Implementation and Abstraction in Mathematics

David McAllester

July 30, 2014
Contents

Introduction 5

1 Inference Rules 9
  1.1 Typed Predicate Calculus . . . . . . . . . . . . . . . . . . . . . 9
  1.2 Structure Types . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2 Overview 15
  2.1 The Semantic Value Function . . . . . . . . . . . . . . . . . . . 15
  2.2 The Abstraction Theorem . . . . . . . . . . . . . . . . . . . . . . 16
  2.3 Bags — a Case Study in Isomorphism . . . . . . . . . . . . . . . 16
  2.4 Simple Isomorphism . . . . . . . . . . . . . . . . . . . . . . . . 17
  2.5 General Isomorphism . . . . . . . . . . . . . . . . . . . . . . . . 18
  2.6 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  2.7 Implementation and Abstraction . . . . . . . . . . . . . . . . . . 20
  2.8 Value Function Invariants . . . . . . . . . . . . . . . . . . . . . 21
  2.9 Abstract Values and Implementations . . . . . . . . . . . . . . . 22
  2.10 Pointification-Composition Commutation . . . . . . . . . . . . . 22
  2.11 The Idempotence Property . . . . . . . . . . . . . . . . . . . . 23
  2.12 Abstraction-Composition Commutation . . . . . . . . . . . . . . 23
  2.13 Value-Abstraction Commutation . . . . . . . . . . . . . . . . . . 23
  2.14 Value-Composition Commutation . . . . . . . . . . . . . . . . . 24
  2.15 Commutation with Inverse . . . . . . . . . . . . . . . . . . . . . 24
  2.16 The Abstraction Theorem . . . . . . . . . . . . . . . . . . . . . 24
  2.17 Voldemort’s Theorem . . . . . . . . . . . . . . . . . . . . . . . . 24

3 Weak-Values 27
  3.1 Pre-Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
  3.2 Operations on Pre-Values . . . . . . . . . . . . . . . . . . . . . . 29
  3.3 Weak Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
  3.4 Algebraic Laws of Composition and Inverse . . . . . . . . . . . . 30
  3.5 Weak-Types as Bijections . . . . . . . . . . . . . . . . . . . . . . 33
4 Implementation and Abstraction
  4.1 Functions .................................................. 35
  4.2 Abstract Values and Implementations ..................... 38
  4.3 Some Template Notation .................................. 39
  4.4 Pointification ............................................. 39
  4.5 Abstraction ............................................... 42

5 The Semantic Value Function
  5.1 Function Types ............................................ 45
  5.2 The Template Value Function ............................. 48
  5.3 The Value Function ....................................... 48
  5.4 A Sanity Check Lemma .................................... 51
  5.5 The Semantic Notation $\tau_\Delta(D)$ ...................... 51

6 The Main Theorems
  6.1 Value-Abstraction Commutation ........................... 53
  6.2 Some Preliminary Lemmas .................................. 57
  6.3 Value-Composition Commutation ........................... 59
  6.4 The Abstraction Theorem .................................. 63

7 Outstanding Issues ............................................ 65
Introduction

This manuscript presents a type-theoretic foundation for mathematics in which each type is associated with an equality relation in correspondence with the standard notions of isomorphism in mathematics. The main result is an abstraction theorem stating that isomorphic objects are inter-substitutable in well-typed contexts.

The notion of isomorphism in mathematics seems related to the notion of an application programming interface (API) in computer software. An API specifies what information and behavior an object provides. Two different implementations can produce identical behavior when interaction is restricted to that allowed by the API. For example, textbooks on real analysis typically start from axioms involving multiplication, addition, and ordering. Addition, multiplication and ordering define an abstract interface — the well-formed statements about real numbers are limited to those that can be defined in terms of the operations of the interface. We can implement real numbers in different ways — as Dedekind cuts or Cauchy sequences. However, these different implementations provide identical behavior as viewed through the interface — the different implementations are isomorphic as ordered fields. The axioms of real analysis specify the reals up to isomorphism. The type system presented here defines interfaces to mathematical objects and each such interface defines a notion of isomorphism.

Types were central to Whitehead and Russell’s Principia Mathematica published in 1912 [1]. This type system was motivated by the need to avoid set-theoretic paradoxes (Russell’s paradox) and did not establish a notion of isomorphism at each type. The type-theoretic approach to avoiding paradoxes was supplanted by Zermelo-Fraenkel set theory with the axiom of choice (ZFC) which appeared in its final form in the early 1920’s. Most mathematicians today recognize ZFC as the foundation of mathematics. Unfortunately, ZFC does not illuminate the working mathematician’s notion of a well-formed statement. It is common in set theory to define the natural numbers in such a way that we have $4 \in 5$. But any working mathematician would agree that $4 \in 5$ is not a well-formed statement of arithmetic — it violates the abstraction barrier, or abstract interface, normally associated with the natural numbers.

The intuitively clear fact that a vector space does not have a canonical basis, or a canonical isomorphism with its dual, is closely related to the concept of isomorphism. A symmetry of an object, such as a rotational symmetry of a circle, is simply an isomorphism of the object with itself — an automorphism. The statement that a vector space has no canonical basis is simply the observation that for any two bases there is an automorphism (a symmetry) of the vector space which maps one basis to the other.

The non-existence of canonical objects has come to be formally explained with the concepts of category theory. The desire to formalize the non-existence of a canonical isomorphism between a vector space and its dual was given in the opening paragraph of the original 1945 Eilenberg and MacLane paper in-
Introducing category theory [2]. Here we take the type-theoretic notion of a well-formedness as fundamental and motivate certain category-theoretic constructs as tools in the proof of the abstraction theorem. The abstraction theorem implies that no basis of a vector space can be named without introducing additional free variables – without additional choices.

While axiomatic set theory largely supplanted type theory as the accepted foundation of mathematics, type theory has continued to evolve as an alternative foundation. This evolution has largely focused on constructive mathematics starting from a 1934 observation by Curry that implication formulas \( P \Rightarrow Q \) (that statement \( P \) implies statement \( Q \)) can be placed in correspondence with function types \( \sigma \to \tau \) (the type of a function mapping objects of type \( \sigma \) to objects of type \( \tau \)) [3]. This has become known as the Curry-Howard isomorphism or the propositions-as-types paradigm. The Curry-Howard isomorphism for propositional logic was extended to a full constructive foundation for mathematics by Coquand and Huet in 1986 [4]. However, the COC system does not associate each type with a notion of isomorphism. The proof of the abstraction theorem given here is based on the more classical propositions-as-Booleans approach.

Types are clearly important in computer programming and type theory has evolved that context. In 1983 John Reynolds published an abstraction theorem loosely analogous to the one proved here [5]. Reynolds’ abstraction theorem is based on logical relations rather than isomorphisms. Logical relations arise in programming because programming languages do not support equality at function types or universal or existential quantification at infinite types.

Recently there has been interest in homotopy type theory (HOTT) [6]. HOTT is a self-described work in progress to produce a synthesis of type theory and homological algebra which might ultimately serve as a new foundation for mathematics. In contrast, the development here has been directed entirely toward proving the abstraction theorem. There is no use of homological algebra.

The abstraction theorem is purely semantic — it can be stated and proved without the introduction of inference rules. However, pedagogically it seems best to start by giving inference rules intended to serve as a foundation for mathematics. This supports the formalist position that ultimately mathematical thought rests on symbolic computation. The semantics, however, is Platonic — semantics is the relationship between symbolic expressions and Platonic mathematics. The semantics allows one to prove that the inference rules are sound — that the formulas provable with those rules are actually true. The abstraction theorem states, in essence, that the inference rule of substitution of equals for equals (the substitution of isomorphics) is sound.

Chapter 1 gives inference rules intended to form a foundation for mathematics. Chapter 2 outlines the remainder of the manuscript including the statement of the abstraction theorem and an outline of its proof. Chapters 3 through 6 give the proof details. Outstanding issues are discussed in chapter 7.

1. A.N. Whitehead and B. Russell. *Principia mathematica*, volume 2. University Press, 1912.
2. S. Eilenberg and S. MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, pages 231–294, 1945.

3. H.B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

4. T. Coquand and G. Huet. The calculus of constructions. INRIA Research Report no. 530, 1986.

5. John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

6. The Univalent Foundations Program. Homotopy Type Theory, Univalent Foundations of Mathematics. [http://hottheory.files.wordpress.com/2013/03/hott-online-611-ga1a258c.pdf](http://hottheory.files.wordpress.com/2013/03/hott-online-611-ga1a258c.pdf) Institute for Advanced Study, April 2013.
Chapter 1

Inference Rules

This chapter gives inference rules for a typed foundation of mathematics. The rules are divided into two groups. The first group contains the inference rules for (typed) predicate calculus. These are the simplest and clearest inference rules. The second group of rules handles structures and structure types. These rules are unusual in that they unify the notion of context from sequent calculi with the notion of structure type such as “group” or “topological space”. We do not give rules for deriving isomorphism relationships. Isomorphism is defined semantically in later chapters. Presumably it is possible to give rules for isomorphism but there are subtleties and, in any case, the abstraction theorem is purely semantic and independent of the inference rules.

1.1 Typed Predicate Calculus

We define a system of inference rules for deriving sequents of the form $\Sigma \vdash \Theta$ where $\Sigma$ is a context consisting of variable declarations and assumptions and $\Theta$ is a statement that is derivable in that context. We will write $e : \tau$ to indicate that $e$ is an instance of type $\tau$. For each integer $i \geq 0$ we introduce a constant \texttt{type$_i$}, representing the type of all types at level $i$. We start with the following inference rules for forming contexts.

1. The empty context is well-formed.
2. $\vdash \texttt{type}_j : \texttt{type}_j$ for $j > i$
3. $\vdash \texttt{Bool} : \texttt{type}_i$
4. If $\Sigma$ is well-formed, $\Sigma \vdash \tau : \texttt{type}_i$, and $x$ is a variable not declared in $\Sigma$, then $\Sigma ; x : \tau$ is well-formed.
5. If $\Sigma$ is well-formed and $\Sigma \vdash \Phi : \texttt{Bool}$ then $\Sigma ; \Phi$ is well-formed.
6. If $\Sigma$ is well-formed and $\Theta$ is a variable declaration or Boolean assumption in $\Sigma$ then $\Sigma \vdash \Theta$. 


7. If $\Sigma$ is well-formed, and we have a rule stating $\vdash \Theta$, then $\Sigma \vdash \Theta$.

Rules 2 and 4 allow us to derive that the single declaration $\alpha : \text{type}_i$ constitutes a well-formed context declaring the variable $\alpha$ to be a type. Further applications of these rules allow one to derive that $\alpha : \text{type}_i$; $x : \alpha$; $P : \text{Bool}$ is a well-formed context where $P$ is a Boolean variable.

If $\sigma$ and $\tau$ are types we will write $\sigma \rightarrow \tau$ for the type of functions from $\sigma$ to $\tau$. We include the following rules for function types and functions. Each rule states that if one can derive the sequents above the line then one can derive the sequent below the line.

$$
\begin{align*}
\Sigma \vdash \tau : \text{type}_i \\
\Sigma \vdash \sigma : \text{type}_i \\
\hline
\Sigma \vdash (\tau \rightarrow \sigma) : \text{type}_i
\end{align*}
$$

$$
\begin{align*}
\Sigma \vdash f : (\tau \rightarrow \sigma) \\
\Sigma \vdash e : \tau \\
\hline
\Sigma \vdash f(e) : \sigma
\end{align*}
$$

Using these rules we can derive

$$
\alpha : \text{type}_i; \ x : \alpha; \ f : \alpha \rightarrow \alpha; \ P : \alpha \rightarrow \text{Bool} \vdash P(f(f(x))) : \text{Bool}.
$$

Functions of more than one argument can be handled by Currying — the type $\tau \times \tau \rightarrow \sigma$ is represented by $\tau \rightarrow (\tau \rightarrow \sigma)$.

Figure 1.1 gives additional inference rules corresponding to the inference rules of sorted first order predicate calculus with equality. In the presence of structure types the substitution rule of figure 1.1 expresses the inter-substitutability of isomorphics.

We also include the following form of the axiom of choice.

$$
\begin{align*}
\Sigma \vdash \forall x : \tau \ \exists y : \sigma \ \Phi[x, y] \\
x \text{ does not occur free in } \sigma \\
\hline
\Sigma \vdash \exists f : \tau \rightarrow \sigma \ \forall x : \tau \ \Phi[x, f(x)]
\end{align*}
$$

1.2 Structure Types

We now extend the type system so as to include types like “group”, “field”, “vector space”, or “topological manifold”. The instances of such a type are the models of a context. For example, we define the concept of a group by saying that a group consists of a set (a type) together with a group operation on that set satisfying certain axioms. For a well-formed context $\Sigma$ we will write $\Sigma$ for type whose instances are the models of $\Sigma$. Instances of a type of the form $\Sigma$ will be called structures. We can define the types $\text{Group}$ and $\text{Field}$ as follows where $\mathcal{A}_{\text{Group}}$ is a Boolean expression giving the group axioms and $\mathcal{A}_{\text{Field}}$ gives
### 1.2. STRUCTURE TYPES

\[
\Sigma \vdash \Phi : \mathbf{Bool} \\
\Sigma \vdash \Psi : \mathbf{Bool} \\
\Sigma \vdash (\Phi \vee \Psi) : \mathbf{Bool} \\
\Sigma \vdash \neg \Phi : \mathbf{Bool}
\]

\[
\Sigma; x : \tau \vdash \Phi[x] : \mathbf{Bool} \\
\Sigma \vdash w : \tau \\
\Sigma \vdash u : \tau \\
\Sigma \vdash (w =_\tau u) : \mathbf{Bool}
\]

\[
\vdash \forall P_1 : \mathbf{Bool}, \ldots, \forall P_n : \mathbf{Bool} \Psi[P_1, \ldots, P_n] \text{ for } \Psi \text{ a Boolean tautology}
\]

\[
\Sigma \vdash \Phi : \mathbf{Bool} \\
\Sigma \vdash \Psi : \mathbf{Bool} \\
\Sigma; \Phi \vdash \Psi \\
\Sigma \vdash \Phi \Rightarrow \Psi \\
\Sigma \vdash \Psi
\]

\[
\Sigma \vdash \forall x : \tau \Phi[x] \\
\Sigma \vdash e : \tau \\
\Sigma \vdash \Phi[e] \\
\Sigma \vdash \forall x : \tau \Phi[x]
\]

\[
\Sigma \vdash e : \tau \\
\Sigma \vdash u =_\tau w \\
\Sigma \vdash w =_\tau s \\
\Sigma \vdash u =_\tau s \\
\Sigma \vdash e =_\tau e \\
\Sigma \vdash w =_\tau u \\
\Sigma \vdash \neg \Phi[x] : \tau \\
\Sigma \vdash w =_\sigma u \\
\Sigma \vdash e =_\tau e[w]
\]

\[
\Sigma; x : \sigma \vdash e[x] : \tau \\
\Sigma \vdash \neg \Phi[x] : \tau \\
\Sigma \vdash w =_\sigma u \\
\Sigma \vdash x \text{ does not occur free in } \tau
\]

\[
\Sigma \vdash \neg \Phi\vee \Psi \\
\Sigma \vdash \exists x : \tau \Phi[x] \text{ abbreviates } \neg \forall x : \tau \neg \Phi[x]. \text{ Under the semantics defined here the equation } G =_{\text{Group}} G' \text{ represents the assertion that } G \text{ and } G' \text{ are isomorphic groups. The substitution rule expresses the inter-substitutability of isomorphics.}
\]

Figure 1.1: Predicate Calculus Rules. We adopt the conventions that \( \Phi \Rightarrow \Psi \) abbreviates \( \neg \Phi \vee \Psi \) and \( \exists x : \tau \Phi[x] \) abbreviates \( \neg \forall x : \tau \neg \Phi[x] \).
the field axioms.

\[
\begin{align*}
\text{Group} & \equiv \alpha : \text{type}_1; \bullet : \alpha \times \alpha \to \alpha; A_{\text{Group}} \\
\text{Field} & \equiv \alpha : \text{type}_1; + : \alpha \times \alpha \to \alpha; \bullet : \alpha \to \alpha; A_{\text{Field}}
\end{align*}
\]

We can think of a variable declared in \(\Sigma\) as a “slot name” or “field name” for a structure \(S : \Sigma\). We will write \(S.x\) for the value that \(S\) assigns to \(x\). We will also use \(\varsigma\) to denote the empty structure, so that we have \(\varsigma : \tau\), and write \(S[y \leftarrow v]\) for the structure which is the extension of \(S\) by an additional field \(y\) with value \(v\). We then have that \(\varsigma[y \leftarrow v]\) is the structure with one field named \(y\) and where that field has value \(v\).

In addition to structure types of the form \(\Sigma\), we also want structure types such as \(\text{PairOf}[\alpha, \beta]\) where \(\alpha\) and \(\beta\) are type variables. More generally we would like types of the form \(\text{PairOf}[\sigma, \tau]\) where \(\sigma\) and \(\tau\) may contain free variables. Other examples of structure types are \(\text{BagOf[\sigma]}\) — the type of multisets of instances of \(\sigma\) — and \(\text{VectorSpace}[F]\) — the type of vector spaces over a given field \(F\). We are tempted to define these types as follows.

\[
\begin{align*}
\text{PairOf}[\sigma, \tau] & \equiv \text{First} : \sigma; \text{Second} : \tau \\
\text{BagOf[\sigma]} & \equiv \alpha : \text{type}_0; f : \alpha \to \sigma \\
\text{VectorSpace}[F] & \equiv \alpha : \text{type}_1; + : \alpha \times \alpha \to \alpha; \bullet : (F, \alpha) \times \alpha \to \alpha; A
\end{align*}
\]

Unfortunately this implementation suffers from a technical difficulty involving the renaming of bound variables in substitutions. For example, the type \(\text{BagOf[\alpha]}\) becomes \(\alpha : \text{type}_0; f : \alpha \to \sigma\). Here the free variable \(\alpha\) has been captured by the bound variable \(\alpha\). We cannot solve this problem by renaming the bound variable \(\alpha\) because \(\alpha\) is a field name for bags and for any bag \(B\) we need to be able to write \(B.\alpha\) for the “index set” of the bag \(B\).

To solve this problem we use structure types of the form \(\Theta_{\Delta}(D)\) where \(\Delta; \Gamma\) is a well-formed context and \(D; \Delta; \Gamma\) so that \(D\) provides the values of the variables declared in \(\Delta\). We now write the the above types as

\[
\begin{align*}
\text{PairOf}[\sigma, \tau] & \equiv \text{First} : \alpha; \text{Second} : \beta; \alpha : \text{type}_1; \beta : \text{type}_1; (\varsigma[\alpha \leftarrow \sigma][\beta \leftarrow \tau]) \\
\text{BagOf[\sigma]} & \equiv \alpha : \text{type}_1; f : \alpha \to \beta; \beta : \text{type}_1; (\varsigma[\beta \leftarrow \sigma]) \\
\text{VectorSpace}[F] & \equiv \alpha : \text{type}_1; + : \alpha \times \alpha \to \alpha; \bullet : (H, \alpha) \times \alpha \to \alpha; {H : \text{Field}(\varsigma[H \leftarrow F])}
\end{align*}
\]

In this version of the type \(\text{BagOf[\sigma]}\) the type expression \(\sigma\) does not occur within the scope of the bound variable \(\alpha\) and we can perform substitutions for the free variables of \(\sigma\) without renaming \(\alpha\). The same is true for free variables in the type expression \(\sigma\) and \(\tau\) in the type \(\text{PairOf}[\sigma, \tau]\) and for free variables in the field expression \(F\) in the type expression \(\text{VectorSpace}[F]\).

Structure types are closely related to dependent sum types. A dependent sum type is written as \(\sum_{x : \tau} \sigma[x]\) and denotes the type whose instances are pairs \((x, y)\) with \(x : \tau\) and \(y : \sigma[x]\). Variable renaming is not an issue for dependent sum types because an instance of a dependent sum type is a pair rather than a structure with field names. However, essentially all type systems involve
1.2. STRUCTURE TYPES

$\vdash \varsigma : \tau_\Delta(\varsigma)$

$\Sigma \vdash D : \tau_\Delta(\varsigma)$

$\Sigma \vdash \Delta : \text{type}_i$

$\Sigma \vdash \varsigma : \tau_\Delta(D)$

$\vdash \tau_\Delta(D) : \text{type}_i$

$\Delta ; \Gamma \vdash \sigma : \text{type}_i$

$\Delta; \Gamma \vdash \Phi : \text{Bool}$

$x$ not declared in $\Delta; \Gamma$

$\vdash \Gamma; x : \sigma \Delta : \text{type}_i$

$\Sigma \vdash G : \Gamma \Delta (D)$

$\Sigma \vdash e : \mathcal{V}_{\Delta, \Gamma} [\tau] D; G$

$\Sigma \vdash G[x \leftarrow e] : \Gamma; \tau_\Delta(D)$

$\Sigma \vdash G[x \leftarrow e] \cdot x = e$

$\Delta; \Gamma \vdash \Theta$

$\Sigma \vdash \mathcal{V}_{\Delta, \Gamma} [\Theta] D; G$

$\Sigma \vdash G[x \leftarrow e] \cdot y = G.y$

$\Sigma \vdash u = w$

$\Sigma \vdash u = w$

$\Sigma \vdash u ; \sigma$

$\Sigma \vdash u = \sigma$

$\Sigma \vdash e ; \sigma$

$\Sigma \vdash e : \tau$

$\Sigma \vdash v : \sigma$

$\Sigma \vdash u =_\sigma v$

Figure 1.2: Structure Rules. Here $\mathcal{V}_{\Delta, \Gamma} [e] D; G$ denotes the result of substituting values from $D$ and $G$ for variables in $e$ — each variable $x$ declared in $\Delta$ is replaced by the expression $D.x$ and each variable $y$ declared in $\Gamma$ is replaced by the expression $G.y$. In the text we use $\Sigma$ as an abbreviation for $\Sigma_\Delta(\varsigma)$. It is highly recommended that in a first reading of the rules one considers only the special case where $\Delta = \varepsilon$. In this case $\Delta$ and $D$ can be removed from all expressions. The use of absolute equality (unsubscripted equality) is a convenience for making the rules more compact — absolute equalities are not Boolean expressions.
contexts with variable declarations. The type system developed here unifies structure types, dependent sum types, and sequent contexts.

Figure 1.2 gives the inference rules for structure types. The inference rules involve a substitution notation analogous to a semantic value function. We defined $V_{\Delta, \Gamma} [e] D; G$ to be the expression resulting from substituting values from $D$ and $G$ for variables in $e$ — each variable $x$ declared in $\Delta$ is replaced by the expression $D.x$ and each variable $y$ declared in $\Gamma$ is replaced by the expression $G.y$. We take $\Sigma$ to be an abbreviation for $\Sigma_x(\varsigma)$. It is highly recommended that in a first reading of the rules one considers only the special case where $\Delta = \varepsilon$. In this case $\Delta$ and $D$ can be remove from all expressions. The use of absolute equality (unsubscribed equality) is a convenience for making the rules more compact — absolute equalities are not Boolean expressions.
Chapter 2

Overview

The proof of the abstraction theorem is contained in chapters 3 through 6. This chapter gives an informal overview of the content of those chapters. The formal definition of the semantic value function is given in chapter 5. The earlier chapters 3 through 4 define the space of semantic values including the space of semantic types. Pedagogically, however, it seems best to start with a discussion of the value function.

2.1 The Semantic Value Function

The semantic value of an expression $e$ under an interpretation $S$ of a context $\Sigma$ is written as $V_{\Sigma} [e] S$. Here $S$ is taken to be a semantic model of the context $\Sigma$ — an assignment of a semantic value to each variable declared in $\Sigma$ consistent with the type declarations and assumptions in $\Sigma$. A semantic model of the context $\Sigma$ is an instance of the semantic value of the type expression $\Sigma$.

To avoid duplicating a large number of symbols we overload notation. For example $\Phi \lor \Psi$ is an expression when $\Phi$ and $\Psi$ are expressions but is a truth value when $\Phi$ and $\Psi$ are truth values. For expressions $\Phi$ and $\Psi$ we have the following where $S$ is a semantic model (a semantic structure).

$$V_{\Sigma} [\Phi \lor \Psi] S = V_{\Sigma} [\Phi] S \lor V_{\Sigma} [\Psi] S$$

The disjunction sign on the left hand side is syntactic while the disjunction on the right hand side is semantic.

The notations $G.x$, $G[y \leftarrow w]$, $\Phi \lor \Psi$ and $\neg \Phi$ all have both semantic and syntactic readings. The meaning of overloaded notation can be formally disambiguated when the subexpressions involved can be (recursively) disambiguated. For example, the notation $G.x$ can be disambiguated provided that $G$ can be disambiguated — if $G$ is a semantic value then $G.x$ is a semantic value while if $G$ is a syntactic expression then $G.x$ is a syntactic expression. For constant symbols we distinguish the syntactic constant from its semantic value. In par-
ticular, Bool, ζ and \texttt{type}_i are constant symbols while \texttt{Bool}^\ast, ζ^\ast and \texttt{type}^\ast_i denote the corresponding semantic values.

Note that Σ is a type expression, not a semantic value. We can write the semantic value of the expression Σ as \( V_\Sigma [\Xi] \) which we will abbreviate as \( V [\Xi] \). We also overload the notion \( x : \sigma \) to mean for all \( \tau : \sigma \) that is a model of the context Σ.

For a Boolean expression \( \Phi \) we will write \( \Sigma \models \Phi \) to mean \( \Sigma \) semantically implies \( \Phi \). More formally we have \( \Sigma \models \Phi \) if for all semantic structures \( S : V [\Xi] \) we have that \( V_\Sigma [\Phi] S \) is the semantic Boolean value “true”. Similarly, we write \( \Sigma \models e : \tau \) to mean for all \( S : V [\Xi] \) we have \( V_\Sigma [\sigma] S = V_\Sigma [\tau] S \).

The notation \( V_\Sigma [\sigma] S \) itself has both a semantic and a syntactic reading. A form of the syntactic reading was introduced in section 1.2. When \( S \) is a syntactic expression \( V_\Sigma [\sigma] S \) is the syntactic expression that results from substituting the expression \( S.x \) for each free occurrence of a variable \( x \) in \( e \). When \( S \) is a semantic value, as above, we have that \( V_\Sigma [\sigma] S \) is a semantic value. For a Boolean formula \( \Phi \) we have \( \Sigma \models \Phi \) if and only if \( \forall S : V [\Xi] V_\Sigma [\Phi] S \). The syntactic and semantic readings of the value function both appear in the following equality where \( G \) is a syntactic expression and \( S \) is a semantic structure.

\[
V_\Sigma [V_T [e] G] S = V_T [e] V_\Sigma [G] S
\]

### 2.2 The Abstraction Theorem

The abstraction theorem is semantic — it is stated purely in terms of the semantic value function without reference to inference rules. The semantic abstraction theorem states that if \( \Sigma; x : \sigma \models e[x] : \tau \), where \( x \) does not occur free in \( \tau \), then for \( S : V [\Xi] \) and \( u, w : V_\Sigma [\sigma] S \) with \( u =_{V_\Sigma [\sigma]} w \), we have

\[
V_\Sigma; x : \sigma [e[x]] S[x \leftarrow u] =_{V_\Xi [\sigma]} V_\Sigma; x : \sigma [e[x]] S[x \leftarrow w].
\]

### 2.3 Bags — a Case Study in Isomorphism

We interpret equality as isomorphism. For example \( =_{\text{Group}} \) and \( =_{\text{Field}} \) should be interpreted as the standard notion of isomorphism for these objects. The general notion of isomorphism covers open structure types such as \texttt{BagOf}[\sigma].

\[
\text{BagOf}[\sigma] \equiv \: \text{type}_i; f : \sigma \to \beta ; \text{type}_i (\zeta [\beta \leftarrow \sigma])
\]

We consider two instances \( B, C : \text{BagOf}[\sigma] \) to be isomorphic if there exists a bijection \( g : B.\alpha \to C.\alpha \) such that for all \( i : B.\alpha \) we have \( (B.f)(i) =_\sigma (C.f)(g(i)) \). We then get that for \( B, C : \text{BagOf}[\sigma] \) we have \( B =_{\text{BagOf}[\sigma]} C \) if and only if \( B \) and \( C \) represent the same multiset of elements of \( \sigma \). It seems intuitively
clear that if two bags \( B \) and \( C \) are equivalent in this sense then they are indistinguishable by (inter-substitutable into) contexts accepting arbitrary instances of \( \text{BagOf}[\sigma] \). Conversely, it is possible to construct a well-formed formula \( \Phi(x, n, B) \) such that for \( x : \sigma \) and \( B : \text{BagOf}[\sigma] \) we have that \( \Phi(x, n, B) \) holds if and only if \( n \) is the count of \( x \) in \( B \). This gives that if two bags represent different multisets then they are distinguishable by well-formed contexts.

2.4 Simple Isomorphism

The type \( \text{BagOf}[\sigma] \) is a special case of a general notion of a “simple” structure type expression. The treatment of isomorphism at simple structure types is straightforward and is given in this section. However, for reasons discussed below, we take a different approach to the general case.

A simple type expression is either the type \( \text{Bool} \), a type variable, or a function type expression \( \sigma \to \tau \) where \( \sigma \) and \( \tau \) are recursively simple type expressions. This is the standard notion of a simple type in the simply-typed lambda calculus. A variable declaration \( x : \tau \) will be called simple if \( \tau \) is a simple type expression. A simple context is a well-formed context in which every variable declaration either has the form \( \alpha : \text{type} \), which declares a type variable, or is a declaration \( x : \tau \) where \( \tau \) is a simple type expression in which each type variable has been previously declared in the context. A simple structure type has the form \( \Gamma \Delta \text{[D]} \) where \( \Delta; \Gamma \) is a simple context. The type \( \text{BagOf}[\sigma] \) as defined above is simple (for any \( \sigma \)).

Now let \( \Pi \Delta(D) \) be a simple structure type, let \( \alpha_1, \ldots, \alpha_n \) be the type variables declared in \( \Delta \) and let \( \beta_1, \ldots, \beta_m \) be the type variables declared in \( \Gamma \). Consider a structure \( G : \Pi \Delta(D) \). We assume that for each type variable \( \alpha_i \) declared in \( \Delta \), the type \( D.\alpha_i \) has a well defined notion of equality \( =_{D.\alpha_i} \). Similarly, we assume that for each type variable \( \beta_j \) declared in \( \Gamma \), the type \( G.\beta_j \) has a well defined notion of equality \( =_{G.\beta_j} \). Now consider a simple declaration \( x : \tau \) where \( \tau \) is a simple type expression over \( \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m \). The type expression \( \tau \) has a value \( \forall \Delta;\Gamma \Pi[\tau] D;G \) under the interpretations of the type variables provided by the structures \( D \) and \( G \). We can define equality (isomorphism) at the type \( \forall \Delta;\Gamma \Pi[\tau] D;G \) recursively by

\[
f = \forall \Delta;\Gamma \Pi[\sigma] D;G \ g \Leftrightarrow \forall x : \forall \Delta;\Gamma \Pi[\sigma] D;G \ f(x) = \forall \Delta;\Gamma \Pi[\tau] D;G \ g(x).
\]

Now consider two structures \( F, G : \Pi \Delta(D) \) such that there exists bijections \( \gamma_{\beta_1} : G.\beta_1 \to H.\beta_1, \ldots, \gamma_{\beta_m} : G.\beta_m \to H.\beta_m \). These bijections induce a bijection

\[
\gamma_\tau : \forall \Delta;\Gamma \Pi[\tau] D;G \to \forall \Delta;\Gamma \Pi[\tau] D;H
\]

at every simple type \( \tau \) over \( \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m \) defined by setting \( \gamma_{\text{Bool}} \) to be the identity function, \( \gamma_{\alpha_i} \) to be the identity function on \( D.\alpha_i \) and defining
γ_σ → η to be the unique bijection satisfying the following commutative diagram.
\[
\begin{align*}
\mathcal{V}_{\Delta;\Gamma}[\sigma] D; G & \xrightarrow{g} \mathcal{V}_{\Delta;\Gamma}[\eta] D; G \\
\downarrow \gamma_\sigma & \downarrow \gamma_\eta \\
\mathcal{V}_{\Delta;\Gamma}[\sigma] D; H & \xrightarrow{\gamma_\sigma \cdot g} \mathcal{V}_{\Delta;\Gamma}[\eta] D; H
\end{align*}
\]
For ∆; Γ a simple context we have \( G = \mathcal{T}_{\Delta(D)} H \) if the bijections \( \gamma_\beta \) can be selected so that for each simple declaration \( x: \tau \) in Γ we have
\[
\gamma_\tau(G.x) = \mathcal{V}_{\Delta;\Gamma}[\tau] D; H.x.
\]

### 2.5 General Isomorphism

Many types of interest are not simple and the definition of isomorphism for simple types does not seem to generalize. For example, the type

\[
\text{VectorSpace}[\mathcal{F}] \equiv \alpha: \text{type}; +: \alpha \times \alpha \to \alpha; \cdot: \alpha \times \alpha \to \alpha; A: \text{Field}(\varsigma[H \leftarrow \mathcal{F}])
\]

is not simple because it involves a variable declared to be of type \( \text{Field} \). Of course it is not difficult to define isomorphism for vector spaces over a given field. However, defining isomorphism at the following structure type is somewhat more challenging.

\[
\alpha: \text{type}_0; a: \alpha; b: \alpha; f: \alpha \to \text{type}_0; g: f(a) \to f(b)
\]

To handle the general case we take an isomorphism-as-value approach. We represent a \( \mathcal{T} \)-isomorphism between \( G, H: \mathcal{T} \) as another value \( M: \mathcal{T} \) such that \( M \) has a left interpretation corresponding to \( G \) and a right interpretation corresponding to \( H \). For example consider graphs. We take an isomorphism between two graphs \( G \) and \( H \) to be a graph \( M \) where each node of \( M \) has a left interpretation as a node of \( G \) and a right interpretation as a node of \( H \) and where the set of nodes of \( M \) define a bijection between the nodes of \( G \) and the nodes of \( H \). An edge of \( M \) then has a left interpretation as an edge of \( G \) and a right interpretation as an edge of \( H \). The graph \( M \) as a whole has a left interpretation which is \( G \) and a right interpretation which is \( H \).

The concept of “point” is central to the semantics developed here. Points are a special class of semantic values. All values are assigned a finite template specifying the position of the points in the value. More explicitly, a template is defined in chapters 3 to 4 to be an expression generated by the following grammar.

\[
\mathcal{T} ::= \text{Bool} \mid \text{Point} \mid \text{TypeOf}(\mathcal{T}) \mid \text{FUN}(\mathcal{T}) \mid \mathcal{S}
\]
\[
\mathcal{S} ::= \varsigma \mid \mathcal{S}[x \leftarrow \mathcal{T}]
\]

For the purposes of chapters 3 and 4 functions are treated identically to types — a function is just a type (or set) of pairs. Each value is tagged with its template so that we can distinguish empty types of different templates.
Each point has both a left index and a right index where the indeces can be anything at all. For two points $p$ and $q$ such that the right index of $x$ equals the left index of $q$ we define the composition $p \circ q$ to be the point whose left index is that of $p$ and whose right index is that of $q$. We define the inverse $p^{-1}$ of a point $p$ to be the result of reversing the right and left indeces. We define the right interpretation of a point $p$, denoted $\text{Right}(p)$, to be $p^{-1} \circ p$. $\text{Right}(p)$ is the point whose left and right indeces are both equal to the right index of $p$. Similarly we define $\text{Left}(p)$ to be $p \circ p^{-1}$.

The operations of $\text{Left}$, $\text{Right}$, composition and inverse are then inherited by objects built form points. In the graph example discussed above, the left and right interpretations of points are inherited by graphs built from points. As with the left and right interpretations, the ability to compose points allows us to compose values built from points. For example, consider two graphs $G$ and $H$ with $\text{Right}(G) = \text{Left}(H)$. We define $G \circ H$ to be the graph whose nodes are the points of the form $p \circ q$ for $p$ a node of $G$ and $q$ a node of $H$ with $\text{Right}(p) = \text{Left}(q)$ and whose edges are pairs of the form $(p \circ p', q \circ q')$ where $(p, q)$ is an edge of $G$ and $(p', q')$ is an edge of $H$ with appropriate right/left compatibilities. More generally, for any values $x$ and $y$ with $\text{Right}(x) = \text{Left}(y)$ we define the composition $x \circ y$. For any value $x$ we can also define the inverse $x^{-1}$ which is simply the result of reversing the left and right index of every point in $x$. For any value $x$ we have $\text{Left}(x) = x \circ x^{-1}$ and $\text{Right}(x) = x^{-1} \circ x$.

Each type must be associated with an equivalence relation. In particular consider point types — types whose members are points. A point specifies a pair of indeces. A type whose members are points defines a binary relation on the indeces occurring in the points. The inverse and composition operations on types of points correspond to the standard composition and inverse operations on binary relations. In the graph isomorphism example it is important that the type of points (nodes) of the graph forms a bijection between the left interpretation and the right interpretation. For a point type to represent a bijection we want that $\text{Left}(\tau)$ and $\text{Right}(\tau)$ are both equivalence relations and that $\tau$ forms a bijection between the left equivalence classes and the right equivalence classes. It turns out that these requirements are equivalent to the single condition that for $x, y, z \in \sigma$ with $x \circ y^{-1} \circ z$ defined we have $x \circ y^{-1} \circ z \in \sigma$. More generally a type $\tau$, not necessarily a type of points, is called back-and-forth closed if for $x, y, z \in \sigma$ with $x \circ y^{-1} \circ z$ defined we have $x \circ y^{-1} \circ z \in \sigma$.

The grammar for templates given above allows types to be embedded arbitrarily deeply in values. In chapter 3 a weak-value is defined to be a value such that all types embedded in that value are back-and-forth closed. Section 3.3 proves the following standard algebraic properties of weak values.

\[
\begin{align*}
\text{Right}(x^{-1}) &= \text{Left}(x) \\
\text{Left}(x^{-1}) &= \text{Right}(x) \\
(x^{-1})^{-1} &= x \\
\text{Right}(x \circ y) &= \text{Right}(y)
\end{align*}
\]
\[ \text{Left}(x \circ y) = \text{Left}(x) \]
\[ x \circ (y \circ z) = (x \circ y) \circ z \]
\[ (x \circ y)^{-1} = y^{-1} \circ x^{-1} \]
\[ x^{-1} \circ x \circ y = y \]
\[ x \circ y \circ y^{-1} = x \]

For a weak-type \( \sigma \) we define the equivalence relation \( \simeq_\sigma \) on the members of \( \sigma \) by saying that \( x \simeq_\sigma y \) if there exists \( z \in \sigma \) such that \( x \circ z^{-1} \circ y \) is defined. Chapter 3 proves that \( \sigma \) is a weak value then \( \simeq_\sigma \) is an equivalence relation.

For \( \models \tau : \text{type} \), — in which case \( \tau \) must not contain free variables — we will have that \( \forall \llbracket \tau \rrbracket \) forms a groupoid — a category in which every morphism is an isomorphism. However, for \( \alpha : \text{type}^* \) the type \( \alpha \) will be back-and-forth closed but will not in general form a groupoid.

### 2.6 Functions

Section 4.1 introduces a more refined notion of function and introduces templates of the form \( \text{Point} \rightarrow T \). A function can be viewed as a set of pairs \((p, y)\) where \( p \) is a point and \( y \) is an arbitrary value. A function can be applied to a value which is not a point by first converting that value to a point. As described below, all values can be converted to (abstracted to) points. We write \( \text{Dom}(f) \) for the type whose members are the points \( p \) such that \((p, y) \in f\) for some \( y \). We require that \( \text{Dom}(f) \) is a well-formed point type, i.e., is back-and-forth closed. We also require that for \( p, q \in \text{Dom}(f) \) there is a unique \( y \) such that \((p, y) \in f\) and write \( f[p] \) for the \( y \) such that \((p, y) \in f\). We define the notation \( f(x) \) by
\[
f(x) = f[x@\text{Point}]\]

where \( x@\text{Point} \) is the conversion of \( x \) to a point.

We also impose the much stronger requirement that for \( p, q \in \text{Dom}(f) \) with \( p \simeq_{\text{Dom}(f)} q \) we have \( f[p] = f[q] \). Note that we have not defined a range type for functions and are requiring absolute equality of function outputs for equivalent function inputs. Also note that the only inference rule for creating functions is the axiom of choice. Lemma 5.7 states, in effect, the soundness of the axiom of choice.

The requirement that the domain types of functions are point types is needed in the proof of the pointification-composition commutation theorem. The requirement that equivalent inputs yield exactly the same output is needed so that a composition of functions (in the sense of left and right interpretations of functions) is also a function.

### 2.7 Implementation and Abstraction

Section 4.5 introduces the notation \( x : \sigma \) as distinct from \( x \in \sigma \). For \( x \in \sigma \) we should think of \( x \) as an abstract member of \( \sigma \) while for \( x : \sigma \) we should think...
of $x$ as an implementation of a member of $\sigma$. For example, consider the type expression $\text{Group}$.

$$\text{Group} \equiv \alpha:\text{type}_1; \bullet: (\alpha \times \alpha \rightarrow \alpha) ; \mathcal{A}_{\text{Group}}.$$

The members of the semantic type $\text{Group}^* = V[\mathcal{G}]$ all have the property that the group elements are points. However, group implementations, such as permutation groups or homotopy groups, can have group elements other than points. An abstract group $G \in \text{Group}^*$ is an object with template

$$\zeta[\alpha \leftarrow \text{TypeOf}(\text{Point})] ; \bullet \leftarrow (\text{Point} \rightarrow (\text{Point} \rightarrow \text{Point}))].$$

A group implementation $G : \text{Group}^*$ can have an arbitrarily large template of the form

$$\zeta[\alpha \leftarrow \text{TypeOf}(\mathcal{T})] ; \bullet \leftarrow (\text{Point} \rightarrow (\text{Point} \rightarrow \mathcal{T})).$$

The notation $x : \sigma$ is used in the inference rules — the inference rules manipulate implementations. However, abstract members are fundamental to the notion of isomorphism — the abstract members of $\sigma$ are precisely the $\sigma$-isomorphisms.

There is a natural order on templates where we write $\mathcal{T} \leq \mathcal{W}$ if $\mathcal{T}$ can be derived from $\mathcal{W}$ by replacing subexpression of the expression $\mathcal{W}$ with $\text{Point}$. For a group implementation $G : \text{Group}^*$ (or for any object $G$) we will write $I(G)$ for the template of $G$. For $G \in \text{Group}^*$ and $G' : \text{Group}^*$ we have $I(G) \leq I(G')$.

For a value $x$ and template $\mathcal{T}$ with $\mathcal{T} \leq I(x)$ we write $x @ \mathcal{T}$ for the result of abstracting substructure of $x$ to points as specified by the template $\mathcal{T}$. Details are given in section 4.3.

For a template of the form $\text{TypeOf}(\mathcal{T})$ we write $\text{Mem}(\text{TypeOf}(\mathcal{T}))$ for the template $\mathcal{T}$. For a (semantic) type $\tau$ and value $x$ we write $x @ \tau$ for $x @ \text{Mem}(I(\tau))$. The notation $x : \tau$ requires (in part) that $x @ \tau$ is defined, i.e., $\text{Mem}(I(\tau)) \leq I(x)$, and that $(x @ \tau) \in \tau$. For $x, y : \tau$ we define $x =_{\tau} y$ to mean that $(x @ \tau) \simeq_{\tau} (y @ \tau)$.

### 2.8 Value Function Invariants

The value function is defined by recursion on syntactic expressions — the value of an expression is defined in terms of values of subexpressions. However, the meaning of the constant symbol $\text{type}_i$ cannot be reduced to the meaning of subexpressions. What does the declaration $\alpha : \text{type}_i$ really say about $\alpha$? What is a type?

Whatever properties types have in general must be such that when we form new types from old types the new types also have those properties. In the rules in chapter 1 types are first class. In particular, types can be included as values in structures. So finding a workable semantic definition of types interacts with finding a workable definition of values. We want the definition of value to support a proof of the abstraction theorem. A property of values which is preserved when we form new values from old values will be called a value function invariant.
A first value function invariant is that types must be back-and-forth closed.
This supports the composition and inverse properties of values and provides an
equivalence relation at each type. The semantic value $\text{type}_i^*$ is defined to be the
type whose members are the back-and-forth closed point types in the universe
$U_i$ (we assume an infinitely ascending collection of universes). One can show
that $\text{type}_i^*$ is itself back-and-forth closed. In fact $\text{type}_i^*$ is closed under inverse
and composition and forms a groupoid. For $\alpha \in \text{type}_i^*$, however, we have that
$\alpha$ is a type-isomorphism — a representation of a bijection between point types.
In general $\alpha$ is back-and-forth closed but is not closed under composition or
inverse and does not form a groupoid.

As with all types, we distinguish $\alpha \in \text{type}_i^*$ from $\alpha : \text{type}_i^*$. The former
states that $\alpha$ is a type of points. The notation $\alpha : \text{type}_i^*$ states that $\alpha$ is a type
implementation — perhaps a type of functions or a type of types. For $j > i$
we have $\text{type}_i^* : \text{type}_j^*$ but $\text{type}_i^* \not\in \text{type}_j^*$. Unfortunately, the definition of the
semantic value $\text{type}_i^*$ does not determine the space of type implementations $\alpha$
satisfying $\alpha : \text{type}_i^*$. It turns out that an additional type invariant is required.

2.9 Abstract Values and Implementations

The second value function invariant can be stated as a restriction on the tem-
plates of values. Section 4.2 defines an abstract value to be a value whose
template can be generated by the following grammar.

\[
\begin{align*}
\mathcal{A} & ::= \text{Bool} | \text{Point} | \text{OfType}(\text{Point}) | \text{Point} \to \mathcal{A} | \mathcal{S} \\
\mathcal{S} & ::= \varsigma | \mathcal{S}[x \leftarrow \mathcal{A}]
\end{align*}
\]

Implementation values have templates generated by the following extension of
the above grammar.

\[
\begin{align*}
\mathcal{I} & ::= \mathcal{A} | \text{OfType}(\mathcal{A}) | \mathcal{W} \\
\mathcal{W} & ::= \varsigma | \mathcal{W}[x \leftarrow \mathcal{I}]
\end{align*}
\]

In addition to the above property that the notation $x : \tau$ requires $x @ \tau \in \tau$, this
notation also requires that $\tau$ is a type implementation and that $x$ is an imple-
mentation value. The proofs of several of the commutation theorems needed for
the abstraction theorem rely on this restriction to implementation values.

2.10 Pointification-Composition Commutation

Section 4.4 proves a pointification-composition abstraction theorem stating that
for abstract values $x$ and $y$, if $\mathcal{I}(x) = \mathcal{I}(y)$ (if they have the same template) then
$x \circ y$ is defined if and only if $(x @ \text{Point}) \circ (y @ \text{Point})$ is defined and if these are
defined then $(x \circ y) @ \text{Point} = (x @ \text{Point}) \circ (y @ \text{Point})$. 
2.11. The Idempotence Property

For a semantic type $\tau$ we define $\text{SubPoint}(\tau)$ to be the type whose members are of the form $x@\text{Point}$ for $x \in \tau$. We show that this pointification-composition commutation theorem implies that for any type implementation $\tau$ (the members of which must be abstract values) and for $x, y \in \tau$ we have

$$x \simeq_{\tau} y \text{ if and only if } (x@\text{Point}) \simeq_{\text{SubPoint}(\tau)} (y@\text{Point}).$$

2.11 The Idempotence Property

Section 4.4 carefully defines pointification so as to yield an idempotence property for abstraction. More specifically, for an implementation value $x$ and templates $\mathcal{T}$ and $\mathcal{W}$ such that $(x@\mathcal{T})@\mathcal{W}$ is defined, and hence $\mathcal{W} \leq \mathcal{T}$, we have

$$(x@\mathcal{T})@\mathcal{W} = x@\mathcal{W}.$$ 

The proof of the value-abstraction commutation theorem relies on this idempotence property.

2.12 Abstraction-Composition Commutation

Section 4.5 proves that abstraction commutes with composition for abstract values. More specifically, for abstract values $x$ and $y$ with $I(x) = I(y)$ and for $\mathcal{A}$ an abstract template with $\mathcal{A} \leq I(x) = I(y)$ then $x \circ y$ is defined if and only if $(x@\mathcal{A}) \circ (y@\mathcal{A})$ is defined and if either is defined then

$$(x \circ y)@\mathcal{A} = (x@\mathcal{A}) \circ (y@\mathcal{A}).$$

2.13 Value-Abstraction Commutation

In order to state the value-abstraction commutation theorem we first define a template evaluation function $V[e]I$ where $e$ is an expression and $\mathcal{T}$ is an implementation value template and such that $V[e]I$ is a template for $e$. More precisely for $S:V[\Sigma]$ and value $e$ such that $V[e]S$ is defined we have

$$I(V[e]S) = V[e]I(S).$$

The value-abstraction commutation theorem states that for $S:V[\Sigma]$ and for any template abstract value template $\mathcal{A}$ such that $(S@\mathcal{A}):V[\Sigma]$ we have

$$V[e](S@\mathcal{A}) = (V[e]S)@V[e]A.$$ 

An important special case of this occurs when $\Sigma = \Phi:\text{Bool}$. In this case we have the following.

$$V[e](\Phi@V[\Sigma]) = (V[e]\Phi)@V[\Phi]\text{Mem}(I(V[\Sigma])) = (V[e]\Phi)@\text{Bool} = (V[e]\Phi)S$$
2.14 Value-Composition Commutation

Consider $G, H : \mathcal{V}[\Sigma]$ with $G \circ H$ defined. The value-composition commutation theorem states that

$$\forall \Sigma [e] (G \circ H) = (\forall \Sigma [e] G) \circ (\forall \Sigma [e] H)$$

2.15 Commutation with Inverse

The symmetry of left and right implies that abstraction and the value function both commute with inverse. In particular we have

$$(x^{-1})@A = (x@A)^{-1}$$

and

$$\forall \Sigma [e] (S^{-1}) = (\forall \Sigma [e] S)^{-1}.$$ 

2.16 The Abstraction Theorem

Suppose $\Sigma; x: \sigma \models e[x]: \tau$ where $x$ does not occur free in $\tau$ and consider $S : \mathcal{V}[\Sigma]$. Let $\sigma^*$ abbreviate $\forall \Sigma [\sigma] S$ and $\tau^*$ abbreviate $\forall \Sigma [\tau] S$. For $v : \sigma^*$ let $e^*[v]$ abbreviate $\forall \Sigma;x: \sigma [e[x]] S[x \leftarrow v]$. Under these abbreviations the abstraction theorem states that for $u, w : \sigma^*$ we have

$$u =_\sigma^* w \implies e^*[u] =_\tau^* e^*[w].$$

To prove this we note that by the definition of $u =_\sigma^* w$ we have that there exists $z \in \sigma^*$ with $(u@\sigma^*) \circ z^{-1} \circ (w@\sigma^*)$ defined. Omitting some details, the various commutation theorems and the idempotence property then allow the following derivation.

$$e^*(u@\sigma^* \circ z^{-1} \circ w@\sigma^*)@\tau^*$$

$$= (e^*[u@\sigma^*] \circ e^*[z]^{-1} \circ e^*[w@\sigma^*])@\tau^*$$

$$= e^*[u@\sigma^*]@\tau^* \circ (e^*[z]@\tau^*)^{-1} \circ e^*[w@\sigma^*]@\tau^*$$

$$= (e^*[u]@\tau(u))@\tau^* \circ (e^*[z]@\tau^*)^{-1} \circ (e^*[w]@\tau(w))@\tau^*$$

$$= e^*[u]@\tau^* \circ (e^*[z]@\tau^*)^{-1} \circ e^*[w]@\tau^*$$

which yields $e^*[u] =_\tau^* e^*[w]$.

2.17 Voldemort’s Theorem

It seems intuitively clear that certain objects cannot be named. If a circle is defined in a coordinate-free way there is no way to name a point on the circle — any two points satisfy exactly the same coordinate-free properties. A similar statement holds for a basis for a vector space or an isomorphism between
a vector space and its dual — no basis or isomorphism can be named. We will call the inability to name non-canonical objects “Voldemort’s theorem”. Voldemort’s theorem states that if there exists a symmetry of an object \( G \) (an isomorphism of \( G \) with itself) carrying \( x \) to \( y \) then no well-type formula can specify \( x \) as distinct from \( y \).

**Definition 2.1.** We say that there is no canonical instance of a type expressions \( \sigma \) for \( \Sigma \) and \( S \) if \( \Sigma \models \sigma : \text{type}, \) and \( S : \mathcal{V}[\Sigma] \) and for all \( u : \mathcal{V}[\sigma] S \) there exists \( w : \mathcal{V}_\Sigma[\sigma] S \) such that \( w \neq \mathcal{V}_\Sigma[\sigma] u \) but \( S[x \leftarrow u] = \mathcal{V}_\Sigma[\sigma] S[x \leftarrow w] \).

For some particular \( S : \mathcal{V}[\Sigma] \) there may be no canonical instance of \( \sigma \) while for other instance \( S' : \mathcal{V}[\Sigma] \) canonical instances of \( \sigma \) may exist. For example, suppose \( \Sigma \) is the type \( \text{DiGraph} \) of directed graphs. The total directed graph on 5 nodes has no canonical node while a non-empty tree always has a distinguished root. Voldemort’s theorem states that if there is no canonical instance of \( \sigma \) for \( \Sigma \) and \( S \) then no instance of \( \sigma \) can be named by \( \Sigma \)-theoretical properties. More specifically, we have the following.

**Theorem 2.2** (Voldemort’s Theorem). If there is no canonical instance of \( \sigma \) for \( \Sigma \) and \( S \) then for any expression \( \Phi[x] \) satisfying \( \Sigma \models x : \sigma \models \Phi[x] : \text{Bool}, \) and any \( u : \mathcal{V}_\Sigma[\sigma] S \) with \( \mathcal{V}_\Sigma : x : \sigma \Phi[x] \) \( S[x \leftarrow u] = T, \) there exists \( w : \mathcal{V}_\Sigma[\sigma] S \) with \( w \neq \mathcal{V}_\Sigma[\sigma] u \) but \( \mathcal{V}_\Sigma : x : \sigma \Phi[x] \) \( S[x \leftarrow w] = T. \)

**Proof.** By the definition of non-canonicality there exists \( w : \mathcal{V}_\Sigma[\sigma] S \) with \( S[x \leftarrow u] = \mathcal{V}_\Sigma[\sigma] S[x \leftarrow w] \). This implies that there exists \( G[x \leftarrow s] \in \mathcal{V}[\Sigma : x : \sigma] \) with

\[
S[x \leftarrow u] \circ \mathcal{V}[\Sigma : x : \sigma] \circ G[x \leftarrow s]^{-1} \circ S[x \leftarrow w] \text{ defined. }
\]

This definedness implies that \( G \) must be a \( \Sigma \)-symmetry of \( S \). By value-composition commutation, and the Boolean special case of value-abstraction commutation, we have the following.

\[
\mathcal{V}_\Sigma : x : \sigma \Phi[x] S[x \leftarrow u] \circ (\mathcal{V}_\Sigma : x : \sigma \Phi[x] G[x \leftarrow s])^{-1} \circ \mathcal{V}_\Sigma : x : \sigma \Phi[x] S[x \leftarrow w]
\]

But this gives that \( \mathcal{V}_\Sigma : x : \sigma \Phi[x] S[x \leftarrow u] \) must have the same truth value as \( \mathcal{V}_\Sigma : x : \sigma \Phi[x] S[x \leftarrow w]. \) \( \square \)
Chapter 3

Weak-Values

All objects will be taken from a model of set theory with an infinite number of inaccessible cardinals. The existence of inaccessible cardinals is not provable in ZFC, the traditional foundation of mathematics. However, it is generally believed that assuming the existence of an infinite number of inaccessibles does not lead to paradoxes. Gödel proved that no sufficiently rich system can prove its own consistency. Hence any system of inference rules capturing the human notion of mathematical proof must have the property that some not-humanly-provable assumption is required to prove the consistency of the human system. Assuming the existence of an infinite number of inaccessibles seems benign.

We omit the definition of an inaccessible cardinal but note that by assuming an infinite sequence of inaccessible cardinals one can construct an infinite sequence of universes $U_0$, $U_1$, $U_2$, $\ldots$ with $U_i \in U_j$ for $j > i$ and where each $U_i$ is closed under all the methods of forming sets allowed by the axioms of set theory. In the remainder of this manuscript we will implicitly assume that any value constructed in standard ways from elements of $U_i$ is itself in $U_i$. A set constructed from a reference to $U_i$ as a whole is not in $U_i$ but instead in $U_j$ for $j > i$. For example, the semantic value of the function space $\text{type}_i \rightarrow \text{type}_i$ is not in $U_i$ but instead in $U_j$ for $j > i$.

Starting from set theory we let “0” denote the empty set and let “1” denote $\{0\}$. We implement the pair $(x, y)$ as $\{x, \{x, y\}\}$. We implement lists by implementing the empty list as the empty set and implementing a non-empty list as a pair $(x, r)$ where $x$ is the first element of the list and $r$ is the rest of the list (which might or might not be empty). We implement bit strings as lists of 0s and 1s. We implement a byte as a string of eight bits. We can implement byte strings as lists of bytes. We define a symbol to be a list of bytes and write symbols using the standard ASCII conventions. For example, we have the symbols "FOO" and "BAR". An expression is defined recursively to be either a symbol or a list of expressions.
3.1 Pre-Values

We now define a simplified kind of type expression that we will call a template. A template is an expression generated by the following grammar.

\[
T ::= \text{Bool} | \text{Point} | \text{TypeOf}(T) | \text{FUN}(T) | S
\]

\[
S ::= \varsigma | S[x \leftarrow T]
\]

Each pre-value is tagged with a template. More specifically, a pre-value is a pair \((T, u)\) where \(T\) is a template called the template tag of the pre-value, and \(u\) is an object with structure specified by \(T\). More formally, pre-values are defined recursively to be one of the following.

- A Boolean value which is either the pair \((\text{Bool}, 1)\) or the pair \((\text{Bool}, 0)\).
- A point which is a pair of the form \((\text{Point}, (i, j))\) where \(i\) and \(j\) are arbitrary sets called the left index and right index respectively.
- A type-tagged pre-type which is a pair \((\text{TypeOf}(T), O)\) where \(T\) is a template and \(O\) is a set of pre-values all having template tag \(T\). (For example, in the pre-type \((\text{TypeOf(Point)}, O)\) we must have that \(O\) is a set of points.)
- A function-tagged pre-type which is a pair \((\text{FUN}(T), O)\) where \(T\) is a template and \(O\) is a set of pre-values with template tag \(T\).
- A pre-structure which is either the empty structure \((\varsigma, 0)\) or a pre-structure \((S[x \leftarrow T], (G, x, v))\) where \(G\) is a pre-structure with template \(S\) and \(v\) is pre-value with template \(T\).

Note that we can distinguish between empty types of different templates. In this chapter functions will be treated identically to types — a function will ultimately be just a type of pairs.

**Definition 3.1.** For a pre-type \(\sigma = (\text{TypeOf}(T), O)\) we will write \(v \in \sigma\) for \(v \in O\) in which case \(v\) will be called a member of \(\sigma\).

We will write \(\text{Bool}^*\) for the Boolean type \((\text{TypeOf}(\text{Bool}), \{((\text{Bool}, 1), (\text{Bool}, 0))\})\).

We will write \(\varsigma^*\) for \((\varsigma, 0)\) and if \(G\) is a pre-structure with template \(S\) and \(v\) is a pre-value with template \(T\) we will write \(G[x \leftarrow v]\) for the pre-structure \((S[x \leftarrow T], (G, x, v))\). We say that a pre-structure \(G\) assigns a value to variable \(y\) if \(G\) has the form \(H[x \leftarrow v]\) where either \(x = y\) or \(H\) assigns a value to \(y\). If \(G\) assigns a value to \(y\) then we define \(G.y\) by the following rules.

\[
H[y \leftarrow v], y = v \\
H[x \leftarrow v], y = H.y \text{ for } y \neq x
\]

We will write \(T\) as an abbreviation for \((\text{Bool}, 1)\) and \(F\) as an abbreviation for \((\text{Bool}, 0)\). For Boolean values \(\Phi\) and \(\Psi\) we will write \(\Phi \lor \Psi\) for the Boolean value which is the disjunction of \(\Phi\) and \(\Psi\) and \(\neg \Phi\) for Boolean value that is the negation of \(\Phi\).
3.2 Operations on Pre-Values

We now give mutually recursive definitions of operations on pre-values. More specifically we define the left operator **Left**, the right operator **Right**, the inverse operation \( {}^{-1} \) and the composition operation \( \circ \). All operations preserve template tags. The operation **Left** is defined as follows where \( O^{-1} \) is the set of values of the form \( x^{-1} \) for \( x \in O \) and where \( O_1 \circ O_2 \) is the set of values of the form \( x \circ y \) for \( x \in O_1 \) and \( y \in O_2 \) with \( x \circ y \) defined.

\[
\text{Left}((\text{Bool}, v)) = (\text{Bool}, v) \\
\text{Left}((\text{Point}, (a, b))) = (\text{Point}, (a, a)) \\
\text{Left}((\text{TypeOf}(\mathcal{T}), O)) = (\text{TypeOf}(\mathcal{T}), O \circ O^{-1}) \\
\text{Left}((\text{FUN}(\mathcal{T}), O)) = (\text{FUN}(\mathcal{T}), O \circ O^{-1}) \\
\text{Left}(((\varsigma, 0))) = (\varsigma, 0) \\
\text{Left}((\mathcal{S}[x \leftarrow \mathcal{T}], G[x \leftarrow v])) = (\mathcal{S}[x \leftarrow \mathcal{T}], \text{Left}(G)[x \leftarrow \text{Left}(v))].
\]

The operation **Right** is defined similarly but with \( \text{Right}((\text{Point}, (a, b))) = (\text{Point}, (b, b)) \) and using \( O^{-1} \circ O \) in place of \( O \circ O^{-1} \).

We define the inverse operation as follows.

\[
(\text{Bool}, v)^{-1} = (\text{Bool}, v) \\
(\text{Point}, (a, b))^{-1} = (\text{Point}, (b, a)) \\
(\text{TypeOf}(\mathcal{T}), O)^{-1} = (\text{TypeOf}(\mathcal{T}), O^{-1}) \\
(\text{FUN}(\mathcal{T}), O)^{-1} = (\text{FUN}(\mathcal{T}), O^{-1}) \\
((\varsigma, 0))^{-1} = (\varsigma, 0) \\
(\mathcal{S}[x \leftarrow \mathcal{T}], G[x \leftarrow v])^{-1} = (\mathcal{S}[x \leftarrow \mathcal{T}], G^{-1}[x \leftarrow v^{-1}]).
\]

The composition \( x \circ y \) is defined when \( \text{Right}(x) = \text{Left}(y) \). In this case the composition is defined by the following rules.

\[
(\text{Bool}, v) \circ (\text{Bool}, v) = (\text{Bool}, v) \\
(\text{Point}, (a, b)) \circ (\text{Point}, (b, c)) = (\text{Point}, (a, c)) \\
(\text{TypeOf}(\mathcal{T}), O_1) \circ (\text{TypeOf}(\mathcal{T}), O_2) = (\text{TypeOf}(\mathcal{T}), O_1 \circ O_2) \\
(\text{FUN}(\mathcal{T}), O_1) \circ (\text{FUN}(\mathcal{T}), O_2) = (\text{FUN}(\mathcal{T}), O_1 \circ O_2) \\
((\varsigma, 0)) \circ (\varsigma, 0) = (\varsigma, 0) \\
(\mathcal{S}[x \leftarrow \mathcal{T}], G[x \leftarrow v]) \circ (\mathcal{S}[x \leftarrow \mathcal{T}], H[x \leftarrow w]) = (\mathcal{S}[x \leftarrow \mathcal{T}], (G \circ H)[x \leftarrow (v \circ w))].
\]

**Lemma 3.2.** For pre-values \( x \) and \( y \) we have \( \text{Left}(x) = x \circ x^{-1} \) and \( \text{Right}(x) = x^{-1} \circ x \).

**Proof.** The proof is by induction on the size of the template of \( x \). The result is immediate for Booleans and points. The results follows directly from the definition for types and follows directly from the induction hypothesis for structures. \(\square\)
### 3.3 Weak Values

A point specifies a pair of indeces. A pre-type whose members are points defines a binary relation on the indeces occurring in the points. The inverse and composition operations on pre-types of points correspond to the standard composition and inverse operations on binary relations. If types are to represent bijections then for a point-type \( \tau \) we want that \( \text{Left}(\tau) \) and \( \text{Right}(\tau) \) are both equivalence relations and that \( \tau \) forms a bijection between the left equivalence classes and the right equivalence classes. It turns out that these requirements are equivalent to the single condition for \( x, y, z \in \sigma \) with \( x \circ y^{-1} \circ z \) defined we have \( x \circ y^{-1} \circ z \in \sigma \).

**Definition 3.3.** A pre-type \( \sigma \) is said to be back-and-forth closed if for all \( x, y, x \in \sigma \) with \( (x \circ y^{-1}) \circ z \) defined we have \( (x \circ y^{-1}) \circ z \in \sigma \).

**Definition 3.4.** We define a point type to be a type \( \sigma \) whose members are points and such that \( \sigma \) is back-and-forth closed.

**Definition 3.5.** A weak-value is one of the following.

- A Boolean value or a point.
- A type-tagged pre-type \((\text{OfType}(T), \mathcal{O})\) such that every member of \( \mathcal{O} \) is a weak-value and \( \mathcal{O} \) is back-and-forth closed.
- A function-tagged pre-type \((\text{FUN}(T), \mathcal{O})\) such that every member of \( \mathcal{O} \) is a weak-value and \( \mathcal{O} \) is back-and-forth closed.
- A pre-structure \( G \) such that for every variable \( x \) assigned in \( G \) we have that \( G.x \) is a weak-value.

**Definition 3.6.** A type-tagged weak-value will be called a weak type. A weak type will be said to form a groupoid if its members are closed under composition and inverse.

### 3.4 Algebraic Laws of Composition and Inverse

**Lemma 3.7.** Weak-values satisfy the following properties.

1. If \( x \) is a weak-value then \( x^{-1} \) is a weak-value and if \( x \) and \( y \) are weak-values with \( x \circ y \) defined then \( x \circ y \) is a weak-value.
2. \( (x^{-1})^{-1} = x \)
3. \( \text{Left}(x^{-1}) = \text{Right}(x) \) and \( \text{Right}(x^{-1}) = \text{Left}(x) \)
4. \( (x \circ y)^{-1} = y^{-1} \circ x^{-1} \)
5. If \( x \circ y \) is defined then \( \text{Left}(x \circ y) = \text{Left}(x) \) and \( \text{Right}(x \circ y) = \text{Right}(y) \).
6. \((x \circ y) \circ z = x \circ (y \circ z)\).

7. If \(x \circ y\) is defined then \(x^{-1} \circ x \circ y = y\) and \(x \circ y \circ y^{-1} = x\).

8. (composition partner) For weak-types \(\sigma\) and \(\tau\) with \(\sigma \circ \tau\) defined we have that for all \(x \in \sigma\) there exists \(y \in \tau\) with \(x \circ y\) defined and for all \(y \in \tau\) there exists \(x \in \sigma\) with \(x \circ y\) defined.

\[\text{Proof.}\] We prove by structural induction on templates that for any template \(T\) all of these conditions hold for all abstract instances of \(T\). We consider a template \(T\) and assume that all of the conditions hold for all abstract instances of proper subexpressions (proper subtemplates) of \(T\). It then suffices to show that all of the conditions hold for abstract instances of \(T\). We consider each condition. We do not consider the case of functions as the proof for functions is always the same as the proof for types.

1. The result is immediate for Booleans and points. For structures the result follows straightforwardly from the induction hypothesis. Now consider a type weak-value \(\sigma = (\text{TypeOf}(T), \mathcal{O})\). We have assumed all the properties of the lemma for members of \(\sigma\). In particular, for \(x \in \sigma\) we have that \(x^{-1}\) is a weak-value which implies that every abstract instance of \(\sigma^{-1}\) is a weak-value. We must also show that \(\sigma^{-1}\) satisfies the closure property of type weak-values. Consider \(x^{-1}, y^{-1}, z^{-1} \in \sigma^{-1}\) with \(x, y, z \in \sigma\) and with \(((x^{-1} \circ (y^{-1})^{-1}) \circ z^{-1})\) defined. Since \(x, y\) and \(z\) are instances of a (proper) subexpression of \(T\) we have that all properties hold for \(x, y\) and \(z\). In particular we have that \(x^{-1} \circ (y^{-1})^{-1} \circ z^{-1} = (z \circ y^{-1} \circ x)^{-1}\). By the closure property of \(\sigma\) we have that \(z \circ y^{-1} \circ x\) is an abstract instance of \(\sigma\) and we are done. Finally consider two type weak types \(\sigma\) and \(\tau\) with template \(T\) and with \(\sigma \circ \tau\) defined. By the induction hypothesis on the members of \(\sigma\) and \(\tau\) we get that every member of \(\sigma \circ \tau\) is a weak-value. To show the closure property consider \((x_1 \circ y_1) \circ (x_2 \circ y_2)^{-1} \circ (x_3 \circ y_3)\) with \(x_i \circ y_i \in \sigma \circ \tau\). Since \(\sigma \circ \tau\) is defined we have \(\text{Right}(\sigma) = \text{Left}(\tau)\). It then follows from the definition of \(\text{Right}\) and \(\text{Left}\) that every weak-value of the form \(x_1^{-1} \circ x_2\) for \(x_1, x_2 \in \sigma\) is equal to some weak-value of the form \(y_1 \circ y_2^{-1}\) for \(y_1, y_2 \in \tau\) and vice-versa. We then have the following.

\[
(x_1 \circ y_1) \circ (x_2 \circ y_2)^{-1} \circ (x_3 \circ y_3) = x_1 \circ (y_1 \circ y_2^{-1}) \circ x_2^{-1} \circ x_3 \circ y_3 = x_1 \circ (x_4^{-1} \circ x_5) \circ x_2^{-1} \circ x_3 \circ y_3 = ((x_1 \circ x_4^{-1} \circ x_5) \circ x_2^{-1} \circ x_3) \circ y_3 = x_7 \circ y_3
\]

2. We must show \((x^{-1})^{-1} = x\). The case where \(x\) is a Boolean value or a point is immediate from the definitions. The case where \(x\) is a structure follows directly from the induction hypothesis. For type we must show that for any set of weak-values \(\mathcal{O}\) such that the lemma holds for all elements of \(\mathcal{O}\) we have that \((\mathcal{O}^{-1})^{-1} = \mathcal{O}\). But this follows immediately from the definition of \(\mathcal{O}^{-1}\) and the induction hypothesis on the elements of \(\mathcal{O}\).

3. We will show \(\text{Left}(x^{-1}) = \text{Right}(x)\). The case of \(\text{Right}(x^{-1}) = \text{Left}(x)\) is similar. If \(x\) is a Boolean value or point the result follows directly from the
definitions. The case of structure weak-values follows straightforwardly from the induction hypothesis. For a type \( \sigma \), we have that the members of \( \text{Left}(\sigma^{-1}) \) are the values of the form \( x^{-1} \circ (y^{-1})^{-1} \) for \( x, y \in O \). On the other hand the members of \( \text{Right}(\sigma) \) are the values of the form \( x^{-1} \circ y \) for \( x, y \in O \). But by the induction hypothesis we have \( (y^{-1})^{-1} = y \) and hence the two value sets are the same.

4. We must show \((x \circ y)^{-1} = y^{-1} \circ x^{-1}\). Note that \( x \circ y \) is defined if and only if \( \text{Left}(x) = \text{Right}(y) \). By the above cases, this is the same as the conditions under which \( y^{-1} \circ x^{-1} \) is defined. We now show equality in the case where both are defined. The case of Boolean values and points is immediate. The case of structures follows straightforwardly from the induction hypothesis. For types \( \sigma \) and \( \tau \) with \( \sigma \circ \tau \) defined we have that the members of \( (\sigma \circ \tau)^{-1} \) are the values of the form \( (x \circ y)^{-1} \) for \( x \in \sigma \) and \( y \in \tau \). The members of \( \tau^{-1} \circ \sigma^{-1} \) are the values of the form \( y^{-1} \circ x^{-1} \). But by the induction hypothesis these are the same sets.

5. We will show \( \text{Left}(x \circ y) = \text{Left}(x) \). The case of \( \text{Right}(x \circ y) = \text{Right}(y) \) is similar. The case of Boolean values and points follows directly from the definitions. The case of structures follows straightforwardly from the induction hypothesis. Now consider \( \sigma \) and \( \tau \) with \( \sigma \circ \tau \) defined. We will use \( x \) to range over members of \( \sigma \) and \( y \) range over members of \( \tau \). We first show that every member of \( \text{Left}(\sigma \circ \tau) \) is an member of \( \text{Left}(\sigma) \). Working though the definitions we have that a member of \( \text{Left}(\sigma \circ \tau) \) has the form \( (x_1 \circ y_1) \circ (x_2 \circ y_2)^{-1} \). By the induction hypothesis we have the following.

\[
(x_1 \circ y_1) \circ (x_2 \circ y_2)^{-1} = x_1 \circ (y_1 \circ y_2^{-1}) \circ x_2^{-1} = x_1 \circ (x_3^{-1} \circ x_4) \circ x_2^{-1} = ((x_1 \circ x_3^{-1}) \circ x_4) \circ x_2^{-1} = x_5 \circ x_2^{-1} \in \text{Left}(\sigma)
\]

For the converse we consider an abstract instance \( x_1 \circ x_2^{-1} \) of \( \text{Left}(\sigma) \) and we have the following.

\[
x_1 \circ x_2^{-1} = x_1 \circ x_2^{-1} \circ x_2 \circ x_2^{-1} = x_1 \circ (x_2^{-1} \circ x_2) \circ x_2^{-1} = x_1 \circ (y_1 \circ y_2^{-1}) \circ x_2^{-1} = (x_1 \circ y_1) \circ (y_2^{-1} \circ x_2^{-1}) = (x_1 \circ y_1) \circ (x_2 \circ y_2)^{-1} \in \text{Left}(\sigma \circ \tau)
\]

6. Clause 5 implies that if \((x \circ y) \circ z\) is defined then \( x \circ (y \circ z) \) is defined and vice-versa. We consider the case where both are defined. For Boolean values and points the result follows from the definitions. For the case of structures the result follows straightforwardly from the induction hypothesis. Now consider types \( \sigma, \tau \) and \( \gamma \) where \( \sigma \circ \tau \circ \gamma \) is defined. The abstract instances of \( (\sigma \circ \tau) \circ \gamma \) are the values of the form \( (x \circ y) \circ z \) for \( x \in \sigma, y \in \tau \) and \( z \in \gamma \). But by the induction hypothesis these are the same as the members of \( \sigma \circ (\tau \circ \gamma) \).

7. We will show \( x^{-1} \circ x \circ y = y \). The case of \( x \circ y \circ y^{-1} \) is similar. For Booleans and points the result follows from the definitions. For structures the
result follows straightforwardly from the induction hypothesis. Now consider types $\sigma$ and $\tau$ with $\sigma \circ \tau$ defined. We must show $\sigma^{-1} \circ \sigma \circ \tau = \tau$. We will let $x$ range over abstract instances of $\sigma$ and $y$ range over abstract instances of $\tau$. We first show that every abstract instance $y$ of $\tau$ is an abstract instance of $\sigma^{-1} \circ \sigma \circ \tau$. By clause 8 of the induction hypothesis there exists $x \in \sigma$ with $x \circ y$ defined. By the induction hypothesis we then have $y = (x^{-1} \circ x \circ y) \in (\sigma^{-1} \circ \sigma \circ \tau)$. Finally for $x_1^{-1} \circ x_2 \circ y_1 \in \sigma^{-1} \circ \sigma \circ \tau$ we have

\[
(x_1^{-1} \circ x_2) \circ y_1 = (y_2 \circ y_1^{-1}) \circ y_1 \in \tau.
\]

8. We will show if $\sigma \circ \tau$ is defined and $y \in \tau$ then there exists $x \in \sigma$ with $x \circ y$ defined. Letting $x$ range over abstract instances of $\sigma$ and $y$ range over abstract instances of $\tau$ we have

\[
y = (y \circ y^{-1}) \circ y = (x_1^{-1} \circ x_2) \circ y
\]

which implies that $x_2 \circ y$ is defined. \hfill \box

### 3.5 Weak-Types as Bijections

**Definition 3.8.** For a weak-type $\sigma$ and for $x, y \in \sigma$ we will write $x \simeq_{\sigma} y$ if there exists $z \in \sigma$ with $x \circ z^{-1} \circ y$ defined.

**Lemma 3.9.** For any weak-type $\sigma$ we have that $\simeq_{\sigma}$ is an equivalence relation.

**Proof.** For any $x \in \sigma$ we have $x \circ x^{-1} \circ x$ is defined and hence $x \simeq_{\sigma} x$. To show symmetry suppose $x \simeq_{\sigma} y$ with $x \circ s^{-1} \circ y$ defined. In this case we have that $y \circ (x \circ s^{-1} \circ y)^{-1} \circ x$ is defined and hence $y \simeq_{\sigma} x$. For transitivity consider $x \simeq_{\sigma} y$ and $y \simeq_{\sigma} z$ with $x \circ s^{-1} \circ y$ and $y \circ u^{-1} \circ z$ defined. We then have that $x \circ (u \circ y^{-1} \circ s)^{-1} \circ z$ is defined and hence $x \simeq_{\sigma} z$. \hfill \box

**Lemma 3.10** (composables are equivalent). If $\sigma$ is a weak-type then for $x, y \in \sigma$ with $x \circ y^{-1}$ defined or with $x^{-1} \circ y$ defined we have $x \simeq_{\sigma} y$.

**Proof.** We consider the case of $x \circ y^{-1}$ defined. In this case $x \circ y^{-1} \circ y$ is also defined. \hfill \box

**Lemma 3.11** (Left and Right are Injective). For any weak-type $\sigma$ and $x, y \in \sigma$ we have $x \simeq_{\sigma} y$ if and only if $\text{Left}(x) \simeq_{\text{Left}(\sigma)} \text{Left}(y)$ if and only if $\text{Right}(x) \simeq_{\text{Right}(\sigma)} \text{Right}(y)$.

**Proof.** We will show that $x \simeq_{\sigma} y$ if and only if $\text{Left}(x) \simeq_{\text{Left}(\sigma)} \text{Left}(y)$. First suppose that $x \simeq_{\sigma} y$ in which case there exists $z \in \sigma$ with $x \circ z^{-1} \circ y$ defined. We then have that $x \circ x^{-1} \circ z \circ z^{-1} \circ y \circ y^{-1}$ is defined which equals $(x \circ x^{-1}) \circ (z \circ z^{-1})^{-1} \circ (y \circ y^{-1})$ and hence $\text{Left}(x) \simeq_{\text{Left}(\sigma)} \text{Left}(y)$. Now suppose
that \( \text{Left}(x) \simeq_{\text{Left}(\sigma)} \text{Left}(y) \). In this case there exists \( z_1, z_2 \in \sigma \) such that 
\[
(x \circ x^{-1}) \circ (z_1 \circ z_2^{-1})^{-1} \circ (y \circ y^{-1})
\]
is defined. In this case we have
\[
(x \circ x^{-1}) \circ (z_1 \circ z_2^{-1})^{-1} \circ y = x \circ (x^{-1} \circ z_2 \circ z_1^{-1}) \circ y
\]
\[
= x \circ (z_1 \circ z_2^{-1} \circ x)^{-1} \circ y
\]
\[
= x \circ z_3^{-1} \circ y.
\]
and we have \( x \simeq \sigma y \).

**Lemma 3.12** (Left and Right are Surjective). For a weak-type \( \sigma \) the operation \( \text{Left} \) from abstract instances of \( \sigma \) to abstract instances of \( \text{Left}(\sigma) \) is onto in the sense that for all \( z \in \text{Left}(\sigma) \) there exists \( x \in \sigma \) with \( z \simeq_{\text{Left}(\sigma)} \text{Left}(x) \). A similar statement holds for right.

**Proof.** Consider \( x_1 \circ x_2^{-1} \in \text{Left}(\sigma) \). We have that \((x_1 \circ x_2^{-1}) \circ (x_1 \circ x_2^{-1})\) is defined and by the equivalence of composites (lemma 3.10) we then have \( \text{Left}(x_1) \simeq_{\text{Left}(\sigma)} x_1 \circ x_2^{-1} \).

Lemmas 3.11 and 3.12 together imply that a weak-type defines a bijection. More specifically, the operation \( \text{Left} \) defines a bijection from \( \sigma \) to \( \text{Left}(\sigma) \) and similarly for \( \text{Right}. \) These two bijections together define a bijection between \( \text{Left}(\sigma) \) and \( \text{Right}(\sigma) \).

**Lemma 3.13** (Composition Preserves Equivalence). For weak-types \( \sigma \) and \( \tau \) with \( \sigma \circ \tau \) defined and for \( x_1, x_2 \in \sigma \) and \( y_1, y_2 \in \tau \) with \( x_1 \circ y_1 \) and \( x_2 \circ y_2 \) defined, we have \((x_1 \circ y_1) \simeq_{\sigma \circ \tau} (x_2 \circ y_2)\) if and only if \( x_1 \simeq \sigma x_2 \) if and only if \( y_1 \simeq \tau y_2 \).

**Proof.** We will let \( x \) range over abstract instances of \( \sigma \) and \( y \) range over abstract instances of \( \tau \). We first show that \((x_1 \circ y_1) \simeq_{\sigma \circ \tau} (x_2 \circ y_2)\) implies \( x_1 \simeq \sigma x_2 \). If \((x_1 \circ y_1) \simeq_{\sigma \circ \tau} (x_2 \circ y_2)\), then by definition there exists \( x_3 \) and \( y_3 \) with \((x_1 \circ y_1) \circ (x_3 \circ y_3)^{-1} \circ (x_2 \circ y_2)\) defined. We then have that the following are defined.

\[
x_1 \circ y_1 \circ (x_3 \circ y_3)^{-1} \circ x_2 = x_1 \circ (y_1 \circ y_2^{-1}) \circ x_3^{-1} \circ x_2
\]
\[
= x_1 \circ (x_4^{-1} \circ x_5) \circ x_3^{-1} \circ x_2
\]

Lemma 3.10 now implies \( x_1 \simeq \sigma x_2 \).

We now show that \( x_1 \simeq \sigma x_2 \) implies \( y_1 \simeq \tau y_2 \). By lemma 3.11 we have that \( x_1 \simeq \sigma x_2 \) if and only if \( \text{Right}(x_1) \simeq_{\text{Right}(\sigma)} \text{Right}(x_2) \). But \( \text{Right}(x_1) = \text{Left}(y_1) \), \( \text{Right}(x_2) = \text{Left}(y_2) \) and \( \text{Right}(\sigma) \) equals \( \text{Left}(\tau) \). So we have that \( x_1 \simeq \sigma x_2 \) if and only if \( \text{Left}(y_1) \simeq_{\text{Left}(\tau)} \text{Left}(y_2) \) if and only if \( y_1 \simeq \tau y_2 \).

Finally we show that if \( x_1 \simeq \sigma x_2 \) and \( y_1 \simeq \tau y_2 \) then \( x_1 \circ y_1 \simeq_{\sigma \circ \tau} x_2 \circ y_2 \). By definition there exists \( x_3 \) and \( y_3 \) with \( x_1 \circ x_3 \circ x_2 \) and \( y_2 \circ y_3 \circ y_1 \) defined. This implies that \( \text{Right}(x_3) = \text{Right}(x_1) \) = \( \text{Left}(y_1) \) = \( \text{Left}(y_3) \) and we have that \( x_3 \circ y_3 \) is defined. We then have that \((x_1 \circ y_1) \circ (x_3 \circ y_3)^{-1} \circ (x_2 \circ y_2)\) is defined and hence \( x_1 \circ y_1 \simeq_{\sigma \circ \tau} x_2 \circ y_2 \).
Chapter 4

Implementation and Abstraction

Implementation and abstraction are central to this manuscript. As discussed in the chapter 2 for any type $\sigma$ we distinguish be a member of $\sigma$, written $x \in \sigma$, and an implementation of a member of $\sigma$, written $x : \sigma$. The proof of the abstraction theorem requires that the space of implementations is actually a different value space from the space of “abstract” values — the values that are members of types. We start by introducing a more concrete notion of function.

4.1 Functions

Definition 4.1. For syntactic expressions $x$ and $y$ we will write $\text{Pair}(x, y)$ for the syntactic expression $\varsigma[\text{first} \leftarrow x][\text{second} \leftarrow y]$. For pre-values $x$ and $y$ we will write $\text{Pair}(x, y)$ for the semantic value $\varsigma^* [\text{first} \leftarrow x][\text{second} \leftarrow y]$.

Definition 4.2. For a function-tagged pre-type $f = (\text{FUN}(\ldots), \mathcal{O})$ we will write $\text{Pair}(x, y) \in f$ as an alternate notation for $\text{Pair}(x, y) \in \mathcal{O}$. We also write $\text{Dom}(f)$ for the type whose members the values $x$ such that there exists a $y$ with $\text{Pair}(x, y) \in f$.

Definition 4.3. A function-tagged pre-value of the form

$$f = (\text{FUN}(\varsigma[\text{first} \leftarrow \text{Point}][\text{second} \leftarrow T]), \mathcal{O})$$

is a function if $\text{Dom}(f)$ is a point type (is back-and-forth closed), for $\text{Pair}(x, y) \in f$ we have that $y$ is a weak value, and for $x_1, x_2 \in \text{Dom}(f)$ with $x_1 \simeq_{\text{Dom}(f)} x_2$ and for $\text{Pair}(x_1, y_1) \in f$ and $\text{Pair}(x_2, y_2) \in f$ we have $y_1 = y_2$.

We now have that if $\text{Pair}(x, y_1) \in f$ and $\text{Pair}(x, y_2) \in f$ then $y_1 = y_2$. This allows us to define the following notation.

Definition 4.4. For a function $f$ and for $x \in \text{Dom}(f)$ we define $f[x]$ to be the unique $y$ such that $\text{Pair}(x, y) \in f$. 

35
A function has the property that for \(x \simeq_{\text{Dom}(f)} y\) we have \(f[x] = f[y]\). Note that the second equation is set-theoretic (absolute) equality. We note that the identity function on a point type \(\sigma\) — the function-tagged type \(f\) with \(\text{Dom}(f) = \sigma\) and such that \(f[x] = x\) for all \(x \in \sigma\) — is not a function when the \(\pi\) equivalence classes contain more than one element. The identity function fails to satisfy \(f[x] = f[y]\) for \(x \simeq y\) with \(x \neq y\). However, we also note that the only inference rule for introducing functions is the axiom of choice. Furthermore, as required by the axiom of choice, for any point type \(\sigma\) there is a function \(f\) with \(\text{Dom}(f) = \sigma\) and such that for all \(x \in \sigma\) we have \(f[x] =_{\sigma} x\).\(^1\)

**Lemma 4.5.** Functions are weak values.

**Proof.** Each pair in \(f\) is a weak value so we need only show that \(f\) is back-and-forth closed. We must show that for \(\text{Pair}(x_1, y_1) \in f\), \(\text{Pair}(x_2, y_2) \in f\), \(\text{Pair}(x_3, y_3) \in f\) with \(x_1 \circ x_2^{-1} \circ x_3\) defined and \(y_1 \circ y_2^{-1} \circ y_3\) defined we have \(\text{Pair}(x_1 \circ x_2^{-1} \circ x_3, y_1 \circ y_2^{-1} \circ y_3) \in f\). But we have \((x_1 \circ x_2^{-1} \circ x_3) \in \text{Dom}(f)\) and \(x_1 \simeq_{\text{Dom}(f)} x_2 \simeq_{\text{Dom}(f)} x_3 \simeq_{\text{Dom}(f)} x_1 \circ x_2^{-1} \circ x_3\). By the definition of a function we then have that \(y_1 = y_2 = y_3 = f[x_1 \circ x_2^{-1} \circ x_3]\) which proves the result. \(\blacksquare\)

**Lemma 4.6.** If \(f\) is a function then \(\text{Left}(f)\) is a function where we have \(\text{Dom}(\text{Left}(f)) = \text{Left}(\text{Dom}(f))\) and \(\text{Left}(f)[\text{Left}(x)] = \text{Left}(f[x])\) and similarly for Right.

**Proof.** We will let \(x\) range over abstract instances of \(\text{Dom}(f)\). For \(x_1 \circ x_2^{-1}\) defined we have \(x_1 \simeq_{\text{Dom}(f)} x_2\) and hence \(f[x_1] = f[x_2]\). This implies that the pairs in \(\text{Left}(f)\) have the form \(\text{Pair}(x_1 \circ x_2^{-1}, \text{Left}(f[x_1]))\). This also shows that \(\text{Left}(f)\) has exactly one output value for each input value and hence the notation \(\text{Left}(f)[x_1 \circ x_2^{-1}]\) is well defined. We then have \(\text{Left}(f)(x_1 \circ x_2^{-1}) = \text{Left}(f[x_1]) = \text{Left}(f[x_2])\). We then have \(\text{Left}(f)[\text{Left}(x)] = \text{Left}(f[x \circ x^{-1}]) = \text{Left}(f[x])\). We also have that \(\text{Dom}(\text{Left}(f))\) is the set of values of the form \(x_1 \circ x_2^{-1}\) which is the same as the set of instances of \(\text{Left}(\text{Dom}(f))\). Finally we must show that \(\text{Left}(f)\) is a function — that equivalent inputs yield the same output. Consider \((x_1 \circ x_2^{-1}) \in \text{Left}(\text{Dom}(f))\) and \((x_3 \circ x_4^{-1}) \in \text{Left}(\text{Dom}(f))\) with \(x_1 \circ x_3^{-1} \simeq_{\text{Left}(\text{Dom}(f))} x_3 \circ x_4^{-1}\). This implies that there exists \(x_5\) and \(x_6\) with \(x_1 \circ x_2^{-1} \circ (x_5 \circ x_6^{-1})^{-1} \circ x_3 \circ x_4^{-1}\) defined. This implies \(x_1 \simeq_{\text{Dom}(f)} x_3\) which implies

\[
\text{Left}(f)[x_1 \circ x_2^{-1}] = \text{Left}(f[x_1]) = \text{Left}(f[x_3]) = \text{Left}(f[x_3 \circ x_4^{-1}]).
\]

\(^1\)The definition of a function used here is not compatible with dependent function types (dependent product types). For example, there is no function \(f\) such that for all point types \(\sigma\) we have \(f(\sigma) : \sigma\).
Lemma 4.7. For two functions \( f \) and \( g \) we have that \( f \circ g \) is defined if and only if \( \text{Dom}(f) \circ \text{Dom}(g) \) is defined and for all \( x \in \text{Dom}(f) \) and \( y \in \text{Dom}(g) \) with \( x \circ y \) defined we have that \( f[x] \circ g[y] \) is also defined.

Proof. Suppose that \( f \circ g \) is defined. We then have \( \text{Right}(f) = \text{Left}(g) \) and by Lemma 1.6 we have that \( \text{Right}(\text{Dom}(f)) = \text{Dom}(\text{Right}(f)) = \text{Dom}(\text{Left}(g)) = \text{Left}(\text{Dom}(g)) \). So we have \( \text{Dom}(f) \circ \text{Dom}(g) \) is defined. Now for \( x \in \text{Dom}(f) \) and \( y \in \text{Dom}(g) \) with \( x \circ y \) defined we have \( \text{Right}(f[x]) = \text{Right}(f)[\text{Right}(x)] = \text{Left}(g)[\text{Left}(y)] = \text{Left}(g[y]) \). Hence we have that \( f[x] \circ g[y] \) is defined.

Conversely, suppose that \( \text{Dom}(f) \circ \text{Dom}(g) \) is defined and for all \( x \in \text{Dom}(f) \) and \( y \in \text{Dom}(g) \) with \( x \circ y \) defined we have that \( f[x] \circ g[y] \) is defined. We must show that the set of pairs in \( \text{Right}(f) \) is the same as the set of pairs in \( \text{Left}(g) \). We will show that every pair in \( \text{Right}(f) \) is in \( \text{Left}(g) \). The converse is similar. Let \( x \) range over elements of \( \text{Dom}(f) \) and let \( y \) range over elements of \( \text{Dom}(g) \). The members of \( \text{Right}(f) \) are the pairs of the form \((x_1^{-1} \circ x_2, f[x_1]^{-1} \circ f[x_2])\) for \( x_1, x_2 \in \text{Dom}(f) \). By the equivalence of composites (Lemma 3.10) any such pair has the property that \( x_1 \simeq_{\text{Dom}(f)} x_2 \) in which case \( f[x_1] = f[x_2] \). So the members of \( \text{Right}(f) \) can also be characterized as the pairs of the form \((x_1^{-1} \circ x_2, \text{Right}(f[x_1]))\). Similarly the members of \( \text{Left}(g) \) are the pairs of the form \((y_1 \circ y_2^{-1}, \text{Left}(f[y_1]))\). Now consider a particular pair \((x_1^{-1} \circ x_2, \text{Right}(f[x_1]))\) in \( \text{Right}(f) \). It now suffices to show that this pair is in \( \text{Left}(g) \). Since \( \text{Right}(\text{Dom}(f)) = \text{Left}(\text{Dom}(g)) \) we have that \( x_1^{-1} \circ x_2 = y_1 \circ y_2^{-1} \) for some \( y_1 \) and \( y_2 \). We then have that \( \text{Right}(x_1) = \text{Left}(y_1) \) and hence \( x_1 \circ y_1 \) is defined and hence \( \text{Right}(f[x_1]) = \text{Left}(g[y_1]) \). We now have that \((x_1^{-1} \circ x_2, \text{Right}(f[x_1]))\) can be rewritten as \((y_1 \circ y_2^{-1}, \text{Left}(g[y_1]))\) in \( \text{Left}(g) \). □

Lemma 4.8. If \( f \) is a function then \( f^{-1} \) is a function with \( f^{-1}[x^{-1}] = f[x]^{-1} \).

Lemma 4.9. If \( f \) and \( g \) are functions with \( f \circ g \) defined then \( f \circ g \) is a function with \( \text{Dom}(f \circ g) = \text{Dom}(f) \circ \text{Dom}(g) \) and for \( x \in \text{Dom}(f) \) and \( y \in \text{Dom}(g) \) with \( x \circ y \) defined we have \( (f \circ g)[x \circ y] = f[x] \circ g[y] \).

Proof. Suppose that \( f \circ g \) is defined. By Lemma 4.7 we have that \( \text{Dom}(f) \circ \text{Dom}(g) \) is defined and for \( x \in \text{Dom}(f) \) and \( y \in \text{Dom}(g) \) with \( x \circ y \) defined we have that \( f[x] \circ g[y] \) is defined. We then have that the pairs of \( f \circ g \) are exactly the pairs of the form \( \text{Pair}(x \circ y, f[x] \circ g[y]) \) with \( x \circ y \) defined. So if \( f \circ g \) is a function then we have \( (f \circ g)[x \circ y] = f[x] \circ g[y] \). To show that \( f \circ g \) is a function let \( x \) range over members of \( \text{Dom}(f) \) and let \( y \) range over instances of \( \text{Dom}(g) \). We have that if \((x_1 \circ y_1) \simeq_{\text{Dom}(f)} \circ \text{Dom}(g) \) \((x_2 \circ y_2) \) then there exists \( x_3 \) and \( y_3 \) with \( x_1 \circ y_1 \circ (x_3 \circ y_3)^{-1} \circ x_2 \circ y_2 \) defined. We then have that \( x_1 \circ y_1 \circ y_3^{-1} \circ x_3^{-1} \circ x_2 \) is defined. But since \( \text{Left}(\text{Dom}(g)) = \text{Right}(\text{Dom}(f)) \) we have that \( y_1 \circ y_3^{-1} \) can be written as \( x_4^{-1} \circ x_5 \) which by Lemma 3.10 gives that \( x_1, x_4, x_5, x_3 \) and \( x_2 \) are all equivalent under \( \simeq_{\text{Dom}(f)} \). This gives \( f[x_1] = f[x_2] \). Similarly we get \( g[y_1] = g[y_2] \). We then have \( (f \circ g)[x_1 \circ y_1] = (f \circ g)[x_2 \circ y_2] \) and hence \( f \circ g \) is a function. □
CHAPTER 4. IMPLEMENTATION AND ABSTRACTION

4.2 Abstract Values and Implementations

Definition 4.10. We define an abstract value to be a Boolean value, a point, a point type, a function \( f \) such that for \( x \in \text{Dom}(f) \) we have that \( f[x] \) is an abstract value, the empty structure, or a structure of the form \( G[x \leftarrow v] \) where \( G \) and \( v \) are abstract values.

Definition 4.11. We let \( \text{Point} \rightarrow A \) abbreviate the template \( \text{FUN}(\varsigma[\text{first} \leftarrow \text{Point}][\text{second} \leftarrow A]) \).

Definition 4.12. An abstract template is a template \( A \) generated by the following grammar.

\[
A ::= \text{Bool} | \text{Point} | \text{TypeOf(Point)} | \text{Point} \rightarrow A | S
\]

\[
S ::= \varsigma | S[x \leftarrow A]
\]

Lemma 4.13. All abstract values have abstract templates.

Lemma 4.14. All abstract values are also weak values.

Proof. The proof is by induction on template size. The case of functions is implied by lemma 4.5.

Lemma 4.15. Abstract values are closed under inverse and composition.

Proof. The proof is by induction on templates. The case of functions is implied by lemma 4.9.

Definition 4.16. A type implementation is a weak type whose members are abstract values.

Definition 4.17. An implementation value is either an abstract value, a type implementation, or a structure \( G[x \leftarrow v] \) where \( G \) and \( v \) are implementation values.

Definition 4.18. An implementation template is a template generated by \( I \) in the following grammar.

\[
I ::= A | \text{TypeOf}(A) | W
\]

\[
W ::= \varsigma | W[x \leftarrow I]
\]

\[
A ::= \text{Bool} | \text{Point} | \text{TypeOf(Point)} | \text{Point} \rightarrow A | S
\]

\[
S ::= \varsigma | S[x \leftarrow A]
\]

Lemma 4.19. The template of an implementation value is an implementation template.
A permutation group has a domain type which is a weak subtype of a function type. A permutation group is an implementation value but is not an abstract value.

**Lemma 4.20.** Implementation values are closed under inverse and composition

**Proof.** The proof is by induction on template using lemma 4.15 as a base case.

### 4.3 Some Template Notation

The following notation will be generally useful. For an implementation value \( v \) we define \( I(v) \) to be the template of \( v \). For a type template \( \text{OfType}(A) \) we define \( \text{Mem}(\text{OfType}(A)) \) to be the template \( A \). For a function template \( \text{Point} \rightarrow A \) we define \( \text{Range}(\text{Point} \rightarrow A) \) to be the template \( A \). For a structure template \( I \) assigning a template to the variable \( x \) we define \( I.x \) to be the template that \( I \) assigns to \( x \). This notation is particularly useful in defining the template value function in section 5.2.

### 4.4 Pointification

**Definition 4.21.** For an implementation value \( x \) the following rules define \( x@\text{Point} \) and \( \text{SubPoint}(x) \).

- If \( x \) is a point then \( x@\text{Point} = x \) and otherwise
  \[
  x@\text{Point} = (\text{Point}, \text{Left}(\text{SubPoint}(x)), \text{Right}(\text{SubPoint}(x))).
  \]

- If \( x \) is a Boolean value, the empty structure, or a point then \( \text{SubPoint}(x) = x \).

- If \( \sigma \) is a type then \( \text{SubPoint}(\sigma) \) is the type whose members are the set of points of the form \( x@\text{Point} \) for \( x \in \sigma \).

- If \( f \) is a function then \( \text{SubPoint}(f) \) is the function \( g \) such that \( \text{Dom}(g) = \text{Dom}(f) \) and for \( x \in \text{Dom}(f) \) we have that \( g[x] = f[x]@\text{Point} \).

- \( \text{SubPoint}(G[x \leftarrow v]) = G@\text{Point}[x \leftarrow v@\text{Point}] \).

**Theorem 4.22** (Pointification-Composition Commutation). For any two abstract values \( x \) and \( y \) with \( I(x) = I(y) \) we have the following.

1. \( x \circ y \) is defined if and only if \( x@\text{Point} \circ y@\text{Point} \) is defined.
2. If \( x \circ y \) and \( x@\text{Point} \circ y@\text{Point} \) are defined then
   \[
   (x \circ y)@\text{Point} = x@\text{Point} \circ y@\text{Point}.
   \]
**Proof.** The proof is by induction on the shared template of \( x \) and \( y \). We note that the grammar for templates of abstract values does not allow abstract values to be either function types or structure types.

**Property 1.** We first note that for values \( x \) and \( y \) we have that \( x@\text{Point} \circ y@\text{Point} \) is defined if and only if \( \text{SubPoint}(x) \circ \text{SubPoint}(y) \) is defined. So it suffices to show that \( x \circ y \) is defined if and only if \( \text{SubPoint}(x) \circ \text{SubPoint}(y) \) is defined.

**Property 1 for Booleans, points, the empty structure, and point types.** If \( x \) is a point, Boolean value, the empty structure, or a point type we have \( \text{SubPoint}(x) = x \). In this case it is immediate that \( x \circ y \) is defined if and only if \( \text{SubPoint}(x) \circ \text{SubPoint}(y) \) is defined.

**Property 1 for structures.** Consider structures \( x = F[z ← u] \) and \( y = H[z ← w] \). We must show that \( x \circ y \) is defined if and only if \( \text{SubPoint}(x) \circ \text{SubPoint}(y) \) is defined. In this case we must show that \( F[z ← u] \circ G[z ← w] \) is defined if and only if \( f@\text{Point}[z ← u@\text{Point}] \circ H@\text{Point}[z ← w@\text{Point}] \) is defined. But this following directly from the induction hypothesis.

**Property 1 for functions.** Now consider two functions \( x = f \) and \( y = g \). We have \( \text{Dom}(\text{SubPoint}(f)) = \text{Dom}(f) \) and \( \text{Dom}(\text{SubPoint}(g)) = \text{Dom}(g) \). One can verify that \( f \circ g \) is defined if and only if \( \text{Dom}(f) \circ \text{Dom}(g) \) is defined and for all \( u \in \text{Dom}(f) \) and \( w \in \text{Dom}(g) \) with \( u \circ w \) defined we have that \( f[u] \circ g[w] \) is defined. Similarly, \( \text{SubPoint}(f) \circ \text{SubPoint}(g) \) is defined if and only if \( \text{Dom}(f) \circ \text{Dom}(g) \) is defined and for all \( u \in \text{Dom}(f) \) and \( w \in \text{Dom}(g) \) with \( u \circ w \) defined we have \( f[u]@\text{Point} \circ g[w]@\text{Point} \) is defined. The equivalence of these conditions follows from the induction hypothesis.

**Property 2.** We will first show that it suffices to prove

\[
\text{SubPoint}(x \circ y) = \text{SubPoint}(x) \circ \text{SubPoint}(y).
\]  

(4.1)

To show that (4.1) suffices we note that property 2 is immediate if \( x \) and \( y \) are points and if \( x \) and \( y \) are not points but (4.1) holds then we have

\[
(x \circ y)@\text{Point} = (\text{Point}, \ (\text{Left}(\text{SubPoint}(x \circ y)), \text{Right}(\text{SubPoint}(x \circ y))))
\]

\[
= (\text{Point}, \ (\text{Left}(\text{SubPoint}(x) \circ \text{SubPoint}(y)), \text{Right}(\text{SubPoint}(x) \circ \text{SubPoint}(y))))
\]

\[
= (\text{Point}, \ (\text{Left}(\text{SubPoint}(x)), \text{Right}(\text{SubPoint}(y))))
\]

\[
= x@\text{Point} \circ y@\text{Point}
\]

We now prove (4.1) by a case analysis on the structure of \( x \).

**Property 2 for Booleans, points, the empty structure, and point types.** If \( x \) is a point, Boolean value, the empty structure, or a point type we have \( \text{SubPoint}(x) = x \) and (4.1) is immediate.
4.4. POINTIFICATION

Property 2 for structures. For structures $x = F[z \leftarrow u]$ and $y = H[z \leftarrow w]$ the result is implied by the induction hypothesis as follows.

\[
\text{SubPoint}(x \circ y) = (F \circ H)@\text{Point}[z \leftarrow (u \circ u)@\text{Point}]
\]
\[
= (F@\text{Point} \circ H@\text{Point})[z \leftarrow (u@\text{Point} \circ w@\text{Point})]
\]
\[
= F@\text{Point}[z \leftarrow u@\text{Point}] \circ H@\text{Point}[z \leftarrow w@\text{Point}]
\]
\[
= \text{SubPoint}(x) \circ \text{SubPoint}(y)
\]

Property 2 for functions. Now consider two functions $x = f$ and $y = g$. We have $\text{Dom}(\text{SubPoint}(f)) = \text{Dom}(f)$ and $\text{Dom}(\text{SubPoint}(g)) = \text{Dom}(g)$. Let $u$ range over members of $\text{Dom}(f)$ and $w$ range over members of $\text{Dom}(g)$. The pairs in $\text{SubPoint}(f \circ g)$ are the pairs of the form

\[
\text{Pair}(u \circ w, (f[u] \circ f[w])@\text{Point}) = \text{Pair}(u \circ w, (f[u]@\text{Point} \circ g[w]@\text{Point})].
\]

But these are also exactly the pairs of $\text{SubPoint}(f) \circ \text{SubPoint}(g)$.

\[\square\]

Definition 4.23. For a function $f$ and for $x@\text{Point} \in \text{Dom}(f)$ we write $f(x)$ for $f[x@\text{Point}]$.

Lemma 4.24. For a function $f$ and for $x@\text{Point} \in \text{Dom}(f)$ we have $f(x)^{-1} = f^{-1}(x^{-1})$

Lemma 4.25. For functions $f$ and $g$ with $f \circ g$ defined, and for $x@\text{Point} \in \text{Dom}(f)$ and $y@\text{Point} \in \text{Dom}(g)$ where $x \circ y$ is defined, we have $(f \circ g)(x \circ y) = f(x) \circ f(y)$.

Proof. Lemma 4.22 yields

\[
(f \circ g)(x \circ y) = (f \circ g)[(x \circ y)@\text{Point}]
\]
\[
= (f \circ g)[x@\text{Point} \circ y@\text{Point}]
\]
\[
= f[x@\text{Point}] \circ g[y@\text{Point}]
\]
\[
= f(x) \circ f(y)
\]

\[\square\]

Lemma 4.26. For any type implementation $\sigma$ we have $\text{SubPoint}(\sigma)$ is a point type (is back-and-forth closed).

Proof. We immediately have that all members of $\text{SubPoint}(\sigma)$ are points and hence are abstract values. To show that $\text{SubPoint}(\sigma)$ is back-and-forth closed consider $x, y, z \in \text{SubPoint}(\sigma)$ with $x \circ y^{-1} \circ z$ defined. We have that there exists $x', y', z' \in \sigma$ with $x = x'@\text{Point}$, $y = y'@\text{Point}$ and $z = z'@\text{Point}$. Note that $\mathcal{I}(x') = \mathcal{I}(y') = \mathcal{I}(z')$. Lemma 4.22 then yields the following.

\[
x \circ y^{-1} \circ z = x'@\text{Point} \circ y'@\text{Point}^{-1} \circ z'@\text{Point}
\]
\[
= x'@\text{Point} \circ y'^{-1}@\text{Point} \circ z'@\text{Point}
\]
\[
= (x' \circ y'^{-1} \circ z')@\text{Point} \in \text{SubPoint}(\sigma)
\]

\[\square\]
Definition 4.27. We define \( \text{type}^*_i \) to be the type whose members are all point types in the universe \( U_i \).

Lemma 4.28. We have \( \alpha : \text{type}^*_i \) if only if \( \alpha \) is a type implementation in the universe \( U_i \).

4.5 Abstraction

Definition 4.29. For an implementation value \( x \) and abstract template \( A \) the projection \( x @ A \) is defined if one of the following rules applies and the recursive abstractions are all defined. If \( x @ A \) is defined the applicable rule specifies its value.

- \( x @ \text{Point} \) was defined in the previous section.
- For a Boolean value \( x \) we have \( x @ \text{Bool} = x \) and for the empty structure \( \zeta^* \) we have \( \zeta^* @ \zeta = \zeta^* \).
- For a function \( f \), if for all \( x \in \text{Dom}(f) \) we have that \( f[x] @ A \) is defined, we define \( f @ (\text{Point} \to A) \) to be the function \( g \) with template \( \text{Point} \to A \) such that \( \text{Dom}(g) = \text{Dom}(f) \) and such that for \( x \in \text{Dom}(f) \) we have \( g[x] = f[x] @ A \).
- For a structure \( G[x ← v] \) with \( G @ S \) and \( v @ A \) defined we have \( G[x ← v] @ S[x ← A] = G @ S[x ← v @ A] \).
- For an abstract type \( \sigma \) we have \( \sigma @ \text{TypeOf}(\text{Point}) = \text{SubPoint}(\sigma) \).

We now define the relation \( \leq \) on templates such that \( A \leq I \) if the template \( A \) can be derived from \( I \) by replacing subexpressions of \( I \) with \( \text{Point} \).

Definition 4.30. We define \( \leq \) to be the least relation on templates satisfying the following rules.

\[
\begin{align*}
\text{Bool} & \leq \text{Bool} \\
\text{Point} & \leq I \text{ for any } I \\
\text{TypeOf}(A) & \leq \text{TypeOf}(I) \text{ if } A \leq I \\
\text{Point} \to A & \leq \text{Point} \to I \text{ if } A \leq I \\
\zeta & \leq \zeta \\
S[x ← A] & \leq W[x ← I] \text{ if } S \leq W \text{ and } A \leq I
\end{align*}
\]

Lemma 4.31. For an implementation value \( x \) and abstract value template \( A \) we have that \( x @ A \) is defined if and only if \( A \leq \mathcal{I}(x) \).

Proof. The proof is straightforward induction on the template \( \mathcal{I}(x) \). \( \square \)
4.5. ABSTRACTION

Each of the following four lemmas can be proved either by induction on the template \( A \) or by induction on the template \( I(x) \).

**Lemma 4.32.** If \( x@A \) is defined then \( I(x@A) = A \).

**Lemma 4.33.** \( x@I(x) = x \).

**Lemma 4.34.** If \( x@A \) is defined then \( x@A@Point = x@Point \)

**Lemma 4.35** (Idempotence Property). For \((x@I)@A \) defined we have that \((x@I)@A = x@A \).

**Definition 4.36.** For an abstract type \( \sigma \) we define \( x@\sigma \) to be \( x@\text{Mem}(I(\sigma)) \) (see section 4.3 for notational definitions). We write \( x:\sigma \) to indicate that \( x@\sigma \) defined and \( (x@\sigma) \in \sigma \).

**Lemma 4.37.** For \( x:\sigma \) we have \( x@\sigma = x \) and hence \( x:\sigma \).

**Proof.** This is implied by lemma 4.33.

**Theorem 4.39** (Abstraction-Composition Commutation). For abstract values \( x \) and \( y \) with \( I(x) = I(y) \), and for an abstract template \( A \) with \( x@A \) and \( y@A \) defined, we have

1. \( x \circ y \) is defined if and only if \((x@A) \circ (y@A)\) is defined,
2. and if \( x \circ y \) and \((x@A) \circ (y@A)\) are defined then \((x \circ y)@A = (x@A) \circ (y@A)\).

**Proof.** The proof is by induction on \( A \) and involves the following cases.

**Points.** If \( A = \text{Point} \) then the result follows from lemma 4.22.

**Booleans.** If \( A = \text{Bool} \) then \( x \) and \( y \) must be Boolean values in which case \( x@A = x \) and \( y@A = y \) and the result is immediate.

**Point types.** If \( A \) is \( \text{TypeOf(Point)} \) then \( x \) and \( y \) must both be point types. In this case we again have \( x@A = x \) and \( y@A = y \) and the result is immediate.

**Structures.** If \( A \) is \( \varsigma \) then \( x \) and \( y \) must both be \( \varsigma^* \) and we have \( x@A = x \) and \( y@A = y \). Now suppose \( A = S[x \leftarrow A'] \) and consider \( x = F[z \leftarrow u] \) and \( y = H[z \leftarrow w] \). By definition we have

\[
(F[z \leftarrow u])@S[x \leftarrow A'] = (F@S)[x \leftarrow u@A'] \\
(G[z \leftarrow w])@S[x \leftarrow A'] = (G@S)[x \leftarrow w@A']
\]

The result now follows from the induction hypothesis.
Functions. Finally suppose that $\mathcal{A} = \text{Point} \rightarrow \mathcal{A}'$ and consider two functions $x = f$ and $y = g$. We have $\text{Dom}(f@\mathcal{A}) = \text{Dom}(f)$ and $\text{Dom}(g@\mathcal{A}) = \text{Dom}(g)$. We first verify property 1. By lemma 4.7 we have that $f \circ g$ is defined if and only if $\text{Dom}(f) \circ \text{Dom}(g)$ is defined and for all $u \in \text{Dom}(f)$ and $w \in \text{Dom}(g)$ we have $f[u] \circ g[w]$ is defined. Similarly, $f@\mathcal{A} \circ g@\mathcal{A}$ is defined if and only if $\text{Dom}(f) \circ \text{Dom}(g)$ is defined and for all $u \in \text{Dom}(f)$ and $w \in \text{Dom}(g)$ with $u \circ w$ defined we have $f[u@\mathcal{A}'] \circ g[w@\mathcal{A}']$ is defined. The equivalence of these conditions follows from the induction hypothesis. So we get that $f \circ g$ is defined if and only if $(f@\mathcal{A}) \circ (g@\mathcal{A})$ is defined.

To verify property 2 let $u$ range over members of $\text{Dom}(f)$ and $w$ range over members of $\text{Dom}(g)$. The pairs in $(f \circ g)@\mathcal{A}$ are the pairs of the form $\text{Pair}(u \circ w, (f[u] \circ f[w])@\mathcal{A}') = \text{Pair}(u \circ w, (f[u@\mathcal{A}'] \circ g[w@\mathcal{A}'])).$

But these are also exactly the pairs of $f@\mathcal{A} \circ g@\mathcal{A}$.

Lemma 4.40. For any type implementation $\sigma$ and $x, x' : \sigma$ we have $x =_\sigma x'$ if and only if $x =_{\text{SubPoint}(\sigma)} x'$.

Proof. First suppose that $x =_\sigma x'$. In that case there exists $z \in \sigma$ with $(x@\sigma) \circ z^{-1} \circ (x'@\sigma)$ defined. But then have that $(x@\sigma)@\text{Point} \circ (z@\text{Point})^{-1} \circ (x'@\sigma)@\text{Point}$ is defined. But we have that $(x@\sigma)@\text{Point} = x@\text{Point}$ and $(x'@\sigma)@\text{Point} = x'@\text{Point}$. This implies that $x@\text{Point} \in \text{SubPoint}(\sigma)$ and $x'@\text{Point} \in \text{SubPoint}(\sigma)$ and $(x@\text{Point}) \circ (z@\text{Point})^{-1} \circ (x'@\text{Point})$ is defined which implies $x =_{\text{SubPoint}(\sigma)} x'$. Conversely, suppose that $x =_{\text{SubPoint}(\sigma)} x'$ with $x, x' : \sigma$. We have that there exists $z@\text{Point} \in \text{SubPoint}(\sigma)$ with $z \in \sigma$ such that $x@\text{Point} \circ (z@\text{Point})^{-1} \circ (y@\text{Point})$ is defined. But since $\mathcal{I}(x@\sigma) = \mathcal{I}(z) = \mathcal{I}(x'@\sigma)$ we have $(x@\sigma) \circ z^{-1} \circ (x'@\sigma)$ is defined and hence $x =_\sigma x'$.

\qed
Chapter 5

The Semantic Value Function

The value function must be specified for each kind of expression. For example, for disjunctions we define $\mathcal{V}_\Sigma[[\Phi \lor \Psi]]S$ to be $\mathcal{V}_\Sigma[[\Phi]]S \lor \mathcal{V}_\Sigma[[\Psi]]S$. As in the case of disjunction, most semantic value specifications simply replace syntactic notation by semantic notation. Before giving all the cases of the value function we first define the semantic notation $\sigma \rightarrow \tau$ for semantic types $\sigma$ and $\tau$.

5.1 Function Types

Definition 5.1. For type implementations $\sigma$ and $\tau$ we define $\sigma \rightarrow \tau$ to be the type whose template is $\text{TypeOf}(\text{Point} \rightarrow \text{Mem}(I(\tau)))$ and where the members of $\sigma \rightarrow \tau$ are the functions $f$ with $\text{Dom}(f) = \text{SubPoint}(\sigma)$ and such that for $x \in \text{Dom}(f)$ we have $f[x] \in \tau$.

Lemma 5.2. For any type implementations $\sigma$ and $\tau$ we have

$$(\sigma \rightarrow \tau) = (\text{SubPoint}(\sigma) \rightarrow \tau)$$

Lemma 5.3. For any two type implementations $\sigma$ and $\tau$ and template $A$ such that $\tau@\text{TypeOf}(A)$ is defined we have

$$(\sigma \rightarrow \tau)@\text{TypeOf}(\text{Point} \rightarrow A) = \sigma \rightarrow (\tau@\text{TypeOf}(A))$$

Proof. Both type implementations have template $\text{TypeOf}(\text{Point} \rightarrow A)$. We must show that they have the same members. Consider

$$f \in (\sigma \rightarrow \tau)@\text{TypeOf}(\text{Point} \rightarrow A).$$

We have that there exists $f' \in (\sigma \rightarrow \tau)$ with $f = f'@\text{Point} \rightarrow \tau$. We have that $f$ and $f'$ have the same domain $\text{SubPoint}(\sigma)$ and for $x \in \text{SubPoint}(\sigma)$
we have \( f[x] = f'[x] \circ A \). This implies \( f[x] \in (\tau \circ \text{TypeOf}(A)) \) which implies \( f \in \sigma \rightarrow (\tau \circ \text{TypeOf}(A)) \).

Conversely consider \( f \in (\sigma \rightarrow (\tau \circ \text{TypeOf}(A))) \). Now we need to show
\[
\text{Lemma 5.4. For type implementations } \sigma \text{ and } \tau \text{ we have that } \sigma \rightarrow \tau \text{ is a type implementation.}
\]

\[\text{Proof.}\]Since every member of a type implementation must be an abstract value, we have that for \( f \in \sigma \rightarrow \tau \) and for \( x \in \text{Dom}(f) \) we have that \( f[x] \in \tau \) and hence \( f[x] \) is an abstract value. This implies that \( f \) is an abstract value. We must also show that the type \( \sigma \rightarrow \tau \) is back and forth closed. We have already shown that functions are weak-values. We must now show that for \( f, g, h \in \sigma \rightarrow \tau \) with \( f \circ g^{-1} \circ h \) defined we have \( (f \circ g^{-1} \circ h) \in (\sigma \rightarrow \tau) \). By lemma 4.33 we have that \( f \circ g^{-1} \circ h \) is a function, that \( \text{Dom}(f \circ g^{-1} \circ h) = \text{SubPoint}(\sigma) \) and that for \( x \in \text{SubPoint}(\sigma) \) we have
\[
(f \circ g^{-1} \circ h)[x] = (f \circ g^{-1} \circ h)[x \circ x^{-1} \circ x] = f[x] \circ g[x]^{-1} \circ h[x] \in \tau
\]

\[\text{Lemma 5.5. For } f : \sigma \rightarrow \tau \text{ and for } x : \sigma \text{ we have } f(x) : \tau.\]

\[\text{Proof.}\]For \( x : \sigma \) we have \((x \circ \sigma) \circ \text{Point} = x \circ \text{Point} \in \text{SubPoint}(\sigma)\) which implies that \( f(x) = f[x] \circ \text{Point} \) is defined. Furthermore, we have \( f : \sigma \rightarrow \tau \) which implies that \( f \circ \text{Point} \in (\sigma \rightarrow \tau) \). This implies that \( f[x] \circ \text{Point} \in \text{Mem}(\mathcal{I}(\tau)) \). By \text{Lemma 4.34} we have \[\text{Lemma 5.6. We say that } \Phi[\cdot, \cdot] \text{ is an equivalence-respecting relation on } \sigma \times \tau \text{ if for } x : \sigma \text{ and } y : \tau \text{ we have that } \Phi[x, y] \text{ is a truth value and for } x = _\sigma x' \text{ and } y = _\tau y' \text{ we have } \Phi[x, y] = \Phi[x', y'].\]

\[\text{Definition 5.6. For any implementation types } \sigma \text{ and } \tau \text{ and any equivalence-respecting relation } \Phi[\cdot, \cdot] \text{ on } \sigma \times \tau, \text{ if for all } x : \sigma \text{ there exists } y : \tau \text{ such that } \Phi[x, y] \text{ then there exists } f : \sigma \rightarrow \tau \text{ such that for all } x : \sigma \text{ we have } \Phi[x, f(x)].\]

\[\text{Lemma 5.7. For any implementation types } \sigma \text{ and } \tau \text{ and any equivalence-respecting relation } \Phi[\cdot, \cdot] \text{ on } \sigma \times \tau, \text{ if for all } x : \sigma \text{ there exists } y : \tau \text{ such that } \Phi[x, y] \text{ then there exists } f : \sigma \rightarrow \tau \text{ such that for all } x : \sigma \text{ we have } \Phi[x, f(x)].\]

\[\text{Proof.}\]For \( x : \sigma \) define \( |x|_\sigma \) to be the set of \( x' : \sigma \) with \( x = _\sigma x' \). For each such equivalence class \( |x|_\sigma \) arbitrarily select a value \( y(|x|_\sigma) \in \tau \) such that for all \( x : \sigma \) we have \( \Phi[x, y(|x|_\sigma)] \). We now take \( f \) to be the function with \( \text{Dom}(f) = \text{SubPoint}(\sigma) \) and with \( f(x) = y(|x|_\sigma) \). Lemma 4.34 ensures that this definition of \( f \) is well formed and that \( f \) is a function.
5.1. FUNCTION TYPES

Lemma 5.8. For \( f, g : \sigma \to \tau \) we have \( f =_{(\sigma \to \tau)} g \) if and only if for all \( x : \sigma \) we have \( f(x) =_\tau g(x) \).

Proof. First consider \( f, g : \sigma \to \tau \) with \( f =_{(\sigma \to \tau)} g \). In that case there exists \( h \in \sigma \to \tau \) such that \( f \circ (\sigma \to \tau) \circ h^{-1} \circ g \circ (\sigma \to \tau) \) is defined. One can show that lemma 4.3 now implies that for \( x : \sigma \) we have that \( f(x) =_\tau h(x) =_\tau g(x) \) is defined which implies \( f(x) =_\tau g(x) \). Conversely, suppose that \( f(x) =_\tau g(x) \) for all \( x : \sigma \). We now define a function \( h \in \sigma \to \tau \) by selecting, for each equivalence class \( [x]_\sigma \) a value \( h([x]_\sigma) \) such that \( f([x]_\sigma) \circ h([x]_\sigma) \circ g([x]_\sigma) \) is defined. We then get \( h \in \sigma \to \tau \) with \( f =_\tau h \) defined and hence \( f =_{(\sigma \to \tau)} g \). \( \square \)

Lemma 5.9. For type implementations \( \sigma_1, \sigma_2, \tau_1 \) and \( \tau_2 \) with \( \sigma_1 \circ \sigma_2 \) defined and \( \tau_1 \circ \tau_2 \) defined we have \( (\sigma_1 \circ \sigma_2) \to (\tau_1 \circ \tau_2) = ((\sigma_1 \to \tau_1) \circ (\sigma_2 \to \tau_2)) \).

Proof. We must first show that \( (\sigma_1 \to \tau_1) \circ (\sigma_2 \to \tau_2) \) is defined. We will show that every instance of \( \text{Right}(\sigma_1 \to \tau_1) \) is an instance of \( \text{Left}(\sigma_2 \to \tau_2) \). The converse is similar. We will let \( f \) range over members of \( \sigma_1 \to \tau_1 \) and let \( g \) range over members of \( \sigma_2 \to \tau_2 \). We must show that every function of the form \( f_1^{-1} \circ f_2 \) can be written as \( g_1 \circ g_2^{-1} \). We will let \( x \) range over members of \( \text{SubPoint}(\sigma_1) \); let \( y \) range over members of \( \text{SubPoint}(\sigma_2) \). We have that \( f_1^{-1} \circ f_2 \) is the function whose pairs have the form \( (x_1^{-1} \circ x_2, f_1[x_1^{-1}] \circ f_2[x_1]) \). Since \( \text{Right}(\tau_1) = \text{Left}(\tau_2) \) the value \( f_1[x_1^{-1}] \circ f_2[x_1] \) can be written as \( w_1 \circ w_2^{-1} \) for some \( w_1, w_2 \in \tau_2 \). The values \( w_1 \) and \( w_2 \) can be selected as a pair of the equivalence class \( [x_1] \). Note that by lemma 4.39 we have that \( \text{Right}(\text{SubPoint}(\sigma_1)) = \text{SubPoint}(\text{Right}(\sigma_1)) = \text{SubPoint}(\text{Left}(\sigma_2)) = \text{Left}(\text{SubPoint}(\sigma_2)) \). Hence we have that \( \text{SubPoint}(\sigma_1) \circ \text{SubPoint}(\sigma_2) \) is defined. We will write \( x \leftrightarrow y \) if \( \text{Right}(x) =_{\text{Right}(\text{SubPoint}(\sigma_1))} \text{Left}(y) \). By lemmas 5.11 and 5.12 the relation \( \leftrightarrow \) defines a bijection between the equivalence classes of \( \text{SubPoint}(\sigma_1) \) and \( \text{SubPoint}(\sigma_2) \). Using this bijection, we can take the values \( w_1 \) and \( w_2 \) to be function of the equivalence classes of \( \text{SubPoint}(\sigma_2) \). This defines two functions \( g_1, g_2 \in \sigma_2 \to \tau_2 \). We can rewrite \( x_1^{-1} \circ x_2 \) as \( y_1 \circ y_2^{-1} \). When this is done we have that \( [y_1] \) is the class in correspondence with \( [x_1] \). This implies that \( f_1[x_1^{-1}] \circ f_2[x_1] = g_1[y_1] \circ g_2[y_2]^{-1} \). So the pair \( (x_1^{-1} \circ x_2, f_1[x_1^{-1}] \circ f_2[x_1]) \) is equal to the pair \( (y_1 \circ y_2^{-1}, g_1[y_1] \circ g_2[y_2]^{-1}) \). Hence every pair in \( f_1^{-1} \circ f_2 \) is also a pair in \( g_1 \circ g_2^{-1} \). The mapping from pairs in \( f_1^{-1} \circ f_2 \) to pairs in \( g_1 \circ g_2^{-1} \) is onto so that every pair in \( g_1 \circ g_2^{-1} \) is also a pair in \( f_1^{-1} \circ f_2 \). This gives \( f_1^{-1} \circ f_2 = g_1 \circ g_2^{-1} \).

Given that \( (\sigma_1 \to \tau_1) \circ (\sigma_2 \to \tau_2) \) is defined we must now prove the equation.

We will show containment of instances in each direction. For \( h \in (\sigma_1 \to \tau_1) \circ (\sigma_2 \to \tau_2) \) we must show \( h \in (\sigma_1 \circ \sigma_2) \to (\tau_1 \circ \tau_2) \). By definition \( h \) has the form \( f \circ g \) with \( f \in (\sigma_1 \to \tau_1) \) and \( g \in (\sigma_1 \circ \sigma_2) \). The pairs of \( h \) then have the form \( (x \circ y, f[x] \circ g[y]) \). It remains only to show that if

\[
x_1 \circ y_1 \simeq_{\text{SubPoint}(\sigma_1) \circ \text{SubPoint}(\sigma_2)} x_2 \circ y_2
\]

then \( f[x_1] \circ g[y_1] = f[x_2] \circ g[y_2] \). But this follows directly from lemma 5.13.

Finally consider \( h \in (\sigma_1 \circ \sigma_2) \to (\tau_1 \circ \tau_2) \). We must show \( h \in (\sigma_1 \to \tau_1) \circ (\sigma_2 \to \tau_2) \). More specifically we must show that \( h \) can be written as \( f \circ g \) for \( f \in (\sigma_1 \to \tau_1) \) and \( g \in (\sigma_1 \circ \sigma_2) \).
τ₁ and g ∈ (σ₂ → τ₂). By lemma 4.39 we have that SubPoint(σ₁ ◦ σ₂) = SubPoint(σ₁) ◦ SubPoint(σ₂). For each equivalence class C of SubPoint(σ₁) ◦ SubPoint(σ₂) we have a well-defined value of h which we can write as h[C]. We have h[C] ∈ (τ₁ ◦ τ₂). So for each class C we can select f[C] and g[C] such that h[C] = f[C] ◦ g[C] and with f[C] ∈ τ₁ and g[C] ∈ τ₂. Lemma 3.13 now implies that we can take f(C) to define the function f ∈ σ₁ → τ₁ and g(C) to define g ∈ σ₂ → τ₂.

5.2 The Template Value Function

Figure 5.1 lists the constructions used in forming expressions. The semantic value function and the template value function are both defined on the expressions that can be built from these constructions.

There is a technical difficulty in defining the value function arising from the fact that we require empty types to be assigned templates. To compute templates for empty types we introduce an auxiliary value function $\bar{V}[e]I$ having the property that when $V_{\Sigma}[e]S$ is defined we will have that $I(V_{\Sigma}[e]S) = \bar{V}[e]I(S)$. The template value function is simpler than the value function and we define it first. The template value function is defined in figure 5.2.

5.3 The Value Function

The recursive definition of the value function is well founded — each recursive call reduces a well-founded order. Figure 5.3 defines a well founded order $\prec$ such that in the definition of $V_{\Sigma}[e]S$ all recursive references are of the form $V_{\Gamma}[s]G$ with $(\Gamma, s) \prec (\Sigma, e)$. The well-foundedness of the rules defining $V_{\Sigma}[e]S$ implies that the rules uniquely specify whether or not $V_{\Sigma}[e]S$ is defined, and when it is defined, what the value is. The pair $(\epsilon, \epsilon)$ is the minimum pair under the order $\prec$ we have that $V_{\Sigma}[e]S$ is defined provided that $S$ is the empty structure $\epsilon^*$ and $V_{\epsilon}[\epsilon]\epsilon^*$ is defined to be the type whose only member is $\epsilon^*$. For a closed expression $e$ (one with no free variables) we will write $V[e]$ for $V_{\epsilon}[e]\epsilon^*$. The base case defines $V[\epsilon]$. We will say that $V_{\Gamma}[s]G$ precedes $V_{\Sigma}[e]$ when $(\Gamma, s) \prec (\Sigma, e)$. The order $\prec$ has the property that for any nonempty context $\Sigma$ we have that $V[\Sigma]$ precedes $V_{\Sigma}[e]$ for any $e$. So when defining $V_{\Sigma}[e]$ we can assume that the set of general instances $S : V[\Sigma]$ is previously defined (even when $\Sigma$ is empty).

Each case in the definition of the value function corresponds to one of the expression types listed figure 5.1. For each kind of expression $e$ the definition specifies when $V_{\Sigma}[e]$ is defined (when $e$ is well formed in the context $\Sigma$) and, in the case where it is defined, specifies the value denoted by $V_{\Sigma}[e]S$ for any $S : V[\Sigma]$. All values returned by the value function — all values of the form $V_{\Sigma}[e]S$ — will be implementation values as defined in chapter 4. The claim that all defined values are implementation values is nontrivial — this claim requires simultaneous proof with the proof of commutation properties of the value
5.3. THE VALUE FUNCTION

|                | typeᵢ   | x         |
|----------------|---------|-----------|
| Bool           | f(e)    | s =ᵣ w   |
| σ → τ         |         |           |
| Φ ∨ Ψ          | ¬Φ      | ∀ x:σ Φ[x]|
| τΔ(D)         | τ₁ : x: τΔ(D) | τ₂Δ(D)   |
| σ             | G[x ← e] | G.x      |

Figure 5.1: Kinds of expressions. The expression Σ is treated as an abbreviation for Σₑ(σ).

|                      |                      |
|----------------------|----------------------|
| \( \mathcal{V}[σΔ(D)]\ I \) | TypeOf(\( \mathcal{V}[Γ] (\mathcal{V}[D] I) \)) |
| \( \mathcal{V} [e] I \) | ζ             |
| \( \mathcal{V}[Γ: x: τ] I \) | (\( \mathcal{V}[Γ] I \) [x ← Mem(\( \mathcal{V}[τ] I) \)]) |
| \( \mathcal{V}[Γ: Φ] I \) | \( \mathcal{V}[Γ] I \) |
| \( \mathcal{V}[Bool] I \) | TypeOf(Bool)     |
| \( \mathcal{V}[ζ] I \) | ζ             |
| \( \mathcal{V} [typeᵢ] I \) | TypeOf(Point)   |
| \( \mathcal{V}[x] I \) | I.x            |
| \( \mathcal{V} [σ → τ] I \) | TypeOf(Point → Mem(\( \mathcal{V}[τ] I) \)) |
| \( \mathcal{V}[f(e)] I \) | Range(\( \mathcal{V}[f] I \)) |
| \( \mathcal{V}[G[x ← e]] I \) | (\( \mathcal{V}[G] I \) [x ← \( \mathcal{V}[e] I \)]) |
| \( \mathcal{V}[G.x] I \) | (\( \mathcal{V}[G] I \).x |
| \( \mathcal{V}[Φ] I \) | bool for Φ Boolean |

Figure 5.2: The Template Value Function. For an expression \( e \) built from the constructs in figure 5.1 and an implementation value template \( I \), the template \( \mathcal{V}[e] I \) is defined by the above rules where the template is undefined if no rule applies or if some expression on the right hand side of the rule is undefined. Notations are defined as follows. The rules also define \( \mathcal{V}[Σ] I \) for a context Σ. The template notations used in this definition are defined in section 4.3.
For a context $\mathcal{S}$ type containing Definition 5.11. of values values $G$ that implementation values. However, in order for function. Our initial definitions do not assume that all defined values are implementation values.

The following two mutually recursive definitions are well-founded by reduction of the ordering $\prec$. In these definitions it is important to recall that that the notation $x : \tau$ means that $x$ is an implementation value such that $x @ \tau \in \tau$. This is especially important when considering $S : V[\Sigma]$.

**Definition 5.10.** For a context $\Sigma$ and syntactic expressions $e$ and $\tau$ we write $\Sigma \models e : \tau$ to mean that $V[\Sigma]$, $V[\Sigma][e]$ and $V[\Sigma][\tau]$ are defined, that $V[\Sigma]$ is a type implementation whose members are structures, and for $S : V[\Sigma]$ we have that $V[\Sigma][\tau]S$ is a type implementation and $(V[\Sigma][e]S) : (V[\Sigma][\tau]S)$. For $\Phi$ not of the form $e : \sigma$ we write $\Sigma \models \Phi$ to mean that $V[\Sigma]$ is defined and is a type implementation whose members are structures, that $V[\Sigma][\Phi]$ is defined, and for $S : V[\Sigma]$ we have $V[\Sigma][\Phi]S = T$.

**Definition 5.11.** We say that $V[\Sigma][e]$ is defined if either $\Sigma = \varepsilon$ and $e = \tau$, in which case we define $V[\Sigma]$ to be the type containing $\varepsilon^*$, or $V[\Sigma]$ is defined and is a type implementation and for $S : V[\Sigma]$ one of the following rules defines $V[\Sigma][e]S$.

- $V[\Sigma][\Delta(D)]$: If $\Sigma \models D : \exists$ then for $S : V[\Sigma]$ we define $V[\Sigma][\Delta(D)]S$ to be the type containing $\varepsilon^*$.

- $V[\Sigma][\Delta(D)]$: If $\Sigma \models D : \exists$, and $\Delta, \Gamma \models \tau : \text{type}$, and for $S : V[\Sigma]$ we have that $V[\Sigma][\Delta(D)]I(S)$ is defined, then for $S : V[\Sigma]$ we define $V[\Sigma][\Delta(D)]S$ to be the pair of the template $V[\Gamma : x : \Delta(D)][I(S)]$ and the set of values of the form $G[x \leftarrow v]$ for $G \in V[\Sigma][\Delta(D)]S$ and $v \in V[\Gamma : x : \Delta(D)]V[\Sigma][\Delta(D)]S; G$.

- $V[\Sigma][\Delta(D)]$: If $\Sigma \models D : \exists$ and $\Delta, \Gamma \models \Phi : \text{Bool}$ then for $S : V[\Sigma]$ we define $V[\Sigma][\Delta(D)]S$ to be the pair of the template of $V[\Sigma][\Delta(D)]S$ and the set of values values $G \in V[\Sigma][\Delta(D)]S$ such that $\Delta, \Gamma, \Phi : (V[\Sigma][\Delta(D)]S; G) = T$.

- $V[\Sigma][\text{Bool}]$: For $S : V[\Sigma]$ we define $V[\Sigma][\text{Bool}]S$ to be $\text{Bool}^*$.

- $V[\Sigma][\text{type}]$: For $S : V[\Sigma]$ we define $V[\Sigma][\text{type}]S$ to be $\text{type}^*$.
5.4. A SANITY CHECK LEMMA

- \( \mathcal{V}_{\Sigma}[s] \): For \( S : \forall [\Sigma] \) we define \( \mathcal{V}_{\Sigma}[s] S \) to be \( \varsigma^* \)
- \( \mathcal{V}_{\Sigma}[x] \): If for all \( S : \forall [\Sigma] \) we have that \( S \) is a structure assigning a value to \( x \) then we define \( \mathcal{V}_{\Sigma}[x] S \) to be \( S.x \).
- \( \mathcal{V}_{\Sigma}[\sigma \rightarrow \tau] \): If \( \Sigma \models \sigma : \text{type}_i \) and \( \Sigma \models \tau : \text{type}_i \) then for \( S : \forall [\Sigma] \) we define \( \mathcal{V}_{\Sigma}[\sigma \rightarrow \tau] S \) to be \( \mathcal{V}_{\Sigma}[\sigma] S \rightarrow \mathcal{V}_{\Sigma}[\tau] S \).
- \( \mathcal{V}_{\Sigma}[f(e)] \): If \( \mathcal{V}_{\Sigma}[f] \) and \( \mathcal{V}_{\Sigma}[e] \) are defined, and for all \( S : \forall [\Sigma] \) we have that \( \mathcal{V}_{\Sigma}[f] S \) is a function with \( \mathcal{V}_{\Sigma}[e] S : \text{Dom}(\mathcal{V}_{\Sigma}[f] S) \), then we define \( \mathcal{V}_{\Sigma}[f(e)] S \) to be \((\mathcal{V}_{\Sigma}[f] S)(\mathcal{V}_{\Sigma}[e] S)\).
- \( \mathcal{V}_{\Sigma}[G[x \leftarrow e]] \): If \( \mathcal{V}_{\Sigma}[G] \) and \( \mathcal{V}_{\Sigma}[e] \) are defined and for all \( S : \forall [\Sigma] \) we have that \( \mathcal{V}_{\Sigma}[G] S \) is a structure not assigning a value to \( x \) then we define \( \mathcal{V}_{\Sigma}[G[x \leftarrow e]] S \) to be \( (\mathcal{V}_{\Sigma}[G] S)[x \leftarrow \mathcal{V}_{\Sigma}[e] S] \).
- \( \mathcal{V}_{\Sigma}[G.x] \): If \( \mathcal{V}_{\Sigma}[G] \) is defined and for all \( S : \forall [\Sigma] \) we have that \( \mathcal{V}_{\Sigma}[G] S \) is a structure assigning a value to the variable \( x \) then for \( S : \forall [\Sigma] \) we define \( \mathcal{V}_{\Sigma}[G.x] S \) to be \( (\mathcal{V}_{\Sigma}[G] S).x \).
- \( \mathcal{V}_{\Sigma}[e =_\sigma w] \): If \( \Sigma \models e : \sigma \) and \( \Sigma \models w : \sigma \) we define \( S : \forall [\Sigma] \) to be \( T \) if \( \mathcal{V}_{\Sigma}[e] S =_{\mathcal{V}_{\Sigma}[\sigma]} S \mathcal{V}_{\Sigma}[w] S \).
- \( \mathcal{V}_{\Sigma}[(\forall x : \tau. \Phi)] \): If \( \Sigma ; x : \tau \models \Phi : \text{Bool} \) then for \( S : \forall [\Sigma] \) we define \( \mathcal{V}_{\Sigma}[(\forall x : \tau. \Phi)] S \) to be \( T \) if for all \( v : \mathcal{V}_{\Sigma}[\tau] S \) we have \( \mathcal{V}_{\Sigma; x : \tau. \Phi} S[x \leftarrow v] = T \).
- \( \mathcal{V}_{\Sigma}[(\Phi \lor \Psi)] \): If \( \Sigma \models \Phi : \text{Bool} \) and \( \Sigma \models \Psi : \text{Bool} \) then for \( S : \forall [\Sigma] \) we have \( \mathcal{V}_{\Sigma}[(\Phi \lor \Psi)] S = \mathcal{V}_{\Sigma}[(\Phi) S \lor \mathcal{V}_{\Sigma}[(\Psi)] S \).
- \( \mathcal{V}_{\Sigma}[(\neg \Phi)] \): If \( \Sigma \models \Phi : \text{Bool} \) then for \( S : \forall [\Sigma] \) we have \( \mathcal{V}_{\Sigma}[(\neg \Phi)] S = \neg \mathcal{V}_{\Sigma}[(\Phi)] S \).

5.4 A Sanity Check Lemma

We will eventually prove that if \( \mathcal{V}_{\Sigma}[e] \) is defined then for \( S : \forall [\Sigma] \) we have that \( \mathcal{V}_{\Sigma}[e] S \) is an implementation value. Until we have proved this however, we have the following much weaker lemma.

Lemma 5.12. If \( \mathcal{V}_{\Sigma}[e] \) is defined then for \( S : \forall [\Sigma] \) we have that \( \mathcal{V}_{\Sigma}[e] S \) is a pre-value with \( \mathcal{I}(\mathcal{V}_{\Sigma}[e] S) = \mathcal{V}[e] \mathcal{I}(S) \).

These two properties can be proved by simultaneous induction on the definition of \( \mathcal{V}_{\Sigma}[e] \) (by induction on the order \( \prec \) on the pair \((\Sigma, e)) \). We omit the details.

5.5 The Semantic Notation \( \tau_\Delta(D) \)

Lemma 5.13. We have that \( \mathcal{V}_{\Sigma}[\tau_\Delta(D)] \) is defined if and only if \( \Sigma \models D : \Xi \) and \( \mathcal{V}_{\Sigma, \Delta}[] \) is defined.

The proof is an induction on the length of \( \Gamma \).

The following notation will be useful.
Definition 5.14. For $D : \mathcal{V}[\mathfrak{X}]$, and for $\mathcal{V}[\mathfrak{X}; \Gamma]$ defined, we define $\mathcal{V}_\Delta(D)$ by the following rules.

- We define $\mathcal{V}_\Delta(D)$ to be the type containing the empty structure.
- We define $\mathcal{V}_\Delta(D)$ to be the pre-type whose template is $\mathcal{V}[\Gamma ; x : \tau] \mathcal{I}(D)$ and whose members are the abstract values of the form $G[x \leftarrow v]$ for $G \in \mathcal{V}_\Delta(D)$ and $v \in \mathcal{V}_\Delta, \Gamma[\tau](D; G)$.
- We define $\mathcal{V}_\Delta(D)$ to be the pre-type whose template is that of $\mathcal{V}_\Delta(D)$ and whose members are those values $G \in \mathcal{V}_\Sigma[\mathcal{V}_\Delta(D)] S$ such that $\mathcal{V}_\Delta, \Gamma[\Phi](D; G) = T$.

We now have the following lemma whose proof is by induction on the length of $\Gamma$.

Lemma 5.15. For $\Sigma \models D : \mathfrak{X}$, for $S : \mathcal{V}[\mathfrak{X}]$ and for $\mathcal{V}[\mathfrak{X}; \Gamma]$ defined we have

$$\mathcal{V}_\Sigma[\mathcal{V}_\Delta(D)] S = \mathcal{V}_\Delta(\mathcal{V}_\Sigma[\mathcal{V}_\Delta(D)] S)$$

Equation 5.1 provides a more concise and compositional specification of semantic values of the form $\mathcal{V}_\Sigma[\mathcal{V}_\Delta(D)] S$. 

We now have that $\mathcal{V}_\Delta(D)$ has both a syntactic and a semantic reading depending on whether $D$ is a syntactic expression or a semantic structure.
Chapter 6

The Main Theorems

This chapter proves two commutation theorems and the abstraction theorem.

6.1 Value-Abstraction Commutation

Theorem 6.1 (Value-Abstraction Commutation). For $\forall: [x]$ defined and for $S: \forall [x]$ and template $A$ such that $(S@A): \forall [x]$ we have

$$\forall: [x] S = (\forall: [x] S) @ (\forall: [x] A).$$

Proof. The proof is by induction on the definition of $\forall: [x]$. In the proof we will use $u^*[S]$ as an abbreviation for $\forall: [u] S$. We must show

$$e^*[S@A] = (e^*[S]) @ (\forall: [x] A).$$

- If $e^*[S]$ does not depend on $S$ (when $e$ is a constant symbol or does not contain free variables) we have

  $$e^*[S@A] = e^*[S] = e^*[S] @ (\forall: [x] \exists(S)) = e^*[S] @ (\forall: [x] A).$$

- If $e$ is a variable $x$ we have

  $$e^*[S@A] = (S@A).x = (S.x) @ (A.x) = (e^*[S]) @ (\forall: [x] A).$$

- For a Boolean expression $\Phi$ we must show $\Phi^*[S@A] = \Phi^*[S]$. There are several subcases of Boolean expressions:
• For Boolean combinations the result follows directly from the induction hypothesis. For example we have

\[(\Phi \lor \Psi)^*[S@A] = \Phi^*[S@A] \lor \Psi^*[S@A]\]

\[= \Phi^*[S] \lor \Psi^*[S]\]

\[= (\Phi \lor \Psi)^*[S].\]

• Now suppose that \(e\) is an equation \(u =_\sigma w\). Using lemmas 4.40 and 4.35 and the induction hypothesis we have

\[(u =_\sigma w)^*[S@A] = u^*[S@A] = \sigma^*[S@A] \lor w^*[S@A]\]

\[= u^*[S@A]@\text{Point} \approx \sigma^*[S@A]@\text{TypeOf(Point)} \lor w^*[S@A]@\text{Point}\]

\[= u^*[S]@\text{Point} \approx \sigma^*[S]@\text{TypeOf(Point)} \lor w^*[S]\]

\[= (u =_\sigma w)^*[S].\]

• Now suppose that \(e\) is a quantified formula \(\forall x: \tau \Phi\). In this case we have

\[e^*[S@A] = T\]

iff for all \(v: \tau^*[S@A]\) \(\Phi^*[S@A][x \leftarrow v]\)

iff for all \(v: ((\tau^*[S])@\text{Point} \approx \tau^*[S]@\text{TypeOf(Point)} [x \leftarrow v] \lor w^*[S@A][x \leftarrow v])\)

iff for all \(w: (\tau^*[S]) \Phi^*[S][x \leftarrow w]@\text{Mem(\(\tau\)[A])}\)

iff for all \(w: \tau^*[S]\) \(\Phi^*[S][x \leftarrow w]\]

iff \(e^*[S] = T\)

• If \(e\) is a type of the form \(\tau_\Delta(D)\) then \(e^*[S]\) is independent of \(S\) and the result is similar to that of constants or closed expressions.

• Now suppose \(e\) has the form \(\tau(x_\Delta(D))\). In general \(e^*[S@A]\) and \(e^*[S@\text{Mem}(\tau)[A]\) have the same template. When \(e\) is a type expression \(\sigma\) it suffices to prove that \(\sigma^*[S@A]\) and \(\sigma^*[S]@\text{Mem}(\tau)[A]\) have the same members.

  – To show \(e^*[S@A] \subseteq e^*[S@\text{Mem}(\tau)[A]\) we note that a member of \(e^*[S@A]\) has the form \(G[x \leftarrow v]\) for

\[G \in (\tau_\Delta(D))^*[S@A]\]

\[= (\tau_\Delta(D))^*[S]@\text{Mem}(\tau)[\tau_\Delta(D)][A]\]

\[G = G'@\text{Mem}(\tau)[\tau_\Delta(D)][A]\) for some \(G' \in \tau_\Delta(D^*(S))\)

and

\[v \in \mathcal{V}_{\Delta,\Gamma} [\tau'] ((D^*[S@A]); G).\]

We now have

\[(D^*[S]; G') : \mathcal{V}_{\Delta'}[\tau'].\]
6.1. VALUE-ABSTRACTION COMMUTATION

and also

\[(D^*[S]; G') @ ((\overline{\nabla} [D]; \mathcal{A}); \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A})) \] 

Hence we can apply the induction hypothesis to

\[\mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S]; G') @ (\overline{\nabla} [D]; (\mathcal{A}; \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}))) \] 

We can now perform the following calculation.

\[ v \in \mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S@\mathcal{A}]; G) \] 

\[ = \mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S]; G') @ ((\overline{\nabla} [D]; \mathcal{A}); \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A})) ) \] 

\[ = \mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S]; G') @ ((\overline{\nabla} [D]; \mathcal{A}); \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}))) \) 

\[ v = v' @ (\overline{\nabla} [\tau] ((\overline{\nabla} [D]; \mathcal{A}); \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}))) \] 

for \( v' \in \mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S]; G') \) 

This calculation yields that a value \( G[x \leftarrow v] \in e^*[S@\mathcal{A}] \) can be rewritten as

\[ G'[x \leftarrow v'] @ \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}) [x \leftarrow \text{Mem}(\overline{\nabla} [\tau] ((\overline{\nabla} [D]; \mathcal{A}); \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}))) \] 

with \( G'[x \leftarrow v'] \in e^*[S] \) which gives \( G[x \leftarrow v] \in e^*[S@\overline{\nabla} [e]; \mathcal{A}] \) and hence \( e^*[S@\mathcal{A}] \subseteq e^*[S@\overline{\nabla} [e]; \mathcal{A}] \) 

To show \( e^*[S@\overline{\nabla} [e]; \mathcal{A}] \subseteq e^*[S@\mathcal{A}] \) consider \( G[x \leftarrow v] \in e^*[S@\overline{\nabla} [e]; \mathcal{A}] \). In this case we have

\[ G = G' @ \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}) \] 

\[ G' \in (\Gamma_{\Delta}(D))^*[S] \] 

\[ v = v' @ \text{Mem}(\overline{\nabla} [\tau] ((\overline{\nabla} [D]; \mathcal{A}); \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}))) \] 

This gives

\[ G \in (\Gamma_{\Delta}(D))^*[S] @ \overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A} \] 

\[ = (\Gamma_{\Delta}(D))^*[S@\mathcal{A}] \] 

\[ v \in \mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S]; G') @ \overline{\nabla} [\tau] ((\overline{\nabla} [D]; \mathcal{A}); \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}))) \] 

\[ = \mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S@\mathcal{A}]; (G' @ \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}))) \] 

\[ = \mathcal{V}_{\Delta, \Gamma}[\tau] ( (D^*[S@\mathcal{A}]; G) ) \] 

We then have \( G[x \leftarrow v] \in e^*[S@\mathcal{A}] \).

• Now suppose that \( e \) has the form \( \overline{\nabla} [\Gamma_{\Delta}(D)] \). In this case the members of \( e^*[S@\mathcal{A}] \) are those structures

\[ G \in (\Gamma_{\Delta}(D))^*[S@\mathcal{A}] \] 

\[ = (\Gamma_{\Delta}(D))^*[S] @ (\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}) \] 

\[ G = G' @ \text{Mem}(\overline{\nabla} [\Gamma_{\Delta}(D)]; \mathcal{A}) \] 

for some \( G' \in \Gamma_{\Delta}(D^*[S]) \)
such that \( \varphi_{\Delta, \Gamma} \Phi (D^*[S@A]; G) = T \). On the other hand, \( e^*[S@\langle \varphi [e] @A \rangle) \) is the set of such \( G \) such that \( \varphi_{\Delta, \Gamma} \Phi (D^*[S]; G') = T \). By the same argument as in the previous case, the induction hypothesis can be applied to \( \varphi_{\Delta, \Gamma} \Phi \). The following calculation then shows that the two restriction on \( G \) are the same.

\[
\begin{align*}
\varphi_{\Delta, \Gamma} \Phi (D^*[S@A]; G) &= \varphi_{\Delta, \Gamma} \Phi ((D^*[S] @ \varphi [D] @A) ; G) \\
&= \varphi_{\Delta, \Gamma} \Phi (\langle D^*[S] @ \varphi [D] @A \rangle ; (G' @ \mem (\varphi [\Gamma\Delta(D)] @A))) \\
&= \varphi_{\Delta, \Gamma} \Phi ((D^*[S]; G') @ ((\varphi [D] @A) ; \mem (\varphi [\Gamma\Delta(D)] @A))) \\
&= \varphi_{\Delta, \Gamma} \Phi (D^*[S]; G')
\end{align*}
\]

- Now suppose that \( e \) has the form \( \sigma \rightarrow \tau \). Lemma 5.2 yields the following.

\[
\begin{align*}
e^*[S@A] &= \sigma^*[S@A] \rightarrow \tau^*[S@A] \\
&= \sigma^*[S@\varphi [\sigma] @A] \rightarrow \tau^*[S@\varphi [\tau] @A] \\
&= (\sigma^*[S@\varphi [\sigma] @A] @ \typeof (\text{Point}) \rightarrow \tau^*[S@\varphi [\tau] @A] \\
&= \sigma^*[S] @ \typeof (\text{Point}) \rightarrow \tau^*[S@\varphi [\tau] @A] \\
&= \sigma^*[S] \rightarrow \tau^*[S@\varphi [\tau] @A]
\end{align*}
\]

One the other hand, lemma 5.3 yields

\[
\begin{align*}
e^*[S@\varphi [e] @A] &= (\sigma^*[S] \rightarrow \tau^*[S]) @ \typeof (\text{Point} \rightarrow \mem (\varphi [\tau] @A)) \\
&= \sigma^*[S] \rightarrow \tau^*[S@\varphi [\tau] @A]
\end{align*}
\]

- Now suppose that \( e \) has the form \( f(w) \). In this case we have

\[
\begin{align*}
(f(w))^*[S@A] &= f^*[S@A] (w^*[S@A]) \\
&= f^*[S@A] (w^*[S@\varphi [f] @A]) @ \text{Point} \\
&= f^*[S@A] (w^*[S@\varphi [w] @A]) @ \text{Point} \\
&= f^*[S@A] (w^*[S@\varphi [f] @A]) @ \text{Point} \\
&= (f^*[S@\varphi [f] @A]) @ \text{Point}
\end{align*}
\]

We have that \( f^*[S@A] = (f^*[S@\varphi [f] @A]) \) must be a function. This implies that the template \( \varphi [f] @A \) must have the form \( \text{Point} \rightarrow B \) for some template \( B \). We then have

\[
\begin{align*}
(f(w))^*[S@A] &= (f^*[S@\varphi [f] @A]) (w^*[S@\text{Point}] \\
&= (f^*[S] (w^*[S@\text{Point}]) @B \\
&= (f^*[S]) (w^*[S]) @B \\
&= (f(w))^*[S@\varphi [f(w)] @A]
\end{align*}
\]

- Now suppose that \( e \) has the form \( G[x \leftarrow w] \). In this case we have

\[
\begin{align*}
(G[x \leftarrow w])^*[S@A] &= (G^*[S@\varphi [G] @A]) (x \leftarrow w^*[S@\varphi [w] @A] \\
&= (G^*[S]) (x \leftarrow w^*[S]) @ (\varphi [G] @A) (x \leftarrow \varphi [w] @A) \\
&= (G[x \leftarrow w])^*[S@\varphi [G[x \leftarrow w]] @A]
\end{align*}
\]
6.2. SOME PRELIMINARY LEMMAS

The following definitions and lemmas will be useful in the proof of the value-composition commutation theorem.

**Definition 6.2.** We say that $\mathcal{V}_S[e]$ commutes with composition if for $S, W : \mathcal{V} \to \Sigma$ with $S \circ W$ defined we have that $(\mathcal{V}_S[e] S) \circ (\mathcal{V}_S[e] W)$ is also defined and $\mathcal{V}_S[e] (S \circ W) = (\mathcal{V}_S[e] S) \circ (\mathcal{V}_S[e] W)$.

**Lemma 6.3.** If $\mathcal{V}_e[\tau]^*$ is defined and is a type implementation, and $\mathcal{V}_e[\tau]$ commutes with composition, then $\mathcal{V}[\tau]$ forms a groupoid (is closed under composition and inverse).

**Proof.** It suffices to show that $\mathcal{V}[\tau] \circ \mathcal{V}[\tau] = \mathcal{V}[\tau]$ and $\mathcal{V}[\tau]^{-1} = \mathcal{V}[\tau]$. For this we note the following.

\[
\mathcal{V}_e[\tau]^* = \mathcal{V}_e[\tau] (\varsigma^* \circ \varsigma^*) = (\mathcal{V}_e[\tau]^* \circ (\mathcal{V}_e[\tau]^*)^*) = \mathcal{V}_e[\tau] ((\varsigma^*)^{-1}) = (\mathcal{V}_e[\tau]^*)^{-1}
\]

**Lemma 6.4.** If $\mathcal{V}_S[\Phi]$ is defined, and for $S : \mathcal{V} \to \Sigma$ we have that $\mathcal{V}_S[\Phi] S$ is a Boolean value, then $\mathcal{V}_S[\Phi]$ commutes with composition if and only if for all $S : \mathcal{V} \to \Sigma$ we have $\mathcal{V}_S[\Phi] \mathbf{Left}(S) = \mathcal{V}_S[\Phi] S$.

**Proof.** We note that if $\mathcal{V}_S[\Phi]$ commutes with composition then

\[
\mathcal{V}_S[\Phi] \mathbf{Left}(S) = \mathcal{V}_S[\Phi] (S \circ S^{-1}) = (\mathcal{V}_S[\Phi] S) \circ (\mathcal{V}_S[\Phi] S)^{-1} = \mathbf{Left}(\mathcal{V}_S[\Phi] S)
\]

Conversely suppose that for all $S : \mathcal{V} \to \Sigma$ we have $\mathcal{V}_S[\Phi] \mathbf{Left}(S) = \mathbf{Left}(\mathcal{V}_S[\Phi] S)$. We first note that

\[
\mathcal{V}_S[\Phi] S = (\mathcal{V}_S[\Phi] S^{-1})^{-1} = \mathcal{V}_S[\Phi] (S^{-1}) = \mathcal{V}_S[\Phi] \mathbf{Left}(S^{-1}) = \mathcal{V}_S[\Phi] \mathbf{Right}(S)
\]
Now for $U, W : \mathcal{V}[\Sigma]$ with $U \circ W$ defined we have
\[
\nu_\Sigma [\Phi](U \circ W) = \nu_\Sigma [\Phi] \operatorname{Left}(U \circ W) = \nu_\Sigma [\Phi] \operatorname{Left}(U) = \nu_\Sigma [\Phi] U = \nu_\Sigma [\Phi] \operatorname{Right}(U \circ W) = \nu_\Sigma [\Phi] \operatorname{Right}(W) = \nu_\Sigma [\Phi] W
\]
This implies $\nu_\Sigma [\Phi](U \circ W) = (\nu_\Sigma [\Phi] U) \circ (\nu_\Sigma [\Phi] W)$.

**Definition 6.5.** For any two pre-types $\sigma$ and $\tau$ we define $\sigma \ast \tau$ to be the pre-type whose members are the values of the form $x \circ y$ for $x \in \sigma$ and $y \in \tau$ and $x \circ y$ defined.

The difference between $\sigma \ast \tau$ and $\sigma \circ \tau$ is simply that $\sigma \ast \tau$ is always defined while $\sigma \circ \tau$ is only defined when $\operatorname{Right}(\sigma) = \operatorname{Left}(\tau)$.

**Lemma 6.6.** If $\nu_\Sigma [e]$ is defined, and $\mathcal{V}[\Sigma]$ forms a groupoid, and for $S : \mathcal{V}[\Sigma]$ we have that $\nu_\Sigma [e] S$ is an implementation value, and for all $S, W : \mathcal{V}[\Sigma]$ with $S \circ W$ defined we have $\nu_\Sigma [e] (S \circ W) = \nu_\Sigma [e] S \ast \nu_\Sigma [e] W$, then $\nu_\Sigma [e]$ commutes with composition.

**Proof.** We must show that for $S, W : \mathcal{V}[\Sigma]$ with $S \circ W$ defined we have that $(\nu_\Sigma [e] S) \circ (\nu_\Sigma [e] W)$ is defined. We are given $S^{-1} \circ S = W \circ W^{-1}$. We now have
\[
\nu_\Sigma [e] (S^{-1} \circ S) = \nu_\Sigma [e] (S^{-1}) \ast (\nu_\Sigma [e] S) = (\nu_\Sigma [e] S)^{-1} \ast (\nu_\Sigma [e] S) = (\nu_\Sigma [e] S)^{-1} \circ (\nu_\Sigma [e] S) = \operatorname{Right}(\nu_\Sigma [e] S) = \nu_\Sigma [e] (W \circ W^{-1}) = (\nu_\Sigma [e] W) \ast \nu_\Sigma [e] (W^{-1}) = (\nu_\Sigma [e] W) \ast (\nu_\Sigma [e] W)^{-1} = (\nu_\Sigma [e] W) \circ (\nu_\Sigma [e] W)^{-1} = \operatorname{Left}(\nu_\Sigma [e] W)
\]
6.3 Value-Composition Commutation

Lemma 6.7. If $\mathcal{V}_\Sigma[e]$ is defined then for $S : \mathcal{V}[\Sigma]$ we have $\mathcal{V}_\Sigma[e](S^{-1}) = (\mathcal{V}_\Sigma[e] S)^{-1}$.

Theorem 6.8 (Value-Composition Commutation). If $\mathcal{V}_\Sigma[e]$ is defined then

1. For $S : \mathcal{V}[\Sigma]$ we have that $\mathcal{V}_\Sigma[e] S$ is an implementation value.

2. $\mathcal{V}_\Sigma[e]$ commutes with composition.

Proof. We prove both statements simultaneously by induction on $\prec$. We consider each case in the definition of the value function assuming that the above properties hold for recursive invocations of the value function. Note that in all cases other than the degenerate base case we have that $\mathcal{V}[\Sigma]$ is prior and by the induction hypothesis we have that $\mathcal{V}[\Sigma]$ commutes with composition. Lemma 6.3 then implies that $\mathcal{V}[\Sigma]$ forms a groupoid.

- $\mathcal{V}_\Sigma[\tau_\Delta(D)]$: This case is immediate as $\mathcal{V}_\Sigma[\tau_\Delta(D)] S$ is always the type containing $\preceq$.
- $\mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)]$: In this case we are given $\Sigma \models D : \Sigma$ and $\Delta : \Gamma \models \tau : \text{type}_t$.

Property 1. To prove property 1 it suffices to show that $\mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)] S$ is back-and-forth closed. Consider $G[x \leftarrow v], H[x \leftarrow s]$ and $F[x \leftarrow w]$ in $\mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)] S$ with $G[x \leftarrow v] \circ H[x \leftarrow s]^{-1} \circ F[x \leftarrow w]$ defined. We must show that this composition is also a member of $\mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)] S$. But the composition can be written as $(G \circ F^{-1} \circ H)[x \leftarrow (v \circ s^{-1} \circ w)]$. Since $\mathcal{V}_\Sigma[\tau_\Delta(D)] S$ is a type implementation we have $(G \circ F^{-1} \circ H) \in \mathcal{V}_\Sigma[\tau_\Delta(D)] S$. It now suffices to show

$$(v \circ s^{-1} \circ w) \in \mathcal{V}_{\Delta,\Gamma}[\tau] ((\mathcal{V}_\Sigma[D] S); (G \circ F^{-1} \circ H)).$$

Let $D^*$ abbreviate $\mathcal{V}_\Sigma[D] S$. By property 2 of the induction hypothesis we have

$$\mathcal{V}_{\Delta,\Gamma}[\tau] (D^*; (G \circ F^{-1} \circ H)) = \mathcal{V}_{\Delta,\Gamma}[\tau] ((D^* \circ (D^*)^{-1} \circ D^*); (G \circ F^{-1} \circ H)) = \mathcal{V}_{\Delta,\Gamma}[\tau] ((D^*; G) \circ (D^*; F)^{-1} \circ (D^*; H)) = (\mathcal{V}_{\Delta,\Gamma}[\tau] (D^*; G)) \circ (\mathcal{V}_{\Delta,\Gamma}[\tau] (D^*; F))^{-1} \circ (\mathcal{V}_{\Delta,\Gamma}[\tau] (D^*; K))$$

which proves the result.

Property 2. By lemma 6.6 it suffices to show that for $S, W : \mathcal{V}[\Sigma]$ with $S \circ W$ defined we have

$$(\mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)] (S \circ W) = (\mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)] S) * (\mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)] W)).$$

We will show that every member of the left hand side is a member of the right hand side and vice-versa. Consider a member of the left hand side $I[x \leftarrow s] \in \mathcal{V}_\Sigma[\Gamma : x : \tau_\Delta(D)] (S \circ W)$. We have $I \in \mathcal{V}_\Sigma[\tau_\Delta(D)] (S \circ W)$ which by the induction hypothesis implies $I \in (\mathcal{V}_\Sigma[\tau_\Delta(D)] S) \circ (\mathcal{V}_\Sigma[\tau_\Delta(D)] W)$. Hence there
exists \( G \in \mathcal{V}_2 \llbracket \Gamma \Delta(D) \rrbracket \) \( S \) and \( H \in \mathcal{V}_2 \llbracket \Gamma \Delta(D) \rrbracket \) \( W \) such that \( I = G \circ H \). We now consider the inserted value \( s \). We have

\[
\begin{align*}
    s & \in \mathcal{V}_2; \Gamma \llbracket \tau \rrbracket ((\mathcal{V}_2; \llbracket D \rrbracket S \circ W); I) \\
    & = \mathcal{V}_2; \llbracket \Gamma \rrbracket \llbracket \tau \rrbracket ((\mathcal{V}_2; \llbracket D \rrbracket S) \circ (\mathcal{V}_2; \llbracket D \rrbracket W); (G \circ H)) \\
    & = \mathcal{V}_2; \llbracket \Gamma \rrbracket \llbracket \tau \rrbracket ((\mathcal{V}_2; \llbracket D \rrbracket S; G) \circ (\mathcal{V}_2; \llbracket D \rrbracket W; H)) \\
    & = (\mathcal{V}_2; \llbracket \Gamma \rrbracket \llbracket \tau \rrbracket ((\mathcal{V}_2; \llbracket D \rrbracket S; G)) \circ (\mathcal{V}_2; \llbracket D \rrbracket W; H))
\end{align*}
\]

But this implies that there exists \( v \in \mathcal{V}_2; \llbracket \tau \rrbracket (\mathcal{V}_2; \llbracket D \rrbracket S; G) \) and \( w \in \mathcal{V}_2; \llbracket \tau \rrbracket (\mathcal{V}_2; \llbracket D \rrbracket W; H) \) with \( s = v \circ w \). We now have \( I[x \leftarrow s] = (G[x \leftarrow v]) \circ (H[x \leftarrow w]) \) which gives

\[
I[x \leftarrow s] \in (\mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket S \rrbracket) \ast (\mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket W \rrbracket).
\]

Conversely, consider

\[
G[x \leftarrow v] \circ H[x \leftarrow w] \in (\mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket S \rrbracket) \ast (\mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket W \rrbracket).
\]

We have \( (G \circ H) \in (\mathcal{V}_2; \llbracket \Gamma \Delta(D) \rrbracket \llbracket S \rrbracket) \ast (\mathcal{V}_2; \llbracket \Gamma \Delta(D) \rrbracket \llbracket W \rrbracket) \) and from the induction hypothesis we have \( (G \circ H) \in (\mathcal{V}_2; \llbracket \Gamma \Delta(D) \rrbracket \llbracket S \circ W \rrbracket) \). We now consider the value \( v \circ w \). We have

\[
\begin{align*}
    v \circ w & \in (\mathcal{V}_2; \llbracket \Gamma \Delta(D) \rrbracket \llbracket S \rrbracket) \ast (\mathcal{V}_2; \llbracket \Gamma \Delta(D) \rrbracket \llbracket W \rrbracket)
\end{align*}
\]

This gives \( (G \circ H)[x \leftarrow (v \circ w)] \in (\mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket S \circ W \rrbracket) \).

\* \( \mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket S \rrbracket \): In this case we are given \( \Sigma \models \Delta; \Gamma \models \Phi \ast \text{Bool} \).

**Property 1.** We must show that for \( S : \mathcal{V}[\Sigma] \) we have that \( \mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket S \rrbracket \) is back and forth closed. Consider \( G, H, F \in \mathcal{V}_2; \llbracket \Gamma; x; \tau \Delta(D) \rrbracket \llbracket S \rrbracket \) with \( G \circ H^{-1} \circ F \) defined. By the induction hypothesis we have that \( \mathcal{V}_2; \llbracket \Gamma \Delta(D) \rrbracket \llbracket S \rrbracket \) is an implementation value and hence is back-and-forth closed. Hence we have \( (G \circ H^{-1} \circ F) \in \mathcal{V}_2; \llbracket \Gamma \Delta(D) \rrbracket \llbracket S \rrbracket \). It remains only to show that \( \mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket \llbracket (\mathcal{V}_2; \llbracket D \rrbracket S; (G \circ H^{-1} \circ F)) = T \). Letting \( D^* \) abbreviate \( \mathcal{V}_2; \llbracket D \rrbracket S \) we have

\[
\begin{align*}
    \mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket \llbracket \Phi \rrbracket (D^*; (G \circ H^{-1} \circ F)) \\
    = \mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket \llbracket \Phi \rrbracket ((D^* \circ (D^*)^{-1} \circ D^*); (G \circ H^{-1} \circ F)) \\
    = \mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket \llbracket \Phi \rrbracket ((D^*; G) \circ (D^*; H)^{-1} \circ (D^*; F)) \\
    = \mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket \llbracket \Phi \rrbracket (D^*; G) \circ (\mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket \llbracket \Phi \rrbracket (D^*; F))^{-1} \circ \mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket \llbracket \Phi \rrbracket (D^*; H) \\
    = T
\end{align*}
\]

**Property 2.** By lemma 5.6 it suffices to show that for \( S, W : \mathcal{V}[\Sigma] \) with \( S \circ W \) defined we have

\[
\mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket (D) (S \circ W) = (\mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket (D) S) \ast (\mathcal{V}_2; \llbracket \Gamma \Delta \rrbracket (D) W).
\]
We will show that every member of the left hand side is a member of the right hand side and vice-versa. Consider a member of the left hand side \( I \in \mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket (S \circ W) \). By the induction hypothesis implies \( I \in (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket S) \circ (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket W) \). Hence there exists \( G \in \mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket S \) and \( H \in \mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket W \) such that \( I = G \circ H \). We now have

\[
\mathcal{V}_{\Delta, \Gamma} [\Phi] ((S \circ W); I) = T \\
= \mathcal{V}_{\Delta, \Gamma} [\Phi] ((S \circ W); (G \circ H)) \\
= (\mathcal{V}_{\Delta, \Gamma} [\Phi] (S; G)) \circ (\mathcal{V}_{\Delta, \Gamma} [\Phi] (W; H))
\]

But this implies that \( \mathcal{V}_{\Delta, \Gamma} [\Phi] (S; G) = T \) and \( \mathcal{V}_{\Delta, \Gamma} [\Phi] (W; H) = T \). We now have

\[
I = G \circ H \in (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket S) \ast (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket W).
\]

Conversely, consider \( G \circ H \in (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket S) \ast (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket W) \). We have \( (G \circ H) \in (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket S) \circ (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket W) \) and from the induction hypothesis we have \( (G \circ H) \in \mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket (S \circ W) \). But we have

\[
\mathcal{V}_{\Delta, \Gamma} [\Phi] ((S \circ W); (G \circ H)) \\
= \mathcal{V}_{\Delta, \Gamma} [\Phi] (((\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket S; G) \circ (\mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket W); H)) \\
= T
\]

This gives \( (G \circ H) \in \mathcal{V}_\Sigma \llbracket \Gamma \cdot \Phi_\Delta (D) \rrbracket (S \circ W) \).

- \( \mathcal{V}_\Sigma [\text{Bool}] \), \( \mathcal{V}_\Sigma [\text{type}] \) and \( \mathcal{V}_\Sigma [\cdot] \): The result is immediate for any constant symbol whose value is an implementation value.

- \( \mathcal{V}_\Sigma [x] \): For property 1 we note that for \( S : \mathcal{V} \llbracket x \rrbracket \) we have that \( S \), and hence \( S.x \), are implementation values. For property 2 we note that for \( S, W : \mathcal{V} \llbracket x \rrbracket \) we have \( (S \circ W).x = S.x \circ W.x \).

- \( \mathcal{V}_\Sigma [\sigma \rightarrow \tau] \): Property 1 follows from lemma 5.4. Property 2 follows from lemma 5.4.

- \( \mathcal{V}_\Sigma [f(e)] \): Property 1 follows from the definition of a function. For property 2 consider \( S, W : \mathcal{V} \llbracket x \rrbracket \) with \( S \circ W \) defined. Using the induction hypothesis and lemma 5.1 we get the following.

\[
\mathcal{V}_\Sigma [f(e)] (S \circ W) = ((\mathcal{V}_\Sigma [f] S) \circ (\mathcal{V}_\Sigma [f] W)) ((\mathcal{V}_\Sigma [e] S) \circ (\mathcal{V}_\Sigma [e] W)) \\
= ((\mathcal{V}_\Sigma [f] S) \circ (\mathcal{V}_\Sigma [f] W)) ((\mathcal{V}_\Sigma [e] S) \circ (\mathcal{V}_\Sigma [e] W)) @ \text{Point} \\
= ((\mathcal{V}_\Sigma [f] S) \circ (\mathcal{V}_\Sigma [f] W)) ((\mathcal{V}_\Sigma [e] S) \circ (\mathcal{V}_\Sigma [e] W)) @ \text{Point} \\
= (\mathcal{V}_\Sigma [f] S)(\mathcal{V}_\Sigma [e] S) @ \text{Point} \circ (\mathcal{V}_\Sigma [f] W)(\mathcal{V}_\Sigma [e] W) @ \text{Point} \\
= (\mathcal{V}_\Sigma [f] S)(\mathcal{V}_\Sigma [e] S) \circ (\mathcal{V}_\Sigma [f] W)(\mathcal{V}_\Sigma [e] W) \\
= (\mathcal{V}_\Sigma [f(e)] S) \circ (\mathcal{V}_\Sigma [f(e)] W)
\]

- \( \mathcal{V}_\Sigma [G [x \leftarrow e]] \) and \( \mathcal{V}_\Sigma [G.x] \): In both cases both properties follow straightforwardly from the induction hypothesis.
• $\forall x: \tau \Phi$: We must show that for $S: \mathcal{V}[\Sigma]$ we have $\forall x: \tau \Phi \models \mathcal{V}[\mathcal{S}]$ where $e^*$ abbreviates $\mathcal{V}[e]$, and $w^*$ and $\sigma^*$ abbreviate analogous values, and $\mathcal{V}$ is the template of members of $\sigma^*$. Note that by the induction hypothesis we have $\mathcal{V}_{\Sigma}[e] \mathcal{Left}(S) = \mathcal{Left}(e^*)$ and similarly for $\sigma$ and $w$.

$$\mathcal{V}_{\Sigma}[e =_{\sigma} w] S = T = e^* =_{\sigma^*} w^* = (e^* @ \mathcal{V}) =_{\sigma^*} (w^* @ \mathcal{V}) = \mathcal{Left}(e^* @ \mathcal{V}) =_{\mathcal{Left}(\sigma^*)} \mathcal{Left}(w^* @ \mathcal{V}) = (\mathcal{Left}(e^*) @ \mathcal{V}) =_{\mathcal{Left}(\sigma^*)} (\mathcal{Left}(w^*) @ \mathcal{V}) = \mathcal{Left}(e^*) =_{\mathcal{Left}(\sigma^*)} \mathcal{Left}(w^*) = \mathcal{V}_{\Sigma}[e =_{\sigma} w] \mathcal{Left}(S) = T$$

• $\forall x: \tau \Phi$: We must show that for $S: \mathcal{V}[\Sigma]$ we have that $\forall x: \tau \Phi \models \mathcal{V}[\mathcal{S}]$ for all $u: \tau^*$ we have

$$\mathcal{V}_{\Sigma}[\forall x: \tau \Phi] S[x ← u] = T$$

and consider $z: \mathcal{V}[[\Phi]] \mathcal{Left}(S)$. We must show

$$\mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(S)[x ← z] = T.$$

We first observe the following.

$$\mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(S)[x ← z] = \mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(S)[x ← z @ \tau^*] = \mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(\mathcal{Left}(S)[x ← z @ \tau^*]) = \mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(\mathcal{Left}(S))[x ← \mathcal{Left}(z @ \tau^*)] = \mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(S)[x ← \mathcal{Left}(z @ \tau^*)]$$

By the induction hypothesis we have $\forall x: \tau \Phi \models \mathcal{V}[\mathcal{S}] = \mathcal{Left}(\tau^*)$ which gives $\forall x: \mathcal{Left}(\tau^*)$. We now have $z @ \tau^* = u @ w^{-1}$ for some $u, w \in \tau^*$. We can then continue the calculation as follows.

$$= \mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(S)[x ← \mathcal{Left}(u)] = \mathcal{V}_{\Sigma}[\forall x: \tau \Phi] \mathcal{Left}(S[x ← u]) = \mathcal{V}_{\Sigma}[\forall x: \tau \Phi] S[x ← u] = T$$

Now suppose that for all $z: \mathcal{Left}(\tau^*)$ we have

$$\forall x: \tau \Phi \models \mathcal{V}[\mathcal{S}] = T.$$
6.4. THE ABSTRACTION THEOREM

But we can show \( \text{Left}(u) : \text{Left}(\tau^*) \) as follows.

\[
\text{Left}(u) \circ \tau^* = (u \circ u^{-1}) \circ \tau^* = (u \circ \tau^*) \circ (u \circ \tau^*)^{-1} \in \text{Left}(\tau^*)
\]

This gives

\[
\forall \Sigma; x : [\Phi] (\text{Left}(S)[x \leftarrow \text{Left}(u)]) = T.
\]

• Finally suppose that \( e \) is a disjunction or negation. These cases follow directly from the induction hypothesis. For example we have

\[
\forall \Sigma; [\Phi \lor \Psi] \text{Left}(S) = (\forall \Sigma; [\Phi] \text{Left}(S)) \lor (\forall \Sigma; [\Psi] \text{Left}(S))
\]

\[
= (\forall \Sigma; [\Phi] S) \lor (\forall \Sigma; [\Psi] S)
\]

\[
= \forall \Sigma; [\Phi \lor \Psi] S
\]

\[
\square
\]

6.4 The Abstraction Theorem

**Theorem 6.9 (Abstraction).** If \( \Sigma; x : \sigma \models e[x] : \tau \), where \( x \) does not occur free in \( \tau \), then for \( S : \Sigma \) and \( u, w : \forall \Sigma; [\sigma] S \) with \( u = \forall \Sigma; [\sigma] S w \), we have

\[
\forall \Sigma; x : \sigma [e[x]] S[x \leftarrow u] = \forall \Sigma; [\tau] S \quad \forall \Sigma; x : \sigma [e[x]] S[x \leftarrow w].
\]

**Proof.** Suppose \( \Sigma; x : \sigma \models e[x] : \tau \) where \( x \) does not occur free in \( \tau \) and consider \( S : \forall \Sigma \) and \( u, w : \forall \Sigma; [\sigma] S \) with \( u = \forall \Sigma; [\sigma] S w \). Let \( \sigma^* \) abbreviate \( \forall \Sigma; [\sigma] S \). By the definition of \( u = \sigma^* w \) we have that there exists \( z \in \sigma^* \) with \( (u \circ \sigma^*) \circ z^{-1} \circ (w \circ \sigma^*) \) defined and hence \( ((u \circ \sigma^*) \circ z^{-1} \circ (w \circ \sigma^*)) \in \sigma^* \). We can now make the following...
calculation.

\[
\begin{align*}
\nu_{\Sigma; x: \sigma} & [[e[x]] S[x \leftarrow (u@\sigma^* \circ z^{-1} \circ w@\sigma^*)]@\tau^* \\
= & \nu_{\Sigma; x: \sigma} [[e[x]] (S \circ S^{-1} \circ S)[x \leftarrow (u@\sigma^* \circ z^{-1} \circ w@\sigma^*)]@\tau^* \\
= & \nu_{\Sigma; x: \sigma} [[e[x]] ((S[x \leftarrow u@\sigma^*]) \circ (S[x \leftarrow z])^{-1} \circ (S[x \leftarrow w@\sigma^*]))]@\tau^* \\
= & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow u@\sigma^*]]) @\tau^*) \\
\circ & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow z]])^{-1} @\tau^*) \\
\circ & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow w@\sigma^*]]) @\tau^*) \\
= & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow u]]) \@\nu [[e[x]] I(S)[x \leftarrow M(I(\sigma^*))]) @\tau^*) \\
\circ & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow z]])^{-1} @\tau^*) \\
= & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow w]]) \@\nu [[e[x]] I(S)[x \leftarrow M(I(\sigma^*))]) @\tau^*) \\
= & (\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow u]) @\tau^* \\
\circ & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow z]])^{-1} @\tau^*) \\
\circ & ((\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow w]]) @\tau^*) \\
\end{align*}
\]

which yields

\[
\nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow u] =_{\nu_{\Sigma; x: \sigma}} \nu_{\Sigma; x: \sigma} [[e[x]] S[x \leftarrow w].
\]

\qed
Chapter 7

Outstanding Issues

A first outstanding issue is the construction of inference rules allowing for the derivation of equivalence (isomorphism). This would seem to require the introduction of linguistics constructs that violate the abstraction barriers imposed by types. For example, we might introduce a formula $x \Rightarrow y$ to indicate that $z \in \tau$ and $(x@\tau) \circ (y@\tau)$ is defined (without requiring $x:\tau$ or $y:\tau$). However, the truth of this formula is not preserved under substitution of isomorphics. Expressions not respecting the substitution of isomorphics could be viewed as “impure” and the substitution rule could be restricted to pure expressions. One can imagine inference rules that mix pure and impure constructs. In any case we have left the construction of rules for deriving isomorphism relations for future work.

A second outstanding issue is the possibility of some form of abstraction theorem handling homomorphisms, continuous maps and logical relations. It seems clear that categories other than groupoids are appropriate for discussing very general notions such as homomorphisms. The goal here would be to explicitly motivate more general category-theoretic structure, or perhaps other constructions, by proving some form of abstraction theorem for well-typed expressions.

There are also issues related to connections between the foundations of mathematics and cognitive science. A first such issue is that the notion of isomorphism seems to arise in common sense. The common sense conception of number seems clearly related to isomorphism of finite sets — the equivalence classes of $\equiv_{\text{type}_0}$. Similarly, ordinal numbers and bags also seem related to the notion of isomorphism. It is also common sense that things of different sizes can be the same shape. This expresses a common-sense understanding of symmetries of scale.

A second issue relating mathematical foundations and common sense is that of understanding the relationship, if any, between the mathematical notion of structure type and the notion of “frame” that arises in natural language semantics [1]. Frames of natural language, such as the buy/sell frame, can be used quite creatively as in “The editor’s endorsement was purchased by adding those planks to the platform”. It seems plausible that creative instantiation
of linguistic frames is formally related to creating interpretations (implementations) of structure types. In both cases values are being assigned to the symbols of the structure type or frame. This might even extend the use of frames as metaphores [2].

A final issue is the relationship between type-theoretic mathematical foundations and the teaching of mathematics. The semantics presented here is far too advanced to be taught as part of a course designed to introduce students to mathematical rigor. However, the inference rules of section 1 seem quite accessible and could be taught as purely syntactic rules. For this purpose it would be very helpful to have rules for deriving isomorphisms. At the very least students should be taught to distinguish between those statements that are well-formed and those that are not.

1. C.J. Fillmore. Scenes-and-frames semantics. *Linguistic structures processing*, 59:55–88, 1977.

2. G. Lakoff and M. Johnson. *Metaphors we live by*, volume 111. Chicago London, 1980.