriching popular imperative languages with OOP features (with a multitude of examples, most known of which is C++), or creating new OOP-imperative style languages (Java). In fact, the distinction between imperative and functional programming style is generally orthogonal to the distinction between structured and object oriented programming. So it
would be interesting to see whether there is a way to combine OOP with the functional programming paradigm trying to get the best of both worlds. Several approaches have been tried.

FC++ [26] is a C++ library designed to make available to C++ programmers certain features that are commonly found in the functional programming paradigm, such as higher order functions and lazy evaluation. It is a very interesting and useful work, but we claim that this is not really what we mean by marrying functional with object oriented programming. The reason is that the features implemented are not necessarily characteristics of functional programming, even though by tradition they are usually found in functional languages. They are features that are simply not genuinely supported by C++, but they don’t turn it into a functional language. Clearly this is not the way we want to go in this thesis.

The fundamental characteristic of a functional language is that it is heavily dependent on the concept of a mathematical function and therefore on referential transparency and lack of side effects (or controlled use thereof). To retain the functional flavor, it seems that we have to start from such a pure-functional language and proceed by adding object orientation, unlike FC++.

CLOVER [4] is a functional / OOP language that is designed this way and it seems that very important lessons are learned from its design about the nature of both paradigms. Among many major and minor technical decisions that are explained in [4], we believe that one issue is especially worth discussing, since it emerges as the main design difficulty: the conflict between the lack-of-state
nature of functional programming and the notion of object state.

The problem is that objects in OOP are very much conceived as entities with state, something alien to functional programming. So for example our C++ stack object starts as the empty stack:

```cpp
Stack s;
```

Later we change its state by pushing something

```cpp
s.push(3);
```

And then pushing something else

```cpp
s.push(2);
```

`s` is now an object with different behavior, because a call to `s.pop()` will return a different number than it did before. But it is the same object as far as the programmer is concerned.

In fact `s` is the variable and not the value. A variable can change its value. An object is a value, and remains unchanged: an empty stack is not equivalent to a stack containing one number, the same way number 0 is not equal to 1. So a functional approach to object orientation would result in rejecting this idea of object as state (object is the variable and the value) and incorporating the idea of object as value.

Taking this a little bit further, we argue that a simpler and purer approach to OOP would be to incorporate this idea of object-value in imperative style languages as well. `s` still remains our variable, but now an object is a value and our hypothetical code would look like:
Stack s;

s := s.push(3);

The new value is returned by the method and it's assigned to the same variable. This way we get the same clear distinction between variables and values that we already had for simple data types. Understandably, a good compiler for such a language should know how to avoid excessive copying of objects through careful dataflow analysis, but we don't concern ourselves with this matter here.

Note also that there is a related conflict between referential transparency and encapsulation. Referential transparency is critical to functional programming, but it seems to leave no place for encapsulation: If encapsulation is to be supported, then we want the hidden state of an object to be non-accessible. But for referential transparency, if we equate an object with its implementation, we end up having total access to its internal implementation.

To work around this apparent contradiction, we argue that a mathematical model that supports both referential transparency and encapsulation, should be based on behavioral equivalence. Then an object is not equated to its implementation (which is supposed to remain hidden) but to its behavior and referential transparency is applied to this behavioral level, without hindering encapsulation. We will see later exactly how we can do that, when we build OOP semantics in terms of theories.
4.1.9 Summary

In this section we gather together the strong and weak points that the OOP paradigm seems to possess, according to our previous analysis.

Strong points:

- The notions of object and encapsulation provide a more natural way to model a problem.
- Via encapsulation, implementation hiding is better in OOP.
- Behavioral equivalence is a better modelling of real-problem equivalence.
- Classes enhance the object idea and allow for code and design reuse.
- Similarly, inheritance enhances the class idea and allows for even finer code and design reuse.
- Inclusion polymorphism is supported in a very natural way.
- The idea of abstract classes is essential in modelling real-world problems.

Weak points: The first three consider current OOP languages, the rest are more general observations about the paradigm.

- Behavioral equivalence is not supported by current languages because it's undecidable.
- Multiple inheritance is being avoided by language designers because it seems hard to understand.
- Parametric polymorphism is not well supported in current OOP languages.
- OOP does not limit overriding power, which contrary to its modular philosophy helps introduce errors from a new module to previously correct modules.

- OOP gives a way to formalize the entities in a problem, but not their behavior, which is still informally stated in the designer's mind.

- Code increases from general to specific, via inheritance. Generalizing classes is not permitted. This keeps the cost of undetected design errors high.

- The variable/value distinction found in simple types is blurred in OOP languages. This gives values a state and makes it hard to imagine an OOP functional language. However, the necessity of the existence of object state is a misconception.

- Referential transparency seems to be in conflict with encapsulation, so languages tend to support only one of them.

### 4.2 Formal semantics for object orientation

Our discussion above was an informal introduction to the ideas encountered in OOP. Various attempts were made over the years of OOP predominance to formalize the concepts by developing semantics for them. Formal semantics should be understood as an effort towards deeply understanding an otherwise vague idea, by giving it exact mathematical definition. Ideally, a semantics should point out holes and ambiguities of the analyzed concepts, while at the same
time revealing and developing a more general way of understanding the ideas and a more general framework of their use. In this section, we briefly present three of the formal semantics developed for OOP which we believe contribute very much, according to this point of view. They are Cardelli and Wegner's quantified types [8], $F$-bounded polymorphism [7] and William Cook's denotational semantics [9] (we are especially interested in the idea that "inheritance is not subtyping" [10]).

However, to the best of our knowledge, no attempt has ever been done so far to include within OOP a complete specification language (that would compensate for the "lack of object invariants" point we made earlier), because the research is oriented towards modelling, understanding and improving OOP characteristics such as polymorphism and inheritance. Also, all these semantics do not pay enough attention to unification of concepts. We believe that the formal theory-based semantics we present in the next section is superior in this sense, as it is primarily concerned with how we can introduce this specification formalism into the OOP modularity and how to unify as much as possible. Unification by itself proves a powerful way to discover new possibilities for the paradigm.

4.2.1 Quantification of types

The first semantics we are going to explore is presented in a very influential early paper by Cardelli and Wegner [8]. Their work aims in particular in expressing formally various kinds of polymorphism and inheritance. They succeed in providing a general framework for understanding polymorphic types and
their survey of polymorphism is very helpful (largely based on previous work by Strachey [31]).

Types

To Cardelli and Wegner, a type is a set of values. But not all sets of values are types, we are rather interested in a countable number of them, which we deem interesting. This selection apparently depends on the design of our language. Typically a type system for any language looks like that

\[
Type ::= BasicType | ConstructedType
\]

\[
BasicType ::= \text{Int} | \text{Bool} | \text{Real}...
\]

\[
ConstructedType ::= \text{Array}(Type) | Type \rightarrow Type...
\]

that is, a set of primitive types and a set of type constructors with which infinite (but countable) different types may be produced. The set of all values (and superset of all types) is called Top.

A type operator operates on types and returns types, for example

\[
\text{Arrow}[T] = T \rightarrow T
\]

is the type operator that returns the type of functions from \(T\) to \(T\), where \(T\) is a parameter type, or

\[
\text{Pair}[T] = T \times T
\]

etc. Type operators can be recursively defined as in

\[
\text{List}[T] = [\text{nil} : \text{Unit}, \text{cons} : \{\text{head} : T, \text{tail} : \text{List}[T]\}]
\]
A language is polymorphic if it allows a value to have multiple types. If we think of types as sets of values, this is a very natural choice, because a value belongs not to one, but to uncountably many sets. However, monomorphic (i.e. non-polymorphic) languages don't take this into account, with the result that they have to provide "type-coercion" systems to "convert" an integer into real when "necessary".

**Universal quantification**

$T \rightarrow U$ is the type of functions from $T$ to $U$. Many times we want a function to be polymorphic, that is to operate on various kinds of values in a similar way. $T = Top$ is too general however, it'll do for the identity function but for not much more, because a declaration $x : Top$ does not say absolutely anything useful about $x$.

Suppose for example that we want to create a function $twice$ that applies its function parameter twice, that is (in untyped $\lambda$ calculus notation):

$$twice = \lambda f. \lambda x. f(f(x))$$

What is the type of this function? It is not as general as $Top \rightarrow Top$, because it can only take a functional argument $a \rightarrow b$. Notice that also $a$ and $b$ have to be the same, so it really takes an argument $a \rightarrow a$. The return value is also of type $a \rightarrow a$. This is true for all types $a$, so we denote this in Cardelli and Wegner notation:

$$twice : \forall a. (a \rightarrow a) \rightarrow (a \rightarrow a)$$
and we call it a universally quantified type. We write in "fun", their toy pro-
gramming language:

\[ \text{twice} = \text{all}(\text{fun}(f : t \rightarrow t)(\text{fun}(x : t)f(f(x)))) \]

where \( \text{fun} \) is their way of saying \( \lambda \). To call \( \text{twice} \) one must first specialize it
giving it an exact type for \( a \):

\[ \text{twice}[\text{Int}](\text{succ})3 = 5 \]

Universally quantified types thus exactly model parametric polymorphism.

**Existential quantification**

*Existentially quantified types* is another addition that naturally comes next. The
universal type quantification

\[ \forall a. A[a] \]

describes values for which, given any type \( a \), they are of type \( A[a] \). Similarly,

\[ \exists a. A[a] \]

describes values for which there exists a type, such that they are of type \( A[a] \).

For example

\[ (3, 4) : \exists a. a \times a \]

Of course it is also

\[ (3, 4) : (\text{Int}, \text{Int}) \]

*Top* can be expressed as a very general existentially quantified type:

\[ \text{Top} = \exists a. a \]

106
Note that $\exists a. a \times a$ is the type of any pair, even though the same type variable $a$ is used in both sides of $\times$, because any pair belongs to $\text{Top} \times \text{Top}$.

Existentially quantified types hide the structure within their parameter type, leaving exposed only the structure we want to use as an interface. This is a proper modelling for encapsulation. To create an encapsulated object, such as the stack we were previously discussing, we hide structure within the existential quantification:

$$\text{Stack} = \exists a. \{ \text{push} : (\text{Int} \times a) \to a, \text{pop} : a \to (a, \text{Int}) \}$$

$\text{Stack}$ can now be used only through $\text{push}$ and $\text{pop}$ fields, because they’re all we know about $\text{Stack}$. Both $\forall$ and $\exists$ are used in an example in [8] to create a generic stack that is both parametrically polymorphic (with respect to the type of values it stores) and abstract (with respect to the exact implementation type).

**Subtyping**

Type $A$ is a subtype of type $B$, and we write $A \leq B$ if $A$ is a subset of $B$. Several subtyping rules apply to this point, including function subtyping which is similar to our function inclusion axiom:

$$s \to t \leq s' \to t' \Leftrightarrow s' \leq s \land t \leq t'$$

Such subtyping also applies to records (which would be redundant to mention, if records were defined as functions from text to values). $A \leq B$ means essentially that $A$ values can be used wherever $B$ values are expected, which opens the way for inheritance (as instance of subtyping) and inclusion polymorphism.
The ≤ relation can be used for bounded universal and existential quantification, that is quantification that ranges over the subtypes of a given type. Cardelli and Wegner claim that bounded quantification is strictly more expressive, because it can retain "type information". They give this example in "fun":

```
type Point = {x : Int, y : Int}
```

```
value moveX0 = fun(p : Point, dx : Int) p.x := p.x + dx; p
```

```
value moveX = all[P ≤ Point]fun(p : P, dx : Int) p.x := p.x + dx; p
```

`moveX` does not retain type information, because it returns a `Point` value, not otherwise specified, even when legitimately called with a value that belongs to a `Point` subtype, say `Tile`. `moveX` doesn't suffer from this because of parametricity of the type information:

```
moveX[Tile](tile)
```

returns a `Tile` value.

We disagree with this take on polymorphism, that assumes that the result of the evaluation of an expression should contain both its value and its type (more correctly: one of its types). Of course, it is a matter of different philosophies. Cardelli and Wegner play it type-safe throughout the whole paper, so their claim is consistent with their philosophy.

**Critique**

[8] provides a thorough and complete study of polymorphism and a satisfactory formalism. Those features are constrained however to keep type checking easy
(at least decidable) and to maintain type safety. Perhaps this is not a right point to limit the expressiveness of a type language. We advocate a different approach: we don’t want the undecidability barrier or any other constraint to prevent us from coming up with an expressive specification language. We eventually want to replace type-checking with program proofs and we believe that even in undecidable specification languages the majority of the proofs should be trivial.

4.2.2 F-bounded polymorphism

Quantification and bounded quantification of types were not expressive enough to cover all cases of polymorphism and it proves especially weak with recursively defined types (very frequent in OOP). The attempt is revised in a later paper [7] by Canning, Cook, Hill, Olthoff and Mitchell who invent an interesting expansion they call $F$-bounded polymorphism. To demonstrate the problems with bounded quantification, the authors of [7] give two examples, one of “positive recursion” and one of “negative recursion.

Motivating examples

In the positive recursion example, suppose we have a recursively defined type:

$$Movable = Rec \, mv. \{move = Real \times Real \rightarrow mv\}$$

(In this notation $Rec$ is the fixpoint function. It binds variable $mv$). The recursion is positive, because the fixpoint parameter $mv$ occurs at the right side of $\rightarrow$, which is monotonic with respect to its right side argument. Now let

$$translate = fun(x : Movable)x.move(1.0, 1.0)$$
be a polymorphic function. \textit{translate} always returns a \textit{Moveable} object, nothing more specific, whereas we would probably want a polymorphic function that would always return a result of the same type as the parameter.

In the negative recursion example, the fixpoint parameter assumes the anti-monotonic left side of $\rightarrow$. Let

$$PartialOrder = Rec \; po. \{ \text{lesseq} : po \rightarrow bool \}$$

be a partially ordered type and

$$\text{minimum} : \forall t \subseteq PartialOrder.t \rightarrow t \rightarrow t$$

a function that returns the minimum of two elements of that type. If we have a type \textit{Number} that contains \text{lesseq}, we can’t use it with \text{minimum}, because \text{Number} $\subseteq$ \text{PartialOrder} would require (because of the antimonotonic nature of the left side of $\rightarrow$) \text{PartialOrder} $\subseteq$ \text{Number}, which is intuitively false (we want \text{PartialOrder} to contain any type that is partially ordered and not the other way around).

\textit{F}-bounded quantification

Let \textit{F} be a type function. Then

$$\forall t \subseteq F[t].\sigma$$

is an \textit{F}-bounded quantified type. The innovation is that the bounded variable is introduced to the bounding function, unlike [8].

Now the \textit{translate} example can be:

$$translate = \text{Fun}[t \subseteq \{ \text{move} : \text{Real} \times \text{Real} \rightarrow t \}]\text{fun}(z : t).z.\text{move}(1.0, 1.0)$$
which translates for any type $t$ that contains a `move` field that takes two real
numbers and returns an object of the exact same type $t$.

For the negative recursion example, the authors suggest the replacement of
`PartialOrder` with function

$$F \text{PartialOrder}[t] = \{ \text{lesseq} : t \rightarrow \text{bool} \}$$

and now $\text{Number} \subseteq F \text{PartialOrder}[	ext{Number}]$ and

$$\text{minimum} : \forall t \subseteq F \text{PartialOrder}[t], t \rightarrow t \rightarrow t$$

In all cases the set of types that are $F$-bounded by a type function $F$ is

$$F \text{BOUND}[F] = \{ t \mid t \subseteq F[t] \}$$

and to be able to quantify over this set is admittedly a gain in expressiveness
compared to bounded quantification.

4.2.3 Inheritance is not subtyping

$F$-bounded polymorphism offered a valuable insight to the research team that
developed it, concerning inheritance in OOP. In [10] W.Cook et al. argue
that the current understanding of inheritance as subtyping is wrong, and their
argument is based on W.Cook's denotational semantics for inheritance [9] and
$F$-bounded polymorphism ideas. The core argument lies again on recursively
defined objects, especially "negative recursion" in this case. Suppose that we
have a class $C$ that defines a method $eq : C \rightarrow \text{bool}$. To inherit this class into
class $D$, where $D$ has the same recursive structure plus an additional element
does not make $D$ a subtype, because, $D.\text{eq} : D \to \text{bool}$ would then be a supertype of $C.\text{eq}$. In this paper’s notation:

$$C = \mu t.\{\text{eq} : t \to \text{bool}\}$$

$$D = \mu t.\{\text{eq} : t \to \text{bool}, a : \text{bool}\}$$

we see that $C$ and $D$ have the same recursive structure. They are both $F$-bounded by function

$$F[t] = \{\text{eq} : t \to \text{bool}\}$$

but none is a subtype of the other.

Thus languages that unify subtyping and inheritance tend to run to problems with negative recursion, which they solve by more restrictions. In C++ for instance, we are not permitted to override the type of the method, so $D$ would have to contain a method $\text{eq} : C \to \text{bool}$, which is not what we want it to be.

Cook et al. are right to say that $F$-bounded polymorphism suffices to model inheritance the way they mean it (and also the way that it is done in SmallTalk, a non-typed language). A recursively defined object is of type $\mu t.F[t]$. An instantiation must contain the type of the object $\sigma$ which must satisfy the $F$-bound $\sigma \leq F[\sigma]$. For inheritance we come up with a new recursive type $\mu t.G[t]$, such that for all $t$: $G[t] \leq F[t]$. The inherited type constructor is a subtype of the old type constructor, but not all fixpoints of $G$ are subtypes of all fixpoints of $F$. Classes are viewed as parameterized objects and the solution offered for them is similar.

Therefore, subtyping applies to objects, while inheritance is subtyping of
type constructors, which are then used to create recursive objects, not necessarily preserving the subtype relation. This reveals the clumsiness of the (not formally analyzed and adequately understood) design of OOP languages.

In this thesis, this basic distinction is accepted, as we want to do both “inheritance” and “subtyping” in OOP and they are clearly two different things. However, we would prefer another point of view as far as the terminology is concerned. It seems that inheritance is *meant* to be subtyping in OOP and it should be thought of this way. What they mean by “inheritance”, we would rather call “meta-inheritance” or something like that. We claim that what they mean by “inheritance” is not what inheritance originally was thought of in OOP.

For example, in our class *C* mentioned before, suppose that *eq* is originally defined as an equivalence check within *C*. If we are to consider class *D* “inherited”, then this means that we may use a *D* object wherever a *C* object is expected. That definitely means that we may compare a *C* and a *D* object with *eq*. It also means that we may want to compare *D* objects with *C*’s *eq* method. *D*’s new equality method is therefore a *different method* and shouldn’t be an overriding of the original *eq*. If, unlike this proposal, one goes on to define *C* and *D* the way Cook et al. do, then it is clear that *C* and *D* have no subtype relation and therefore *D* cannot be used in general in place of *C*. Their relationship is now a looser one: they have the same recursive structure.

The punch line is that we disagree with their understanding of what inheritance is and what we want to model with it. To us, inheritance is subtyping. What they call “inheritance” is for us a valuable new concept of relation between
As for the equivalence example, we will later return to it, proposing an alternative solution to model various levels of equivalence, based in high-level elementhood and elevating, which were introduced in the previous chapter. We believe that this approach is conceptually clearer and more importantly compatible with subtyping which is very desirable.

### 4.2.4 More related work

This section contains some other research conducted towards formalizing OOP.

Pierce and Turner in [28] created a simple type theory for OOP, emphasizing the use of existential types (like Cardelli and Wegner's [8] and unlike Cook et al. [10]) and they elaborate more than Cardelli and Wegner on OOP as a whole instead of just polymorphism. Their model is theoretically appealing and conceptually very simple.

Bruce [5], also influenced by Cook's [9] built several denotational semantics of increasing complexity, to formalize OOP. In his paper, he comments that OOP is much too complex to be reflected in a simple formalism, argument with which we disagree, because we feel that the ideas behind OOP are not complicated. We find that the problem lies more within the denotational semantics methodology.

Kamin and Reddy summarize in [20] their previous research. They present two semantics for OOP and prove them equivalent. Their discussion focuses mainly on variants of SmallTalk (of increasing complexity), which they model. Their first semantic model is denotational. The second mimics more the concrete implementation of objects and classes in SmallTalk.
An interesting concept is that of mixins, explored in [12]. Mixins are functions from classes to classes, that are mainly invented to substitute multiple inheritance. Flatt et al. support mixins as an expressive, yet easy to understand way to avoid the complexity of multiple inheritance in a programming language.

Finally, we should mention an other line of research that tries to deal with what we called the "too powerful overriding" problem. The term behavioral subtyping is coined for subtyping that doesn't cause objects to behave "surprisingly" in terms of their specification. Leavens has done a lot of work in this field; a recent example that cites also other related work is (written by Leavens and Pigozzi) [21]. This work is closely related to the framework presented below in this thesis, even though the authors are not interested in the specification language, which they abstract away and reason with algebraic models instead.

4.3 A theory theory semantics of OOP

4.3.1 Classes and objects as theories

The main idea of our framework is the unification of classes, objects and formal theories. We consider any object or class to be a formal theory of the bunch θ. That an object o is an instance of a class C is captured by o : C, so a class is really a bunch of all its objects. That makes all classes objects and all objects classes. That also gives the possibility to treat whole bunches of objects as one single object (or class-in our framework they are not distinguished). Encapsulation is achieved by existential quantification, like we did in theories.
Here's a stack class created as a theory:

\[
\begin{align*}
& g_1 = \langle - - \\
& \quad \text{pub} \quad \text{push pop} \\
& \quad \text{hid} \quad \text{list} \\
& \quad \text{forall} \\
& \quad \text{axiom} \quad \text{list} : [*\text{int}] \\
& \quad \quad \text{push} : (n : \text{int} \rightarrow \text{construct("list" \rightarrow [n]^\text{list}})) \\
& \quad \quad \text{pop} : \text{if} \quad \text{list} = [\text{nil}] \\
& \quad \quad \quad \text{then} \quad [\text{nil}] \\
& \quad \quad \quad \text{else} \quad [\text{list 0}; \text{construct("list" \rightarrow \text{tail list}})] \\
& \quad \quad \text{- -} \\
& \end{align*}
\]

\[Stack_1 = \emptyset g_1\]

We defined \(Stack_1\) to be the bunch of all stacks. Any instance of the stack class is a subtheory of \(Stack_1\), and we can get any instance corresponding to some hidden list \(\text{somelist}\) like that:

\[create g_1 ("list" \rightarrow \text{somelist})\]

More than one instances can be taken together. For example the bunch of all stacks that contain positive numbers only is:

\[create g_1 ("list" \rightarrow [*\text{nat}])\]

By the \textit{create}-law of course, all those objects are subtheories of \(Stack_1\).
4.3.2 Behavioral equivalence

Our objects encapsulate their implementation using existential quantification, thus, for example, the list variable \( \textit{list} \) is not visible out of the definition of \( g_1 \).

At this point the reader might reasonably ask: what exactly do we mean by “encapsulation”? If we are able to change the hidden value using \( \textit{create} \) (or deep conjunction) this not truly encapsulation. Three points are to be made here:

- \( \textit{create} \) is not to be provided in a programming language. The class designer is free to use it within the theory as \( \textit{construct} \). The designer may define and use creating methods (called \textit{constructors} in standard OO languages).

- Even if \( \textit{create} \) is allowed, \textit{there is no way for the outsider to create an invalid instance of Stack}_1. \( \textit{create} \) defines the value of the hidden state that corresponds to a correct instance (or correct instances).

- What we mainly want to achieve by encapsulation is not discipline of the programmer, but behavioral equivalence.

The final point is very important: Mathematically \( \textit{list} \) is hidden from the outside world, without having this mathematical encapsulation affect referential transparency. We now show how to create the exact same theory by transforming the hidden part (that is, altering the implementation). Notice that the hidden theory within \( g_1 \) is a subtheory of

\[
L_1 = "\textit{list}" \rightarrow [\ast \text{int}]
\]
Now we define the following transformation $tt$ that operates on $L_1$ objects:

\[
\begin{align*}
    tt & = \langle s : L_1 \\
         & \rightarrow \text{let } list = s \text{"list"} \\
         & \text{in } \text{"isEmpty" } \rightarrow (list = \text{[nil]}) \\
         & \text{if list = [nil]} \\
         & \text{then } \text{all} \\
         & \text{else } \text{"head" } \rightarrow \text{list 0} \\
         & \text{if } \text{tail } \rightarrow tt(\text{tail list}) \\
     \rangle
\end{align*}
\]

Breaking $L_1$ like that:

\[
L_1 = (\text{"list" } \rightarrow \text{[nil]}), (\text{"list" } \rightarrow \text{[int]+[int]})
\]

we can prove easily by the fact that $tt$ is bunch union distributing that:

\[
\begin{align*}
    tt L_1 & = \text{"isEmpty" } \rightarrow \text{true} \\
             & , (\text{"isEmpty" } \rightarrow \text{false} \\
             & \text{if } \text{"head" } \rightarrow \text{int} \\
             & \text{if } \text{tail } \rightarrow tt L_1 \\
     \rangle
\end{align*}
\]

and we call

\[
L_2 = tt L_1
\]

By induction on the number of items in $x \text{"list"}$, it is easily shown that:

\[
\begin{align*}
    inv tt & = \langle x : L_2 \\
             & \rightarrow \text{if } x \text{"isEmpty"} \\
             & \text{then } \text{"list" } \rightarrow \text{[nil]} \\
             & \text{else } \text{"list" } \rightarrow (x \text{"head"})^{+} (inv tt (x \text{"tail"})) \\
     \rangle
\end{align*}
\]

118
And by induction again:

\[ \forall \langle x : L_1 \rightarrow inv \; tt \; (tt \; x) = \text{project} \; x \; "list" \rangle \]

Therefore the transformation preserves \( eq_{g_1} \), because it may also be shown easily that:

\[ \forall \langle \begin{array}{c}
x : \theta \\
\rightarrow \forall \langle \begin{array}{c}
y : \theta \\
create \; g_1 \; x = create \; g_1 \; y = x"list" = y"list"
\end{array} \rangle
\rangle \]

Therefore, by deep transformation, we are sure to have complete theory equivalence:

\[
create \; g_1 \; L_1 = create \; g_2 \; L_2
\]

where \( g_2 \) is defined composing \( g_1 \) with \( inv \; tt \), which gives:

\[
g_2 = \langle \begin{array}{c}
\text{pub}
\quad \text{push} \; \text{pop}
\text{hid}
\quad \text{isEmpty} \; \text{head} \; \text{tail}
\forall \text{all}
\text{axiom}
\quad \text{hidden} : L_2
\land \text{push} : \langle \begin{array}{c}
\text{n} : \text{int}
\rightarrow \text{construct}(\begin{array}{c}
\text{"isEmpty"} \rightarrow \text{false}
\mid \text{"head"} \rightarrow \text{n}
\mid \text{"tail"} \rightarrow \text{public}
\end{array})
\end{array} \rangle
\land \text{pop} : \text{if isEmpty then [nil] else [head; inv tt tail "list"]}
\end{array} \rangle
\]

119
The formal proofs were omitted, because they are not the main point of the section.

4.3.3 Inheritance as containment and overriding

Suppose now that we want to inherit \( \text{Stack}_1 \) in such a way that a new member "size" is exposed that contains the size of the stack. This is a simple case of inheritance that we also demonstrated in C++. Our new stack theory can look like this:

\[
\text{Stack}_3 = \text{Stack}_1 ("size" \to \text{nat})
\]

which is not satisfactory (although the only possible approximation if the only axioms allowed are type axioms, as in our C++ example), because we didn't specify any interesting axiom about the newly added member. A more interesting way to do it would be with deep conjunction that accesses the hidden variable:

\[
\text{Stack}_4 = \text{g}_1 \land
\begin{cases}
\text{pub} & \text{size} \\
\text{hid} & \text{list} \\
\text{forall} & \text{axiom} \quad \text{size} = \#\text{list}
\end{cases}
\]

(Usually OOP languages support two layers of abstraction called \textit{private} members and \textit{protected} members. Protected members are hidden from the outside world but, unlike private members, visible to the inheritors. We don't go into
that level of technical detail here, because it would significantly obscure the presentation.)

Inheritance in our model is modeled thus by bunch containment, as was instantiation. Both are unified in one and single operator. The fact that Stack4 is a subclass of Stack1 is modelled by Stack4 : Stack1. But Stack4 is very abstract yet: it has no concrete implementation. A concrete implementation would typically override methods push and pop to make sure that we still get a Stack4 object after the operation. That leads us to the definition of Stack5, a concrete implementation, such that Stack5 : Stack4:

\[
gs = \langle - - \\
\quad \text{pub } \begin{array}{l}
\text{push pop} \\
\text{hid } \begin{array}{l}
\text{list size (super = create g, hidden)}
\end{array}
\end{array} \\
\quad \text{forall } \begin{array}{l}
\text{axiom } \begin{array}{l}
push : \langle n : \text{int } \rightarrow (\text{"size" } \rightarrow \text{size + 1})|\text{super"push"n)} \\
\wedge \text{pop : if } \begin{array}{l}
\text{size = 0} \\
\text{then } [\text{nil}] \\
\text{else } (\text{"size" } \rightarrow \text{size - 1})|\text{super"pop} \\
\end{array}
\end{array}
\end{array}
\rangle
\]

\[
\text{Stack5} = \Theta gs
\]

It is easily proved that Stack5 : Stack1 and also with induction it can be proved that Stack5 : Stack4. Stack5's definition exactly defines it, which means that Stack5 ≠ inconsistent.

There are two important points to be mentioned here: First, the overriding of the methods could never have the destructive effects that overriding can have in OOP programming, due to the unification of inheritance and bunch
containment. Second, implementation reuse is possible in a very elegant way (look at the definition of super) above.

Indeed, if we were to define incorrect methods push and pop, we would not be able to prove that Stack₂ : Stack₁. To retain the desired subtyping relations, push and pop must obey the function inclusion rule, so they may return values that are included in the values of push and pop of the superclass. Typically this is done by adding fields, as in this case, but it can also be done by restricting the returned value of the method of the superclass. We consider this to be a solution to the overriding problem, we mentioned before. The power of overriding is restricted. Only changes that would be acceptable by the supertype are allowed.

This encourages the heavy reuse of the implementation of the superclass (which is already a usual programming practice in OOP) because it facilitates the proof that Stack₂ : Stack₁. The reuse of the superclass implementation is possible by “translating” the object to the appropriate “pure” Stack₁ object using create. This is conveniently named super, because its use is exactly the same as Java’s keyword super. A this variable similarly would be create gs hidden, or in abbreviated form, construct hidden. Later on, we will systematize this idea, when we introduce class theories.

4.3.4 Structuring class information into theories

We have already presented a core idea of our framework, which is the unification of classes, objects and theories and our idea of how method overriding should be validated. However, our system is still weak in many ways. We are not
yet able to reason about classes as a whole, enforcing sophisticated whole-class properties (not object properties) and providing static information in classes where needed (static information is information about a class as a whole and not specific to a particular object). In our particular stack example what we want and not yet have is:

- A theory that reasons about the properties that other theories must have in order to be considered implementations of stacks. We could already enforce in our theory the basic LILO stack property, that is for any stack \( x \), it must be \( x{\text{push}}n{\text{pop}} = [n; x] \), but we didn’t do so. But still, we cannot say for example that a stack theory must have a \textit{push} method that takes an integer and returns a stack of \textit{the same theory}. This is a property of our implementation theory as a whole and not of its subtheories.

- Axioms about the empty stack. The empty stack is a kind of static information about the stack class.

In this section we explore a way to standardize theories about classes, which will give us all this expressiveness. We will call them, \textit{metatheories}. With metatheories, we will create an abstract \textit{MetaStack} specification that describes exactly what we expect from a stack theory. Compared to C++ abstract classes or Java interfaces, this framework is clearly more expressive, even when it contains only type axioms. For example our C++ abstract class couldn’t say that the result of applying \textit{push} to a stack, should be a stack of the same implementation. We will not use the framework only for type axioms however. We will
introduce the LIFO property as an additional object invariant. This compensates for the inability to wrap such specifications within abstract and concrete class declarations in OOP languages.

Our metatheories will be theories of the bunch

\[ Metatheory = "Class" \rightarrow setaf \theta \]

which means that the described theory for any metatheory \( m \) is \( m\text{"Class"} \). The theory will be wrapped in a set, so the metatheory can reason about it as a whole and not about its subtheories. A metatheory can talk about many theories at the same time, that is \( m\text{"Class"} \) can be a bunch of sets. If a metatheory \( m \) is specific enough to describe one class, then \( m\text{"Class"} \) is the theory that corresponds to the class.

One thing that we will probably want to do with all our class theories is to give them their own high-level elementhood. For any class, we usually want to assign it to at least one such partition that describes its objects. This can be done in the metatheory. In fact we will derive a theory that is both a metatheory and a high-level elementhood, and also has the property that all the elements of the class described are included in the partition. We name all these theories _metaclasses:_

\[
\begin{align*}
\text{Metaclass} &= \text{MetaTheory'HLE} \\
&\quad \& \{ s :: \theta \\
&\quad \rightarrow \sim s\text{"Class"} : \bigcup b \sim s\text{"HLEP"} \}
\end{align*}
\]

124
Now if we have a metaclass $m$, and an object $o : ~ m"{Class}"$, we can always elevate this object to the corresponding high-level element of its class by

$$(~ m"{Elevate}" )o$$

This models Java's `this` and `super` but also generalizes it, because if we know that an object belongs to $n$ different such classes (not necessarily subclassing one-another) we have $n$ such supers.

Notice now, that if we define a theory $\Theta g$ in terms of an `absaxiom` member $g$, we already have an important equivalence partition on the members of the theory, defined by $eq_g$. This partitioning will probably do for most of our purposes. An example of using this exact partitioning (without mentioning it) was shown in the previous section. Because this use of $g$ to create a class and an equivalence partitioning at the same time is going to occur a lot, we are going to give it its own name: `BuildClass`. For any $g : `absaxiom` (this function is not distributing over bunch union):

$$BuildClass ~ g$$

$$= \text{Metaclass}$$

$\exists (s :: \emptyset)$

$$\rightarrow \quad (s"{Class}" = \{\Theta g\})$$

$$\land\quad \text{let} \quad f = (`T : setof \emptyset \rightarrow create ~ g (~ `T))$$

$$\text{in} \quad f(\mathcal{D}f) : \sim s"{HLEP}"
$$

Because of the theorems of `HLE` and the inclusion of all results of `create g` within `HLEP`, if $M = BuildClass ~ g$, then for any theory $t$, it is

$$create ~ g ~ t = (\sim M"{Elevate}")(create ~ g ~ t)$$

125
We will refer to this theorem, as the basic property of BuildClass.

The Metaclass associated with a class theory, or with a collection of class theories is enough to contain any static information about the theory, or to specify exactly what that information should be. Thus, we will introduce the empty stack in our stack metatheory.

Here’s what our metatheory looks like:

\[
\begin{align*}
\text{Metastack}_1 &= \text{Metaclass} \\
&\qquad \theta(\LoadIdentity) \\
&\text{pub Class } HLEP \text{ Elevate Emptystack} \\
&\qquad (c \equiv \text{Class}) \\
&\qquad (HLEP \equiv HLEP) \\
&\qquad (\text{elevate} \equiv \text{Elevate}) \\
&\qquad (\text{emptystack} \equiv \text{Emptystack}) \\
&\forall (s : c) (n : \text{int}) \\
&\text{axiom } c = \text{emptystack}.c\text{"push"int} \\
&\quad \land s\text{"push" : int }\rightarrow c \\
&\quad \land \text{emptystack"pop"} = [\text{nil}] \\
&\quad \land \text{elevate}(s\text{"push"n"pop"}) = \text{elevate}[n ; s] \\
&\quad \land \text{Emptystack : HLEp} \\
&\end{align*}
\]

The specification for any candidate stack implementation \(c\) says:

- \(c\) must be the \text{emptystack} and anything that can be derived by pushing integers to a \(c\) object.

- Any object in \(c\) has a \text{push} method that takes an integer and returns a \(c\) object.

- \text{popping} out of an empty stack gives \([\text{nil}]\).
• The LIFO property holds for all objects.

• The emptystack is a high-level element. It is proved thus, that any stack constructed by applying push several times to an empty stack, is also a high-level element.

We already have such an implementation, Stack5. The metaclass that describes Stack5 will be Buildclass g5. In addition we have to give an implementation for the empty stack. So the whole metatheory that describes our implementation will be:

\[
Metastack_5 = \begin{align*}
\text{BuildClass } g_5 \\
\text{Emptystack } \rightarrow \{\text{create } g_5 (\text{"list" } \rightarrow \text{[nil]})\}
\end{align*}
\]

And it is provable that

\[
Metastack_5 : Metastack_1
\]

which means that our implementation is valid. To prove this claim, notice that because both theories are derived from Metaclass, we can just prove that the additional axioms (those not in Metaclass) in Metastack5 imply the additional axioms in Metastack1. Let

\[
\begin{align*}
\text{emptystack} &= \sim Metastack_5 \text{"Emptystack"} \\
\text{elevate} &= \sim Metastack_5 \text{"Elevate"}
\end{align*}
\]
We have five axioms (whose conjunction is the axiom of $Metastack_1$) to prove.

$$Stack_5 = emptystack, Stack_5 \text{"push" int}$$

$$Stack_5 \text{"push" : int} \rightarrow Stack_5$$

$$emptystack \text{"pop"} = [nil]$$

$$\land \langle s : Stack_5 \rightarrow \land \langle n : int \rightarrow elevate(s \text{"push" n"pop"}) = elevate[n; s] \rangle \rangle$$

$$\{emptystack\} \in Metastack_5 \text{"HELP"}$$

The first axiom is proved, using equivalence transitivity:

$$Stack_5$$

$$\{\text{Restrictions of the hidden theory in } g_5 \text{ and total creation law}\}$$

$$= create g_5 \text{ ("list" } \rightarrow [\star \text{int}])$$

$$\{\text{Bunch union distribution}\}$$

$$= create g_5 \text{ ("list" } \rightarrow [\text{nil}]), create g_5 \text{ ("list" } \rightarrow [\text{int}][\star \text{int}])$$

$$\{\text{Definition of emptystack}\}$$

$$= emptystack, create g_5 \text{ ("list" } \rightarrow [\text{int}][\star \text{int}])$$

$$\{\text{The last operand equals Stack_5 \text{"push" int}\}\}$$

$$= emptystack, Stack_5 \text{"push" int}$$

To prove our claim that

$$Stack_5 \text{"push" int} = create g_5 \text{ ("list" } \rightarrow [\text{int}][\star \text{int}])$$
we utilize transitivity again. Let \( s \) be an element:

\[
\begin{align*}
    s : \text{Stack}_5 \text{"push" int} \\
    & \{ \text{Definitions, simplifications} \} \\
    = \forall \langle \text{list : [\ast int]} \rangle \forall \langle \text{n : int} \rangle \rightarrow s : \text{create } g_5 \ (\text{"list" } \rightarrow [\text{n}]+\text{list}) \\
    & \{ \text{Bunch union distribution} \} \\
    = s : \text{create } g_5 \ (\text{"list" } \rightarrow [\text{int}]+[\ast \text{int}])
\end{align*}
\]

which generalizing proves the desired expression.

The second axiom is proved by what we proved just above:

\[
\begin{align*}
    \text{Stack}_5 \text{"push" int} = \text{create } g_5 \ (\text{"list" } \rightarrow [\text{int}]+[\ast \text{int}]) \\
    & \{ \text{create-law and bunch containment transitivity} \} \\
    \forall \text{Stack}_5 \text{"push" int} : \text{Stack}_5 \\
    & \{ \text{Function inclusion} \} \\
    \forall \text{Stack}_5 \text{"push" : int } \rightarrow \text{Stack}_5
\end{align*}
\]

The third axiom is obviously true by definition of \textit{emptystack}.

We will prove the fourth axiom by proving it for arbitrary \( s \) and \( n \). Let \( s : \text{Stack}_5 \) and \( n : \text{int} \) be elements. Then there is an element \( \text{list : [\ast int]} \), such that

\[
s : \text{create } g_5 \ (\text{"list" } \rightarrow \text{list})
\]

Then:
Which means, again by theorems of elevation (bunch union distribution and idempotence):

\[\text{elevate}(s{\text{"push"}}n{\text{"pop"}}) = \text{elevate}[n;s]\]

which is what we wanted to prove.

Finally, the fifth axiom is proved by the definition of emptystack and the basic property of BuildClass.

We believe that by now the reader is convinced that the expressive power of metatheories is enough for many more things including:

- By containment of metatheories, inheritance of specifications.
- \(F\)-bounded polymorphism and other kinds of type quantification.
- W.Cook's notion of inheritance.

The idea is also perfectly recursive, one may define metametatheories etc.

One relation between Metaclass members that is worth mentioning is the subclassing relation. Subclass is a function that takes a metatheory \(M\) and returns the bunch of all metatheories that describe subclasses of the classes described by \(M\):
4.3.5 Parametric polymorphism

It should be obvious by now that reasoning about a class in a metaclass gives us the opportunity to deal with parametric polymorphism in a very general way, which definitely includes things like \( F \)-bounded polymorphism. For example, the metatheory of all types that are \( F \)-bounded by

\[
F t = \text{"leq"} \rightarrow t \rightarrow \text{bool}
\]

is

\[
FBounded = \{ S : \text{Metaclass} \rightarrow \text{let } t = \sim S\text{"Class" in } t : \text{\"leq"} \rightarrow t \rightarrow \text{bool} \}
\]

Unbounded universal quantification of types would be the bunch of all metaclasses \text{Metaclass}.

So should we want a generic stack of all possible types, we could parameterize our \text{MetaStack} specification into \text{GenericStack}. Suppose we want unbounded quantification (otherwise the parameter could be a sub-bunch of \text{Metaclass}, like \( FBounded \) above). Then the definition looks like that:
\[
\text{GenericStack} = \{ \begin{array}{l}
\text{let } m = \sim M\text{"Class"}
\begin{array}{l}
\text{let } mhlep = \sim M\text{"HLEP"}
\text{in } \text{Metaclass}
\begin{array}{l}
\text{pub } \text{Class } HLEP \text{ Elevate } \text{Emptystack}
\begin{array}{l}
\text{hid } (c =\sim \text{Class})
\text{HLEP} =\sim HLEP
\text{elevate} =\sim \text{Elevate}
\text{emptystack} =\sim \text{Emptystack}
\text{forall } (s : c) (n : m)
\text{axiom } c = \text{emptystack} \land c\text{"push" } m
\land s\text{"push" } m \rightarrow c
\land \text{emptystack}\text{"pop" } = [\text{nil}]
\land \text{elevate}(s\text{"push" } n\text{"pop"}) = \text{elevate}[n : s]
\land \text{Emptystack} : HLEP
\land mhlep : HLEP
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}\]

Of course, it's the same as Metastack, but parameterizing the type of the contents of the list, and forcing it to be a theory instead. Also one more axiom is enforced, that is the equivalence partition of the parameter type must be contained in the equivalence property of the stack, because this way we make sure that pushing a high-level element \(n : m\) results to a high-level element.

Two technical difficulties are made clear here. First, we didn't define simple data structures, like numbers and functions as theories, so we really cannot derive Metastack as a specific case of GenericStack, because \text{int} is not a theory. This is bad, because we could have avoided this problem in a more unified world where everything is a theory, and numbers too are defined as theories. We chose not to do it this way, so that we can utilize what was already there from bunch theory and now we pay the price for that design decision. To
solve this problem, we have two alternatives. We either go back and redesign theory-theory so that numbers and other theories are expressed in it, or we make a Java-like patch and define “wrapper theories” for our simple types, e.g. an *Integer* theory, a *Natural* theory etc. The first approach is more thorough and more satisfactory. However, in this thesis we are just going to pinpoint the need for further unification and leave it at that, which will save us the effort of defining numbers, operations and syntax for arithmetic from scratch.

The second problem is, we did have to write our *Metastack* theory again to parameterize it, even though in practice everything except *int* remains the same. This is the problem of *elaboration tolerance* which will be discussed further in section 4.3.7.

### 4.3.6 Multiple inheritance as conjunction

Multiple inheritance is such a thorny issue in OOP, because of its perceived complexity, that people involved in programming language design or research tend to avoid it altogether. In our framework multiple inheritance is very simple and in fact it has been already used several times in our examples before. Multiple inheritance is no more than intersection of two or more theories. This idea is semantically very clear to grasp.

A complication might arise by the fact that we usually have hidden theories, therefore two possible levels of conjoining axioms. They correspond to different things we want to do, so they are both welcome. Because we have shown many examples of conjunctions of theories (which corresponds to the shallowest level of multiple inheritance), we will only provide here an example of “deep” multiple
inheriting. Imagine a theory `Array` that gives array-like access facilities to a list (of integers again):

```plaintext
\[
\begin{align*}
g_{array} & = \langle \text{pub} \at \text{length} \text{ list} \\
& \quad \text{hid} \forall \text{axiom} \\text{list : \{int\}} \\
& \quad \quad \land \text{length} = \#\text{list} \\
& \quad \quad \land \at \{n : \text{int} \rightarrow \text{list n}\} \\
& \quad \quad - -
\end{align*}
\]

\[Array = \emptyset g_{array}\]
```

We can deeply-conjoin our `absaxiom` members, to get a new theory that will have both stack and array access to the data:

```
StackNArray = g_1 \land g_{array}
```

In our framework both ways of doing multiple inheritance are well defined mathematically, so there is no theoretical problem like "conflicting names". which happens when (at least) two of the classes to be inherited define the same member. Our definition would hold, even if \(g_{array}\) defined a member named `push`. In that case, the two axioms may or may not be in conflict with each other. When they are in conflict, then the defined theory is `inconsistent`. Otherwise, the defined theory is the biggest bunch of theories that satisfy both specifications for that name.

134
The two different ways of doing multiple inheritance are also a solution to another problem that complicates this issue. Many times in a multiple inheritance relation, the two parents $A$ and $B$ have a common ancestor $C$. When this happens, it is unclear in OOP languages whether we should consider a single occurrence of the data values of $C$ for both classes $A$ and $B$, or whether each of them should have its own separate $C$-values. Because usually the data structure is stored in the hidden theory, the first would be implemented by a deep conjunction of $abs axiom$ members defining the theories, while the second would correspond to a mere intersection of the theories. An example of that would be a theory:

$$A \text{StackAndAnArray} = \text{Stack}\_1\text{Array}$$

which contains a stack and, independently, an array. Notice that the two cases are not equivalent: the first is stronger than the second.

4.3.7 Generalization

Generalization of a class or classes is another important feature that OOP languages do not support, as already mentioned before. It can be argued that every major software design revision may include certain generalizations of classes. Given that the way software development is done in a traditional OOP language (that is, refinement-oriented) these generalizations cannot be introduced later on in the design. We have to either re-design or to make ad hoc patches.

To make our conversation more concrete, let's say that a software package
contains a certain class $C$ that implements a specific operation of the program (say a method $f$ within the class). We now decide to make a new version of our package with a considerable amount of new operations. We can have the same interface for the operations, that is, each corresponds to a class that exposes a method $f$.

In current OOP languages, there is no way to create a superclass $O$ of all such operations, without altering the already written code for class $C$, because we must specifically say in $C$'s definition that $C$ inherits $O$. This may be troublesome in many ways. If we don't do it however, there is no way for us to have $O$ and use it for example for inclusion polymorphism: there is no way to get a function that takes a generic $O$ object and calls its particular $f$ method.

In our understanding of OOP however, this augmenting generalization is possible, since inheritance is a well-defined mathematical relation. We can create $O$ and prove that $C : O$, which means that without touching existing specifications we create a superclass of a certain class. The proof may (or may not) be very obvious, for example defining $O$ to be $C, D$ trivially makes it a parent of both $C$ and $D$.

We will now give an example of generalization of one of our stack theories. Before that, we define the Unary meta-theory. This meta-theory describes a theory with only one high-level element:
Unary = BuildClass( --
    pub
    hid
    forall
    axiom true
    -- )
: "Class" → {θ}
| "Elevate" → {θ → θ}

Unary describes theory θ whose only high-level element is θ.

UnaryStack will be a stack of Unary theories:

UnaryStack = GenericStack Unary

which is not terribly interesting since it is a stack where no useful information

We can generalize UnaryStack into a class UnaryStore, by dropping one of

can be stored (except for the number of θ theories in it).

demand that emptystack"pop" be [nil]:

UnaryStore = Metaclass
    θ( --
      pub Class HLEP Elevate Emptystack
      hid (c =~ Class)
         (HLEP =~ HLEP)
         (elevate =~ Elevate)
         (emptystack =~ Emptystack)
      forall (s : c)
      axiom c = emptystack, c"push"θ
           ∧ s"push" : θ → c
           ∧ elevate(s"push"θ"pop") = elevate[θ; s]
           ∧ Emptystack : HLEP
           ∧ {θ} : HLEP
    -- )
where, obviously, \textit{UnaryStack} : \textit{UnaryStore}. Now adding a new axiom to \textit{UnaryStore}, we create a “cousin” theory for stacks, which we call \textit{Semaphore}:

\begin{verbatim}
Semaphore
  = UnaryStack
    \[\Theta(---
       \begin{array}{l}
         \text{pub} \quad \text{Elevate Class} \\
         \text{hid} \quad (c \leftarrow \text{Class}) \ (\text{elevate} \leftarrow \text{Elevate}) \\
         \text{forall} \quad (s : c) \\
         \text{axiom} \quad \text{elevate}(s"\text{pop}""\text{push}"\theta) = \text{elevate} s \\
        \end{array}
    )
\]
\end{verbatim}

That tells us that a semaphore is a stack that can also “increase negatively”.

We “discovered” formally (although admittedly awkwardly) a semantic “relationship” between stacks and semaphores and this relationship (even though it is not clearly a parent-child relationship) made it into the formalism, after we created stacks. This example of course is neither impressive nor particularly useful, but it will do to demonstrate the plausibility of augmenting generalization in our framework.

An interesting and difficult problem that manifests itself now is \textit{elaboration tolerance} (see [25]). While it is (most of the time) relatively easy to reuse specifications for augmenting specialization, that is not true for generalization. We would like to have a good way to generalize a class without having to rewrite it from the beginning. There doesn’t seem to be an easy way to do that in the general case, as we’ve also seen in section 4.3.5.

Coming up with techniques that will give good elaboration tolerance in our specifications (especially \textit{generic} techniques that do that) is an open problem. Our “theory theory” has some abstraction tools that may help in some cases.
such as bunch union, name abstraction, projection and elevation.

4.3.8 References

In our analysis so far, as is the general trend in OOP semantics, we avoided having mutable objects. Moreover, we advocated the separation of variables/states from objects/values and we would also like to see that in a programming language. That is, unlike other OOP formalizations, we don't consider immutable objects to be an abstraction used to simplify the mathematics, but we rather prefer to view it as a permanent design decision in an OOP language. In that sense we are in agreement with [4].

However, this is a heretic approach to OOP and some will disagree with it. Objects are commonly understood as having state and our idea of objects as values may seem too restrictive to be useful. The problem is not evident in simple cases such as the stack example presented above. The simplicity of this example lies on the fact that the object can mimic state by returning its new state after a method is called. This is a universal solution for objects that are stand-alone, such as the stack and they don't refer to other objects that may change independently.

What happens if we have a referenced object within another object? In general that shouldn't be hard to handle either in our framework, because we can consider the referenced object as a part of the referring object. To change the referenced object, one should access it through the referring object.

But the difficulty arises when we have to refer to some "global" state from many different objects, where clearly many objects will have to change as a
result of a change of this state. To give a concrete example, imagine that we want to create a LimitedStack theory, that includes all theories that behave as stacks as long as their size is less than or equal to limit, where limit is a natural value that might globally change for all LimitedStack theories.

To model this situation with immutable objects, we must create a wrapper theory that is going to contain this global information. Change of limit will result in another instance of this wrapper theory, exactly the same way pushing a value resulted in another instance of a stack theory:

\[
\text{WrapperStack} = \Theta \langle \ldots \\
\text{pub} \quad \text{limit LimitedStack} \\
\text{hid} \quad (\text{limitedStack} = \sim \text{LimitedStack}) \\
\text{forall} \quad \text{axiom} \\
\quad \text{limit : nat} \\
\quad \wedge \quad \text{limitedStack} = \text{Stacks}('size' \to 0..\text{limit}+1)) \\
\ldots \rangle
\]

Introducing assignment in our language denoted by :=, we can have a variable \( S \) of type \( \text{WrapperStack} \) and a variable \( x \) of type \( \sim S\text{LimitedStack} \). Changing the limit, would be a command like:

\( \text{S"limit" := something} \)

Of course, for this program to type-check, we must prove that \( x \) will still be a sub-bunch of \( \sim S\text{LimitedStack} \). More details on types as invariants are given in chapter 5.

To handle references, we wrap all information that changes when the referenced value changes within a theory. This way we can have a language in
which the only way to change the state is via assignment (no side-effects, no parameters by reference). This results in an imperative-style language that also manages to respect referential transparency.

4.3.9 Undecidability and expressiveness

This final comment concerns the practicality of our framework. It could be argued that such a view of OOP, blatantly ignoring the undecidability barrier, is not realistic and cannot be used to design a programming language to be used in practice. (Most) existing programming languages have decidable type systems, which cannot guarantee to express everything we want to know about the entities of the problem, but they can certainly guarantee termination of the type-checking process with a definite answer. In our theory theory semantics we went too far: not even considering the “object invariant” idea, even something as common as the inheritance relationship must be proved to hold between two theories. Doing it so, gives us the expressive power to generalize theories and do other interesting things, but one may reasonably wonder if this is really worth trespassing on the undecidability frontier.

What undecidability means, is that there exist instances of the problem that cannot be solved. Undecidability is a characterization of how hard the problem can be at its worst. It doesn’t mean that the problem in the average case is very hard. In our examples, the proofs were quite trivial and that’s what we expect them to be in most of the cases. Our framework is general enough to contain decidable sublanguages, should somebody want to stick to what is guaranteed to work. Of course a theory theory based language, should be able to cope with
its undecidability, with a good prover that would automate the proof of usual cases.

Apart from these objections, one may think that our framework is much more general than what we usually need in a programming language. The whole "object invariant" idea seems exaggerated. What we are doing here, is trying to unify the specification language with the programming language. We end up with one language for both design and programming.

A similar attempt to introduce and check various specifications within programs is the Extended Static Checkers (ESC) for Modula-3 [22] and Java [23] created in Compaq. The difference of this approach is that a new specification language is built for an existing programming language. The fact that the programming language is created before without any provision for the specification languages somewhat restricts the usability of ESCs. Two problems evident with ESC-Java are:

- Because Java's methods do have side-effects and because objects have state, it is impossible to invoke function names within the specification language.

- Specifications are attached to methods. There is no way to express object invariants like the stack properties, even though Java is an OOP language, and therefore the proper modularization of specifications should be done in terms of objects and not in terms of methods.

A language created based on our theory-theory framework could encompass the
ESC technology without these problems. Therefore this framework can prove a very useful approach of the static checking problem in practice.
Chapter 5

A theory theory language

In this chapter, we sketch the design of a language based on our theory theory analysis of OOP. The language is named Alpha. We do not delve into very concrete details about the syntax (or even the semantics of the language). We abstract away all these design decisions, to describe the basic ideas on which this language is to be built. The language is influenced by ProTem [18], an imperative specification and programming language based on bunch theory.

5.1 Expression sub-language into ASCII

This section describes the conversion of basic bunch theory notation to ASCII characters. The following operators are used in Alpha:

- Parentheses stand for themselves. $<$ $>$ replace the $\langle\rangle$ notation. Curly and square brackets stand for themselves. $\langle -$ $-$ $-$ $\rangle$ is also imported as $<-- -->>$.

- $+$ $-$ $*$ $/$ $#$ $\ldots$ $;\ldots$ $,\ldots$ $|$ $'$ $"$ $:$ $= <$ $>$ are used in the same way they are used in bunch theory. $+$ is overloaded to also denote $^\dagger$. 

144
• Multiplication $\times$ is denoted $\times$.

• if then else and let in notation is borrowed.

• $\land \lor \leq \geq \neq$ are respectively denoted by $\land \lor \leq \Rightarrow \neq$.

• $\rightarrow$ denotes $\rightarrow$ and $:\rightarrow$ denotes $\rightarrow$.

• Keywords replace other notation: $\cap \cup \subseteq \notin \emptyset \theta \theta$ are respectively denoted by $\text{intersection union in subseteq powerset solutionTheta theta}$.

• Keywords qu and qub followed by the appropriate operator in parentheses denote the respective quantifier and bunch quantifier.

• Identifiers nil and null are used in Alpha as keywords that describe nil and null of bunch theory.

5.2 Defining theories and metatheories

In Alpha everything exists within theories. To create a theory, we basically define the related axiomatic, using a notation similar to the $\langle -- -- \rangle$ notation introduced in chapter 3.

The following code defines the $^*g_1$ axiomatic:

```
^*g1
= <<<--
pub push pop
hid list
forall
```
axiom list:[[int]

/\ push:<<n:int->construct("list"->[n]+list)>>

/\ pop: if list=nil then [nil]

  else [list 0;construct("list"->[list"tail"])]

-->>

and then the theory Stack1 will be:

"Stack1 = Theta("gl")

We add ", to make gl and Stack1 sets, because they themselves belong in an outer theory and we want to prevent bunch union distribution. Words public and hidden are also reserved and used within the axiom the way we were using them in chapter 3.

A metatheory is a special kind of theory that contains a Class member, that describes another theory. A metaclass also contains a high-level elementhood for that theory. Keyword BuildClass can be used to define a metaclass out of an abs axiom:

"Metastack1 = BuildClass("gl")

Of course, we can always create metatheories as other theories.

Notice that we use "special" names such as int and "tail". In fact, number and list theory should be just theories defined within the standard libraries of Alpha and globally available. "tail" is just a method of the list theory and the special notation we use in Alpha to make our number theory look like conventional arithmetic should be considered just syntactic sugar.
Names $g_1$, $Stack_1$ and $Metastack_1$ belong to another exterior theory, since everything is a theory. Using an identifier within the list of public, hidden or universally quantified names, makes it local. The entities of the outer scopes that have the same name are still accessible if we precede the identifier with one or more ! characters. For each exclamation mark, the next outer entity is referenced.

### 5.3 Generalized bunch containment notation

The expression $A:B$ means that $A$ is a sub-bunch of $B$. Sometimes we also want to say that $A$ is not a bunch-theory element, but that it belongs to a higher level elementhood $H$. We want to say that there is an element $b$ such that $b:B$ and $A=("H" Elevate")b$. The *generalized bunch containment notation* means exactly this:

$$A \in_H B$$

means

$$\text{qu}(\backslash / ) \ll b:B \Rightarrow A=("H" Elevate")b \gg$$

provided of course that

$$B="\text{qub(union)}("H" \text{HLEP})$$

For example, let a high-level elementhood $H$ such that

$$H" \text{HLEP}"=\{0,1,2\},\{3,4,5\}$$

and suppose that we claim that
AGH: 0, \ldots, 6

That last expression is equivalent to

\[ A = 0, 1, 2 \lor A = 3, 4, 5 \]

Should we want to define a name within a theory this way, we should precede it with the, to wrap the high-level element into a set. Otherwise, the name of the theory has to be an element and the theory ends up being inconsistent.

Because a metaclass \( M \) always contains information about both a class \( C \) and a high-level elementhood \( H \) concerning that class, we can use the ofClass operator to abbreviate the fact that an expression is a high-level element of \( H \) and a sub-bunch of \( C \).

\( A \) ofClass \( M \)

abbreviates

\( \text{AG}(M\"HLEP\") \cdot \text{\"M\"Class}\)

The as operator abbreviates elevation:

\( A \) as \( M \)

abbreviates

\( (\text{\"M\"Elevate})A \)

Generalized bunch containment notation may also be used in function parameters. Suppose that we want a function \( f \) that takes a parameter \( p \) of type
C and of high-level elementhood H. In our mathematical notation we wrap the parameter in a set and then we extract it and elevate it within its body:

\[ f = \langle P : \text{setof}.M \to \text{let } p = (\sim H^"\text{Elevate}")(\sim P) \text{ in } \ldots \rangle \]

In Alpha we are allowed to declare the parameter as a set and use generalized bunch containment in its definition:

\[ f = \langle\langle p \rangle\rangle. \text{p@H:M } \to \ldots \rangle \]

and the \texttt{ofClass} notation is of course also allowed.

## 5.4 Imperative sub-language

To add imperative code within Alpha expressions we use the \texttt{do result} notation. The \texttt{do} keyword is followed by optional local variable declarations, then a period and then imperative code and then the \texttt{result} keyword that is followed by an expression. What happens when such an expression is evaluated, is: first the code is executed and then the value of the expression after \texttt{result} is returned. There are no side effects after the execution of the code and the variables declared are no longer visible.

Imperative code is a statement. A statement can be:

- An assignment statement: \( V := E \) where \( V \) is a string of variables (possibly containing lists) and \( E \) is an expression

- A sequence of statements. The sequential operator is \( . \)

- A conditional construct: \texttt{if} \( B \) \texttt{then} \( S_1 \) \texttt{else} \( S_2 \)
- A loop construct: `while B do S`

- An `ok` statement that does nothing

- A statement within parentheses (to alter precedence for use within conditional constructs, and loops)

We don’t include parallelism, message passing and input-output in these commands, but we expect to do so in a more detailed design of Alpha.

### 5.5 Non-deterministic variable declaration

Variable declaration in Alpha starts with the declaration of names. First the keyword `var` is written followed by the names of all local variables separated with spaces. The names should be non-keyword identifiers. This declares the names as local, within the scope of the imperative code. To see the exterior entity a name represents, one uses the exclamation mark syntax again. The declaration of names says nothing about the type of the variables. This is the purpose of the invariant following the declaration in parentheses. This could be used for simple type declarations, but an even more complicated expression, the `invariant`. Local variable values must always make the invariant evaluate to true, that is they belong to a given theory. This theory shouldn’t be inconsistent.

This manner of declaring variables allows for non-determinism. Consider the following program section:

```claration
  while B do S

do
  var n (n:bool). X
```
result ...

This means that in code X, variable n is either true or false and this is non-deterministically so from the beginning of the execution of X to the end. (To make n be non-deterministically any sub-bunch of bool, we need to use set structure again, because variables declared are elements. Set braces are allowed in the list of names following var.) The result of the expression will be the bunch union of the results for all possible values of n.

Now let's suppose that we have a SortingFunction theory that describes all functions that sort integer lists (see chapter 3). The following code returns the sorted version of a list L, without specifying anything more than the fact that we use a sorting function.

```
SortedL = do var {SF} (SF:SortingFunction) .
result SF"f"L
```

This is written in imperative code but it is abstract in a the way that it doesn't specify exactly how the action is to be done. If we have a more specific sorting function theory MergeSort then this code

```
MergeSortedL = do var {SF} (SF:MergeSort) .
result SF"f"L
```

is a refinement of the previous code. Evidently, MergeSortedL:SortedL.
5.6 The prover

Alpha needs a prover to do almost everything. The prover is to be helped by the programmer to prove that the program satisfies the specifications by building a whole theorem database. Each theory has its own collection of interesting theorems that can be further used in other proofs. To force a prover to prove a theorem about a theory \textit{(without including it as an axiom)} the keyword \texttt{prove} is used at the proper scope. For example

\begin{verbatim}
fact = <<<n:nat->if n=0 then 1 else n><fact(n-1)>>
prove qu(\forall)<<<n:nat->fact n=qub(><)(1,..n+1)>>
\end{verbatim}

Syntactically the \texttt{prove} clause should be part of the expression, that is in BNF notation our expression sub-language \( X \) is given by \((E \) denotes a simple expression, without a \texttt{prove} clause):

\[
X ::= E|E \texttt{'}prove' \ X
\]

The way these theorems are classified within their corresponding theories and used within formal proofs is not clear. Much research is needed in this topic.

5.7 Alpha as a multi-level language

In this chapter we presented Alpha as a specification and programming language based on theory-theory. Our main observation and idea at this premature stage of design is the fact that specification and code are unified within one language.

152
In fact we show examples where even imperative code can be further refined and therefore treated as formal specifications.

Taking this idea a bit further, we can consider Alpha as potentially a multi-level language, in the sense that one can program both in high and low level and we can also have programs of varying levels, according to the needs of their modules. We can take low level Alpha down to machine code and strive to define various possible implementations of higher-level operators in machine code instructions using Alpha refinement. This, among others, will give the possibility of configurable compilers, because even code optimizations and code generation schemes can be thought of as libraries of implementations of high-level operations.
Chapter 6

Conclusions

In this thesis we developed theory theory, used it to express OOP entities as theories and then used all this formal construct to design a programming language. Throughout the task, we argued in several points about the usefulness of our approach. Ending this presentation, we gather all these arguments to summarize our philosophy and our contributions.

Theory theory's basic provision is modularity. Theories are expressed as individual mathematical objects, therefore one can reason about more than one theories at once. That makes treatment of theories more systematic than that of [16]. An early attempt to incorporate this modularity into mathematics was Burstall’s [6] but it wasn’t complete in the sense that theories were not considered “first class citizens” (in our framework, theories are unified with functions, therefore they are first class citizens).

Z [30] and B [1] are specification languages that also emphasize modularity of axioms. We find theory theory to be very similar to these two approaches in terms of goals but we also consider it superior in terms of unification and
simplicity. Unification of similar ideas into one and only mathematical image is stressed all over this work as a very important property of a formalism. Unification in general is good practice for mathematical economy and to discover new possibilities (and this especially shows in our treatment of OOP). In theory theory, it offers the capability to use the same language for both design (specifications) and programming (code).

Unification is the main theme even when we incorporate theory theory to describe OOP. We show that classes and objects are unifiable resorting to the bunch idea once more. We also show that they are also describable by formal theories, therefore we unify three different things into one. Our OOP semantics provides a good mathematical description, taking care of semantic difficulties. Subtle issues such as inheritance and overriding are resolved by expressing inheritance and instantiation with bunch inclusion. Known kinds of polymorphism (even F-bounded polymorphism) are supported. We firmly believe that the answers we give to the problems of OOP are made possible through unification.

The idea of incremental generalization is another example of discovering how to fill in semantic gaps, when similar concepts are expressed in one single way mathematically. Because inheritance is modelled as bunch inclusion, it is possible for the user of our model to build their design “backwards”, that is from more specific to more general classes. We have argued that this seems to be a great plus for software engineering.

Because we model objects and classes with theories, we have a far more expressive way to describe the “type” of a class member. In fact, type declarations
are in our framework a special and limited kind of axiom. We are interested in object invariants, that formally but a lot more precisely specify properties of the objects and classes. This is different in several ways from current (even implemented: [22], [23]) ideas for automated verification, first because it advocates creating the specification and programming language together (better even: create one language that does both jobs) and second because it puts OOP specifications where they should be (specifications describe classes and objects, not single methods).

Those benefits from increasing the expressiveness of the specification and programming language come with the price of undecidability. We chose to ignore the matter. We allow our language to be as interesting as possible and we accept that not all valid programs are ever going to be verified. We favor proving facts (such as inheritance, types of variables etc.) as opposed to just declaring them, with the faith that the task shouldn't be hard in well-made programs or designs.

Our last comment should be on the "multi-level" potential of Alpha, that hasn't been explored in depth in this thesis, but seems to emerge as a very nice property, if well-exploited. Because Alpha can operate on both very high (specification) and very low (possibly machine code) levels, it opens the way for interesting research, such as highly configurable compilers (a programmer can decide on the exact machine code implementation of certain high-level constructs, or when and how the code should be optimized), but it also gives the opportunity for a programmer to operate on various levels of implementation, depending on the criticality of performance, correctness and maintainability of
each piece of code.

We therefore hope with this thesis to positively contribute and to open new lines of research in the fields of programming languages, automated verification and software engineering. There are open problems to be addressed and future work to be done in various directions. The next step should be a refinement of the design of Alpha (perhaps even an implementation), which will definitely reveal many problems that didn't manifest themselves during our theoretical analysis. Something like that was of course out of the scope of this thesis.
Appendix A

Operators and precedence

Precedence

The following list contains all operators used in this thesis in order of decreasing precedence:

- () {} \{ \} names constants application

- Unary operators: \(- \ + \ * \ \sim \ \mathcal{D} \ \mathcal{U} \ \#\), universal domain in restricted domain functions and \(\to\) \(\to\)

- Quantifiers

- \(\times \ \div \ \cap \ \&\)

- \(\pm \ \mp \ \ominus \ \oslash \)

- \(; \ \ldots ; \)

- \(, \ \ldots |\)

- \(= \ \neq \ \leq \ \geq \ : \ \in \ \subseteq \)
• \( \land \)

• \( \lor \)

• if then else let in

• \( \geq \leq \)

**Associativity**

Right associative: all unary operators

Left associative all other operators except for

\[ = \neq \lt \leq \gt \geq : \in \subseteq \implies \geq \leq \]

For any two of these operators \( \oplus \) and \( \otimes \), association works like that:

\[(a \oplus b \otimes c) = (a \oplus b) \land (b \otimes c)\]

so for example \( a = b = c \) really means \( a = b \land b = c \).

Also \( \land \) is not left-associative. See details in section 3.3.7.
Appendix B

Theory laws and proofs

Notation

In the following: $T$ and $U$ are theories ($\theta$ members), $f$ and $f_i$ for all natural $i$ are axioms ($axiom$ members), $g$ and $g_i$ for all natural $i$ are axioms with abstraction ($absaxiom$ members), $tz$ and $tz_i$ for all natural $i$ are sub-bunches of $text$, $tt$ is a theory transformation. $eq_g$ is the high level equivalence relation introduced by $g$ for any $g : absaxiom$. Laws are implicitly universally quantified over those names.

Laws

Bunches

Theory

$$\theta = \text{textall}f$$

Axiom

$$axiom = s(\theta \rightarrow \text{bool})$$
Axiom with abstraction

\[ \text{abs axiom} = \sigma(\theta \rightarrow \text{axiom}) \]

Basics
Top/bottom

\text{inconsistent} : \top : \top

Conjunction

\[ \langle f_1, f_2 \rangle = \langle s : \theta \rightarrow f_1 s \land f_2 s \rangle \]

Disjunction

\[ \langle f_1, f_2 \rangle = \langle s : \theta \rightarrow f_1 s \lor f_2 s \rangle \]

Containment

\[ \langle f_1 : f_2 \rangle = \langle s : \theta \rightarrow f_1 s \geq f_2 s \rangle \]

\( \Theta \) definitions

\[
\begin{align*}
  \Theta g \\
  &= \langle s : \theta \rightarrow \langle g s \rangle \neq \text{null} \rangle \\
  &= \langle s : \theta \rightarrow \lor(g s) \rangle
\end{align*}
\]
Laws for create

Definition

\[
\text{create} = \langle g : \text{abs axiom} \rightarrow \langle T : \theta \rightarrow \Theta \langle p :: \theta \rightarrow \langle h :: \theta \rightarrow g \ p \ h \wedge h : T \rangle \rangle \rangle \rangle
\]

The create-law

\[
\text{create } g \ T : \Theta \ g
\]

\(\Theta\text{-create-law}\)

\[
T : \Theta g = \bigvee \langle H : \text{set of } \theta \rightarrow T = \text{create } g (\sim H) \rangle
\]

"Total" creation

\[
\text{create } g \ \theta = \Theta g
\]

Obtaining the axiom

Definition

\[
\text{axiom of } T = \langle s :: \theta \rightarrow s : T \rangle
\]

§-inverse property-I

\[
\S(\text{axiom of } T) = T
\]
§-inverse property-II

\[ \text{axiomof}(\xi f) = f \]

Θ-law

\[ \text{axiomof}(\Theta g) = \langle s :: \theta \to \forall (g s) \rangle \]

Name abstraction
Definition

\[ \text{abstract } T tx = \Theta(p :: \theta \to \langle h :: \theta \to \text{axiomof } T(\text{project } h tx|p) \rangle) \]

Abstraction law

\[ T : \text{abstract } T tx \]

Deep conjunction
Definition

\[ g_1 \land g_2 = \Theta(p :: \theta \to \langle h :: \theta \to g_1 p h \land g_2 p h \rangle) \]

Deep conjunction law

\[ g_1 \land g_2 : \Theta g_1 \land \Theta g_2 \]


create as deep conjunction

\[ create \ g \ T = g \land \theta \rightarrow \langle h : \theta \rightarrow h : T \rangle \]

Transformation

Deep transformation law

\[
\begin{align*}
\exists (a : Dtt \rightarrow eqg a \{ inv \ tt \ (tt a) \}) \\
\forall (a : Dtt) \rightarrow create \ g \ a = create(p : \theta \rightarrow \langle h : \theta \rightarrow g \ p (inv \ tt \ h) \rangle)(tt \ a)
\end{align*}
\]

Projection

Definition

\[ project \ T \ tz = \langle x : tx^{\mathcal{T}T} : text \rightarrow T \ x \rangle \]

Projection law-I

\[ T : project \ T \ tz \]

Projection law-II

\[ T = project \ T \ text \]

Mutual projection

\[ T : U \geq project \ T \ tz : project \ U \ tz \]
Proofs

This section provides formal proofs for the laws stated above. Laws labeled “definitions” are not proved, because they are axioms. Also: bunches laws and the top/bottom law are axioms and they are not proved here.

Basics

The “Basics” section contains laws that are instances of laws in [16]. The $\Theta$ definitions are axioms and their consistency is established by the law that relates $\|$ with $\lor$ in section 2.7.4.

The create-law

\[
create \; g \; T : \Theta g \\
\{ \text{Bunch containment and domain law} \} \\
= \land \langle z : create \; g \; T \rightarrow z : create \; g \; T \geq z : \Theta \; g \rangle
\]

The body of the quantification may be proven:

\[
x : create \; g \; T \\
\{ \text{Definition of create} \} \\
= \lor \langle h :: \theta \rightarrow g \; x \; h \land h : T \rangle \\
\{ \text{Generalization} \} \\
\geq \lor \langle g \; x \rangle \\
\{ \Theta \; \text{definition} \} \\
= x : \Theta g
\]

By transitivity, the body of the quantification is equal to $true$, thus by equational transitivity:
create g T : Θg
\{Equational transitivity\}
= \land (create g T \rightarrow true)
\{Universal quantification laws\}
= true

QED.

The total creation law is immediate consequence of the definitions of create and Θ.

The Θ-create-law

\begin{align*}
T : Θg
\{Total creation\}
= T : create g Θ
\{Properties of ~\}
= T : create g (\sim \{Θ\})
\{Generalization\}
\geq \lor \langle H : setof Θ \rightarrow T = create g (\sim H) \rangle
\end{align*}

\begin{align*}
\lor \langle H : setof Θ \rightarrow T = create g (\sim H) \rangle
\{create-law, transitivity\}
\geq \lor \langle H : setof Θ \rightarrow T : Θg \rangle
\{Quantification of a constant\}
= T : Θg
\end{align*}

By transitivity and antisymmetry, QED.

The $\$-$inverse properties

The first property is proved by equivalence transitivity:
\(\$\) \text{axiomof } T
\{\text{Definition}\}
= \$\langle s : \theta \rightarrow s : T \rangle
\{\text{Solution of containment property}\}
= \theta^T 
\{\text{Inclusion (because } T : \theta)\}
= T

QED.

Similarly, for the second one,

\text{axiomof} (\$f)
\{\text{Definition}\}
= \langle s : \theta \rightarrow s : (\$f) \rangle
\{\text{Solution properties}\}
= \langle s : \theta \rightarrow f s \rangle 
\{\text{Because } \mathcal{D}f = \theta \text{ and by definition of } \langle \rangle \text{ notation}\}
= f

QED. The \(\Theta\)-law is direct consequence of \text{axiomof} and \(\Theta\) definitions.

\textbf{The abstraction law}

First we prove a lemma:

\(\land \langle s : \theta \rightarrow s = \text{project } s \ tx \mid s \rangle\)

Indeed, by the fact that \(tx : \text{text}\), we have that

\(\mathcal{D}(\text{project } s \ tx \mid s) = \text{text}\)

which means:
\[
\text{project } s \ext{ t } z | s
\]
\[
\begin{aligned}
\text{Definition of } | \\
= \langle t : \text{text} \to \text{if } t : \text{t } z \text{ then } s t \text{ else } s t \rangle \\
\text{Case absorption and definition of } ()
\end{aligned}
\]
\[
= s
\]

By transitivity and generalization the lemma has been proven.

Now let \( f = \text{axiom of } T \). We need to prove \( s : T \geq s : \text{abstract } T \ext{ t } z \)
which will give us the law by generalization:

\[
\begin{aligned}
(s : T \geq s : \text{abstract } T \ext{ t } z) \\
\text{Definition of } f, \text{abstract and } \Theta \text{ and } \Sigma^-\text{-inverse properties}
\end{aligned}
\]
\[
= f s \geq \bigvee \langle h : \theta \to f(\text{project } h \ext{ t } z|s) \rangle \\
\text{Lemma}
\]
\[
= f(\text{project } s \ext{ t } z|s) \geq \bigvee \langle h : \theta \to f(\text{project } h \ext{ t } z|s) \rangle \\
\text{Generalization law for } \bigvee
\end{aligned}
\]
\[
= \text{true}
\]

QED. by transitivity.

**Deep conjunction laws**

\[
g_1 \land g_2 : \Theta g_1 \land \Theta g_2
\]
\[
\text{Bunch containment}
\]
\[
= \land \langle x : g_1 \land g_2 \to x : \Theta g_1 \land \Theta g_2 \rangle
\]

To prove this, let \( z : g_1 \land g_2 \). We will prove that \( x : \Theta g_1 \land \Theta g_2 \) which by generalization gives the law:
\[ x : g_1 \land g_2 \]
\[ = \forall (h :: \theta \rightarrow g_1 \land g_2 \land x \cdot h) \]
\[ \{ \text{Definition of } \lambda \} \]
\[ \supset \forall (g_1 x) \land \forall (g_2 x) \]
\[ \{ \text{Splitting laws} \} \]
\[ = x : \Theta g_1 \land x : \Theta g_2 \]
\[ \{ \text{Definition of bunch intersection} \} \]
\[ = x : \Theta g_1 \cap \Theta g_2 \]

QED. by transitivity.

\( create \) expressed by \( \lambda \) is just another writing of the definition of \( create \).

The deep transformation law

First we need to prove a lemma:

\[ create g (inv tt (tt a)) = create (p :: \theta \rightarrow (h :: \theta \rightarrow g \cdot p \cdot (inv tt h)) \cdot (tt a)) \]

which we’ll do by transitivity of equality and starting from the right side of the equation:

\[ create (p :: \theta \rightarrow (h :: \theta \rightarrow g \cdot p \cdot (inv tt h)) \cdot (tt a)) \]
\[ \{ \text{Definition of } create \} \]
\[ = \Theta (p :: \theta \rightarrow (h :: \theta \rightarrow g \cdot p \cdot (inv tt h)) \land h : tt a) \]
\[ \{ \text{Change of domain} \} \]
\[ = \Theta (p :: \theta \rightarrow (h : tt a :: \theta \rightarrow g \cdot p \cdot (inv tt h))) \]
\[ \{ \text{Change of variable} \} \]
\[ = \Theta (p :: \theta \rightarrow (h' : a :: \theta \rightarrow g \cdot p \cdot (inv tt (tt h'))) \]
\[ \{ \text{Change of domain and definition of } create \} \]
\[ = create g (inv tt (tt a)) \]

Then the proof goes by transitivity of equivalence:
\begin{align*}
&\land \langle a : Dtt \to eq_{g} a (inv tt (tt a)) \rangle \\
&\quad \{ \text{Definition of } eq_{g} \} \\
= &\land \langle a : Dtt \to create g a = create g (inv tt (tt a)) \rangle \\
&\quad \{ \text{Lemma} \} \\
= &\land \langle a : Dtt \\
&\qquad \to create g a \\
&\qquad \quad = create (\langle p :: \theta \to \langle h :: \theta \to g p (inv tt h) \rangle (tt a) \\
&\quad \} \\

\text{The change of variable mentioned above goes like this in general:}

&\lor \langle x : f a \to g(f x) \rangle \\
&\quad \{ \text{Bunch containment, domain change law} \} \\
= &\lor \langle x : f a \to \lor \langle y : a \to g(f x) \land x = f y \rangle \rangle \\
&\quad \{ \text{Commutative law, transparency, one-point law} \} \\
= &\lor \langle y : a \to g y \rangle \\

\text{Projection laws}

\text{Projection laws are direct consequences of the function inclusion axiom and the projection definition.}
Appendix C

High level elementhood theory

This appendix formally proves theorems of the $HLE_3$ theory, which of course directly apply to the derived $HLE$ theory. In the following

$$H : HLE_3$$

$$elevate \models \neg (H \text{"Elevate"})$$

$$HLEp \models \neg (H \text{"HLEP"})$$

The laws we are presenting are implicitly universally quantified over these variables:

$$x, y : \bigcup b \ HLEp$$
First notice that \( \text{elevate} \ x \) is defined as \( \sim A \), where

\[
A = \{ X : HLEp \rightarrow x \in X \}
\]

For this function to be well defined, we need to show \( \text{elem} \ A \). That is, we have to show that for any elements \( S \) and \( P \):

\[
S, P : A \supseteq S = P
\]

The proof uses transitivity:

\[
S, P : A \\
\left\{ \text{Definition of } A \right\} \\
= S, P : HLEp \land x \in S \land x \in P \\
\left\{ \text{Axiom of the } HLE_3 \text{ theory and set theory} \right\} \\
\supseteq (S = P \lor S \cap P = \emptyset) \land x \in S \cap P \\
\left\{ \text{Set theory and boolean algebra} \right\} \\
\supseteq S = P
\]

which means \( \text{elem} \ A \) so \( \sim \) can be applied to \( A \) and \( \text{elevate} \) is well defined.

The fundamental property of equivalence partition is reflected on \( \text{elevate} \) thus:

\[
\text{elevate} \ x = \sim X = x \in X
\]

The proof is again by transitivity:
\[ \text{elevate } x = \sim X \]
\[ = \{ \text{Definition of } \sim \} \]
\[ = \{ \text{elevate } x \} = X \]
\[ = \{ \text{Definition of } \text{elevate} \text{ and } \sim \text{ and } \mathcal{E} \} \]
\[ = x : \bigcup \mathcal{B} \mathcal{H} \mathcal{LE} \mathcal{P} \land x \in X \]
\[ = \{ \text{Assumption of the domain of } x \text{ and identity law} \} \]
\[ = x \in X \]

This also proves (for any \( X : \mathcal{H} \mathcal{L} \mathcal{E} \mathcal{P} \)) the \textit{elevation fixpoint law}: \[ \text{elevate } X = \sim X \]

since \( \sim X \in X \) and \( \sim X : \bigcup \mathcal{B} \mathcal{H} \mathcal{L} \mathcal{E} \mathcal{P} \)

The \textit{elevation idempotency law} states that:

\[ \text{elevate}(\text{elevate } x) = \text{elevate } x \]

Proof by transitivity:

\[ \text{elevate}(\text{elevate } x) = \text{elevate } x \]
\[ = \{ \text{Definition of } \sim \text{ and fundamental equivalence property} \} \]
\[ = \text{elevate } x \in \{ \text{elevate } x \} \]
\[ = \{ \text{Set theory} \} \]
\[ = \text{true} \]

Furthermore we also prove the following rules:

\[ \text{elevate } x : \mathcal{D}(\text{elevate } y) \supseteq \text{elevate } y \text{ (elevate } x) = \text{elevate}(y \ x) \]

\[ \text{elevate } x; \text{elevate } y = \text{elevate}(x; y) \]
\[\text{[elevate } x\text{]} = \text{elevate}[x]\]

Proof: For all the above cases, let \(X = \{\text{elevate } x\}\) and \(Y = \{\text{elevate } y\}\).

By this definition, \(X, Y : HLEp\), which means \((HLE_3 \text{ axiom})\):

\[
\begin{align*}
\& (\sim X : D(\sim Y)) \geq \{(\sim Y \sim X) : HLEp\} \\
\& \{\sim X : HLEp\} \\
\& \{\sim X \sim Y : HLEp\} \\
\& \{\text{Because always } z : \text{elevate } z \text{ for } z \text{ being } [x], [x; y] \text{ and } y z\}\} \\
\& \{\text{the fundamental property applies}\}\} \\
\& \{\text{Function inclusion and containment transitivity}\}\} \\
\& \{\text{Containment transitivity}\}\} \\
\& \{\text{containment transitivity}\}\} \\
\& \{\text{Function inclusion and containment transitivity}\}\} \\
\end{align*}
\]

The laws have been proven. Notice that

\[\text{elevate } x : D(\text{elevate } y)\]

suffices to assert that \(y x\) is well-defined. That is because always \(z : \text{elevate } z\).

so:

\[
\begin{align*}
\& \{\text{Function inclusion and containment transitivity}\}\} \\
\& \{\text{Containment transitivity}\}\} \\
\& \{\text{containment transitivity}\}\} \\
\end{align*}
\]
Bibliography


