Type Safety *and* Logical Relations

Typesikkerhed og logiske relationer

Bachelor Project in Computer Science
Danny Nygård Hansen • 201605113

Supervisor: Lars Birkedal

10th June 2024

Department of Computer Science
Aarhus University

Layout and typography by the author
Font: 10 pt KP Serif | 8.5 pt Bera Sans | 8.5 pt `Bera Mono`
Typeset using pdfLaTeX and the *memoir* class
All figures generated using Ti*k*Z

Used packages:

array	cleveref	imakeidx	multicol
babel	csquotes	inputenc	ntheorem
beramono	enumitem	kpfonts	pgffor
berasans	fontenc	mathpartir	tikz
biblatex	graphicx	mathtools	tikz-cd
caption	hyperref	microtype	xcolor

The source code for this project is available on GitHub.

Resumé

I denne opgave studerer vi typesikkerhed af en udgave af System **F** med referencer og rekursive funktioner og typer. Vi tager først den klassiske tilgang, idet vi beviser sætninger om progress og preservation som medfører typesikkerhed for dette sprog.

For et fragment af dette sprog uden referencer og rekursion beviser vi også typesikkerhed ved hjælp af et logisk prædikat, og vi definerer en binær logisk relation som kan benyttes til at vise kontekstuel ækvivalens af programmer.

Abstract

In this report we study type safety of a version of System **F** with references and recursive functions and types. We first take the classical approach, proving progress and preservation theorems which imply type safety for this language.

For a fragment of this language without references and recursion we also prove type safety using a logical predicate, and we define a binary logical relation which can be used to show contextual equivalence of programs.

Contents

Preface

For the computer scientist, programming languages and programming in general are both an important tool and an interesting and fruitful subject of study. While we will briefly discuss practical applications of type systems, this report is chiefly concerned with the latter: In particular, we study how to describe programming languages and their semantics in a manner that is amenable to proof, how type systems give guarantees on runtime safety of programs, and how we can talk about two programs being 'the same'.

The genesis of this project was a project course on logical relations led by Lars Birkedal and Simon Oddershede Gregersen in the autumn of 2023, the purpose of my own project essentially being to write up the solutions to exercises in all their gory detail, for the benefit of current and future students. While this purpose can hardly be said to have survived the writing process, I have tried as far as possible to be clear and pedagogical, with the hope that the present report reads more like a textbook than a journal article.

There is of course another motivation for writing a report like this one: Namely, obtaining a bachelor's degree. And the report must then, in part, show that I have learnt enough to warrant this distinction. Hopefully this is the case, though I will certainly claim that the process of writing it has been a great learning experience.

The end result is a report that is, quite frankly, all over the place. I have been rather (some would say too) detailed in my presentation of the syntax and semantics of programming languages, something which I have rarely, if at all, seen done in the introductory literature. On the other hand I make liberal (some would say gratuitous) references to category theory, topology, algebra and even measure theory to try to explicate the concepts being studied. I hope that the reader not familiar with these topics will forgive me.

Organisation

Chapter 1 is a miscellany of foundational topics that appear throughout the report. It covers the elementary theory of inference rules, induction and recursive definitions, making ample use of Appendix A on order theory. Next abstract syntax is introduced, and we give a rather thorough, if informal, treatment of abstract syntax trees in the style of Harper (2016). Then follows short sections on reduction, partitions and coinduction, whose purpose is more or less to introduce definitions and fix notation.

In Chapter 2 we introduce the two languages we study throughout the rest of the report. These are the pure language **F** which is a variant of

Girard's System **F**, and the extension $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ which among other things include references as well as recursive functions and types. After a tour of the type system of these languages, we prove some of their fundamental properties. Keywords are 'inversion' and 'canonical forms'.

Chapters 3 and 4 are both concerned with type safety. The first of these chapters proves type safety for $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ using the classic approach of 'progress and preservation'. The second instead focuses on **F** and uses a logical predicate to prove type safety.

Next, Chapter 5 introduces (binary) logical relations, which are extensions of logical predicates to two arguments. We consider how to define equivalence of programs, and define a logical relation as a tool to prove equivalence of concrete programs.

Appendix A is a fairly extensive and mostly self-contained appendix covering elementary order theory with a view towards the fixed-point theorems we need to get us off the ground in Chapter 1. We generalise the theory beyond what is necessary for our purposes: It turns out that the theory underlying recursive definitions of things like type systems or operational semantics appears in a more general context in mathematics, in particular in topology and measure theory. While a more elementary approach is sufficient to cover our applications of the theory, we wish to highlight exactly what it is that makes type systems different from e.g. topologies or σ-algebras. This all but necessitates the use of transfinite induction and recursion, but we provide an elementary path through the material for readers either not familiar with the requisite set theory, or readers simply not interested in the aforementioned generalisations.

Appendix B collects the syntax and semantics of our languages. To avoid cluttering the body of the text, this is the only place the complete specifications of the languages are located. To mitigate flipping to and from this appendix, we thus encourage the reader to either open a second copy of this report in another window, or to use a pdf-reader with a 'back'-button. Every cross reference includes a link for precisely this purpose.

Finally, after the bibliography are located two fairly extensive indices that will hopefully help the reader find their way through the report.

Acknowledgments

I am first and foremost greatly indebted to my supervisor Lars Birkedal, and I thank him for his guidance and generosity throughout the project. Thanks are also due to Simon Oddershede Gregersen for his guidance on the logical relations project, to Amin Timany for preparing the initial specification of the languages studied in the project, and to Lau Skorstengaard without whose notes on logical relations there might not have been a project in the first place. Finally I thank Gerth Stølting Brodal and Asger Klinkby Jørgensen for inadvertently inspiring me to study computer science.

Aarhus, June 2024 Danny Nygård Hansen

Preliminaries | 1

This first chapter introduces various topics that will be needed in the sequel. In particular we consider recursive definitions of sets and functions, inference rules and generating functions, abstract syntax, reduction systems, partitions and coinduction.

1.1 ◇ Notation

We begin by fixing notation.

If X is a set, then we denote by $\mathcal{P}(X)$ the power set of X, and for a cardinal κ we furthermore write $\mathcal{P}_\kappa(X)$ for the collection of subsets of X with cardinality strictly less than κ. If $A \in \mathcal{P}_\kappa(X)$, then we also write $A \subseteq_\kappa X$. For ordinals α we do not distinguish between the ordinal ω_α and the cardinal $\aleph_\alpha$, and we write $\omega \coloneqq \omega_0$. Hence $\mathcal{P}_\omega(X)$ denotes the set of finite subsets of X, and $A \subseteq_\omega X$ means that A is a finite subset of X.

Readers unfamiliar with cardinal and ordinal numbers can for our purposes safely think of κ as an element of $\mathbb{N} \cup \{\infty\}$, and of ω as ∞.

If Y is another set, then we denote the set of partial maps from X to Y by $(X \rightharpoonup Y)$. If $f \in (X \rightharpoonup Y)$ then we let $\operatorname{dom} f$ and $\operatorname{ran} f$ denote the domain and range of f, respectively. We also write $(X \rightharpoonup_\kappa Y)$ for the partial functions $f \in (X \rightharpoonup Y)$ with $\operatorname{dom} f \subseteq_\kappa X$. For $x_0 \in X$ and $y_0 \in Y$ we denote by $f[x_0 \mapsto y_0]$ the partial map given by

$$f[x_0 \mapsto y_0](x) = \begin{cases} y_0, & x = x_0, \\ f(x), & x \in \operatorname{dom} f \setminus \{x_0\}. \end{cases}$$

Hence $\operatorname{dom} f[x_0 \mapsto y_0] = \operatorname{dom} f \cup \{x_0\}$, so that if $f \in (X \rightharpoonup_\omega Y)$, then also $f[x_0 \mapsto y_0] \in (X \rightharpoonup_\omega Y)$. Any partial map with empty domain is denoted $\bot$.

We let $\mathbb{N}$ denote the set of natural numbers, including 0, $\mathbb{Z}$ the set of integers, and $\mathbb{N}_+$ the set of strictly positive integers.

The image of a set $A \subseteq X$ under a function $f \colon X \to Y$ is denoted $f[A]$. We similarly write $f^{-1}[B]$ for the preimage of $B \subseteq Y$ under f. The restriction of f to A is denoted $f|_A$.

It is common to write simply $f(A)$ for the image of A under f, relying on context to distinguish this from the value of f at an *element* A of X.

An ***indexed family*** of elements of a set X is a map $I \to X$, where I is any set called the ***index set*** of the family. If $\mathcal{A}$ is an indexed family of elements of X and Y is some set, then we write $\mathcal{A} \subseteq Y$ if the image of $\mathcal{A}$ is a subset of Y. If the image of $i \in I$ under the indexed family is x_i, then we also write $(x_i)_{i \in I}$. If I is the finite set $\{1, \dots, n\}$, then we also denote the family $(x_i)_{i \in I}$ by $\vec{x}$. The set of finite sequences of elements from X is denoted X^*.

1.2 ◇ Definition by recursion

1.2.1. Generating functions

Let $(P, \leq)$ be a poset, and let $f : P \to P$ be a monotone map. We think of f as a ***generating function***, in the sense that given an element $x \in P$ representing a set of states of some system, it generates a new set of states $f(x)$. The fact that f is monotone models that given a larger set of states as input, a larger set of new states is also generated.

We wish to find an element $x \in P$ such that f cannot generate anything new from it—i.e., x should be f-closed—and we don't want this element to contain any 'redundancies': It should contain only those states mandated by f. More precisely, x should be the *least* f-closed element in P. But Lemma A.4.2 says that x is then a fixed-point of f, indeed its smallest fixed-point, and is denoted $\mu(f)$. Hence we may instead focus our attention on fixed-points.

In this generality it is not possible to systematically study the fixed-points of f. Indeed, f may not even have any! Hence we must impose appropriate further requirements on P or f. If P is a power set, then it is in particular a complete lattice, and we thus have access to the Knaster–Tarski fixed-point theorem (cf. Theorem A.4.9), ensuring the existence of both a least and a greatest fixed-point.

We will also have occasion to consider fixed-points of monotone functions defined on posets that are not complete lattices. Most importantly, notice that a set $(X \rightharpoonup Y)$ of partial functions is not generally a complete lattice, but that it does satisfy a collection of weaker properties: The empty map $\bot$ is its least element, and every directed subset D of $(X \rightharpoonup Y)$ has a join, namely the map whose graph is the union of the graphs of the elements in D. This means that $(X \rightharpoonup Y)$ is a dcppo, and thus Kleene's fixed-point theorem applies (cf. Theorems A.4.12 and A.4.13). This also ensures that $\mu(f)$ is the least f-closed element of P. Especially important in the present context is the following:

1.2.1 • Corollary: *Principle of induction.*
Let P be a dcppo and let $f : P \to P$ be monotone. If $y \in P$ is f-closed, then $\mu(f) \leq y$. ∎

1.2.2. Inference rules

When we in Theorem 1.2.5 study how to define functions recursively using inference rules, it will be important that H is indexed. Compare e.g. Pitts (2013, §7.1) who only considers H as a set.

Specialising to the case where P is a power set $\mathcal{P}(X)$, one way to define a generating function is using inference rules. An ***inference rule*** over X is a pair (H, y), where H is an indexed family of elements of X called the ***hypotheses*** of the rule, and $y \in X$ is its ***conclusion***. We denote this rule by[1] $H \Mapsto y$, and if we give this rule the name R, then we also write $R\colon H \Mapsto y$. The index set of H is then also denoted $\mathfrak{i}(R)$.

If R has finitely many premises $\vec{x} = (x_1, \ldots, x_n)$, then we say that R is ***finitary***. In this case we naturally write $R\colon \vec{x} \Mapsto y$, and we also use the

[1] This is not standard notation. Pitts (2013, §7.1) simply denotes it by (H, y), but we prefer to use a slightly more distinctive notation.

notation

$$\frac{x_1 \quad x_2 \quad \cdots \quad x_n}{y}\,R .$$

We allow n to be zero, in which case we call the rule an ***axiom*** and write $\Rightarrow y$.

Given a (usually infinite) collection $\mathcal{R}$ of inference rules, we construct a generating function $F\colon \mathcal{P}(X) \to \mathcal{P}(X)$ by defining $F(A)$ for a subset $A \subseteq X$ as follows: For $y \in X$ we let $y \in F(A)$ if and only if there is a rule $H \Rightarrow y$ in $\mathcal{R}$ with $H \subseteq A$. We say that F is ***represented by*** $\mathcal{R}$. In this case F is clearly monotone, so by Theorem A.4.9 it has a least and a greatest fixed-point. Since it is often unnecessary to explicitly talk about F, we also write $\mu(\mathcal{R})$ and $\nu(\mathcal{R})$ for these fixed-points.

While the collection of inference rules will usually be infinite, they will be presented in the form of finitely many rule *schemas*, as is usual when presenting axioms in (first-order) logic.

By Corollary 1.2.1 we also get a principle of induction for F. However, it is useful to restate induction in terms of the inference rules, since we usually have explicit rules in mind when defining F. If $\mathcal{R}$ is a collection of inference rules on X, then we say that a subset $K \subseteq X$ is ***$\mathcal{R}$-closed*** if, given any rule $A \Rightarrow y$ in $\mathcal{R}$, $A \subseteq K$ implies that $y \in K$.

1.2.2 • Lemma. *If F is represented by a collection $\mathcal{R}$ of inference rules, then a subset $A \subseteq X$ is F-closed if and only if it is $\mathcal{R}$-closed.*

Proof. First assume that A is F-closed so that $F(A) \subseteq A$, and consider a rule $H \Rightarrow y$ from $\mathcal{R}$ with $H \subseteq A$. Since F is represented by $\mathcal{R}$, this implies that $y \in F(A) \subseteq A$, so A is $\mathcal{R}$-closed.

If A is $\mathcal{R}$-closed, then let $y \in F(A)$. Then there is a rule $H \Rightarrow y$ in $\mathcal{R}$ with $H \subseteq A$. But since A is $\mathcal{R}$-closed, this implies that $y \in A$, and so $F(A) \subseteq A$. ■

1.2.3 • Theorem: *Principle of rule induction.*
If $\mathcal{R}$ is a set of inference rules on X and $A \subseteq X$, then $\mu(\mathcal{R}) \subseteq A$ if and only if the following condition holds: For every inference rule $H \Rightarrow y$ in $\mathcal{R}$, $H \subseteq A$ implies $y \in A$.

Proof. Let F be the function represented by $\mathcal{R}$. The condition says precisely that A is $\mathcal{R}$-closed, which by Lemma 1.2.2 is equivalent to A being F-closed. The claim thus follows from Corollary 1.2.1. ■

Notice that the principle of induction follows since fixed-points are closed. But fixed-points are also *consistent*, which leads to the following important result:

1.2.4 • Theorem: *Inversion.*
If $\mathcal{R}$ is a collection of inference rules and $y \in \mu(\mathcal{R})$, then there is a rule $H \Rightarrow y$ in $\mathcal{R}$ such that $H \subseteq \mu(\mathcal{R})$.

Proof. Let F be the function represented by $\mathcal{R}$. Then we in particular have $y \in \mu(F) \subseteq F(\mu(F))$, which by definition of F implies that there is an inference rule $H \Rightarrow y$ with $H \subseteq \mu(F)$, as desired. ■

This is particularly useful if there to any $y \in \mu(\mathcal{R})$ is a *unique* inference rule with y as its conclusion, but sometimes this is not even necessary, as long as we have the right knowledge about which inference rules have y as their conclusion. We will see an application of this in Lemma 2.4.3.

These kinds of results are sometimes proved by rule induction, for instance by Pitts (2013, §7.2) or Harper (2016, Lemma 4.2), and this approach is indeed more elementary, circumventing Theorem A.4.9 or similar results. But notice that the proof above does not have anything to do with induction: Indeed, as mentioned above induction concerns properties of *closed* sets, while inversion follows from the *consistency* of fixed-points. In fact, the proof of Theorem 1.2.4 goes through if $\mu(\mathcal{R})$ is replaced with any F-consistent set.

Inversion is also sometimes justified by appealing to a notion of 'derivation' of elements of $\mu(\mathcal{R})$, by arguing that y is obtained by applying a series of inference rules to the set of axioms. It is then claimed that there is a *last* such application, which must have the form $H \Mapsto y$, and since this application is valid it follows that $H \subseteq \mu(\mathcal{R})$. If all rules in $\mathcal{R}$ are finitary, then it is indeed possible, and in fact very simple, to set up a deductive calculus that does precisely this. This is the approach taken in Hindley (1997, e.g. Remark 2A8.6), using a version of natural deduction.

1.2.3. Recursive functions

As is usual, the above principle of proof by induction allows us to define functions by recursion.

(a) *Total functions.* We begin by proving that we can define total functions recursively. In contrast to recursion on $\mathbb{N}$, it may be possible to obtain an element of the relation $\mu(\mathcal{R})$ by applying multiple different rules, in which case it is not obvious that a purported recursive function is well-defined. We only consider the case where each element in $\mu(\mathcal{R})$ is the conclusion of a unique rule, since this is all we will need in the sequel.

The statement of Theorem 1.2.5 is based on Moschovakis (2006, Corollary 5.12), while the proof is inspired by Davey and Priestley (2002, §8.18) and Moschovakis (2006, §6.30).

1.2.5 • Theorem: *Recursion.*
Let $\mathcal{R}$ be a set of inference rules such that each $y \in \mu(\mathcal{R})$ is the conclusion of a unique rule, and let Z be any set. For each $R \in \mathcal{R}$ let $h_R \colon \mu(\mathcal{R}) \times Z^{\mathrm{i}(R)} \to Z$ be a function. Then there is a unique function $f \colon \mu(\mathcal{R}) \to Z$ such that if $R \colon (x_i)_{i \in \mathrm{i}(R)} \Mapsto y$, then

$$f(y) = h_R\big(y, (f(x_i))_{i \in \mathrm{i}(R)}\big). \tag{1.1}$$

Proof. Define a function $G \colon (\mu(\mathcal{R}) \rightharpoonup Z) \to (\mu(\mathcal{R}) \rightharpoonup Z)$ as follows: For $f \colon \mu(\mathcal{R}) \rightharpoonup Z$ let $\operatorname{dom} G(f)$ be the set of elements $y \in \mu(\mathcal{R})$ such that if $R \colon (x_i)_{i \in \mathrm{i}(R)} \Mapsto y$ is the unique rule with conclusion y, then $(x_i)_{i \in \mathrm{i}(R)} \subseteq \operatorname{dom} f$. For $y \in \operatorname{dom} G(f)$ we then let

$$G(f)(y) = h_R\big(y, (f(x_i))_{i \in \mathrm{i}(R)}\big).$$

Clearly G is monotone, so Theorem A.4.13 implies that G has a least fixed-point f^*.

We prove that f^* is indeed a total function, i.e., that $\mu(\mathcal{R}) \subseteq \operatorname{dom} f^*$. By Theorem 1.2.3 it suffices to show that if $R\colon (x_i)_{i \in \mathrm{i}(R)} \Mapsto y$ is a rule with $(x_i) \subseteq \operatorname{dom} f^*$, then also $y \in \operatorname{dom} f^*$. But notice that $\operatorname{dom} f^* = \operatorname{dom} G(f^*)$, so y lies in $\operatorname{dom} f^*$ just when $(x_i) \subseteq \operatorname{dom} f^*$. Hence f^* is total. It follows that

$$f^*(y) = G(f^*)(y) = h_R\big(y, (f^*(x_i))_{i \in \mathrm{i}(R)}\big)$$

for all rules $R\colon (x_i)_{i \in \mathrm{i}(R)} \Mapsto y$.

Finally, to prove that f^* is unique simply note that any partial function f that satisfies (1.1) is a fixed-point of G, so $f^* \leq f$ by minimality of f^*. But since f^* is total, this implies that $f^* = f$. ■

(b) *Finitary rules.* If every rule in $\mathcal{R}$ is finitary, then it turns out that we can avoid an appeal to Theorem A.4.13, instead using Theorem A.4.12. Of course G is monotone, and we claim that it is compact (cf. Definition A.3.2). Assume that $f \in \operatorname{dom} G$ and $y \in \operatorname{dom} G(f)$. If $\vec{x} \Mapsto y$ is the unique rule with y as conclusion, let $f_0 := f|_{\vec{x}}$. Then f_0 is finite and $y \in \operatorname{dom} G(f_0)$ with

$$\begin{aligned} G(f_0)(y) &= h_R\big(y, f_0(x_1), \dots, f_0(x_n)\big) \\ &= h_R\big(y, f(x_1), \dots, f(x_n)\big) \\ &= G(f)(y), \end{aligned}$$

so G is compact. Hence it is continuous by Proposition A.3.3, so Theorem A.4.12 implies that G has a least fixed-point f^*. The rest of the proof is identical to the above.

(c) *Partial functions.* We also sometimes wish to define *partial* functions recursively. This is no issue due to the correspondence between partial and total functions described in §A.3.1: To define a partial function $f\colon X \rightharpoonup Y$, first define a total function $f_*\colon X \to Y_*$ by recursion, letting $f_*(x) = *$ if x is supposed to lie outside the domain of f. Then restrict f_* to the set $\{x \in X \mid f_*(x) \neq *\}$.

1.3 ◇ Abstract syntax

In this report we have no interest in the concrete syntax of languages; instead we seek an abstract representation of the structure of programs. We describe one formulation of abstract syntax trees—inspired by Harper (2016)—but we do not go into painstaking detail in either the definition of such trees, nor in the proofs of their properties, for a few reasons:

- The definition of abstract syntax trees that support bindings (what Harper calls 'abstract binding trees') is fairly involved, and our naïve understanding of abstract syntax is accurate enough not to cause issues.

- What is important is not the precise formulation of any kind of abstract syntax, but rather its existence and basic properties.
- An abstract syntax that is readable to humans is difficult to formalise in a proof assistant, so if that is our ultimate goal then we will have to revisit these issues anyway.

While our abstract syntax is of course heavily inspired by the λ-calculus, note that the λ-calculus is usually defined[2] by a *concrete* syntax (i.e., as a formal language).

[2] For instance by Barendregt (1984).

1.3.1. Abstract syntax trees

An ***abstract syntax tree*** (or simply ***AST***) is a rooted tree whose leaves are variables, and whose interior nodes are operators. Operators simultaneously play the roles of functions and of binding constructs, analogous to function symbols and quantifiers in formal logic, respectively. Therefore, each operator not only has an arity (which may be zero), for each argument it also specifies which variables it binds inside this argument.

We may use inference rules to define the set of ASTs. Fix some infinite set $\mathcal{V}$ of variables and $\mathcal{O}$ of operators which have been assigned arities. For each $x \in \mathcal{V}$ we have the axiom

Since programs are finite, $\mathcal{V}$ need not be more than countably infinite, but for our purposes we do not need to assume that it is countable.

$$\frac{}{x} \tag{1.2}$$

saying that x itself is an AST. Next, say that the operator $\varphi \in \mathcal{O}$ takes n arguments, and in the ith argument it binds k_i variables. If $a_1, \dots, a_n$ are ASTs and $\vec{x}_i \in \mathcal{V}^{k_i}$, then we have the rule

For instance, we might have an AST on the form $\mathsf{let}(a; x.b)$, which has the desired interpretation of letting $x := a$ in b. Since there are no variables bound in the first argument, we have omitted the period.

$$\frac{a_1 \quad \cdots \quad a_n}{\varphi(\vec{x}_1.a_1; \dots; \vec{x}_n.a_n)} \tag{1.3}$$

This is supposed to denote the application of the operator φ to the ASTs $a_1, \dots, a_n$, in which the variables $\vec{x}_1, \dots, \vec{x}_n$ have been bound, yielding a new AST. It should be clear that any AST is the conclusion of a unique rule; on the other hand, many rules with the same hypotheses have different conclusions.

Of course, for this to be well-defined we need some 'ground set' from which we draw the ASTs, i.e., which plays the role of the set X in §1.2.2. We do not care precisely what X looks like, but if we think of expressions such as $\varphi(\vec{x}_1.a_1; \dots; \vec{x}_n.a_n)$ as being a sort of nested tuple, then we can take X to be the set of all finitely nested tuples whose elements are operators or variables.

We do not wish to think of expressions such as $\varphi(\vec{x}_1.a_1; \dots; \vec{x}_n.a_n)$ as denoting *formal* expressions, as this would defeat the purpose of defining an *abstract* syntax, requiring us to concern ourselves with issues of parsing, unique readability, etc.

1.3.1 • Definition: ***Abstract syntax trees.***
The set of ***abstract syntax trees*** with variables $\mathcal{V}$ and operators $\mathcal{O}$ is the set generated by the inference rules (1.2) and (1.3) and is denoted $\mathcal{A}[\mathcal{V}, \mathcal{O}]$. ▲

(a) ***Free and bound variables.*** Having defined the set of ASTs recursively, Theorem 1.2.5 allows us to recursively define functions on ASTs: We first define what is meant by ***free variables***. This will be a function $\mathrm{FV}\colon \mathcal{A}[\mathcal{V},\mathcal{O}] \to \mathcal{P}(\mathcal{V})$ with the properties

$$\mathrm{FV}(x) = \{x\},$$
$$\mathrm{FV}\bigl(\varphi(\vec{x}_1.a_1;\dots;\vec{x}_n.a_n)\bigr) = \bigcup_{i=1}^{n} \bigl(\mathrm{FV}(a_i) \setminus \vec{x}_i\bigr).$$

The function FV is usually 'defined' simply by writing down the above two equations, but to properly justify this definition we must go through Theorem 1.2.5. We illustrate the application of this theorem once, leaving it implicit in later recursive definitions.

According to the theorem, we must specify for each inference rule R a function $h_R\colon \mathcal{A}[\mathcal{V},\mathcal{O}] \times \mathcal{P}(\mathcal{V})^{\mathrm{i}(R)} \to \mathcal{P}(\mathcal{V})$. If R is a rule as in (1.2), then $\mathrm{i}(R) = \emptyset$ and we let $h_R(a) = \{a\}$. If instead R is as in (1.3), then we let

If the first argument to h_R is instead a variable, then we may let the value of h_R be an arbitrary element of $\mathcal{P}(\mathcal{V})$.

$$h_R\bigl(\varphi(\vec{x}_1.a_1;\dots;\vec{x}_n.a_n), A_1,\dots,A_n\bigr) = \bigcup_{i=1}^{n} (A_i \setminus \vec{x}_i).$$

It is then easy to check that Theorem 1.2.5 yields a function $f = \mathrm{FV}$ with the desired properties.

We similarly define the set of ***bound variables*** in an AST by

$$\mathrm{BV}(x) = \emptyset,$$
$$\mathrm{BV}\bigl(\varphi(\vec{x}_1.a_1;\dots;\vec{x}_n.a_n)\bigr) = \bigcup_{i=1}^{n} \bigl(\vec{x}_i \cup \mathrm{BV}(a_i)\bigr).$$

The set of all variables occurring in an AST a is then $\mathrm{V}(a) := \mathrm{FV}(a) \cup \mathrm{BV}(a)$. If $\mathrm{FV}(a) = \emptyset$, then we say that a is ***closed*** and otherwise that it is ***open***.

(b) ***Sorts.*** We wish to be able to construct ASTs which both contains expressions and types, first of all to obtain a unified way of describing these entities syntactically, as well as to support language features such as type annotations, ascription, and so on. Expressions and types are examples of ***sorts***. We let $\mathcal{S}$ be a set of sorts, and we attribute sorts to variables and operators in the following way:

To build the function type $\tau_1 \to \tau_2$ as an AST we might write $\mathsf{func}(\tau_1;\tau_2)$. An expression of this type could be the explicitly typed lambda abstraction $\mathsf{lam}(\tau_1;x.e)$, which takes as arguments an AST τ_1 of type sort, and an AST e of expression sort in which the parameter x is bound.

- Each variable is assigned a sort. The set of variables with sort s is denoted $\mathcal{V}_s$, and we assume that there are infinitely many variables of each sort.
- Each operator φ of arity n, which binds k_i variables in its ith argument, is assigned $k_1 + \dots + k_n + n + 1$ sorts: One for each bound variable, one for each of the n arguments, and one as its ***return sort***. We denote this collection of sorts by $(\vec{s}_1.t_1;\dots;\vec{s}_n.t_n) \to s$ to match the notation for the application of φ to a series of ASTs. Call this collection of sorts the ***arity*** of φ.

We leave open the question of whether variables and operators can be assigned multiple sorts and arities at the same time, and return to this in §2.1.3. Our languages will only have two sorts, expressions and types, but we could also use sorts for various other purposes, for instance to distinguish between pure and impure parts of the language.[3]

[3]For an example of this, see Harper (2016, Chapter 34).

We assign sorts to ASTs by defining a binary relation $a : s$ with $a \in \mathcal{A}[\mathcal{V},\mathcal{O}]$ and $s \in \mathcal{S}$ as follows: For $x \in \mathcal{V}_s$ we have the rule

$$\frac{}{x : s}$$

and for an operator $\varphi \in \mathcal{O}$ with arity $(\vec{s}_1.t_1;\dots;\vec{s}_n.t_n) \to s$ we have the rule

$$\frac{a_1 : t_1 \qquad \cdots \qquad a_n : t_n}{\varphi(\vec{x}_1.a_1;\dots;\vec{x}_n.a_n) : s}$$

where each $\vec{x}_i \in \mathcal{V}^*$ is a sequence of variables with sorts $\vec{s}_i$. Again we need some ground set to define this relation on, but this is simply $\mathcal{A}[\mathcal{V},\mathcal{O}] \times \mathcal{S}$.

If s is a sort, then we write $\mathrm{FV}_s(a)$ for the set of free variables in a of sort s.

1.3.2. α-equivalence and substitution

Properly defining substitution is rather technical and would take us too far afield, so we give an overview of the problems one has to consider to make it precise.

(a) ***α-equivalence.*** Inspired by the λ-calculus, we say that two ASTs $a, b \in \mathcal{A}[\mathcal{V},\mathcal{O}]$ are ***α-equivalent***, written $a =_\alpha b$, if we may obtain b from a by renaming bound variables. Some care must be taken in that not all renamings are allowed[4], but we won't go into detail. It is clear that $=_\alpha$ is an equivalence relation. Furthermore, one can show that renaming respects sorts in the sense that if $a : s$ and x and y are variables of the same sort, then renaming x to y in a yields an AST of sort s.

[4]For instance, if an operator φ binds two variables x and y in the same argument, say $\varphi(xy.a)$, we clearly do not allow x to be renamed y, or vice versa. If a already contains another free variable z, then we also do not allow x to be renamed z.

It is standard to use the so-called ***Barendregt convention***[5]: This says that whenever ASTs $a_1,\dots,a_n$ occur in the same context, we choose the bound variables in each a_i such that the sets $\mathrm{FV}(a_1) \cup \cdots \cup \mathrm{FV}(a_n)$ and $\mathrm{BV}(a_1) \cup \cdots \cup \mathrm{BV}(a_n)$ are disjoint. In our setting we must also choose the new bound variables to have the same sorts as the original ones. This is clearly possible since we have infinitely many variables of each sort, and we need only perform finitely many renamings.

[5]See Barendregt (1984, 2.1.13), though he of course did not name it after himself, instead calling it the 'variable convention'.

There are also other ways of dealing with bound variables, most notably ***de Bruijn indices*** (see for instance Barendregt 1984, Appendix C). These are less human-friendly but are more amenable to formalisation.

(b) ***Substitution.*** Defining substitution is tricky, not least because of its relationship with α-equivalence. We take the following approach: If a and b are ASTs, then we roughly speaking define $a[b/x]$ to be the

AST[6] obtained by replacing every occurrence of x in a with b, if this is allowed. Some such substitutions will be undefined[7], but one can show that it is always possible to find an AST a' with the same sort as a, such that $a =_\alpha a'$ and $a'[b/x]$ is defined.

It then turns out that substitution preserves α-equivalence, in the sense that if $a =_\alpha a'$ then $a[b/x] =_\alpha a'[b/x]$, as long as the substitutions are defined. Hence we may define substitution on α-equivalence classes by

$$[a]_\alpha[b/x] := \big[a[b/x]\big]_\alpha,$$

where $[a]_\alpha$ is the α-equivalence class of a, and where we choose the representative a such that $a[b/x]$ is defined.

Furthermore, it is of course possible to follow the Barendregt convention while choosing α-equivalent ASTs to perform substitution.

[6]The precise definition of substitution makes it clear that $a[b/x]$ is in fact an AST.

[7]As an example, if $x \neq y$, x is free in a and y is free in b, then the substitution $\varphi(y.a)[b/x]$ is undefined, since the free occurrences of y in b would be 'captured' by the binding of y.

1.4 ⋄ Reduction

1.4.1. Abstract reduction systems

An ***abstract reduction system*** is a pair $(X, \to)$, where X is a set and $\to$ is a binary relation on X called a ***reduction***. In the present context, X will be a set of expressions in a language, and $\to$ will describe how expressions may reduce to other expressions. For instance, the expression $(\lambda x.xy)(\lambda z.z)$ in the untyped λ-calculus reduces to the expression $(\lambda z.z)y$, which we thus write

$$(\lambda x.xy)(\lambda z.z) \to (\lambda z.z)y.$$

The latter expression itself reduces to y, so in total we write

$$(\lambda x.xy)(\lambda z.z) \to^2 y \quad \text{or} \quad (\lambda x.xy)(\lambda z.z) \to^* y,$$

to indicate that the former expression reduces to y in 2 or in any number of steps (including 0), respectively.

An element $x \in X$ is ***reducible*** if there is a $y \in X$ such that $x \to y$, and ***irreducible*** otherwise. An irreducible element is also called a ***normal form***, and if y is irreducible and $x \to^* y$, then y is also said to be *a* normal form of x, and we conversely say that x *has* the (not necessarily unique) normal form y.

We say that x is ***weakly normalising*** or simply ***normalising*** if $x \to^* y$ for some irreducible y; i.e., if x has a normal form. Furthermore, x is ***strongly normalising*** or ***terminating*** if there is no infinite chain $x \to x_1 \to x_2 \to \cdots$ of reductions. We call the recuction $\to$ strongly or weakly normalising if every element of X is. The untyped λ-calculus is clearly not normalising, as witnessed by the divergent combinator $(\lambda x.xx)(\lambda x.xx)$, but it is an important result that the simply typed λ-calculus is strongly normalising (cf. Pierce 2002, Theorem 12.1.6). We will not be concerned with normalisation as such, though the fact that

An abstract reduction system is also called an abstract *rewriting* system to underline the fact that the reduction $\to$ may not in fact describe a 'reduction' in the usual sense of the word.

Recall that a (binary) relation between sets X and Y is simply a subset $R \subseteq X \times Y$. If $(x,y) \in R$, then we also write xRy. If $S \subseteq Y \times Z$ is another relation, recall the composition $S \circ R \subseteq X \times Z$: Writing $R \cdot S := S \circ R$ following Terese (2003, Definition A.1.1(iii)), this is defined so that $x(R \cdot S)z$ if and only if xRy and ySz for some $y \in Y$. If $X = Y$, then we write R^n for the n-fold composition $R \circ \cdots \circ R$ and R^* for the reflexive and transitive closure $\bigcup_{n \in \mathbb{N}} R^n$, where R^0 by definition is equality. The reflexive and transitive closure of a relation $\to$ is also often denoted $\twoheadrightarrow$. Finally, the inverse R^{-1} of the relation R is defined by the property that $xR^{-1}y$ if and only if yRx.

the simply typed λ-calculus is strongly normalising will influence certain definitions in Chapters 4 and 5.

A binary relation $\to$ on X is said to be ***deterministic*** if whenever $x \to y_1$ and $x \to y_2$ we have $y_1 = y_2$. Furthermore, $\to$ is said to have the ***diamond property*** if whenever $x \to y_1$ and $x \to y_2$, there is a $z \in X$ such that $y_1 \to z$ and $y_2 \to z$, cf. Figure 1.1. Finally $\to$ is called ***Church–Rosser*** if the reflexive and transitive closure $\to^*$ has the diamond property. Clearly deterministic relations are Church–Rosser[8].

[8]The converse does not hold. For instance, it is a classic theorem, known as the Church–Rosser theorem (cf. Barendregt 1984, Theorem 3.2.8), that the untyped λ-calculus with so-called full β-reduction is Church–Rosser, but this is not deterministic.

$$\begin{tikzcd}
x \arrow[r] \arrow[d] & y_2 \arrow[d, dashed, "\exists"] \\
y_1 \arrow[r, dashed, "\exists"'] & z
\end{tikzcd}$$

Figure 1.1. The diamond property.

1.4.1 • Lemma. *If the relation $\to$ on X is Church–Rosser, then any element of X has at most one normal form. If $\to$ is also weakly normalising, then every element has a unique normal form.*

Proof. Let $x \in X$, and let y_1 and y_2 be normal forms of x. Since $\to^*$ has the diamond property there is a $z \in X$ such that $y_1 \to^* z$ and $y^*2 \to^* z$. But since y_1 and y_2 are irreducible, we must then have $y_1 = z = y_2$. The second claim is obvious. ∎

1.4.2. Reduction in ASTs

Abstract reduction systems are very general. In the context of programming languages (or formal systems such as the λ-calculus), reductions are induced by the structure of the language in the following way.

Given variables $\mathcal{V}$ and operators $\mathcal{O}$, a ***context*** is an AST from $\mathcal{A}[\mathcal{V},\mathcal{O}]$ in which one sub-AST has been replaced by a 'hole', denoted '$-$'. More precisely, the set of contexts is defined recursively by the rules

$$\frac{}{-} \qquad\qquad \frac{C}{\varphi(\vec{x}_1.a_1;\dots;\vec{x}_{i-1}.a_{i-1};\vec{x}_i.C;\vec{x}_{i+1}.a_{i+1};\dots;\vec{x}_n.a_n)}$$

using the same notation for variables and ASTs as in the definition of $\mathcal{A}[\mathcal{V},\mathcal{O}]$. If C is a context and a is an AST, then we write $C[a]$ for the AST obtained by replacing the hole in C by a. We leave it to the reader to provide a recursive definition of the function $C \mapsto C[a]$. Contrary to substitutions which are capture-avoiding, replacement of the hole in a context is supposed to be capturing. Hence we do *not* identify α-equivalent contexts, nor the AST replacing the hole therein.

Any context C gives rise to a map $\mathcal{A}[\mathcal{V},\mathcal{O}] \to \mathcal{A}[\mathcal{V},\mathcal{O}]$ given by $a \mapsto C[a]$. This in turn induces a composition on contexts such that $C' \circ C$ maps a to $C'[C[a]]$.

Let $\mathcal{C}$ be a collection of contexts. A binary relation R on $\mathcal{A}[\mathcal{V},\mathcal{O}]$ is ***$\mathcal{C}$-compatible*** if $(a,a') \in R$ implies $(C[a],C[a']) \in R$ for all $a,a' \in \mathcal{A}[\mathcal{V},\mathcal{O}]$ and all $C \in \mathcal{C}$. The ***$\mathcal{C}$-compatible closure*** of R is the smallest $\mathcal{C}$-compatible relation extending R. The set $\mathcal{C}$ will usually be clear from context, and in this case we just talk of 'compatibility'.

In the classical theory of the λ-calculus we usually only consider the case where $\mathcal{C}$ is the set of all contexts. In the study of programming languages we will need to consider different classes of contexts, see §2.1.3(c) and §5.2.3.

A ***notion of reduction*** is simply a binary relation R on $\mathcal{A}[\mathcal{V},\mathcal{O}]$. This induces various other relations on $\mathcal{A}[\mathcal{V},\mathcal{O}]$:

- The compatible closure of R is denoted $\to_R$ and is called the ***one-step R-reduction***.
- The reflexive and transitive closure of $\to_R$ is denoted $\twoheadrightarrow_R$ and is simply called the ***R-reduction***.
- The equivalence relation generated by $\twoheadrightarrow_R$ is denoted $=_R$ and is called ***R-convertibility*** or ***R-equivalence***.

Notice that any of these relations give rise to an abstract reduction system on $\mathcal{A}[\mathcal{V},\mathcal{O}]$.

In this context, an ***R-redex*** is an AST a such that $(a,b) \in R$ for some AST b. In this case b is called an ***R-contractum*** of a. An AST a is called an ***R-normal form*** if it is irreducible with respect to $\to_R$. If $a \twoheadrightarrow_R b$ where b is an R-normal form, then we also say that b is an R-normal form *of* a. We finally say that a notion of reduction R is Church–Rosser if the one-step R-reduction $\to_R$ is.

1.5 ◇ Partitions and coinduction

1.5.1. Partitions

Recall that a ***partition*** of a set X is a collection $\mathcal{P}$ of pairwise disjoint subsets of X such that $X = \bigcup \mathcal{P}$. Every partition induces an equivalence relation $\sim_{\mathcal{P}}$ on X such that $x \sim_{\mathcal{P}} y$ if and only if x and y belong to the same set in $\mathcal{P}$. Conversely, every equivalence relation $\sim$ on X induces a partition $\mathcal{P}_\sim$ whose sets are the $\sim$-equivalence classes.

If $\sim$ and $\approx$ are equivalence relations, then we say that $\sim$ is ***finer*** than $\approx$ (and that $\approx$ is ***coarser*** than $\sim$) if $\mathcal{P}_\sim$ is finer than $\mathcal{P}_\approx$, i.e., if for every $A \in \mathcal{P}_\sim$ there is a $B \in \mathcal{P}_\approx$ with $A \subseteq B$. This equivalent to the property that $x \sim y$ implies $x \approx y$ for all $x,y \in X$. On the other hand, this is equivalent to the inclusion $\sim \subseteq \approx$, so coarseness and size are synonyms: One equivalence relation is coarser than another if and only if it is larger (as a set).

Note that depending on which kinds of families of sets one is considering, different definitions of fineness or coarseness are appropriate. For instance, one topology is finer than another if the latter is a *subset* of the former.

1.5.2. Coinduction

Now notice that Theorem A.4.9 (the Knaster–Tarski fixed-point theorem) has the following immediate consequence:

1.5.1 • Corollary: ***Principle of coinduction.***
Let L be a complete lattice and let $f : L \to L$ be monotone. If $y \in P$ is f-consistent, then $y \leq \nu(f)$. ■

In the important special case where L is a power set $\mathcal{P}(X)$ and $F : \mathcal{P}(X) \to \mathcal{P}(X)$ is monotone, recall that the principle of induction (cf. Corollary 1.2.1) allows us to prove that the set $\mu(F)$ has a certain property P, if only we can show that the characteristic set of P is F-closed. On the other hand, the above principle of *coinduction* says that to prove that some element $x \in X$ belongs to $\nu(F)$, it suffices to find an F-consistent set containing x.

Returning to relations, while these are not always defined using a generating function, we can sometimes show that some relation of interest R is the largest or coarsest with some property P. To show that xRy, it thus suffices to find another relation S with property P such that xSy. We will take this approach in Chapter 5.

Language Fundamentals 2

In this chapter we define the languages we will study for the remainder of this report. We begin with the language $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$, defining its syntax, type system and operational semantics. Then we briefly consider a fragment of this language, which we simply call $\mathbf{F}$, that we will return to in Chapters 4 and 5. Next we consider the type system in more detail and describe how to derive new types, as well as how a type system can aid in programming tasks. Finally we prove various properties of the languages' type systems and operational semantics.

2.1 ⋄ The language $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$

The language $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ is named after Girard's System $\mathbf{F}$, which is the simply typed λ-calculus extended with universal types. In addition, our language has existential types '$\exists$', recursive functions and types 'μ', as well as references 'ref'. The language also has products and sums, but we do not display these in the name of the language to avoid clutter.

2.1.1. Syntax

We begin by defining the syntax of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ and describe how the usual representation of such syntax can be understood as defining abstract syntax trees.

As mentioned, $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ have ASTs of two sorts: expressions and types, denoted *Exp* and *Type* respectively. The set of *closed* expressions (i.e., expressions with no free variables) is denoted *ClExp*. Furthermore, we divide the set of variables into two infinite sets, one set *Var* which contains variables of sort *Exp*, and another set *TypeVar* containing variables of sort *Type*. Among the operators we have a countably infinite collection *Loc* of nullary operators, which we think of as locations in memory.

We denote by *Exp* both the expression sort and the set of ASTs of this sort, and similarly for *Type*.

The full specification of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ is found in §B.1. For now let us consider only a subset in order to understand how to read this specification:

$$\begin{array}{lrcl} & x, f & \in & \mathit{Var} \\ & l & \in & \mathit{Loc} \\ & \alpha & \in & \mathit{TypeVar} \\ \mathit{Exp} & e & ::= & 1 \mid x \mid \langle e, e\rangle \mid \mathsf{rec}\, f(x) := e \mid \cdots \\ \mathit{Type} & \tau & ::= & 1 \mid \alpha \mid \tau \times \tau \mid \tau \to \tau \mid \cdots \end{array}$$

The first three lines only serve to introduce notation. The next line defines the operators whose return sort is *Exp*, with the letter e serving the same role as nonterminal symbols do in formal grammars: For instance, the 'production' $e ::= \langle e, e\rangle$ says that there is an operator $\langle -, - \rangle$ with arity $(\mathit{Exp}; \mathit{Exp}) \to \mathit{Exp}$. The production $e ::= x$ simply says that elements of *Var* are expressions[1], and $e ::= l$ says that locations have return sort *Exp*. Finally, in the production $e ::= \mathsf{rec}\, f(x) := e$ it is implicit that the variables f and x are bound by the operator in the expression e.

[1] Strictly speaking this production is redundant, and so is the production $\tau ::= \alpha$, but we include them to make the specification self-contained.

Similarly, the next line says, among other things, that there is a nullary operator $\mathtt{1}$ of sort *Type*, and that there is an operator $- \times -$ of arity $(\mathit{Type}; \mathit{Type}) \to \mathit{Type}$.

Notice that operators such as rec and Ref are written in a sans-serif font, and that the first letter is capitalised when it is an operator whose return sort is *Type*.

2.1.2. Static semantics

(a) *Type contexts.* Consider some expression e of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$. If e contains free variables, then in order to decide the type of e we must, at least in the general case, specify types for those variables. We capture this using a ***type context*** which is a finite partial map $\Gamma\colon \mathit{Var} \rightharpoonup_\omega \mathit{Type}$. If $\Gamma(x) = \tau$ then we also write $(x : \tau) \in \Gamma$ or just $x : \tau$ when Γ is understood. In this way, $x : \tau$ also becomes the *assertion* that x has type τ.

It is common to define a type context as a finite *list* of pairs $x : \tau$. Nothing of significance hangs on this for us, but not having to worry about ordering simplifies some proofs.

If Δ is another type context such that $\operatorname{dom}\Gamma \cap \operatorname{dom}\Delta = \emptyset$, then we denote by Γ, Δ the type context given by

$$(\Gamma,\Delta)(x) = \begin{cases} \Gamma(x), & x \in \operatorname{dom}\Gamma, \\ \Delta(x), & x \in \operatorname{dom}\Delta. \end{cases}$$

That is, Γ, Δ is the join $\Gamma \vee \Delta$ of the two type contexts in the poset $(\mathit{Var} \rightharpoonup \mathit{Type})$.

Furthermore, if $\operatorname{dom}\Delta = \{x_1, \dots, x_n\}$ and $\Delta(x_i) = \tau_i$ for distinct x_i, then we write $\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n$.

We extend the definition of substitution to type contexts: If α is a type variable and τ a type, then we define

$$\Gamma[\tau/\alpha](x) := \Gamma(x)[\tau/\alpha]$$

for all $x \in \operatorname{dom}\Gamma$, taking care to rename bound variables as needed. Notice that since $\operatorname{dom}\Gamma$ is finite, only finitely many such renamings are necessary.

Similarly, we extend the definition of free type variables to type contexts by letting

$$\mathrm{FV}_{\mathit{Type}}(\Gamma) := \bigcup_{x \in \operatorname{dom}\Gamma} \mathrm{FV}_{\mathit{Type}}(\Gamma(x)).$$

(b) *Store typings.* Similarly, the expression e may contain references to locations in memory. A ***store typing*** is a finite partial map $\Sigma\colon \mathit{Loc} \rightharpoonup_\omega \mathit{Type}$ that assigns types to locations. We use the same notation for extending store typings as we did above for type contexts, and substitution and $\mathrm{FV}_{\mathit{Type}}(\Sigma)$ is defined analogously.

(c) *Free type variables.* Notice that using type contexts we can simultaneously keep track of which variables are (potentially) free in a given expression. In order to get something similar for free *type* variables we may collect the free type variables in a (finite) set Ξ and also keep track of this set. If Φ is another such set that is disjoint from Ξ, then we write Ξ,Φ for the union $\Xi \cup \Phi$. If $\Phi = \{\alpha_1,\ldots,\alpha_n\}$ for distinct α_i, then we also write $\Xi,\alpha_1,\ldots,\alpha_n$.

Of course, Ξ,Φ is the join of Ξ and Φ in $\mathcal{P}_\omega(\mathit{TypeVar})$.

We say that a type τ is ***well-formed*** with respect to Ξ if all free type variables in τ lie in Ξ, in which case we write $\Xi \vdash \tau$. If Γ is a type context, then we similarly say that Γ is well-formed with respect to Ξ if $\Xi \vdash \tau$ for all $\tau \in \operatorname{ran}\Gamma$, and we also write $\Xi \vdash \Gamma$. Well-formedness of store typings with respect to Ξ is defined and denoted analogously.

(d) ***Syntactic typing.*** We are now in a position to define the type system for $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$. This is formalised as a 5-ary relation on the set

$$\mathcal{P}_\omega(\mathit{TypeVar}) \times (\mathit{Var} \rightharpoonup_\omega \mathit{Type}) \times (\mathit{Loc} \rightharpoonup_\omega \mathit{Type}) \times \mathit{Exp} \times \mathit{Type}$$

called the ***syntactic typing relation***. Instead of $(\Xi,\Gamma,\Sigma,e,\tau)$, elements of this relation are denoted

$$\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$$

The order in which Ξ, Γ and Σ occur is not massively significant. We have chosen this order since it seems the most convenient when we omit one or more of them as noted below.

and are called ***typing judgments***. We also use this notation to assert that this tuple lies in the typing relation. In this case we say that e is ***well-typed*** with respect to Ξ, Γ and Σ, with type τ. If $\Xi = \emptyset$, then we simply denote the above element by $\Gamma \mid \Sigma \vdash e : \tau$, and we furthermore write $\Sigma \vdash e : \tau$ if also $\Gamma = \bot$. We finally write $\vdash e : \tau$ if also $\Sigma = \bot$, in which case we simply say that e is well-typed.

We define the typing relation using inference rules. As an example, consider the rule

$$\textsc{T-rec}\quad \frac{\Xi \mid \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \mid \Sigma \vdash e : \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{rec}\, f(x) := e : \tau_1 \to \tau_2}.$$

Compare the notation

$$f : \tau_1 \to \tau_2,$$

which says that f is an expression of type $\tau_1 \to \tau_2$, to the notation

$$f\colon \tau_1 \to \tau_2,$$

which says that f is an arrow between objects τ_1 and τ_2 of some category. Notice in particular the spacing around the colon ':'.

This says that given an expression e having (potentially) free variables f and x of appropriate types, we may construct an expression $\mathsf{rec}\, f(x) := e$ of function type, whose parameter type is that of x, and whose return type is the return type of f. Note that the store typing Σ and the set Ξ of free type variables are unchanged when moving from the premise to the conclusion, while the variables f and x are no longer assigned types by the type context in the conclusion.

Consider instead the rule

$$\textsc{T-var}\quad \frac{\Xi \vdash \Gamma \qquad \Xi \vdash \Sigma \qquad \Gamma(x) = \tau}{\Xi \mid \Gamma \mid \Sigma \vdash x : \tau}$$

which assigns types to variables. This rule is not quite on the correct form, since its hypotheses are not elements of the typing relation. The intended interpretation is that given the assumptions $\Xi \vdash \Gamma$, $\Xi \vdash \Sigma$ and $\Gamma(x) = \tau$ we have the rule (in fact the axiom)

$$\frac{}{\Xi \mid \Gamma \mid \Sigma \vdash x : \tau}.$$

2.1.3. Dynamic semantics

The last piece of the definition of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ is its dynamic semantics. We specify an operational (transition) semantics in several steps.

The resulting dynamics consists of performing reductions at three different levels:

- The first level consists of *pure head reductions*. These are reductions that can be performed on expressions that have no subexpressions that can be evaluated, and without reading or modifying the store[2].
- The next level are the *impure* head reductions. Here we again reduce simple expressions, but these may read from or write to the store.
- The final level allows us to perform reductions on complex expressions by reducing subexpressions.

[2]We describe what we mean by 'store' in more detail below.

At each level we define a transition system on machine states using inference rules.

(a) *Pure head reductions.* As mentioned, we do not yet wish to reduce complex expressions, so we need to make precise what we mean by a 'simple' expression. We call an expression a ***value*** if it is to be considered the final result of a computation. We define the set *Val* of values recursively:

$$v ::= 1 \mid l \mid \langle v, v \rangle \mid \cdots$$

We use the letter 'v', often with various decorations, to denote values. The set of *closed* values (i.e., values with no free variables) will be denoted *ClVal*. Furthermore, as the end result of computations, values are naturally supposed to be irreducible, and indeed they turn out to be (cf. Proposition 2.4.9). We denote the set of irreducible expressions by *Irr*.

Furthermore, when performing pure reductions we do not need to take into account memory, so we simply model the machine state by a single expression. The pure head reduction, which we denote by $\to_p$, is thus a binary relation on *Exp*.

As an example of a pure reduction rule, consider the rule

$$\textsc{E-proj}_1\quad \frac{}{\pi_1 \langle v_1, v_2 \rangle \to_p v_1}.$$

Since both v_1 and v_2 are values, these are not supposed to be reduced further, so the expression $\pi_1 \langle v_1, v_2 \rangle$ has no reducible subexpressions.

(b) *Impure head reductions.* Next we must take into account mutable state. We model memory access using a ***store***, which is a finite partial map $\sigma\colon \mathit{Loc} \rightharpoonup_\omega \mathit{Exp}$. Denote by *Sto* the set of stores. We thus model machine states as elements of the product $\mathit{Sto} \times \mathit{Exp}$, and the impure head reduction, denoted $\to_h$, becomes a binary relation on this product.

Some authors use the word ***heap*** instead of store, which is not to be confused with the heap data structure.

All pure head reductions give rise to impure head reductions: We formalise this by adding a rule

$$\textsc{E-pure}\quad \frac{e \to_p e'}{(\sigma, e) \to_h (\sigma, e')}$$

for every store σ. To take an example of a rule that modifies the store, consider

$$\textsc{E-alloc}\quad \frac{l \notin \operatorname{dom} \sigma}{(\sigma, \mathsf{ref}\, v) \to_h (\sigma[l \mapsto v], l)}.$$

This rule allows us to place the value v in the store by wrapping it in a ref expression and evaluating it. Note that this rule is non-deterministic, since the location l is not uniquely specified by the hypothesis: Indeed, since the domain of a store is finite but the set *Loc* of locations is infinite, the value v could be saved at an infinite number of locations. For our purposes we could have modified the rule E-ALLOC to be deterministic, for instance by enumerating the locations and always allocating the location that is smallest with respect to this enumeration. However, in practice memory allocation is (partially) handled by the operating system and is not deterministic[3], so we prefer the the non-deterministic version.

[3]At least from the point of view of the programmer.

(c) *Evaluation contexts.* In order to reduce a complex expression e, we must somehow locate a simple subexpression of e that can be reduced directly. Furthermore, there may be multiple such subexpressions, so we must also decide in which order these should be reduced. We do this by introducing ***evaluation contexts***: Recall that we in §1.4.2 introduced *contexts* which are, roughly speaking, expressions in which one subexpression has been replaced by the hole $-$. A subset of these will serve as evaluation contexts, namely those given by the following recursive definition:

$$E ::= - \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \cdots$$

Again the complete specification can be found in §B.1.

As before, e denotes an expression and v a value. We collect the evaluation contexts in a set *ECtx*.

Recall also that contexts are in general *capturing*, in that the expression e may contain free variables that are captured by bindings in a context C when inserting e into C. But notice that this is not the case for evaluation contexts. This for instance means that the body of a function is not evaluated before the function has been applied to an argument.

This also means that there is another way to define evaluation contexts: We could backtrack and add the hole to our initial specification of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$. In this case the hole cannot be an expression, since this would allow undesirable expressions such as $\langle -,- \rangle$. We could instead let *ECtx* be its own sort, but then for $\langle E,e \rangle$ to be an evaluation context the pairing operator must have three different arities, namely $(Exp;Exp) \to Exp$, $(ECtx;Exp) \to ECtx$ and $(Exp;ECtx) \to ECtx$. If we assign each operator multiple different arities, and if we furthermore let the hole be a *variable*, then one can show that substituting an expression into the hole does indeed yield an expression. Furthermore, one can show that we have $E[e/-] = E[e]$. No matter which definition we use, each E gives rise to a map $Exp \to Exp$ given by $e \mapsto E[e]$.

We can now define the final transition system that formalises the operational semantics of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$. The reduction $\to$ is defined by the single rule schema

$$\textsc{E-head}\quad \frac{(\sigma,e) \to_h (\sigma',e')}{(\sigma,E[e]) \to (\sigma',E[e'])}.$$

Notice that this is *almost* the consistent closure of $\to_h$, except that we also need to keep track of stores.

2.2 ◇ The language fragment F

In addition to the 'full' language $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ we will also have reason to study a fragment that has neither recursive types nor references, which we call **F**. To further simplify we also omit existential types.

Since **F** does not have references, we have no need to keep track of a store. Hence we agree that when studying this language, we model the machine state using only an expression, and we thus omit the store from our notation.

We return to this language in Corollary 2.4.11, and in Chapters 4 and 5 it will be our main focus, but until then all references are to $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ unless otherwise specified.

2.3 ◇ Programming with types

In this section we consider each of the types of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ and discuss in greater or lesser detail what each type can be used for in practice.

2.3.1. Base types

The only ***base type*** in $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ is the unit type $\mathtt{1}$ whose only value is[4] $\mathtt{1}$. This is used for various purposes: to indicate the absense of a value, as a 'sentinel value', or as the value of expressions that are only used for their side effects.

[4]This fact will follow from Lemma 2.4.4(i).

Since the unit type categorically is an empty product, hence a terminal object, we denote the type by $\mathsf{1}$. We could have denoted the unit *value* by $\langle\rangle$ to represent an empty tuple, but we prefer $\mathsf{1}$ for purely aesthetic reasons. No confusion is likely to result from this choice.

Other common base types are booleans, numbers and strings. We construct booleans in §2.3.3 below, and in §2.3.6(b) we will also consider a base type Nat of natural numbers for expository purposes.

Sometimes ***uninterpreted*** base types—i.e., base types that have no constructors or eliminators—are also used. Type variables can be considered to be of this sort[5].

[5]See Pierce (2002, §22.1). Though note that for us variables are by definition different from operators with the same sort.

2.3.2. Products and sums

The product $\tau_1 \times \tau_2$ of types τ_1 and τ_2 should be well-known: It is simply the type of pairs $\langle e_1, e_2\rangle$ of expressions, where e_i has type τ_i.

The sum $\tau_1 + \tau_2$ is perhaps less familiar. Its values are *either* of type τ_1 *or* of type τ_2, and each value of type $\tau_1 + \tau_2$ is decorated to indicate which type it belongs to. Thus the sum is effectively the disjoint union[6] of τ_1 and τ_2. If e_1 and e_2 are expressions of types τ_1 and τ_2, then we denote by $\iota_1\, e_1$ and $\iota_2\, e_2$ their injections into the type $\tau_1 + \tau_2$.

We will see applications of both products and sums below.

[6]Indeed, it is possible to construct a category of types in which products are categorical products and sums are categorical coproducts, i.e., analogous to disjoint unions in the category of sets. For a brief description of such a category see Awodey (2010, §2.5).

2.3.3. Booleans

A boolean type is nothing but a type with two values. Instead of 'hard-coding' booleans into $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$, we may construct such a type using sum types: We can simply use $\mathsf{1}+\mathsf{1}$ whose values are of course $\iota_1\, \mathsf{1}$ and $\iota_2\, \mathsf{1}$. We use Bool as an alias for this sum type, and false and true as aliases for its values, respectively.

To implement an if-expression we define the expression

$$\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2$$

as syntactic sugar for $\mathsf{match}(e, x, e_2, e_1)$, where x is a variable that is not free in either e_1 or e_2. In particular we have, by E-MATCH-INJ$_1$,

$$\begin{aligned}\mathsf{if}\;\mathsf{false}\;\mathsf{then}\; e_1\; \mathsf{else}\; e_2 &= \mathsf{match}(\iota_1\, \mathsf{1}, x, e_2, e_1)\\ &\to_p e_2[\mathsf{1}/x]\\ &= e_2,\end{aligned}$$

and E-MATCH-INJ$_2$ similarly implies that $\mathsf{if}\;\mathsf{true}\;\mathsf{then}\; e_1\; \mathsf{else}\; e_2 \to_p e_1$.

We furthermore obtain the evaluation contexts

$$E ::= \cdots \mid \mathsf{if}\; E\; \mathsf{then}\; e\; \mathsf{else}\; e,$$

showing that we may evaluate the condition but neither of the branches, as expected.

Our definition of false and true require us to change the ordering of the expressions e_1 and e_2 in the match expression below. This slightly inconvenient choice arises from interpreting the type $\mathsf{1}$ as the singleton $\{\mathsf{1}\}$ with the trivial ordering and the sum $\mathsf{1}+\mathsf{1}$ as the *linear* sum of its summands, meaning as the disjoint union equipped with the ordering $\iota_1\, \mathsf{1} < \iota_2\, \mathsf{1}$. Interpreting the smaller of these as false corresponds to interpreting the ordering as logical implication.

This is consistent with the convention that 0 represents false and 1 true given the ordering $0 < 1$.

2.3.4. Options

'Option' is the name given to these types in languages such as ML or Coq, while e.g. Haskell or Agda call them 'maybe' types.

Sum types also allow us to implement option types. Given a type τ we construct a type that can contain either 'nothing' or an expression of type τ. We use the type $\mathsf{1}$ to model the first case, and we collect the types in the sum $\mathsf{1} + \tau$. Let $\mathsf{Opt}\,\tau$ be an alias for this type.

As 'value constructors' we define $\mathsf{none} = \iota_1\,\mathsf{1}$ and $\mathsf{just}\,e = \iota_2\,e$. We can also introduce syntactic sugar for pattern matching on options:

$$\mathsf{which}(e, x, e_1, e_2) = \mathsf{match}(e, x, e_1, e_2).$$

If e is none this reduces to $e_1[\mathsf{1}/x]$, and it reduces to $e_2[e'/x]$ otherwise where $e = \mathsf{just}\,e'$. Notice that this allows x to also be free in e_1, which is probably not the intended behaviour. Notice also that if we implement options and booleans simultaneously, then false and none are aliases for the same value, which is also probably undesirable.

Option types can also be implemented using labeled sums, so-called ***variants*** (cf. Pierce 2002, §11.10), in which option types and Bool can be made disjoint, though a programmer can of course still unintentionally construct values of option type.

2.3.5. An empty type

In fact, universal types are so expressive that we can even use them to implement products and sums, cf. Harper (2016, §16.2).

If no expressions has a type τ, then τ is an empty or ***void*** type. We can implement such a type as $\forall\alpha.\alpha$.

Note that the terminology surrounding empty and unit types is not exactly standardised. For instance, the type `Void` in Java is in fact a unit type, since its only value is `null`. On the other hand, the type `void` in C seems to occupy a role somewhere in between: While no object has type `void`, functions whose return type is `void` can in fact terminate.

2.3.6. Polymorphic types

(a) *Universal types.* Universal types should be familiar to anyone experienced with functional programming, though perhaps in a different guise. We attempt to motivate universal types using a simplified version of a classic example.

This example is based on Pierce (2002, §22.7). Notice that the language studied by Pierce has type annotations and let-expressions; we emulate the latter using lambdas (and products), and the former turns out not to make a difference for our purposes.

Let $\mathsf{id} = \lambda x.x$ be the identity function. The expression

$$\langle \mathsf{id}\,\mathsf{true}, \mathsf{id}\,\mathsf{1} \rangle$$

is then completely unproblematic: Even though the expression id appears in both entries in the pair applied to expressions of different types, the two occurrences of id are simply assigned different types. But a problem arises when we factor id out as follows:

$$(\lambda f.\langle f\,\mathsf{true}, f\,\mathsf{1}\rangle)\,\mathsf{id}. \qquad (2.1)$$

The inversion lemma (which we will meet in Lemma 2.4.3) implies that if id has type ρ, then

$$f : \rho \vdash \langle f\,\mathsf{true}, f\,\mathsf{1}\rangle : \tau$$

for some type τ, which must be on the form $\tau_1 \times \tau_2$. Further application of the inversion lemma then shows that

$$f : \rho \vdash f : \mathsf{Bool} \to \tau_1$$

and

$$f : \rho \vdash f : 1 \to \tau_2$$

simultaneously. But then we must have[7]

$$\mathsf{Bool} \to \tau_1 = \rho = 1 \to \tau_2,$$

which is impossible since Bool and 1 are distinct types.

[7]Notice that this argument does not appeal to *uniqueness* of types, but simply uses the inversion lemma. We return to the question of uniqueness in §2.4.1(c).

However, it seems like we morally should be able to write something like (2.1). Clearly the function id does not care about the type of its argument, and in many languages without static type checking we could easily get away with such an expression.

We solve this problem as follows: Instead of requiring that the types of expressions be closed, we allow them to contain type variables. Just as with expression variables, we then introduce *abstractions* on type variables, lambda expressions that take types instead of expressions as arguments. To distinguish these from ordinary lambda expressions, we use a capital Λ. If e is an expression whose type τ may contain type variables, among these α, we can then write[8] $\Lambda\alpha.e$ to denote an expression which takes a type τ' as argument and yields an expression with type $\tau[\tau'/\alpha]$. The expression $\Lambda\alpha.e$ then becomes polymorphic in the sense that we may instantiate α in τ as any type, though note that if τ has no other free type variables than α, then the type of $\Lambda\alpha.e$ is closed. We denote the type of $\Lambda\alpha.e$ by $\forall\alpha.\tau$.

[8]This is not our official syntax since this does not contain explicit type variables. We return to this below.

For instance, we can define a polymorphic version of the identity function as follows: The expression $\lambda x.x$ has type $(\alpha \to \alpha)$, where α is a type variable, so the expression

$$\mathsf{id} = \Lambda\alpha.\lambda x.x$$

then has type $\forall\alpha.(\alpha \to \alpha)$. To instantiate α as a concrete type, say Bool, we write[9] $\mathsf{id}\,[\mathsf{Bool}]$. Returning to the untypable expression in (2.1), by using the polymorphic version of id we can write

[9]Again our official syntax does not use explicit types in expressions.

$$(\lambda f.\langle f\,[\mathsf{Bool}]\,\mathsf{true}, f\,[1]\,1\rangle)\,\mathsf{id},$$

which is then well-typed.

The above example is fairly artificial, but parametric polymorphism is of course ubiquitous in functional programming: For a simple example, lists are usually polymorphic in the sense that to each type τ there is a type of τ-lists where expressions of such a type are lists whose elements all are of type τ. Using the recursive list type $\tau\text{-}\mathsf{List} = \mu\alpha.1 + (\tau \times \alpha)$ we will construct in §2.3.7, we obtain a polymorphic list type by

$$\mathsf{List} := \forall\beta.\mu\alpha.1 + (\beta \times \alpha).$$

Finally we note that $\mathbf{F}_{\exists,\mu,\text{ref}}$ does not have explicit types. Instead of $\Lambda\alpha.e$ we replace the type variable with a wildcard and write $\Lambda\_.e$, and instead of $e\,[\tau]$ we similarly write $e\,\_$. This of course makes the type checker's job more difficult, but this is not a concern for us in this report.

In fact, existential types can be encoded using universal types, cf. Pierce (2002, §24.3).

(b) ***Existential types.*** Since we can universally quantify over type variables, it seems reasonable to assume that we can also *existentially* quantify over them.

If an expression has the existential type $\exists\alpha.\tau$, then this supposedly means that there is some type τ' such that the expression has type $\tau[\tau'/\alpha]$. Notice that there might be multiple different types τ' such that e has type $\tau[\tau'/\alpha]$: For instance, if e is the expression $\lambda x.1$, then this has type $\tau' \to 1$ for any type τ'. When we abstract away this type τ', it is therefore useful to keep track of it. If an expression e has type $\tau[\tau'/\alpha]$, then we therefore (tentatively) introduce an expression of existential type $\exists\alpha.\tau$ by[10] $\mathsf{pack}\,(e,\tau')$. We keep track of the ***witness type*** τ', but notice that this does not appear in the existential type itself, meaning that it cannot be used to typecheck. We therefore also call it the ***hidden representation type***.

[10]The name 'pack' will make sense shortly. As for universal types this notation is temporary, and we introduce our official notation below.

Furthermore, the expression $\mathsf{pack}\,(\lambda x.1,\tau')$ itself also has multiple types, for instance $\exists\alpha.1 \to 1$ and even $\exists\alpha.\alpha \to \alpha$. Hence it is also useful when introducing an expression of either type to specify which type it should be introduced as, and so we might write something like

Notice that the 'as'-clause is a kind of ***ascription***, cf. Pierce (2002, §11.4).

$$\mathsf{pack}\,(\lambda x.1,\tau')\,\mathsf{as}\,\exists\alpha.\alpha \to \alpha.$$

Existential types can be used as a sort of primitive module system by allowing us to implement abstract data types. Under this interpretation we also think of expressions of existential type as *packages*. As an example, we implement an abstract counter type[11]. Assuming that we have a natural number type Nat, a constructor $\mathsf{0}$ and a successor operation succ, we use Nat as the hidden representation type. Since this is not available to the user, this allows us as 'maintainers' of this package to change the internal representation without issue.

[11]This example is adapted from Pierce (2002, §24.2).

The counter type should be able to create new counters, increment counters, and return the value of a counter. Creating a new counter is simple, we just use the constructor $\mathsf{0}$. To increment a counter we can use the function $\lambda n.\mathsf{succ}\,n$, and to return its value we simply return the natural number used internally, so we use the identity function $\lambda n.n$. In total the internal representation of the type becomes

We here use a tuple with three elements to simplify notation, but this can of course be implemented as two nested pairs.

$$\langle \mathsf{0}, \lambda n.\mathsf{succ}\,n, \lambda n.n\rangle,$$

and when packed with representation type Nat it receives the existential type

$$\exists\alpha.(\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{Nat})).$$

That is, we obtain the expression

$$\mathsf{pack}\,(\langle \mathsf{0}, \lambda n.\mathsf{succ}\,n, \lambda n.n\rangle, \mathsf{Nat})\,\mathsf{as}\,\exists\alpha.(\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{Nat})).$$

This is the expression that is available to users of the package, and notice again that while the hidden representation type Nat does appear in the expression, it is not part of the type and so does not affect type checking.

We also need a way to 'unpack' packages. Given a package p of type $\exists\alpha.\tau$, we need some way of 'importing' p and extracting its components. Say that e is the program in which we wish to use p, and that we wish to bind the contents of p to a variable x. Then x must have the correct type, namely τ (perhaps containing the type variable α). The hidden representation type of p will also become bound to α in e if it occurs free. We may write this as follows:

$$\mathsf{unpack}\, p\, \mathsf{as}\, (x,\alpha)\, \mathsf{in}\, e.$$

More explicitly, say that p is the expression $\mathsf{pack}\,(v,\tau')\,\mathsf{as}\,\exists\alpha.\tau$, where v is a value. We then reduce the above expression as

$$\mathsf{unpack}\,(\mathsf{pack}\,(v,\tau')\,\mathsf{as}\,\exists\alpha.\tau)\,\mathsf{as}\,(x,\alpha)\,\mathsf{in}\,e \to_p e[v/x][\tau'/\alpha].$$

Note that $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ does not have explicit types, so we simplify the pack-unpack notation and instead simply write $\mathsf{pack}\,v$ and $\mathsf{unpack}(p,x,e)$.

2.3.7. Recursive types

A recursive type is, roughly speaking, a type that is defined in terms of itself, or properly contains itself syntactically[12]. In a sense a recursive type is thus somehow infinite, so to properly understand recursive types, we begin by considering how types can be represented as various kinds of objects that can be either finite or infinite.

[12]That is, it is a proper 'subtype' of itself in the sense of sub-ASTs, not in the usual sense of subtyping.

Types can of course be represented using potentially infinite strings, but they are more naturally represented by rooted trees; almost obviously, since we have even defined them as abstract syntax *trees*. This representation is of course very natural: The leaves are either base types or type variables, and the inner vertices are operators whose children are their arguments. An example is given in Figure 2.1.

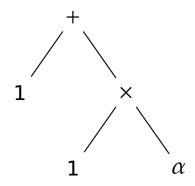

Figure 2.1. Tree representing the type $1+(1\times\alpha)$.

In the other direction we can imagine *infinite* trees that have the same general structure: Leaves are still base types or type variables, inner vertices are operators with the correct number of children, but the tree is infinite. For instance, consider the tree in Figure 2.2. The outermost operator (i.e., the root of the tree) is $\to$, so if this tree is supposed to represent a type, then it must be a function type. The argument type is 1, but notice that the right child of the root is just another copy of the whole tree. Apparently, an expression of this type is a function that takes 1 as an argument and returns a function of the same type as itself. If τ denotes (the type represented by) the whole tree, we thus have $\tau = 1 \to \tau$. We denote τ by $\mu\alpha.1\to\alpha$, motivated by the fact that τ is a fixed-point of the map $\alpha \mapsto 1 \to \alpha$.

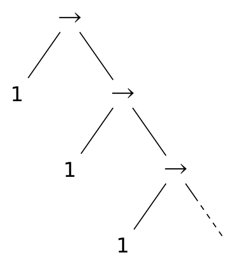

Figure 2.2. An infinite tree representing the 'infinite type' $1\to(1\to(1\to\cdots))$.

For a more immediately useful example, we show how to construct a recursive list type: Recall that in order to construct a linked list, we need a ‘nil’ object to represent the empty list and a ‘cons’‘nil’ operation that pairs two objects. If we use $\mathsf{1}$ to represent the empty list and the built-in pairs $\langle -, - \rangle$ for pairing, then an informal specification of a type τ-List of lists of elements of type τ could look like

$$\tau\text{-List} ::= \mathsf{1} \mid \langle \tau, \tau\text{-List} \rangle.$$

More precisely, we would have $\tau\text{-List} = \mu\alpha.\mathsf{1} + (\tau \times \alpha)$.

What we have outlined is the ***equi-recursive*** approach to recursive types, in which the recursive type τ is by definition *equal* to its ‘unfolded’ counterpart $\mathsf{1} \to \tau$. More generally, a recursive type $\mu\alpha.\tau$ is equal to the type $\tau[\mu\alpha.\tau/\alpha]$. An alternative is the ***iso-recursive*** approach, in which the types $\mu\alpha.\tau$ and $\tau[\mu\alpha.\tau/\alpha]$ are seen as different but somehow *isomorphic*. The isomorphism is witnessed by maps unfold and fold that yield a correspondence between expressions of the two types, as schematically presented in Figure 2.3.

$$\begin{tikzcd} \mu\alpha.\tau \arrow[r, bend left, "\mathsf{unfold}"] & \tau[\mu\alpha.\tau/\alpha] \arrow[l, bend left, "\mathsf{fold}"] \end{tikzcd}$$

FIGURE 2.3. Correspondence between a recursive type $\mu\alpha.\tau$ and its ‘unfolded’ version $\tau[\mu\alpha.\tau/\alpha]$.

We construct a recursive list type. Let τ be the type of the elements of the list, and assume that α is not free in τ. The expression $\iota_1\,\mathsf{1}$ has type $\mathsf{1} + (\tau \times \tau\text{-List})$, so we fold it and obtain the expression $\mathsf{nil} := \mathsf{fold}\,\iota_1\,\mathsf{1}$ with type $\tau\text{-List} = \mu\alpha.\mathsf{1} + (\tau \times \alpha)$. We similarly want a cons operator, and this should take an expression of type τ and prepend this to a τ-list, so it should be of type $\tau \to \tau\text{-List} \to \tau\text{-List}$. Given expressions x and l of type τ and τ-List respectively, the expression $\iota_2\,\langle x, l \rangle$ has type $\mathsf{1} + (\tau \times \tau\text{-List})$, so folding this and abstracting out x and l we obtain

$$\mathsf{cons} := \lambda x.\lambda l.\mathsf{fold}\,\iota_2\,\langle x, l \rangle$$

with the correct type.

In order to use expressions with recursive types we first reduce the argument to fold to a value and apply unfold to extract this value. In practice, if a programming language supports recursive types and takes the iso-recursive approach, folding and unfolding often happens automatically. For instance, in ML a fold is automatically added to each constructor of a recursive type, while an unfold is added during pattern matching[13].

[13]Cf. Pierce (2002, §20.2).

2.3.8. Mutable state

Side effects in programming languages should be nothing new to the reader, but what may be unfamiliar is side effects being reflected by the type system[14]: Notice that the only way side effects can occur is if an expression of reference type $\mathsf{Ref}\,\tau$ is present somewhere.

[14]Users of languages like Haskell will of course be familiar with this concept.

2.4 ◇ Language properties

2.4.1. Type system

(a) ***Well-formedness.*** In §2.1.2(c) we defined what it means for a type context, store typing or just a type to be well-formed with respect to a set of type variables. It is of course desirable that the typing relation is such that if $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ holds, then Γ, Σ and τ are all well-formed with respect to Ξ. By adding appropriate assumptions to the axioms of the typing relation (i.e., to T-VAR, T-UNIT and T-LOC) we ensure that this is the case:

2.4.1 • Proposition. *If $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$, then Γ, Σ and τ are well-formed with respect to Ξ.*

Proof. Proof by rule induction on the typing relation. We only consider some representative cases since many of them are nearly identical. As a general comment, notice that if for instance $\Gamma \subseteq \Delta$ and $\Xi \vdash \Delta$, then also $\Xi \vdash \Gamma$, and similarly for store typings. Furthermore, if τ is a structurally smaller[15] type than τ', then $\Xi \vdash \tau'$ implies $\Xi \vdash \tau$.

[15]We have not defined what it means for one type (or more generally one AST) to be structurally smaller or larger than another, but we assume that the reader is familiar with this notion. Regardless, this comment is only supposed to motivate the proof below.

T-VAR: By assumption we have both $\Xi \vdash \Gamma$, $\Xi \vdash \Sigma$, and $\Xi \vdash \tau$ follows since $(x : \tau) \in \Gamma$ so that $\tau \in \operatorname{ran} \Gamma$.

T-UNIT: Again $\Xi \vdash \Gamma$ and $\Xi \vdash \Sigma$ follow by assumption, and obviously $\Xi \vdash \mathbf{1}$ since $\mathbf{1}$ has no free type variables.

T-PAIR: This follows since $\mathrm{FV}_{Type}(\tau_1 \times \tau_2) = \mathrm{FV}_{Type}(\tau_1) \cup \mathrm{FV}_{Type}(\tau_2)$ by definition of free (type) variables.

T-TLAM: By induction we have both $\Xi, \alpha \vdash \Gamma$ and $\Xi, \alpha \vdash \Sigma$, and since α is not free in either Γ or Σ, it follows that indeed $\Xi \vdash \Gamma$ and $\Xi \vdash \Sigma$. Furthermore,

$$\mathrm{FV}_{Type}(\forall \alpha . \tau) = \mathrm{FV}_{Type}(\tau) \setminus \{\alpha\} \subseteq (\Xi, \alpha) \setminus \{\alpha\} = \Xi,$$

as desired.

T-TAPP: Notice first that $\mathrm{FV}_{Type}(\forall \alpha . \tau) = \mathrm{FV}_{Type}(\tau) \setminus \{\alpha\}$ by definition of free (type) variables. Furthermore, the type variables that are free in the type $\tau[\tau'/\alpha]$ are at most those that are free in τ except α, along with those that are free in τ'.[16] Hence

[16]A proof of this fact would require us to precisely define substitution in ASTs, which we have not done. Thus we settle for an appeal to intuition.

$$\begin{aligned} \mathrm{FV}_{Type}(\tau[\tau'/\alpha]) &\subseteq (\mathrm{FV}_{Type}(\tau) \setminus \{\alpha\}) \cup \mathrm{FV}_{Type}(\tau') \\ &= \mathrm{FV}_{Type}(\forall \alpha . \tau) \cup \mathrm{FV}_{Type}(\tau') \\ &\subseteq \Xi, \end{aligned}$$

where we have used both the induction hypothesis and the assumption $\Xi \vdash \tau'$.

T-LOC: We already have $\Xi \vdash \Gamma$ and $\Xi \vdash \Sigma$ by assumption, and furthermore

$$\mathrm{FV}_{Type}(\mathsf{Ref}\, \Sigma(l)) = \mathrm{FV}_{Type}(\Sigma(l)) \subseteq \mathrm{FV}_{Type}(\Sigma) \subseteq \Xi,$$

as desired. ∎

(b) ***Inversion.*** As we proved in Theorem 1.2.4, every set that is defined using inference rules gives rise to a notion of inversion. We specialise this result to the typing relation, but we first note an important property of the type system:

2.4.2 • *Remark.* We collect the infinite collection of inference rules that define the typing relation into finitely many sets, each of which is represented by a rule schema, namely those listed in §B.2.

Notice then that the conclusions of the schemas are distinct, in the sense that if $H_1 \Mapsto y_1$ and $H_2 \Mapsto y_2$ are instances of different schemas, then $y_1 \neq y_2$. To see this, simply consider every pair of different schemas in §B.2 and check that their conclusions are distinct in this sense. ⌟

Recall the distinction between *rules* and rule *schemas*. We have not given a precise definition of a 'schema', but its meaning is hopefully well-known from logic where we meet the distinction between axioms and axiom schemas (for instance, consider the distinction between the induction axiom of second-order Peano arithmetic and the induction axiom schema of first-order Peano arithmetic). For our purposes we can simply think of a rule schema as a set of rules. An 'instance' of a schema is then just an element of such a set.

2.4.3 • **Lemma:** ***Inversion on typing.***
Assume that $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$.

(i) *If* $e = x$ *is a variable, then* $(x : \tau) \in \Gamma$.

(ii) *If* $e = \mathsf{1}$, *then* $\tau = \mathsf{1}$.

(iii) *If* $e = \langle e_1, e_2 \rangle$, *then* $\tau = \tau_1 \times \tau_2$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_i : \tau_i$.

(iv) *If* $e = \pi_1\, e'$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau \times \tau_2$. *If instead* $e = \pi_2\, e'$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1 \times \tau$.

(v) *If* $e = \iota_1\, e'$, *then* $\tau = \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1$. *If instead* $e = \iota_2\, e'$, *then* $\tau = \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_2$.

(vi) *If* $e = \mathsf{match}(e', x, e_1, e_2)$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma, x : \tau_1 \mid \Sigma \vdash e_1 : \tau$ *and* $\Xi \mid \Gamma, x : \tau_2 \mid \Sigma \vdash e_2 : \tau$.

(vii) *If* $e = \mathsf{rec}\, f(x) := e'$, *then* $\tau = \tau_1 \to \tau_2$ *and* $\Xi \mid \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \mid \Sigma \vdash e' : \tau_2$.

(viii) *If* $e = e_2\, e_1$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_1 \to \tau$.

(ix) *If* $e = \Lambda\_.e'$, *then* $\tau = \forall \alpha.\tau'$ *and* $\Xi, \alpha \mid \Gamma \mid \Sigma \vdash e' : \tau'$. *In particular,* $\alpha \notin \Xi$.

(x) *If* $e = e'\,\_$, *then* $\tau = \tau'[\tau''/\alpha]$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \forall \alpha.\tau'$.

(xi) *If* $e = \mathsf{pack}\, e'$, *then* $\tau = \exists \alpha.\tau'$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau'[\tau''/\alpha]$.

(xii) *If* $e = \mathsf{unpack}(e_1, x, e_2)$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \exists \alpha.\tau'$ *and* $\Xi, \alpha \mid \Gamma, x : \tau' \mid \Sigma \vdash e_2 : \tau$.

(xiii) *If* $e = \mathsf{fold}\, e'$, *then* $\tau = \mu \alpha.\tau'$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau'[\mu \alpha.\tau'/\alpha]$.

(xiv) *If* $e = \mathsf{unfold}\, e'$, *then* $\tau = \tau'[\mu \alpha.\tau'/\alpha]$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \mu \alpha.\tau'$.

(xv) *If* $e = l$ *is a location, then* $l \in \mathrm{dom}\, \Sigma$ *and* $\tau = \mathsf{Ref}\, \Sigma(l)$.

(xvi) *If* $e = \mathsf{ref}\, e'$, *then* $\tau = \mathsf{Ref}\, \tau'$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau'$.

(xvii) *If* $e = e_1 := e_2$, *then* $\tau = \mathsf{1}$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \mathsf{Ref}\, \tau'$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau'$.

(xviii) *If* $e = !\, e'$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \mathsf{Ref}\, \tau$.

Proof. All the above claims are proved in precisely the same way, so we prove one of them, say (iii). Assume that $e = \langle e_1, e_2 \rangle$. Then Theorem 1.2.4 implies that there is a typing rule R whose conclusion is $\Xi \mid \Gamma \mid \Sigma \vdash \langle e_1, e_2 \rangle : \tau$, and whose premises also hold. But Remark 2.4.2 says that there is a unique rule schema that has an instance with this conclusion, and considering all the rules (cf. §B.2) we see that this schema must be T-Pair. Hence R is the rule

$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \qquad \Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

where τ_1 and τ_2 are appropriate types. Comparing the two forms of the conlusion of R, we must have $\tau = \tau_1 \times \tau_2$. As mentioned its premises also hold, yielding the second part of (iii). ■

(c) *Uniqueness of types.* It would obviously be nice if well-typed expressions of $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ had *unique* types. However, this is clearly not the case, since expressions on the form $e\,\_$ can have many different types depending on the type τ' in T-Tapp. As a consequence, not even values have unique types, since a value can have non-values as subexpressions, for instance the expression e in either $\mathsf{rec}\, f(x) := e$ or $\Lambda\_.e$.

Had we made different choices when designing $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$, we could have had uniqueness of types, for instance if the language had explicit type annotations as we briefly explored in §2.3.6. Notice however that this requires expressions to have types as sub-ASTs, something which $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ does not allow.

On the other hand, notice that the inversion lemma puts some restrictions on the form the types of an expression can take. For instance, (iii) says that even though a pair expression $\langle e_1, e_2 \rangle$ might have many different types, all of them must be on the form $\tau_1 \times \tau_2$.

This observation leads to a partial converse of the inversion lemma. While the inversion lemma assigns types to expressions, the following result assigns expressions to types:

2.4.4 • Lemma: *Canonical forms.*
Assume that $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau$ where v is a value.

(i) *If $\tau = 1$, then $v = 1$.*

(ii) *If $\tau = \tau_1 \times \tau_2$, then $v = \langle v_1, v_2 \rangle$.*

(iii) *If $\tau = \tau_1 + \tau_2$, then either $v = \iota_1 v'$ or $v = \iota_2 v'$.*

(iv) *If $\tau = \tau_1 \to \tau_2$, then $v = \mathsf{rec}\, f(x) := e$.*

(v) *If $\tau = \forall \alpha.\tau'$, then $v = \Lambda\_.e$.*

(vi) *If $\tau = \exists \alpha.\tau'$, then $v = \mathsf{pack}\, v'$.*

(vii) *If $\tau = \mu \alpha.\tau'$, then $v = \mathsf{fold}\, v'$.*

(viii) *If $\tau = \mathsf{Ref}\, \tau'$, then v is a location.*

Proof. The proof of each claim is identical, so we only prove one, say (v). The proof consists of going through each form a value can have (cf. §B.1), and seeing that only a value on the form $\Lambda\_.e$ can have a type on the form $\forall\alpha.\tau'$. For instance, we claim that this is the case for a value $v' = \mathsf{rec}\, f(x) := e$. By Lemma 2.4.3(vii) the type of v' must be on the form $\tau_1 \to \tau_2$, which is different from $\Lambda\_.e$. Hence $v \neq v'$. ■

(d) *Weakening.* We begin with a couple of easy results, the first of which is sometimes known as ***weakening***, and the second of which we conversely might call ***strengthening***.

2.4.5 • Lemma: *Weakening.*
If

(i) Ξ *and* Ξ' *are finite sets of type variables with* $\Xi \subseteq \Xi'$,

(ii) Γ *and* Γ' *are type contexts with* $\Gamma \subseteq \Gamma'$ *and* $\Xi \vdash \Gamma'$, *and*

(iii) Σ *and* Σ' *are store typings with* $\Sigma \subseteq \Sigma'$ *and* $\Xi \vdash \Sigma'$,

then

$$\Xi \mid \Gamma \mid \Sigma \vdash e : \tau \quad \textit{implies} \quad \Xi' \mid \Gamma' \mid \Sigma' \vdash e : \tau$$

Proof. This is a fairly straightforward induction on type derivations. Notice that whenever a type variable α or a pair $x : \tau$ appears in a hypothesis but not in the conclusion of a rule, then the (type) variable in question is bound in the conclusion. To perform the corresponding inductive step we can thus change the bound variables in the conclusion so that $\alpha \notin \Xi'$ and $x \notin \operatorname{dom}\Gamma'$. For the store typing there is no such issue since the store typings in the hypotheses and the conclusion are always the same. ■

2.4.6 • Lemma: *Strengthening.*
Assume that $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$. *If* Γ, Σ *and* τ *are well-formed with respect to* $\Phi \subseteq \Xi$, *then also* $\Phi \mid \Gamma \mid \Sigma \vdash e : \tau$.

Proof. This is an easy proof by rule induction on the typing relation. It also follows from Lemma 2.4.8 below: Since $\Xi \setminus \Phi$ is finite we may assume that $\Xi = \Phi, \alpha$, and since α is not free in either Γ, Σ or τ, we may choose any type τ' with $\Phi \vdash \tau'$ (for instance 1) and substitute it into α.■

(e) *Substitution.* Next a couple of results on how the type relation interacts with substitution.

2.4.7 • Lemma. *If* $\Xi \mid \Gamma, z : \rho \mid \Sigma \vdash e : \tau$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash d : \rho$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e[d/z] : \tau$.

Proof. The proof is by rule induction in $\Xi \mid \Gamma, z : \rho \mid \Sigma \vdash e : \tau$. More precisely, we prove the following claim:

> *For all* $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ *the following holds: If* $\Xi \mid \Gamma \mid \Sigma \vdash d : \rho$, *then either* $\Gamma(z) \neq \rho$ *or else* $\Xi \mid \Gamma \setminus (z : \rho) \mid \Sigma \vdash e[d/z] : \tau$.

This is then a claim about a general judgment $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$, and the proof is by rule induction on the typing relation. But if $(z : \rho)$ is not present in the type context in the conclusion of a rule, then the induction step for that rule is trivial. Hence we may assume that said type context always contains $(z : \rho)$, and we thus write $\Gamma, z : \rho$.

The proof for many of the inductive steps are identical, so we only present a few representative cases.

T-VAR: Assume that $\Gamma(x) = \tau$ and that $\Xi \mid \Gamma \mid \Sigma \vdash d : \rho$. If $x = z$ then $x[d/z] = d$, so the claim holds in this case. If instead $x \neq z$, then $x[d/z] = x$. But since $\Gamma(x) = \tau$, we have $\Xi \mid \Gamma \mid \Sigma \vdash x : \tau$ without $(z : \rho)$ as required.

In the axioms T-VAR, T-UNIT and T-LOC we should also assume that the type contexts and store typings are well-formed with respect to the set Ξ of type variables, but nothing in the proof rests on this assumption.

T-UNIT: This is clear since $\Xi \mid \Gamma \mid \Sigma \vdash \mathbf{1} : \mathbf{1}$ always holds.

T-PAIR: Assume that the claim holds for $\Xi \mid \Gamma, z : \rho \mid \Sigma \vdash e_i : \tau_i$ for $i \in \{1, 2\}$, and assume that $\Xi \mid \Gamma \mid \Sigma \vdash d : \rho$. By induction we have $\Xi \mid \Gamma \mid \Sigma \vdash e_i[d/z] : \tau_i$, so an application of T-PAIR implies that $\Xi \mid \Gamma \mid \Sigma \vdash \langle e_1[d/z], e_2[d/z] \rangle : \tau_1 \times \tau_2$. But since $\langle e_1[d/z], e_2[d/z] \rangle = \langle e_1, e_2 \rangle [d/z]$, the claim follows.

T-MATCH: Assume that the claim holds for $\Xi \mid \Gamma, z : \rho \mid \Sigma \vdash e : \tau_1 + \tau_2$ and $\Xi \mid \Gamma, x : \tau_i, z : \rho \mid \Sigma \vdash e_i : \tau$ for $i \in \{1, 2\}$, and assume that $\Xi \mid \Gamma \mid \Sigma \vdash d : \rho$. By Lemma 2.4.5 also $\Xi \mid \Gamma, x : \tau_i \mid \Sigma \vdash d : \rho$, so by induction we have $\Xi \mid \Gamma \mid \Sigma \vdash e[d/z] : \tau_1 + \tau_2$ and $\Xi \mid \Gamma, x : \tau_i \mid \Sigma \vdash e_i[d/z] : \tau$, so by applying T-MATCH we get $\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{match}(e[d/z], x, e_1[d/z], e_2[d/z]) : \tau$. But since[17] $\mathsf{match}(e[d/z], x, e_1[d/z], e_2[d/z]) = \mathsf{match}(e, x, e_1, e_2)[d/z]$, the claim follows.

[17] Here we use that $x \neq z$ and that $x \notin \operatorname{dom} \Gamma$, so that z is not bound by the match-expression, and so that x is not free in d.

T-LOC: This is clear since a location l contains no free variables. ■

2.4.8 • Lemma. *If $\Xi, \alpha \mid \Gamma \mid \Sigma \vdash e : \tau$ and τ' is a type with $\Xi \vdash \tau'$, then $\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e : \tau[\tau'/\alpha]$.*

Proof. The proof is by rule induction in the typing relation. However, we prove the following claim:

> *For all $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ the following holds: If $\Xi \setminus \{\alpha\} \vdash \tau'$, then either $\alpha \notin \Xi$ or else $\Xi \setminus \{\alpha\} \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e : \tau[\tau'/\alpha]$.*

If $\alpha \notin \Xi$ in the conclusion of a rule, then the induction step for that rule is trivial. Hence we may assume that α does lie in the set of type variables in the conclusion, and we write Ξ, α. The assumption of well-formedness of τ' then becomes $\Xi \vdash \tau'$ as in the statement of the lemma.

Since many of the inductive steps are basically identical, we only prove some of them.

T-VAR: Assume that $\Xi, \alpha \vdash \Gamma$ and $\Xi, \alpha \vdash \Sigma$ and $\Gamma(x) = \tau$. Since $\Xi \vdash \tau'$, the variable α is not free in τ', and so $\Xi \vdash \Gamma[\tau'/\alpha]$ and $\Xi \vdash \Sigma[\tau'/\alpha]$. Furthermore, $\Gamma[\tau'/\alpha](x) = \Gamma(x)[\tau'/\alpha] = \tau[\tau'/\alpha]$, so an application of T-VAR implies that $\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash x : \tau[\tau'/\alpha]$.

T-pair: Assume that $\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e_i : \tau_i[\tau'/\alpha]$ for $i \in \{1,2\}$. An application of T-pair then yields

$$\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash \langle e_1, e_2 \rangle : \tau_1[\tau'/\alpha] \times \tau_2[\tau'/\alpha],$$

and since $\tau_1[\tau'/\alpha] \times \tau_2[\tau'/\alpha] = (\tau_1 \times \tau_2)[\tau'/\alpha]$, we are done.

T-rec: Assume that $\Xi \mid (\Gamma, f : \tau_1 \to \tau_2, x : \tau_1)[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e : \tau_2[\tau'/\alpha]$. This means that

$$\Xi \mid \Gamma[\tau'/\alpha], f : \tau_1[\tau'/\alpha] \to \tau_2[\tau'/\alpha], x : \tau_1[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e : \tau_2[\tau'/\alpha],$$

so applying T-rec we get

$$\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash \mathsf{rec}\, f(x) := e : \tau_1[\tau'/\alpha] \to \tau_2[\tau'/\alpha].$$

Since $\tau_1[\tau'/\alpha] \to \tau_2[\tau'/\alpha] = (\tau_1 \to \tau_2)[\tau'/\alpha]$, the claim follows.

We use β instead of α to denote the type variable appearing in the rule schema T-Tlam. Since we have assumed that α lies in the set of type variables in the conclusion of each rule, $\alpha \neq \beta$.

T-Tlam: Let $\beta \notin \Xi$ be a type variable. Since $\Xi \vdash \tau'$ we also have $\Xi, \beta \vdash \tau'$, so by induction we have

$$\Xi, \beta \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e : \tau[\tau'/\alpha].$$

An application of T-Tlam then yields

$$\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash \Lambda\_.e : \forall \beta.\tau[\tau'/\alpha].$$

Since $\beta \notin \Xi$ and $\Xi \vdash \tau'$, the variable β is not free in τ', and since also $\alpha \neq \beta$, we have $\forall\beta.\tau[\tau'/\alpha] = (\forall\beta.\tau)[\tau'/\alpha]$. The claim follows.

T-Tapp: Assume that $\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e : (\forall\beta.\tau)[\tau'/\alpha]$, and let τ'' be a type with $\Xi, \alpha \vdash \tau''$. Since β only occurs bound and we identify ASTs up to α-equivalence, we may rename it and assume that $\alpha \neq \beta$ and that $\beta \notin \Xi$. Hence $(\forall\beta.\tau)[\tau'/\alpha] = \forall\beta.\tau[\tau'/\alpha]$. We furthermore have $\Xi \vdash \tau''[\tau'/\alpha]$, so by applying T-Tapp we get[18]

$$\Xi \mid \Gamma[\tau'/\alpha] \mid \Sigma[\tau'/\alpha] \vdash e\,\_ : \tau[\tau'/\alpha][\tau''[\tau'/\alpha]/\beta].$$

Since $\alpha \neq \beta$ and β is not free in τ', we have[19] $\tau[\tau'/\alpha][\tau''[\tau'/\alpha]/\beta] = \tau[\tau''/\beta][\tau'/\alpha]$, which proves the claim. ∎

[18] It is easy to forget that we cannot simply commute the substitutions $[\tau''/\beta]$ and $[\tau'/\alpha]$ since α might occur free in τ''. If we could, then we would not have to perform the substitution $[\tau'/\alpha]$ on τ''.

[19] This follows by what in the context of the λ-calculus is sometimes called the ***substitution lemma***, see for instance Barendregt (1984, Lemma 2.1.16).

2.4.2. Operational semantics

Let us temporarily call an expression e a 'redex' if there is an expression e' and stores σ and σ' such that $(\sigma, e) \to_h (\sigma', e')$.

First we note that, as advertised, values are indeed irreducible:

2.4.9 • Proposition. *Every value is irreducible.*

Proof. Proof by induction on values. Let v be a value. If v is on one of the forms $\mathtt{1}$, $\mathsf{rec}\, f(x) := e$, $\Lambda\_.e$ or is a location, then this is clear since neither of these are redexes, and none of them are on the form $E[e']$ for an evaluation context E and an expression e'.

The inductive step is similar in all cases, so we only consider the case $v = \langle v_1, v_2 \rangle$. This is not a redex, so assume that there is an evaluation context E and an expression d such that $v = E[d]$ and $d \rightarrow_h d'$ for some d'. Then E must be on the form $\langle E', e' \rangle$ or $\langle v', E'' \rangle$: In the first case it follows that $v_1 = E'[d]$, and in the second that $v_2 = E''[d]$. Both are impossible since by induction both v_1 and v_2 are irreducible. Hence v is not on the form $E[d]$ and hence is not reducible. ■

Next recall that we in §2.1.3(b) noted that the presence of references in $\mathbf{F}_{\exists,\mu,\mathrm{ref}}$ means that it is not deterministic. Furthermore, it is also clearly not even weakly normalising since it has recursive functions, hence effectively infinite loops. But considering instead the fragment $\mathbf{F}$ we have the following result:

2.4.10 • Theorem: ***Determinism.***
In $\mathbf{F}$*, the reduction* $\rightarrow$ *is deterministic.*

Proof. Assume that $e \rightarrow e_1$ and $e \rightarrow e_2$. We must then show that $e_1 = e_2$. First notice that for redexes d, d_1, d_2, if $d \rightarrow_h d_1$ and $d \rightarrow_h d_2$ then $d_1 = d_2$: This follows by noticing that exactly one head reduction rule applies to d.

Next let E_1 and E_2 be evaluation contexts and d_1 and d_2 redexes such that $E_1[d_1] = E_2[d_2]$. We claim that $E_1 = E_2$ and $d_1 = d_2$. The proof is by induction in E_1.

$E_1 = -$: Assume towards a contradiction that $E_2 \neq -$. We consider only a few of the other possibilities since the rest are similar. If $E_2 = \langle E, e \rangle$, then it follows that $d_1 = \langle E[d_2], e \rangle$. But this is impossible since the right-hand side is not a redex.

If instead $E_2 = \pi_1\, E$ then $d_1 = \pi_1\, E[d_2]$, and then d_1 must be on the form $\pi_1 \langle v_1, v_2 \rangle$. Hence $v_1 = E[d_2]$, but this is impossible since v_1 is irreducible by Proposition 2.4.9.

$E_1 = \langle E, e \rangle$: Most possibilities for E_2 are clearly impossible since it must be a pair. If $E_2 = \langle E', e' \rangle$ then it follows that $E[d_1] = E'[d_2]$ and $e = e'$, so by induction we have $E = E'$ and $d_1 = d_2$.

The other possibility is $E_2 = \langle v, E' \rangle$. In this case $E[d_1] = v$, which is impossible since v is irreducible by Proposition 2.4.9.

$E_1 = \langle v, E \rangle$: This is the opposite of the above case.

$E_1 = E\ e$ *or* $E_1 = v\ E$: These are similar to the above two cases.

Remaining cases: The remaining possibilities for E_1 are identical, so we give a single example. If $E_1 = \pi_1\, E$, then E_2 must also be on the form $\pi_1\, E'$. It follows that $E[d_1] = E'[d_2]$, so by induction $E = E'$ and $d_1 = d_2$.

Returning to the claim to be proved, since $e \rightarrow e_1$ and $e \rightarrow e_2$ there are evaluation contexts E_1 and E_2 as well as redexes d_1, d_2, d'_1, d'_2 such that $e = E_1[d_1]$, $e_1 = E_1[d'_1]$ and $d_1 \rightarrow_h d'_1$, and such that $e = E_2[d_2]$, $e = E_2[d'_2]$ and $d_2 \rightarrow_h d'_2$. The above first implies that $E_1 = E_2$ and $d_1 = d_2$, and next that $d'_1 = d'_2$. ■

2.4.11 • Corollary. *In* **F**, *if* $e \to^* e_1$ *and* $e \to^* e_2$ *with* e_1 *and* e_2 *irreducible, then* $e_1 = e_2$. ■

We could now go ahead and study whether **F** is also weakly or even strongly normalising. And indeed it turns out to be, but the presence of polymorphism makes the proof rather difficult[20]. In Chapter 5 the fact that **F** is normalising will affect how we choose to define certain concepts, but we will not use this fact in any proofs.

[20]See Pierce (2002, §23.5) and the references therein.

Type Safety I: Progress *and* Preservation 3

Having become acquainted with $\mathbf{F}_{\exists,\mu,\text{ref}}$ we being our study of type safety. In this chapter we take a classical approach to type safety, by proving the properties *progress* and *preservation*.

3.1 ⋄ Safety and well-typedness

3.1.1. Safety defined

It is important to consider what kinds of safety a type system is supposed to give us. For instance, we are not so ambitious as to require of the type system that it is able to catch algorithmic errors, or errors in logic.

On the other hand, well-typedness of programs is supposed to tell us something about how its subexpressions are combined to form a complete program. For instance, in a language with numbers and strings we would hope that if a program is well-typed, then that program does not attempt to e.g. perform arithmetic with strings or index into numbers. In other words, the program does not attempt to perform 'illegal' operations. Conversely, the operational semantics of the language is supposed to capture the *legal* operations.

Furthermore, we analyse well-formed programs into two categories: Those in which there is computation still to be done, and those in which there is not. While we cannot hope that all programs will eventually reach the latter state, it turns out well-typed programs do not get 'stuck'. Either they do indeed terminate in a 'finished' state, or else they run forever. More precisely we have the following:

3.1.1 • Definition: ***Safety.***
An expression e is ***safe*** if for all expressions e_1 and stores σ_1, if $(\perp, e) \to^* (\sigma_1, e_1)$, then either e_1 is a value, or else there is an expression e_2 and a store σ_2 such that $(\sigma_1, e_1) \to (\sigma_2, e_2)$. ▲

As mentioned in §2.1.3(a), the expressions that we have singled out as *values* are those corresponding to programs that are 'finished'.

3.1.2. Type safety

We thus wish to show that if an expression is well-typed, then it is safe. This is indeed the case, and we thus also talk about ***type safety***. There are various ways of proving this result, and the classical proof is in two steps:

- We first show that if an expression is well-typed, then it is either a value, or else it can reduce in one step to another expression. That is, it can make ***progress***. For reasons that will become clear, we will not assume that the store is empty, so we also need to assume that the contents of the store is well-typed.

- Next we show that *if* a well-typed expression reduces in one step to another expression, *then* that expression is also well-typed, so that well-typedness is ***preserved*** by the operational semantics. Again we need to take care that well-typedness of the store is well-typed.

Taken together, these will imply the main theorem:

3.1.2 • Theorem: ***Type safety.***
If an expression e is well-typed, i.e. if $\vdash e : \tau$, then e is safe. ■

More precisely, this will follow immediately from Theorem 3.2.1 and Theorem 3.3.1 below.

3.2 ◇ Progress

As mentioned above, it is not enough that an expression is well-typed for it to make progress, the store must also be well-typed. A store $\sigma : Loc \rightharpoonup_\omega Exp$ is ***well-typed*** with respect to Ξ, Γ and Σ if $\operatorname{dom}\sigma = \operatorname{dom}\Sigma$ and $\Xi \mid \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$ for all $l \in \operatorname{dom}\sigma$. In this case we write $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$, and just as for the typing relation we omit Ξ and Γ if they are empty.

There are at least a couple of ways to approach proving the progress theorem. Most naturally the proof goes by induction on the typing relation, since the claim concerns well-typed expressions (or more precisely typing judgments, i.e., elements of the typing relation). Since our typing rules are syntax-directed in the sense that there is a single typing rule (schema) for each type of expression, we could also prove the theorem by induction on the structure of expressions, but this does not generalise as readily and the proof is no simpler—indeed, quite the contrary.

We are now ready to prove the progress theorem. The reader may wish to read in parallel the proof and the remark immediately following it.

3.2.1 • Theorem: ***Progress.***
If $\Sigma \vdash e : \tau$, then either e is a value or else, for any store σ with $\Sigma \vdash \sigma$, there exists an expression e' and a store σ' such that $(\sigma, e) \to (\sigma', e')$.

Augmenting the claim in this way ensures that we can perform the induction on the entire 5-ary relation, which is important since this relation is the one that is defined by the inference rules, *not* the corresponding ternary relation obtained by restricting the 5-ary relation to the subset where $\Xi = \emptyset$ and $\Gamma = \bot$.

Proof. The proof is by rule induction on the typing relation $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$, but the claim to be proved is augmented by 'or either $\Xi \neq \emptyset$ or $\Gamma \neq \bot$'. Hence we only need to prove each case when $\Xi = \emptyset$ and $\Gamma = \bot$, since otherwise the claim trivially holds for the conclusion of each rule. Furthermore, since the store is relevant for only a few reductions, we suppress it from the notation in most of the cases below, simply taking about expressions reducing to other expressions and not distinguishing between the reductions $\to_p$ and $\to_h$.[1]

[1] We are essentially restricting to the *pure* subset of the language so the subscript p is redundant, and we thus use $\to_h$ to denote the one-step reduction.

T-VAR: Since we may assume that $\Gamma = \bot$, this is vacuously true.

T-UNIT: Since 1 is a value, this follows.

T-PAIR: Assume that the claim holds for $\Sigma \vdash e_1 : \tau_1$ and $\Sigma \vdash e_2 : \tau_2$. If both e_1 and e_2 are values, then $\langle e_1, e_2 \rangle$ is also a value, so assume that only $e_1 = v_1$ is a value and that $e_2 \to e_2'$. Since the only rule that generates the reduction $\to$ is E-HEAD, it follows that e_2 is on the form $E[d_2]$ and e_2' is on the form $E[d_2']$, where E is an evaluation context and d_2 and d_2' are expressions such that $d_2 \to_h d_2'$. Letting $E' = \langle v_1, E \rangle$ it follows that $\langle v_1, e_2 \rangle = E'[d_2]$ and $\langle v_1, e_2' \rangle = E'[d_2']$, and so

$$\langle v_1, e_2 \rangle = E'[d_2] \to E'[d_2'] = \langle v_1, e_2' \rangle$$

by E-HEAD. If instead e_1 is not a value, then the same argument (using the evaluation context $\langle E, e_2 \rangle$) yields the same result.

T-PROJ$_1$ *and* T-PROJ$_2$: Assume that the claim holds for $\Sigma \vdash e : \tau_1 \times \tau_2$. If e is a value, then it is on the form $\langle v_1, v_2 \rangle$ by Lemma 2.4.4(ii), where v_1 and v_2 are values. Hence $\pi_1\, e = \pi_1 \langle v_1, v_2 \rangle$, and this reduces to v_1 via $\to_h$ by E-PROJ$_1$. Choosing the evaluation context $E = -$, E-HEAD implies that $\pi_1 \langle v_1, v_2 \rangle \to v_1$. If instead e is not a value, then by induction there is some e' such that $e \to e'$. Hence there are expressions d and d' and an evaluation context E such that $e = E[d]$, $e' = E[d']$ and $d \to_h d'$. Letting $E' = \pi_1\, E$ we have

$$\pi_1\, e = E'[d] \to E'[d'] = \pi_1\, e',$$

as desired. The case T-PROJ$_2$ is analogous.

T-INJ$_1$ *and* T-INJ$_2$: Assume that the claim holds for $\Sigma \vdash e : \tau_1$. If e is a value v, then so is $\iota_1\, v$. If instead $e \to e'$, then as before $e = E[d]$, $e' = E[d']$ and $d \to_h d'$. Letting $E' = \iota_1\, E$ we get $\iota_1\, e = E'[d]$ and $\iota_1\, e' = E'[d']$, so $\iota_1\, e \to \iota_1\, e'$. The case T-INJ$_2$ is similar.

T-MATCH: Assume that the claim holds for[2] $\Sigma \vdash e : \tau_1 + \tau_2$. If e is a value, then by Lemma 2.4.4(iii) it must be on the form $\iota_1\, v$ or $\iota_2\, v$ for a value v. Hence the expression $\mathsf{match}(e, x, e_1, e_2)$ can reduce by either E-MATCH-INJ$_1$ or E-MATCH-INJ$_2$, so it reduces by E-HEAD (using the evaluation context $E = -$). If instead $e \to e'$, then by the same argument as in previous cases with $E' = \mathsf{match}(E, x, e_1, e_2)$, it follows that $\mathsf{match}(e, x, e_1, e_2)$ reduces.

[2]Notice that while the typing rule T-MATCH has multiple hypotheses, only the first one is relevant for progress, since the only evaluation contexts whose outermost operators are match operators are on the form $\mathsf{match}(E, x, e_1, e_2)$. Hence the other two hypotheses cannot affect the one-step reduction of a match expression.

T-REC: This is obvious since $\mathsf{rec}\, f(x) \coloneqq e$ is a value.

T-APP: Assume that the claim holds for $\Sigma \vdash e_1 : \tau_1$ and $\Sigma \vdash e_2 : \tau_1 \to \tau_2$. If e_2 is a value, then by Lemma 2.4.4(iv) it must be on the form $\mathsf{rec}\, f(x) \coloneqq e$. If also e_1 is a value, then the claim follows by E-REC-APP. If $e_2 = v$ is a value but e_1 is not, then $e_1 \to e_1'$. The same argument as in previous cases with $E' = v\, E$ shows that $v\, e_1$ reduces. Finally, if e_2 is not a value, then $e_2 \to e_2'$, and choosing $E' = E\, e_1$ proves the claim.

T-TLAM: This is obvious since $\Lambda\_.e$ is a value.

T-TAPP: Assume that the claim holds for $\Sigma \vdash e : \forall \alpha.\tau$. If e is a value, then by Lemma 2.4.4(v) it must be on the form $\Lambda\_.e'$, so the claim follows from E-TAPP-TLAM (via E-HEAD using the evaluation context $E = -$). If e is not a value, then $e \longrightarrow e'$ for some expression e' by induction. Hence there are expressions d and d' such that $d \longrightarrow_h d'$, and such that $e = E[d]$ and $e' = E[d']$ for some evaluation context E. Letting $E' = E\,\_$ we thus have $e\,\_ = E'[d]$ and $e'\,\_ = E'[d']$, proving the claim.

T-PACK: Assume that the claim holds for $\Sigma \vdash e : \tau[\tau'/\alpha]$. If e is a value, then so is $\mathsf{pack}\, e$. Otherwise $e \longrightarrow e'$ for some expression e' by induction. The same argument as before using the evaluation context $\mathsf{pack}\, E$ for an appropriate E yields the claim.

T-UNPACK: Assume that the claim holds for[3] $\Sigma \vdash e_1 : \exists\alpha.\tau$. If e_1 is a value, then it must be on the form $\mathsf{pack}\, v$ by Lemma 2.4.4(vi), so an application of E-UNPACK-PACK yields the claim. Otherwise $e \longrightarrow e'$ for some e', and we use the evaluation context $\mathsf{unpack}(E, x, e_2)$ for an appropriate E.

[3]As in the case T-MATCH mentioned above, only the first hypothesis is relevant.

T-FOLD: Assume that the claim holds for $\Sigma \vdash e : \tau[\mu\alpha.\tau/\alpha]$. If e is a value then so is $\mathsf{fold}\, e$; otherwise $e \longrightarrow e'$ for some e' by induction, and it follows that $\mathsf{fold}\, e \longrightarrow \mathsf{fold}\, e$.

T-UNFOLD: Assume that the claim holds for $\Sigma \vdash e : \mu\alpha.\tau$. If e is a value then Lemma 2.4.4(vii) implies that $e = \mathsf{fold}\, v$, and the claim follows by E-UNFOLD-FOLD. Otherwise $e \longrightarrow e'$, and so also $\mathsf{unfold}\, e \longrightarrow \mathsf{unfold}\, e'$.

T-LOC: Locations are values, so this is obvious.

T-ALLOC: Assume that the claim holds for $\Sigma \vdash e : \tau$. If e is a value, then the claim follows by applying E-ALLOC, noting that there always exists a location $l \notin \operatorname{dom}\sigma$. Otherwise there is an expression e' and a store σ' such that $(\sigma, e) \longrightarrow (\sigma', e')$ by induction. But then there is an evaluation context E and expressions d and d' such that $e = E[d]$, $e' = E[d']$ and $(\sigma, d) \longrightarrow_h (\sigma', d')$. Letting $E' = \mathsf{ref} E$ we have $\mathsf{ref} e = E'[d]$ and $\mathsf{ref} e' = E'[d']$, so E-HEAD implies that $(\sigma, \mathsf{ref}\, e) \longrightarrow (\sigma', \mathsf{ref}\, e')$.

T-STORE: Assume that the claim holds for $\Sigma \vdash e_1 : \mathsf{Ref}\,\tau$ and $\Sigma \vdash e_2 : \tau$, and let σ be a store with $\Sigma \vdash \sigma$. If e_1 is a value, then by Lemma 2.4.4(viii) it is a location l, and by Lemma 2.4.3(xv) we have $l \in \operatorname{dom}\Sigma = \operatorname{dom}\sigma$. If e_2 is also a value, then the claim follows from E-STORE. If $e_1 = l$ is a value but e_2 is not, then there are some e_2' and σ' such that $(\sigma, e_2) \longrightarrow (\sigma', e_2')$. Again writing $e_2 = E[d_2]$ and $e_2' = E[d_2']$ with $(\sigma, d) \longrightarrow_h (\sigma', d')$, we use the evaluation context $l := E$. Finally, if e_1 is not a value, then the same argument using instead $E := e_2$ yields the claim.

T-LOAD: Assume that the claim holds for $\Sigma \vdash e : \mathsf{Ref}\,\tau$, and let σ be a store with $\Sigma \vdash \sigma$. If e is a value, then as before it is a location l, and $l \in \operatorname{dom}\sigma$. It then follows from E-LOAD that $(\sigma, !\,l) \longrightarrow_h (\sigma, v)$, where $v = \sigma(l)$. If instead $(\sigma, e) \longrightarrow (\sigma', e')$ for an expression e' and a store σ', then we simply use the evaluation context $!\,E$ for an appropriate E. ∎

Let us pause to consider how each case in the progress theorem was proved.

- Some cases (T-PAIR, T-INJ$_1$, T-INJ$_2$, T-PACK, T-FOLD and T-ALLOC) are proved directly by induction. That is, the claim for the conclusion of a rule follows directly from the induction hypothesis.

  The former five cases have the property that if the hypotheses contain values, then so does the conclusion. In fact, notice that these are precisely the rules whose conclusions contain expressions that *may* be values but are not necessarily. Hence we only need to consider when the hypotheses do not contain values, but since the (expression in the) conclusion is structurally larger[4] than the hypotheses, we may hope that there is an evaluation context that allows us to transfer reducibility of the hypotheses to the conclusion. And indeed this is the case.

  For the case T-ALLOC, notice that even if v is a value, $\mathsf{ref}\, v$ is not. However, we may also quickly dispense with this case by noting that E-ALLOC allows us to reduce $\mathsf{ref}\, v$ immediately.

- Other cases (T-PROJ$_1$, T-PROJ$_2$, T-MATCH, T-APP, T-TAPP, T-UNPACK, T-UNFOLD, T-STORE and T-LOAD) require an appeal to the canonical forms lemma (cf. Lemma 2.4.4). Notice that these are precisely the rules (except for T-ALLOC discussed above) whose conclusions contain expressions that can never be values. When the expressions in the hypotheses are reducible, then so are the conclusions by an appropriate choice of evaluation context.

  However, when all the hypotheses contain values, this argument of course does not work. Still, the conclusions are not values so there must be some other reason for why they are reducible. Considering the reduction rules, this requires the subexpressions (i.e., the hypotheses of the typing rules) to have particular forms: For instance, for $\pi_1\, v$ to be reducible by E-PROJ$_1$, v must be on the form $\langle v_1, v_2\rangle$. And this is precisely what the canonical forms lemma allows us to conclude.

  Notice also that T-STORE is special in that the proof in this case requires an explicit[5] application of the inversion lemma (cf. Lemma 2.4.3): For note that to apply E-STORE we must know that $l \in \operatorname{dom} \sigma$ which the canonical forms lemma doesn't say anything about[6].

- Still other cases (T-UNIT, T-REC, T-TLAM and T-LOC) are fairly trivial since their conclusions are *always* values.

- The final case is T-VAR, which is also trivial since $\Gamma = \bot$.

[4]We again remind the reader that we have not defined what 'structurally larger' means, which is no matter since the present discussion is purely expository.

[5]As opposed to implicit through the canonical forms lemma

[6]Indeed it cannot, since it says nothing about stores at all, but is only concerned with statics.

3.3 ◇ Preservation

Having proved that *if* an expression is well-typed *then* it is reducible (unless it is a value), we now show that reductions preserve well-typedness.

In our statement of the progress theorem we had to take into account whether the store was well-typed or not, but we did not have reason to consider what actually happens to the store when performing a reduction. Recall from Definition 3.1.1 that safety is a property of expressions and not of machine states (i.e., of store-expression pairs), since we assume that the store is initially empty. But the store can of course change during evaluation, so we need to ensure that not only is well-typedness of expressions preserved by reduction, so is well-typedness of stores. The typing rule T-STORE ensures than types are preserved when we modify already allocated locations in the store.

However, the store can also *grow*, and we must take this into account when we try to make precise the formulation of the preservation theorem. Just because the initial store σ is well-typed with respect to some store typing Σ, it does not follow that the resulting store σ' is also well-typed with respect to Σ. But as mentioned we only need to ensure that the newly allocated locations—i.e., those in $\operatorname{dom}\sigma' \setminus \operatorname{dom}\sigma$—have the correct types. Hence it suffices to find a store typing Σ' that agrees with Σ on $\operatorname{dom}\Sigma$, and with respect to which σ' is well-typed.

Furthermore, notice that to prove that well-typed expressions are safe it suffices to show that reduction preserves *well-typedness*, and not types themselves. We may indeed consider whether it is even true that types are preserved, i.e., that if e reduces to e' and e has some type τ, then e' also has type τ. For some type systems, notably those with subtyping, this is not the case, but for ours it will turn out to be.

While the proof of the progress theorem was by induction on the typing relation, the proof of preservation is somewhat different. Pierce (2002, Theorem 8.3.3) proves preservation for a simple language by induction on *derivations* of typing judgments, but this of course requires one to have a notion of derivations (and subderivations) of judgments, which is unnecessary and overly complicates the proof[7].

[7]In the author's opinion, though Harper (2016, §6.1) seems to share this sentiment.

Another approach is proof by induction on the reduction relation, which is the approach taken by Harper (2016, Theorem 6.2). The operational semantics of the language studied by Harper is formalised as an ordinary structural dynamics, in which the reduction relation is in fact defined recursively. However, since the dynamics of our language is *contextual* (in that it is described using evaluation contexts), a variation of this approach is more appropriate.

We prove the preservation theorem in three steps as follows:

- Evaluation contexts 'reflect'[8] well-typedness Lemma 3.3.2: If $E[e]$ is well-typed, then so is e. This is a straightforward corollary of the inversion lemma.

[8]Compare reflection of limits by functors.

- Evaluation contexts preserve types (Lemma 3.3.3): If e and e' have the same type, then so do $E[e]$ and $E[e']$. This follows directly by rule induction in E. Combined with the above result, this allows us to reduce the proof to the case where e reduces to e' in a single head reduction step.

- Head reduction preserve types (Lemma 3.3.4.): If $e \to_h e'$ and e has type τ, then so does e'. The proof of this step consists of judicious application of the inversion lemma along with various technical results on the interplay between typing and substitution.

We delay the proofs of these results and first show more precisely how they imply preservation:

3.3.1 • Theorem: ***Preservation.***
If

$$\Xi \mid \Gamma \mid \Sigma \vdash e : \tau, \quad \Xi \mid \Gamma \mid \Sigma \vdash \sigma \quad \textit{and} \quad (\sigma, e) \to (\sigma', e'),$$

then there exists some store typing Σ' with $\Sigma \subseteq \Sigma'$ such that

$$\Xi \mid \Gamma \mid \Sigma' \vdash e' : \tau \quad \textit{and} \quad \Xi \mid \Gamma \mid \Sigma' \vdash \sigma'.$$

Proof. By definition of the reduction relation, there exist an evaluation context E and expressions d and d' such that $e = E[d]$, $e' = E[d']$, and $(\sigma, d) \to_h (\sigma', d')$. By Lemma 3.3.2 there is some type τ' such that $\Xi \mid \Gamma \mid \Sigma \vdash d : \tau'$. Next it follows from Lemma 3.3.4 that $\Xi \mid \Gamma \mid \Sigma' \vdash d' : \tau'$ for some store typing Σ' with $\Sigma \subseteq \Sigma'$ and $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$. By Lemma 2.4.5 we also have $\Xi \mid \Gamma \mid \Sigma' \vdash d : \tau'$, so it follows from Lemma 3.3.3 that $\Xi \mid \Gamma \mid \Sigma' \vdash E[d'] : \tau$ as desired. ∎

3.3.2 • Lemma. *If E is an evaluation context, e is an expression and $\Xi \mid \Gamma \mid \Sigma \vdash E[e] : \tau$, then $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau'$ for some type τ'.*

Proof. The proof is by rule induction on E. If $E = -$, then the claim is obvious, since then $E[e] = e$. Hence we assume that E is obtained from an evaluation context E' by some application of an inference rule, so that the induction hypothesis holds for E'. The inductive step is identical in all cases, so we illustrate the argument in the case $E = \langle E', e' \rangle$.

In this case $E[e] = \langle E'[e], e' \rangle$, and Lemma 2.4.3(iii) implies that $\Xi \mid \Gamma \mid \Sigma \vdash E'[e] : \tau''$ for some type τ''. By induction we thus have $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau'$ for some τ'. ∎

3.3.3 • Lemma. *If $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau$ for the same type τ, then $\Xi \mid \Gamma \mid \Sigma \vdash E[e] : \tau'$ and $\Xi \mid \Gamma \mid \Sigma \vdash E[e'] : \tau'$ for the same type τ'.*

Proof. The proof is a straightforward rule induction in E: Simply apply the typing rules corresponding to each evaluation context. ∎

3.3.4 • Lemma: ***Preservation for head reduction.***
If

$$\Xi \mid \Gamma \mid \Sigma \vdash e : \tau, \quad \Xi \mid \Gamma \mid \Sigma \vdash \sigma \quad \textit{and} \quad (\sigma, e) \to_h (\sigma', e'),$$

then there exists some store typing Σ' with $\Sigma \subseteq \Sigma'$ such that

$$\Xi \mid \Gamma \mid \Sigma' \vdash e' : \tau \quad \textit{and} \quad \Xi \mid \Gamma \mid \Sigma' \vdash \sigma'.$$

Proof. We simply check all cases.

E-PROJ$_1$ *and* E-PROJ$_2$: Assume that $e = \pi_1 \langle v_1, v_2 \rangle$ and $e' = v_1$ for values v_1 and v_2. Then Lemma 2.4.3(iv) implies first that $\Xi \mid \Gamma \mid \Sigma \vdash \langle v_1, v_2 \rangle : \tau \times \tau_2$ for some type τ_2, and then Lemma 2.4.3(iii) implies[9] that $\Xi \mid \Gamma \mid \Sigma \vdash v_1 : \tau$. Similarly if $e = \pi_2 \langle v_1, v_2 \rangle$ and $e' = v_2$

[9]That is, Lemma 2.4.3(iii) implies that $\tau \times \tau_2 = \tau_3 \times \tau_4$ and $\Xi \mid \Gamma \mid \Sigma \vdash v_1 : \tau_3$ for some types τ_3 and τ_4. But we of course have $\tau = \tau_3$.

E-MATCH-INJ$_1$ *and* E-MATCH-INJ$_2$: Assume that $e = \mathsf{match}(\iota_1\, v, x, e_1, e_2)$ and $e' = e_1[v/x]$. Then Lemma 2.4.3(vi) implies that $\Xi \mid \Gamma \mid \Sigma \vdash \iota_1\, v : \tau_1 + \tau_2$ and $\Xi \mid \Gamma, x : \tau_1 \mid \Sigma \vdash e_1 : \tau$. Next Lemma 2.4.3(v) yields $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau_1$, so Lemma 2.4.7 implies that $\Xi \mid \Gamma \mid \Sigma \vdash e_1[v/x] : \tau$. Similarly if $\mathsf{match}(\iota_2\, v, x, e_1, e_2)$ and $e' = e_2[v/x]$.

E-REC-APP: Assume that

$$e = (\mathsf{rec}\, f(x) := e'')\, v \quad \text{and} \quad e' = e''[(\mathsf{rec}\, f(x) := e'')/f][v/x].$$

Then Lemma 2.4.3(viii) first implies that $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau_1$ and $\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{rec}\, f(x) := e'' : \tau_1 \to \tau$, and Lemma 2.4.3(vii) applied to the latter judgment implies that $\Xi \mid \Gamma, f : \tau_1 \to \tau, x : \tau_1 \mid \Sigma \vdash e'' : \tau$. But then two applications of Lemma 2.4.7 imply that[10] $\Xi \mid \Gamma \mid \Sigma \vdash e''[(\mathsf{rec}\, f(x) := e'')/f][v/x] : \tau$ as desired.

[10]Strictly speaking, the first application of Lemma 2.4.7 requires that $\Xi \mid \Gamma, x : \tau_1 \mid \Sigma \vdash \mathsf{rec}\, f(x) := e'' : \tau_1 \to \tau$, but this follows by Lemma 2.4.5. Notice that there is no issue of well-formedness of τ_1, since $\tau_1 \to \tau$ is already well-formed with respect to Ξ.

E-TAPP-TLAM: Assume that $e = (\Lambda\_.e')\,\_$. Then Lemma 2.4.3(x) first implies that $\tau = \tau'[\tau''/\alpha]$ and $\Xi \mid \Gamma \mid \Sigma \vdash \Lambda\_.e' : \forall \alpha.\tau'$, and an application of Lemma 2.4.3(ix) then yields $\Xi, \alpha \mid \Gamma \mid \Sigma \vdash e' : \tau'$ and $\alpha \notin \Xi$. Next, Proposition 2.4.1 implies that $\Xi \vdash \Gamma$ and $\Xi \vdash \Sigma$, and also that $\Xi \vdash \tau$.

Assume that α is free in τ. Then also $\Xi \vdash \tau''$, so Lemma 2.4.8 implies that $\Xi \mid \Gamma[\tau''/\alpha] \mid \Sigma[\tau''/\alpha] \vdash e' : \tau'[\tau''/\alpha]$. But α is not free in either Γ or Σ, and $\tau = \tau'[\tau''/\alpha]$, so this implies that $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau$ as desired.

If instead α is not free in τ, then $\tau = \tau'$. Since Γ, Σ and τ are well-formed with respect to Ξ, the claim follows from Lemma 2.4.6.

E-UNPACK-PACK: Assume that

$$e = \mathsf{unpack}(\mathsf{pack}\, v, x, e_2) \quad \text{and} \quad e' = e_2[v/x].$$

Then Lemma 2.4.3(xii) implies that $\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\, v : \exists \alpha.\tau'$ and that $\Xi, \alpha \mid \Gamma, x : \tau' \mid \Sigma \vdash e_2 : \tau$. Next Lemma 2.4.3(xi) applied to the first judgment yields[11] $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau'[\tau''/\alpha]$.

[11]Strictly speaking, the inversion lemma first says that $\mathsf{pack}\, v$ has type $\exists \beta.\rho$ for some type ρ, and then that v has type $\rho[\rho'/\beta]$ for some ρ'. But since we already know that $\mathsf{pack}\, v$ has type $\exists \alpha.\tau'$, this implies that v has type $\tau'[\tau''/\alpha]$.

We then claim that $\Xi \mid \Gamma, x : \tau'[\tau''/\alpha] \mid \Sigma \vdash e_2 : \tau$. If α is not free in τ', then $\tau'[\tau''/\alpha] = \tau'$ so this follows from Lemma 2.4.6, and if α is free in τ', then since $\Xi \vdash \tau'[\tau''/\alpha]$ we also have $\Xi \vdash \tau''$, so the claim follows from Lemma 2.4.8. Finally, since v and x have the same type Lemma 2.4.7 implies that $\Xi \mid \Gamma \mid \Sigma \vdash e_2[v/x] : \tau$ as desired.

E-UNFOLD-FOLD: Assume that

$$e = \mathsf{unfold}\, \mathsf{fold}\, v \quad \text{and} \quad e' = v.$$

Then Lemma 2.4.3(xiv) implies that $\mathsf{unfold}\, \mathsf{fold}\, v$ and $\mathsf{fold}\, v$ have types $\tau'[\mu\alpha.\tau'/\alpha]$ and $\mu\alpha.\tau'$ respectively, and Lemma 2.4.3(xiii) in turn implies that v has type[12] $\tau'[\mu\alpha.\tau'/\alpha]$, as desired.

[12]As in the case E-UNPACK-PACK this is not strictly true.

E-ALLOC: In this case $e = \mathsf{ref}\, v$, $e' = l$, $\sigma' = \sigma[l \mapsto v]$, and $l \notin \operatorname{dom} \sigma$. By Lemma 2.4.3(xvi) we have $\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{ref}\, v : \mathsf{Ref}\, \tau'$ for some τ', and we further have $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau'$. Now letting $\Sigma' = \Sigma[l \mapsto \tau']$, it follows from T-LOC that $\Xi \mid \Gamma \mid \Sigma' \vdash l : \mathsf{Ref}\, \Sigma'(l)$, so we have both $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$ and $\Xi \mid \Gamma \mid \Sigma' \vdash l : \mathsf{Ref}\, \tau'$.

E-STORE: Here $e = l := v$, $e' = \mathtt{1}$ and $\sigma' = \sigma[l \mapsto v]$ with $l \in \operatorname{dom} \sigma$. Notice first that $l := v$ and $\mathtt{1}$ both have type $\mathtt{1}$ by Lemma 2.4.3(xvii) and (ii), so that $\Xi \mid \Gamma \mid \Sigma \vdash \mathtt{1} : \mathtt{1}$ as required.

By inversion we also have $\Xi \mid \Gamma \mid \Sigma \vdash l : \mathsf{Ref}\, \tau'$ and $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau'$ for some type τ', and Lemma 2.4.3(xv) implies that $\tau' = \Sigma(l)$. Furthermore, since $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$, we have $\Xi \mid \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$. Since $v = \sigma'(l)$ it thus follows that $\Xi \mid \Gamma \mid \Sigma \vdash \sigma'(l) : \Sigma(l)$, so $\Xi \mid \Gamma \mid \Sigma \vdash \sigma'$.

E-LOAD: Here $e = {!}\, l$, $e' = v$ and $\sigma = \sigma'$ with $\sigma(l) = v$. It follows from Lemma 2.4.3(xviii) that $\Xi \mid \Gamma \mid \Sigma \vdash l : \mathsf{Ref}\, \tau$, and Lemma 2.4.3(xv) implies that $\mathsf{Ref}\, \tau = \mathsf{Ref}\, \Sigma(l)$, which in turn yields $\tau = \Sigma(l)$. But since $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$ we have $\Xi \mid \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$, or in other words, $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau$. ■

4 Type Safety II: Logical Predicates

In this chapter we use the method of logical relations, or more property logical *predicates*, to prove type safety for the language **F** we presented in §2.2.

The logical predicates allow us to define an alternative 'semantic' notion of type safety, which turns out to be stronger than the 'syntactic' notion of type safety we studied in the previous chapter. The goal is then to prove that well-typed closed expressions are semantically type safe, which as it turns out is quite simple.

4.1 ⋄ A logical predicate

A ***logical predicate*** is more properly a *type-indexed* predicate on expressions: Roughly speaking, for each type τ there is a predicate, or property, P_τ whose extension is somehow determined by τ. For us the predicate will also strictly speaking depend on a set of type variables and a type context.

In some applications an expression e having the property P_τ requires e to be well-typed with type τ. One such application is the proof of strong normalisation of the simply typed λ-calculus (cf. Pierce 2002, §12.1), in which the predicate P_τ is defined recursively on τ in a fairly straightforward way. Since our goal is precisely to study type safety, we will not be making any assumptions about well-typedness.

4.1.1. Interpretations of syntax

We first define a type-indexed ***expression interpretation***, which will somehow capture those expressions that have a given type. As we will see, this definition will depend on a ***value interpretation***, and this will be defined recursively on types, sometimes also depending on the *expression* interpretations of smaller types.

The definition of the expression interpretation is 'uniform', in the sense that it is defined in the exact same way for all types. On the other hand, the value interpretations are constructed to capture the values of each type individually. For closed types this is straightforward, but since our language includes polymorphic types, we first of all need a way to interpret type variables, and second of all a way to keep track of the interpretations of type variables inside the expression and value interpretations.

Furthermore, it will turn out that value interpretations must include only *closed* values, so we need a way to 'close' values. Since all free variables are assigned types in the type context, we do this by defining a ***context interpretation***.

We will see how to do this below. For now, since the expression and value interpretations are defined by mutual recursion, we fix notation: Let Ξ be a finite set of type variables, τ a type, and Γ a type context. We write $\mathcal{D}[\![\Xi]\!]$ for the interpretation of the type variables in Ξ, and for $\rho \in \mathcal{D}[\![\Xi]\!]$ we write $\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$ and $\mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho$ for the expression and value interpretations respectively of τ with respect to ρ. Finally, the interpretation of Γ is denoted $\mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$. If Ξ is empty, and thus $\rho = \bot$, we also write $\mathcal{E}[\![\tau]\!]$ and similarly for the other interpretations.

The symbol '$\triangleright$' only serves as a delimiter between the set of type variables and the type itself. Another natural choice would be '$\vdash$', but we follow Gunter (1992) in preferring a different symbol, not to confuse it with the '$\vdash$' from various typing relations.

(a) ***Interpretation of type variables.*** We begin by defining the interpretation of type variables, since its definition will turn out to not depend on the definition of the other interpretations.

Since a type variable α is supposed to represent any type whatsoever, if α is not bound then we must *choose* its interpretation, just as we choose the type of variables in a type context Γ. We might hope that we could define the interpretation of α in terms of value interpretations[1]: Simply choose a type τ and let the interpretation of α be the value interpretation of τ, i.e., the set of value of type τ.

[1] Or in terms of *expression* interpretations. But the idea is that a type is determined by its values, so we might as well use the value interpretation.

This is however not a possibility. For recall that we need the interpretation of type variables in order to even define the expression interpretation of universal types. For the recursion to be well-founded, we thus cannot choose τ to be a universal type, which of course contradicts the idea that α should represent an *arbitrary* type.

Instead we 'model' the interpretation of α on value interpretations. The value interpretation of a type is (as it will turn out) just a set of closed values, so we just let α be an arbitrary set of closed values, called a ***semantic type***. We let $\mathit{SemType} := \mathcal{P}(\mathit{ClVal})$ be the set of all such semantic types.

The interpretation of a type variable is thus just a semantic type. More generally, if Ξ is a (usually finite) set of type variables, then we let

$$\mathcal{D}[\![\Xi]\!] := \{\rho \colon \mathit{TypeVar} \rightharpoonup \mathit{SemType} \mid \operatorname{dom}\rho = \Xi\}.$$

That is, an element ρ of $\mathcal{D}[\![\Xi]\!]$ is a map that assigns to each variable in Ξ a semantic type, so that ρ is a simultaneous interpretation of all variables in Ξ.

(b) ***Expression interpretation.*** Let Ξ be a set of type variables, and let $\rho \in \mathcal{D}[\![\Xi]\!]$. We assume below that all types are well-formed with respect to Ξ (cf. §2.1.2(c)). The ***expression interpretation*** of τ is the set $\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$ of closed[2] expressions e with the property that if e' is an irreducible expression and $e \to^* e'$, then $e' \in \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho$. That is,

[2] We make sense of this assumption in §4.1.1(c) below.

$$\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho := \{e \in \mathit{ClExp} \mid \forall e' \in \mathit{Irr} \colon e \to^* e' \Rightarrow e' \in \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho\}.$$

An alternative definition of the expression interpretation is

The original definition is taken from Skorstengaard (2019, §3.2), while this alternative is adapted from Skorstengaard (2019, §4.4).

$$\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho := \{e \in \mathit{ClExp} \mid \exists v \in \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho \colon e \to^* v\}.$$

Compared to the original definition above, this on the one hand postulates the *existence* of a value (belonging to the correct value interpretation) to which e reduces, and on the other it only talks about a *single* such value.

We return to the differences between these two definitions in §5.5.

(c) *Value interpretation.* The ***value interpretation*** of τ will be a set of closed[3] values of our language of type τ. For most types this is simply a case of looking at the syntax in §B.1 and collecting all values of that type.

[3] In the proof of Lemma 4.2.2 we will see why the assumption of closedness is necessary.

It is easy to define the value interpretation for the unit type, and for products and sums:

$$\begin{aligned}
\mathcal{V}[\![\Xi \triangleright \mathbb{1}]\!]_\rho &:= \{1\}, \\
\mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho &:= \{\langle v_1, v_2\rangle \mid v_1 \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho, v_2 \in \mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho\}, \\
\mathcal{V}[\![\Xi \triangleright \tau_1 + \tau_2]\!]_\rho &:= \{\iota_1\, v \mid v \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho\} \cup \{\iota_2\, v \mid v \in \mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho\}.
\end{aligned}$$

Notice that $\mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$ is just the Cartesian product

$$\mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho \times \mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho,$$

where we use the pair operator $\langle -, - \rangle$ as a pairing scheme instead of the usual Kuratowski pairs $(a, b) = \{\{a\}, \{a, b\}\}$.

Similarly, $\mathcal{V}[\![\Xi \triangleright \tau_1 + \tau_2]\!]_\rho$ is the disjoint union

$$\mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho \sqcup \mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho,$$

where the 'tags' are provided by the injections ι_1 and ι_2.

Notice that each value interpretation is well-defined, since they are defined in terms of the value interpretations of strictly smaller types.

For function types things are slightly more complicated: A value of type $\tau_1 \to \tau_2$ is an expression $\lambda x.e$, where e is an expression in which the variable x may be free, and which has type τ_2 if $x : \tau_1$. This means that we may substitute x for expressions of type τ_1, but since our language is call-by-value we only need to consider *values* of type τ_1. That is, e should be such that $e[v/x]$ is an expression of type τ_2 whenever v is a value of type of τ_1:[4]

$$\mathcal{V}[\![\Xi \triangleright \tau_1 \to \tau_2]\!]_\rho := \{\lambda x.e \mid \forall v \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho \colon e[v/x] \in \mathcal{E}[\![\Xi \triangleright \tau_2]\!]_\rho\}.$$

[4] We see here why the expression interpretation had to contain only *closed* expressions: For notice that then the expression $\pi_1 \langle 1, y\rangle$ would lie in $\mathcal{E}[\![\mathbb{1}]\!]$ since it reduces to 1. Hence the open value $\lambda x.\pi_1 \langle 1, y\rangle$ would be an element of $\mathcal{V}[\![\tau \to \mathbb{1}]\!]$, which is not allowed.

This is again well-defined, since $\mathcal{E}[\![\Xi \triangleright \tau_2]\!]_\rho$ is defined directly in terms of $\mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho$, and τ_2 is a strictly smaller type than $\tau_1 \to \tau_2$.

We finally consider type variables and universal types. Recall that we already have an interpretation of type variables as semantic types, so we simply let[5]

$$\mathcal{V}[\![\Xi \triangleright \alpha]\!]_\rho := \rho(\alpha).$$

[5] Recalling that α is assumed to be well-formed with respect to Ξ, which in this case just means that $\alpha \in \Xi$.

Thus we finally see the reason for indexing value (and hence expression) interpretations by ρ. Next, if α is *not* an element of Ξ, then we define

$$\mathcal{V}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho := \{\Lambda\_.e \mid \forall T \in \mathit{SemType} \colon e \in \mathcal{E}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha \mapsto T]}\}.$$

Recall that the set Ξ, α is only well-defined if $\alpha \notin \Xi$. Of course we can always α-convert $\forall\alpha.\tau$ so that this is the case (since Ξ is finite), in which case α may be free in τ.

Let us pause to consider the definition of $\mathcal{V}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho$: A value of type $\forall\alpha.\tau$ is of course on the form $\Lambda\_.e$ for some *expression* e of the proper type. Notice that the only type variable that can be free in the type of e but not in the type of $\Lambda\_.e$ is α, so assuming that we already know how to deal with the free type variables in $\forall\alpha.\tau$—i.e., the variables in Ξ—it suffices to consider how to extend this with an extra type variable. That is, we must consider how to extend an interpretation of Ξ to an interpretation of Ξ, α. But this is easy: If ρ is an interpretation of Ξ, just choose any semantic type T and extend this interpretation to $\rho[\alpha \mapsto T]$.

Notice also that since expressions of our language do not contain types as subexpressions (that is, they do not contain explicit type annotations), they in particular cannot contain type variables. When choosing an interpretation of a type variable we thus do not need to consider how this should affect expressions containing that type variable.

(d) ***Context interpretation.*** In order to close an expression we use a ***value substitution***, which is a finite partial map $\gamma\colon \mathit{Var} \rightharpoonup_\omega \mathit{ClVal}$. Given a type context Γ, the collection of all such γ with $\operatorname{dom}\gamma = \operatorname{dom}\Gamma$ *interprets* Γ, and we call this collection the ***context interpretation*** of Γ. If Ξ is a set of type variables, $\rho \in \mathcal{D}[\![\Xi]\!]$, and Γ is well-formed with respect to Ξ, then we define

$$\mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho := \{\gamma\colon \operatorname{dom}\Gamma \to \mathit{ClVal} \mid \forall x \in \operatorname{dom}\Gamma\colon \gamma(x) \in \mathcal{V}[\![\Xi \triangleright \Gamma(x)]\!]_\rho\}.$$

Given a value substitution $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$, we extend this to a map $\mathit{Exp} \to \mathit{Exp}$ as follows:

We could also use Theorem 1.2.5 to define the extension of γ, but this would not yield the explicit characterisation of γ that we will need.

4.1.1 • Definition. Let $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$. If $\operatorname{dom}\gamma = \{x_1,\dots,x_n\}$ and $e \in \mathit{Exp}$, then we define

$$\gamma(e) := e[\gamma(x_1)/x_1]\cdots[\gamma(x_n)/x_n],$$

yielding a map $\gamma\colon \mathit{Exp} \to \mathit{Exp}$. ▲

That is, we apply γ to e by substituting each variable x_i in the domain of γ with the closed value $\gamma(x_i)$. Notice that this is well-defined since each $\gamma(x_i)$ is closed[6], so the order of the substitutions does not matter. Furthermore, this is indeed an extension of γ since $x_i[\gamma(x_i)/x_i] = \gamma(x_i)$ by the definition of substitution.

[6] Again we rely on an appeal to intuition in lieu of a precise definition of substitution.

4.1.2. Semantic typing

We are finally in a position to define the logical predicate we will study:

4.1.2 • Definition: ***Semantic typing.***
Let Ξ be a finite set of type variables, Γ a type context with $\Xi \vdash \Gamma$, and let $e \in \mathit{Exp}$ and $\tau \in \mathit{Type}$. Then we write

$$\Xi \mid \Gamma \vDash e : \tau$$

if and only if

$$\forall \rho \in \mathcal{D}[\![\Xi]\!], \gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho\colon \gamma(e) \in \mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho.$$ ▲

In case $\Xi = \emptyset$ or $\Gamma = \bot$ we simply write $\Gamma \vDash e : \tau$ or $\vDash e : \tau$ as appropriate. The significance of this definition is captured by the following result:

4.1.3 • Proposition. *If $\vDash e : \tau$, then e is safe.*

Proof. Assuming that $e \longrightarrow^* e'$ we must show that e' is either reducible or a value. If e' is irreducible, then since $e \in \mathcal{E}[\![\tau]\!]$ we have $e' \in \mathcal{V}[\![\tau]\!]$ by the definition of the expression interpretation. In particular, e' is a value as desired. ∎

Hence to prove that the *syntactic* notion of well-typedness implies safety, it suffices to show that syntactic well-typedness implies semantic well-typedness. This is called the ***fundamental property*** of the semantic typing relation, and it is the goal of the rest of this chapter. The proof method is obviously rule induction, and the induction itself it phrased in terms of a series of ***compatibility lemmas***.

4.1.3. Properties of interpretations

Before moving on we note some technical properties of the objects we have defined so far.

4.1.4 • Lemma. (i) $\mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho \subseteq e \in \mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$.

(ii) *If $e \in \mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$ is irreducible, then $e \in \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho$.*

Proof. Both parts follow from the definition of the expression interpretation since values are irreducible (Proposition 2.4.9). ∎

4.1.5 • Lemma. *Let $\Xi \subseteq \Phi$ be finite sets of type variables, and let $\rho \in \mathcal{D}[\![\Xi]\!]$ and $\rho' \in \mathcal{D}[\![\Phi]\!]$ with $\rho \leq \rho'$. Furthermore let τ be a type and Γ a type context that are well-formed with respect to Ξ. Then we have the inclusions*

$$\mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho \subseteq \mathcal{V}[\![\Phi \triangleright \tau]\!]_{\rho'},$$
$$\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho \subseteq \mathcal{E}[\![\Phi \triangleright \tau]\!]_{\rho'},$$
$$\mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho \subseteq \mathcal{G}[\![\Phi \triangleright \Gamma]\!]_{\rho'}$$

Proof. The first two inclusions are proved by induction in τ. This is clear if τ is either $\mathbf{1}$ or a type variable, and the induction step is obvious for product and sum types. It is also easy to see that the induction step for function types holds by considering their value interpretation.

Next consider the type $\forall \alpha.\tau$ with $\alpha \notin \Phi$. The second inclusion follows easily. Furthermore, $\Xi, \alpha \subseteq \Phi, \alpha$ and $\rho[\alpha \mapsto T] \leq \rho'[\alpha \mapsto T]$ for all $T \in$ *SemType*, so the first inclusion also follows.

The third inclusion follows directly from the first. ∎

4.1.6 • Lemma: *Compositionality.*
Let $T = \mathcal{V}[\![\Xi \triangleright \tau']\!]_\rho$, and assume that $\Xi \vdash \tau'$. Then

$$\mathcal{V}[\![\Xi \triangleright \tau[\tau'/\alpha]]\!]_\rho = \mathcal{V}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha \mapsto T]},$$
$$\mathcal{E}[\![\Xi \triangleright \tau[\tau'/\alpha]]\!]_\rho = \mathcal{E}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha \mapsto T]}.$$

Proof. The proof is by induction in τ. The only case that is nontrivial is the case $\forall\beta.\tau$ with $\beta \neq \alpha$ and $\beta \notin \Xi$. Since τ' is well-formed with respect to Ξ, β is not free in τ'. Hence $(\forall\beta.\tau)[\tau'/\alpha] = \forall\beta.\tau[\tau'/\alpha]$, and so

$$\begin{aligned}
\mathcal{V}[\![\Xi \triangleright (\forall\beta.\tau)[\tau'/\alpha]]\!]_\rho &= \mathcal{V}[\![\Xi \triangleright \forall\beta.\tau[\tau'/\alpha]]\!]_\rho \\
&= \left\{ \Lambda\_.e \;\middle|\; \begin{array}{l} \forall S \in \mathit{SemType}: \\ e \in \mathcal{E}[\![\Xi, \beta \triangleright \tau[\tau'/\alpha]]\!]_{\rho[\beta\mapsto S]} \end{array} \right\} \\
&= \left\{ \Lambda\_.e \;\middle|\; \begin{array}{l} \forall S \in \mathit{SemType}: \\ e \in \mathcal{E}[\![\Xi, \beta, \alpha \triangleright \tau]\!]_{\rho[\beta\mapsto S, \alpha\mapsto T]} \end{array} \right\} \\
&= \mathcal{V}[\![\Xi, \alpha \triangleright \forall\beta.\tau]\!]_{\rho[\alpha\mapsto T]}. \qquad \blacksquare
\end{aligned}$$

4.2 ⋄ Compatibility

The reader that is less interested in details or is prepared to take things purely on intuition may feel free to skip the following lemma and only consider the cases T-VAR, T-MATCH, T-LAM, T-TLAM and T-TAPP of Lemma 4.2.2.

4.2.1 • LEMMA. (i) *If $\langle e_1, e_2\rangle \longrightarrow^* e'$, then there are expressions e_1' and e_2' such that $e' = \langle e_1', e_2'\rangle$, and such that $e_i \longrightarrow^* e_i'$.*

(ii) *If $\pi_1\, e \longrightarrow^* e'$, then either $e' = v_1$ is a value and $e \longrightarrow^* \langle v_1, v_2\rangle$, or else $e' = \pi_1\, e''$ such that $e \longrightarrow^* e''$.*

(iii) *If $e_1\, e_2 \longrightarrow^* e'$, then either $e_1\, e_2 \longrightarrow^* e''[v/x] \longrightarrow^* e'$ where $e_1 \longrightarrow^* \lambda x.e''$ and $e_2 \longrightarrow^* v$, or else $e' = e_1'\, e_2'$ where $e_i \longrightarrow^* e_i'$.*

Proof. *Proof of (i)*: The proof is by induction on the length of the reduction $\langle e_1, e_2\rangle \longrightarrow^* e'$, it suffices to prove the claim when this is a one-step reduction. There exists an evaluation context E and expressions d and d' such that $\langle e_1, e_2\rangle = E[d]$ and $e' = E[d']$, and such that $d \longrightarrow_h d'$. Notice that E must either be on the form $\langle E', e''\rangle$ or $\langle v, E'\rangle$ for an evaluation context E', an expression e'' and a value v. In the former case we have $E[d] = \langle E'[d], e''\rangle$, so we must have $e_1' = E'[d]$ and $e_2' = e''$. We similarly have $e' = \langle E'[d'], e''\rangle$, and we notice that $E'[d] \longrightarrow^* E'[d']$ (indeed this happens in one step) and $e'' \longrightarrow^* e''$ as desired. If instead $E = \langle v, E'\rangle$, then the argument is similar.

Proof of (ii): The proof is by induction on the length n of the reduction $\pi_1\, e \longrightarrow^* e'$. For $n = 0$ we have $e' = \pi_1\, e$ and $e \longrightarrow^* e$, so the claim holds. Assuming that it holds for some n, suppose that $\pi_1\, e \longrightarrow^n e_1' \longrightarrow e_2'$. Then e_1' cannot be a value since values are irreducible by Proposition 2.4.9, so $e_1' = \pi_1\, e_1''$ with $e \longrightarrow^* e_1''$ by induction. Now, $e_1' = E[d_1]$ and $e_2' = E[d_2]$ with $d_1 \longrightarrow_h d_2$, where E is either the hole or on the form $\pi_1\, E'$. If E is the hole, then $\pi_1\, e_1'' \longrightarrow_h e_2'$, which is only possible if $e_1'' = \langle v_1, v_2\rangle$ and $e_2' = v_1$. In this case we thus indeed have $e \longrightarrow^* \langle v_1, v_2\rangle$. If instead $E = \pi_1\, E'$, then $e_1' = \pi_1\, E'[d_1]$ and $e_2' = \pi_1\, E'[d_2]$, the first of which implies that

$e_1'' = E'[d]$ by the induction hypothesis. But since $d_1 \to_h d_2$ we also have $E'[d_1] \to E'[d_2]$, and so $e \to^* e_1'' = E'[d_1] \to E'[d_2]$ as desired.

Proof of (iii): By induction on the length n of the reduction $e_1 e_2 \to^* e'$. If $n = 0$ then the claim is obvious, so assume that is holds for some n and that $e_1\, e_2 \to^n e' \to e'''$. We consider each disjunct in the induction hypothesis: First assume that $e_1\, e_2 \to^* e''[v/x] \to^* e' \to e'''$, where $e_1 \to^* \lambda x.e''$ and $e_2 \to^* v$. Then we have $e''[v/x] \to^* e'''$, proving the claim.

Instead assume that $e' = e_1'\, e_2'$ and $e_i \to^* e_i'$. Then $e_1'\, e_2' \to e'''$, so $e_1'\, e_2' = E[d]$ and $e''' = E[d']$ with $d \to_h d'$, and E is either the hole or on one of the forms $E'\, e_2''$ and $v_1\, E'$. If E is the hole, then we must have $e_1' = \lambda x.e''$ and $e_2' = v_2$, in which case $e''' = e''[v_2/x]$. If instead $E = E'\, e_2''$, then $e_1'\, e_2' = E'[d]\, e_2''$ and $e''' = E'[d']\, e_2''$, implying that $e_1' = E'[d] \to E'[d']$ and $e_2' = e_2''$ as desired. The final case is similar. ■

4.2.2 • Lemma: *Compatibility.*
The semantic typing relation $\vDash$ satisfies all inference rules in §B.2 pertaining to the language **F**.[7]

[7] We trust that the reader is able to discern which inference rules are relevant.

Proof. T-var: Assume that $\Xi \vdash \Gamma$ and that $\Gamma(x) = \tau$. Let $\rho \in \mathcal{D}[\![\Xi]\!]$ and $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$. By definition of the context interpretation we have $\gamma(x) \in \mathcal{V}[\![\Xi \triangleright \Gamma(x)]\!]_\rho = \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho$, which implies that $\gamma(x) \in \mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$ by Lemma 4.1.4(i).

T-unit: Since 1 is a closed value, this follows immediately from Lemma 4.1.4(i).

T-pair: Let $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$, and let e' be an irreducible expression such that $\langle \gamma(e_1), \gamma(e_2)\rangle = \gamma(\langle e_1, e_2\rangle) \to^* e'$. By Lemma 4.2.1(i) this implies that $e' = \langle e_1', e_2'\rangle$ for appropriate expressions e_1' and e_2', and furthermore that $\gamma(e_1) \to^* e_1'$ and $\gamma(e_2) \to^* e_2'$. By induction we then have $\gamma(e_i) \in \mathcal{E}[\![\Xi \triangleright \tau_i]\!]_\rho$. Notice that e_1' and e_2' are both irreducible since e' is,[8] so $e_i' \in \mathcal{V}[\![\Xi \triangleright \tau_i]\!]_\rho$ by definition of expression interpretations. But then $e' = \langle e_1', e_2'\rangle \in \mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$, which implies that $\gamma(\langle e_1, e_2\rangle) \in \mathcal{E}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$ as desired.

[8] Some irreducible expressions have reducible subexpressions, but in this case, if either e_1' or e_2' were reducible then e' would clearly also be.

T-proj$_1$ *and* T-proj$_2$: Both cases are proved in the same way, so we only prove the case T-proj$_1$.

Assume that $\Xi \mid \Gamma \vDash e : \tau_1 \times \tau_2$, let $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$, and let e' be irreducible such that $\pi_1\, \gamma(e) = \gamma(\pi_1\, e) \to^* e'$. We consider each case of Lemma 4.2.1(ii): First assume that $e' = v_1$ is a value and $\gamma(e) \to^* \langle v_1, v_2\rangle$. Since $\langle v_1, v_2\rangle$ is a value, and hence irreducible by Proposition 2.4.9, the hypothesis implies that[9] $\langle v_1, v_2\rangle \in \mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$. It follows that $v_1 \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho$, so $\gamma(\pi_1\, e) \in \mathcal{E}[\![\Xi \triangleright \tau_1]\!]_\rho$ as desired.

[9] Note that we cannot conclude directly that $\langle v_1, v_2\rangle \in \mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$ by using that the v_i are values of type τ_i, since this requires that types are preserved under reductions. Of course we know this from Theorem 3.3.1, but we do not want to assume this result here.

Next assume that $e' = \pi_1\, e''$ and that $\gamma(e) \to^* e''$. If e'' is reducible then so is $\pi_1\, e''$, so assume that e'' is irreducible. The hypothesis then implies that $e'' \in \mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$, which means that e'' is on the form $\langle v_1, v_2\rangle$ with $v_i \in \mathcal{V}[\![\Xi \triangleright \tau_i]\!]_\rho$. But then $e' = \pi_1\, \langle v_1, v_2\rangle$ reduces to v_1, which is a contradiction.

T-INJ$_1$ *and* T-INJ$_2$: These cases are almost identical to T-PAIR, so we omit it.

T-MATCH: This case is similar to T-PROJ$_1$ and T-PROJ$_2$, so we only sketch the proof. Assume that $\Xi \mid \Gamma \vDash e : \tau_1 + \tau_2$ and that $\Xi \mid \Gamma, x : \tau_i \vDash e_i : \tau$ for $i \in \{1,2\}$. Choosing $x \notin \operatorname{dom}\Gamma$ we have

$$\gamma\Big(\mathsf{match}(e, x, e_1, e_2)\Big) = \mathsf{match}(\gamma(e), x, \gamma(e_1), \gamma(e_2)).$$

When this reduces[10] we first reduce $\gamma(e)$, and by induction this reduces to an element of $\mathcal{V}[\![\Xi \triangleright \tau_1 + \tau_2]\!]_\rho$, i.e., an expression on the form $\iota_1\, v$ or $\iota_2\, v$ for a closed value v. Applying E-MATCH-INJ$_1$ or E-MATCH-INJ$_2$ we obtain either $\gamma(e_1)[v/x]$ or $\gamma(e_2)[v/x]$, and these reduce[11] to an element of $\mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho$ by induction, as desired.

[10]Here we appeal to the reader's intuition instead of proving an appropriate version of Lemma 4.2.1.

[11]Here we use that v is a *closed* value so that we may commute the substitution $x \mapsto v$ with the substitutions defining $\gamma(e_i)$. Notice that the same assumption is needed in the case T-LAM below.

T-LAM: Let $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$, and let e' be an irreducible expression such that $\gamma(\lambda x.e) \to^* e'$. Notice that by choosing $x \notin \operatorname{dom}\Gamma$ we have $\gamma(\lambda x.e) = \lambda x.\gamma(e)$. Since $\lambda x.\gamma(e)$ is a value it is irreducible by Proposition 2.4.9, so $e' = \lambda x.\gamma(e)$. To show that this lies in $\mathcal{V}[\![\Xi \triangleright \tau_1 \to \tau_2]\!]_\rho$, let $v \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho$ and notice that

$$\gamma(e)[v/x] = \gamma[x \mapsto v](e) \in \mathcal{E}[\![\Xi \triangleright \tau_2]\!]_\rho$$

by the induction hypothesis, since $\gamma[x \mapsto v] \in \mathcal{G}[\![\Xi \triangleright \Gamma, x : \tau_1]\!]_\rho$. Thus $\gamma(\lambda x.e) \in \mathcal{E}[\![\Xi \triangleright \tau_1 \to \tau_2]\!]_\rho$ as desired.

T-APP: Let $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$, and let e' be an irreducible expression such that $\gamma(e_1)\,\gamma(e_2) = \gamma(e_1\, e_2) \to^* e'$. We consider in turn each case of Lemma 4.2.1(iii): First assume that $\gamma(e_1)\,\gamma(e_2) \to^* e''[v/x] \to^* e'$ where $\gamma(e_1) \to^* \lambda x.e''$ and $\gamma(e_2) \to^* v$. Each of the expressions on the right-hand sides are values and hence irreducible by Proposition 2.4.9, so the hypothesis implies that they lie in $\mathcal{V}[\![\Xi \triangleright \tau_1 \to \tau_2]\!]_\rho$ and $\mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho$ respectively. But then $e''[v/x] \in \mathcal{E}[\![\Xi \triangleright \tau_2]\!]_\rho$, so since e' is irreducible it lies in $\mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho$ as desired.

Instead assume that $e' = e_1' e_2'$ where $\gamma(e_i) \to^* e_i'$. Since e' is irreducible, then so are the e_i', so the hypothesis implies that $e_1' \in \mathcal{V}[\![\Xi \triangleright \tau_1 \to \tau_2]\!]_\rho$ and $e_2' \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho$. Hence e_1' is on the form $\lambda x.e_1''$ and e_2' is a value. But then $e_1'\, e_2'$ is reducible, which is a contradiction.

T-TLAM: Assume that $\Xi, \alpha \mid \Gamma \vDash e : \tau$ holds, and consider $\rho \in \mathcal{D}[\![\Xi]\!]$ and a $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$. Assume that e' is an irreducible expression with $\gamma(\Lambda\_.e) \to^* e'$. Since $\gamma(\Lambda\_.e) = \Lambda\_.\gamma(e)$ is a value it is irreducible, so it must equal e'.

Now let $T \in \mathit{SemType}$. By Lemma 4.1.5 we also have $\gamma \in \mathcal{G}[\![\Xi, \alpha \triangleright \Gamma]\!]_{\rho[\alpha \mapsto T]}$, so the hypothesis implies that $\gamma(e) \in \mathcal{E}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha \mapsto T]}$, as desired.

T-TAPP: Assume that $\Xi \mid \Gamma \vDash e : \forall\alpha.\tau$ holds. Consider a $\rho \in \mathcal{D}[\![\Xi]\!]$ and a $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$. We then have $\gamma(e\,\_) = \gamma(e)\,\_$, and by induction[12] $\gamma(e)$ reduces to an element $\Lambda\_.e' \in \mathcal{V}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho$. Hence $\gamma(e)\,\_$ reduces to $e' \in \mathcal{E}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha \mapsto T]}$ for any $T \in \mathit{SemType}$. This in turn reduces to an element of $\mathcal{V}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha \mapsto T]}$, which then lies in $\mathcal{V}[\![\Xi \triangleright \tau[\tau'/\alpha]]\!]_\rho$ by Lemma 4.1.6. Hence it follows that $\gamma(e\,\_) \in \mathcal{V}[\![\Xi \triangleright \tau[\tau'/\alpha]]\!]_\rho$ as desired. ■

[12]Again we are only sketching the proof to avoid proving another case of Lemma 4.2.1.

We finally arrive at the promised result:

4.2.3 • THEOREM: *Fundamental property.*
If $\Xi \mid \Gamma \vdash e : \tau$, *then* $\Xi \mid \Gamma \vDash e : \tau$. ∎

Logical Relations 5

Having studied logical predicates we now move on to binary logical relations. While the construction of such a relation is very similar to the construction of its unary counterpart, it will be used very differently. In particular we will discuss how to tell whether two programs are in some sense equivalent, and how to use logical relations to prove that they are.

5.1 ◇ Equational reasoning

In the study of programming languages we are obviously interested in the evaluation or reduction of programs, as we have been in previous chapters. But there are other questions one might ask about languages, and in this chapter we are concerned with the question of when two programs are 'equivalent' in some sense.

There are various ways of defining equivalence, or equality, of programs, and to take some examples:

- Most directly one might postulate axioms and inference rules for equivalence. For instance, one axiom might be the equivalence of $\pi_1 \langle e_1, e_2 \rangle$ and e_1, since these expressions are clearly supposed to be identical. A rule might then be, using our usual notation for inference rules,
$$\frac{e_1 \sim e_2}{\lambda x.e_1 \sim \lambda x.e_2},$$
saying that two lambda expressions are equivalent if their bodies are.

  The desire is then for the additional structure of the language (i.e., its operational semantics and type system if it has one) to respect these equivalences somehow.

- If a language has a notion of reduction R, then we can simply let this induce a compatible equivalence relation $=_R$ (as described in §1.4.2) and take this to describe equivalence of programs. Of course, $=_R$ depends on which set of contexts we use to induce the compatible one-step reduction $\to_R$: We could consider only *evaluation* contexts, some other restricted set of contexts, or (as is standard in the untyped λ-calculus) use all contexts.

- Below we consider ***contextual equivalence***, by which expressions are considered equivalent if they 'behave' the same in all contexts with respect to the operational semantics.

- We finally define a ***logical relation***, which is reminiscent of the logical predicate we defined in Chapter 4, except that the interpretation of e.g. a type will be a set of *pairs* of expressions. This then also yields a notion of equivalence of programs.

The (untyped) λ-calculus is an equational theory usually defined by axioms and rules, namely by the axiom schemas

$$(\lambda x.M)N \sim M[N/x], \qquad (\beta)$$
$$\lambda x.Mx \sim M, \qquad (\eta)$$

along with the axioms for compatible equivalence relations. The axiom (η) is sometimes omitted.

Compatible equivalence relations are also induced by β- and η-reduction, which are given by

$$\big((\lambda x.M)N, M[N/x]\big) \in \beta,$$
$$(\lambda x.Mx, M) \in \eta.$$

Letting $\beta\eta := \beta \cup \eta$, these induce compatible equivalence relations $=_\beta$ and $=_{\beta\eta}$. One can then show that the equivalence $\sim$ is precisely $=_\beta$ or $=_{\beta\eta}$, depending on the inclusion of the axiom (η) in $\sim$ (cf. Barendregt 1984, Propositions 3.2.1 and 3.3.2).

The definition of an equational theory is obviously complicated by the presence of a type system, and this will concern us below.

5.2 ⋄ Contextual equivalence

Before defining contextual equivalence we discuss some desirable properties that such a notion of equivalence should have.

5.2.1. Adequate congruences

Let us first be more precise about what kinds of relations we are studying:

Our treatment of congruences is adapted from Birkedal et al. (2013).

5.2.1 • Definition: *Type-indexed relation.*
A ***type-indexed relation*** is a set of tuples (Ξ,Γ,e,e',τ), where e_1 and e_2 are expressions such that $\Xi \mid \Gamma \vdash e : \tau$ and $\Xi \mid \Gamma \vdash e' : \tau$. If $\mathcal{R}$ is such a relation, then we write $\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau$ if $(\Xi,\Gamma,e,e',\tau) \in \mathcal{R}$. ▲

If $\Xi = \emptyset$ or $\Gamma = \bot$, then we omit these from the notation as usual.

Put another way, each set Ξ of type variables, type context Γ, and type τ gives rise to a relation $\mathcal{R}_{\Xi,\Gamma,\tau}$ on the set

$$Exp_{\Xi,\Gamma,\tau} := \{e \in Exp \mid \Xi \mid \Gamma \vdash e : \tau\}$$

such that $(e,e') \in \mathcal{R}_{\Xi,\Gamma,\tau}$ if and only if $\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau$. Thus we may transfer properties and constructions pertaining to binary relations on a set to type-indexed relations: We say that $\mathcal{R}$ has property P if all $\mathcal{R}_{\Xi,\Gamma,\tau}$ have property P. For instance, $\mathcal{R}$ is an equivalence relation just when all $\mathcal{R}_{\Xi,\Gamma,\tau}$ are equivalence relations on their respective sets.

Similarly, if $\mathcal{S}$ is another type-indexed relation then we may form the composition $\mathcal{S} \circ \mathcal{R}$ by letting

$$(\mathcal{S} \circ \mathcal{R})_{\Xi,\Gamma,\tau} := \mathcal{S}_{\Xi,\Gamma,\tau} \circ \mathcal{R}_{\Xi,\Gamma,\tau}.$$

We can in particular form powers $\mathcal{R}^n$, and in the case $n = 0$ we get the identity $\mathcal{I}$ with the property that $\mathcal{I}_{\Xi,\Gamma,\tau}$ is just equality on $Exp_{\Xi,\Gamma,\tau}$. Furthermore, the inverse of $\mathcal{R}$ is given by $(\mathcal{R}^{-1})_{\Xi,\Gamma,\tau} := (\mathcal{R}_{\Xi,\Gamma,\tau})^{-1}$.

The type-indexed relations of interest should of course respect the structure of the language somehow, and in particular respect the type system. We formalise this by requiring that a relation satisfy rules derived from the typing rules. For instance, in the rule

$$\textsc{T-pair}\quad \frac{\Xi \mid \Gamma \vdash e_1 : \tau_1 \qquad \Xi \mid \Gamma \vdash e_2 : \tau_2}{\Xi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

we replace occurrences of e_i with $e_i \mathrel{\mathcal{R}} e_i'$ to obtain the rule

$$\textsc{R-pair}\quad \frac{\Xi \mid \Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1' : \tau_1 \qquad \Xi \mid \Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2' : \tau_2}{\Xi \mid \Gamma \vdash \langle e_1, e_2 \rangle \mathrel{\mathcal{R}} \langle e_1', e_2' \rangle : \tau_1 \times \tau_2}.$$

That is, $\mathcal{R}$ should be 'compositional' in the sense that two expressions of the same form are equivalence whenever their corresponding subexpressions are equivalent. We collect all these so-called ***compatibility rules*** in §B.4.

5.2.2 • **Definition.** Let $\mathcal{R}$ be a type-indexed relation. Then $\mathcal{R}$ is ***compatible*** if it satisfies the rules in §B.4, and a ***congruence*** if it is a compatible equivalence relation. ▲

A congruence thus gives rise to an equational theory that respects types.

Finally, we also want a congruence to say something reasonable about which programs are equivalent. As described above, we desire a notion of equivalence based on the execution of programs, motivating the following definitions. First we say that a ***complete program*** is a closed expression of type Bool. Then we define:

5.2.3 • **Definition: *Kleene equality.***
Two complete programs e_1 and e_2 are ***Kleene equal*** if for all values v of type Bool, $e_1 \to^* v$ if and only if $e_2 \to^* v$. In this case we write $e_1 \simeq e_2$. ▲

Kleene equality is clearly an equivalence relation. Depending on the language under consideration, different definitions of Kleene equality are useful. We return to this point in §5.5.

5.2.4 • **Definition.** A type-indexed relation $\mathcal{R}$ is ***adequate*** if $\vdash e_1 \mathrel{\mathcal{R}} e_2 :$ Bool implies $e_1 \simeq e_2$. ▲

Let us note some (mostly) easy to verify properties of type-indexed relations:

This list of properties and the proof of Proposition 5.2.5 are based on Pitts (2005, Theorem 7.5.3).

- The identity relation $\mathcal{I}$ is an adequate congruence. Conversely, every compatible relation contains $\mathcal{I}$, i.e., is reflexive (this is an easy proof by induction on the compatibility rules in §B.4).
- A composition of adequate relations is adequate, and the inverse of an adequate relation is adequate. The same is true for compatible relations.
- A union of adequate relations is adequate. A *nonempty* union of compatible relations is also compatible if it is transitive (this is less obvious, see the proof of Proposition 5.2.5).

Below we will explicitly construct the adequate congruence of interest, and it will turn out that this is the coarsest, or largest, adequate congruence. However, it is fairly easy to prove that this exists without constructing it explicitly:

Note that though we are working in a power set, we cannot appeal to Knaster–Tarski's fixed-point theorem (cf. Theorem A.4.9) to establish the existence of a largest adequate congruence. For note that while the rules in §B.4 that define compatibility do—when taken alone—give rise to a generating function, neither the definition of a type-indexed relation nor of adequacy do.

5.2.5 • **Proposition.** *There exists a coarsest adequate and compatible type-indexed relation. Furthermore, this is a congruence.*

Proof. Consider the relation

$$\mathcal{S} := \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is adequate and compatible}\}.$$

Then $\mathcal{S}$ contains $\mathcal{I}$ and is thus reflexive, and it is also symmetric and transitive since the set above is closed under composition and taking inverses. Furthermore, $\mathcal{S}$ is adequate.

It remains to be shown that a nonempty union of compatible relations is compatible if it is transitive. This is proved by induction on the rules in §B.4. For rules with a single hypothesis this is immediate, and for rules with multiple hypothesis it follows by using transitivity. As an example, consider the rule R-PAIR and assume that $\Xi \mid \Gamma \vdash e_i \; \mathcal{S} \; e_i' : \tau_i$ for $i \in \{1,2\}$. There then exist compatible $\mathcal{R}_1, \mathcal{R}_2 \subseteq \mathcal{S}$ such that $\Xi \mid \Gamma \vdash e_i \; \mathcal{R}_i \; e_i' : \tau_i$, and since each $\mathcal{R}_i$ is compatible (and in particular reflexive) we have

$$\Xi \mid \Gamma \vdash \langle e_1, e_2 \rangle \; \mathcal{R}_1 \; \langle e_1', e_2 \rangle : \tau_1 \times \tau_2$$

and

$$\Xi \mid \Gamma \vdash \langle e_1', e_2 \rangle \; \mathcal{R}_2 \; \langle e_1', e_2' \rangle : \tau_1 \times \tau_2.$$

These judgments then also hold with each $\mathcal{R}_i$ replaced with $\mathcal{S}$, so the claim follows from transitivity of $\mathcal{S}$. ∎

5.2.2. Typed contexts

Recall from §1.4.2 that a context, roughly speaking, is an expression in which a subexpression is replaced by the hole $-$. In order to define contextual equivalence we first need to consider which contexts we may apply to which expressions. Of course the argument to a context will always be well-typed (in some type context), and the resulting expression should also be well-typed.

It is also possible to define context types using typing rules, just as we have done for types of expressions. One can then show that typed contexts preserve properties so that they obey our definition of typed contexts (see e.g. Harper 2016, §46.1).

We extend the typing of expressions to contexts as follows: If C is a context with the property that $\Xi \mid \Gamma \vdash e : \tau$ implies $\Xi' \mid \Gamma' \vdash C[e] : \tau'$, then we say that the type of C is $(\Xi \mid \Gamma \triangleright \tau) \multimap (\Xi' \mid \Gamma' \triangleright \tau')$ and write $C : (\Xi \mid \Gamma \triangleright \tau) \multimap (\Xi' \mid \Gamma' \triangleright \tau')$. If any of Ξ, Ξ', Γ or Γ' are empty then we omit them from the notation, for instance writing $\tau \multimap \tau'$.

Composition of typed contexts is naturally well-defined:

5.2.6 • LEMMA. *If $C : (\Xi \mid \Gamma \triangleright \tau) \multimap (\Xi' \mid \Gamma' \triangleright \tau')$ and $C' : (\Xi' \mid \Gamma' \triangleright \tau') \multimap (\Xi'' \mid \Gamma'' \triangleright \tau'')$ are typed contexts, then $C' \circ C : (\Xi \mid \Gamma \triangleright \tau) \multimap (\Xi'' \mid \Gamma'' \triangleright \tau'')$.*

Proof. This is obvious from the definition of composition of contexts (cf. §1.4.2) and the definition of context types. ∎

Some authors define compatibility using contexts directly, e.g. Harper (2016, §46.1) (though note that Harper only considers compatibility of equivalence relations, and thus jumps straight to congruences).

Typed contexts also gives us an alternative definition of compatibility of type-indexed relations which is more closely related to the notion of contextual equivalence which we define below.

5.2.7 • **Lemma.** *A type-indexed relation $\mathcal{R}$ is compatible if and only if*

$$\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau \quad \textit{implies} \quad \Xi' \mid \Gamma' \vdash C[e] \mathrel{\mathcal{R}} C[e'] : \tau'$$

for all contexts $C : (\Xi \mid \Gamma \triangleright \tau) \multimap (\Xi' \mid \Gamma' \triangleright \tau')$.

Proof. Since $\mathcal{R}$ is automatically reflexive, each direction follows by induction: The 'only if' direction follows by induction on the structure on C, and the 'if' direction follows by induction on the compatibility rules in §B.4. ■

5.2.3. Contextual equivalence

A ***program context*** is a context of type $(\Xi \mid \Gamma \triangleright \tau) \multimap \mathsf{Bool}$.

5.2.8 • **Definition: *Contextual equivalence.***
If $\Xi \mid \Gamma \vdash e_1 : \tau$ and $\Xi \mid \Gamma \vdash e_2 : \tau$, then e_1 and e_2 are ***contextually equivalent*** if $C[e_1] \simeq C[e_2]$ for all program contexts $C : (\Xi \mid \Gamma \triangleright \tau) \multimap \mathsf{Bool}$. In this case we write $\Xi \mid \Gamma \vdash e_1 \cong^{ctx} e_2 : \tau$. ▲

Contextual equivalence is indeed an equivalence relation since Kleene equality is; see also §5.5.

5.2.9 • **Theorem.** *Contextual equivalence is the coarsest adequate congruence.*

Proof. Contextual equivalence is clearly adequate since the hole is a context of type $\mathsf{Bool} \multimap \mathsf{Bool}$. We show compatibility using the characterisation in Lemma 5.2.7, so assume that $\Xi \mid \Gamma \vdash e_1 \cong^{ctx} e_2 : \tau$ and let $C : (\Xi \mid \Gamma \triangleright \tau) \multimap (\Xi' \mid \Gamma' \triangleright \tau')$. Given a program context $C' : (\Xi' \mid \Gamma' \triangleright \tau') \multimap \mathsf{Bool}$ we have $C' \circ C : (\Xi \mid \Gamma \triangleright \tau) \multimap \mathsf{Bool}$ by Lemma 5.2.6, so $(C' \circ C)[e_1] \simeq (C' \circ C)[e_2]$ since e_1 and e_2 are contextually equivalence. But this means that $C'[C[e_1]] \simeq C'[C[e_2]]$, so since C' was arbitrary $C[e_1]$ and $C[e_2]$ are also contextually equivalent.

Conversely let $\mathcal{R}$ be an adequate congruence. If $\Xi \mid \Gamma \vdash e_1 \mathrel{\mathcal{R}} e_2 : \tau$ then $\vdash C[e_1] \mathrel{\mathcal{R}} C[e_2] : \mathsf{Bool}$ for any program context C by compatibility, and adequacy implies that $C[e_1] \simeq C[e_2]$. ■

5.3 ◇ A logical relation

It is usually difficult to prove the contextual equivalence of two expressions directly. However, due to the characterisation in Theorem 5.2.9 of contextual equivalence as the coarsest adequate congruence—and even the coarsest adequate and compatible relation by Proposition 5.2.5—we can instead use *coinduction* (cf. §1.5.2): In order to prove the judgment $\Xi \mid \Gamma \vdash e_1 \cong^{ctx} e_2 : \tau$, we instead show that $\Xi \mid \Gamma \vdash e_1 \mathrel{\mathcal{R}} e_2 : \tau$ for some other adequate and compatible relation $\mathcal{R}$. We next define such a relation.

The approach is roughly the same as in Chapter 4 when we defined a logical *predicate* in order to study type safety, except that the interpretations of values, expressions and so on are now sets of *pairs*.

5.3.1. Interpretations of syntax

We define the same kinds of interpretations as we did in Chapter 4, and we use the same notation.

(a) ***Interpretation of type variables.*** As was the case in §4.1.1, we begin by defining the interpretation of type variables. Previously we interpreted a type variable α as a semantic type, i.e., as a set of closed values. Our approach in the second case is similar, in that an interpretation of α will be an arbitrary *binary relation* on closed values, i.e., a subset of $\mathit{ClVal} \times \mathit{ClVal}$. We let $\mathit{Rel} := \mathcal{P}(\mathit{ClVal}^2)$ be the collection of such relations, and we can thus define

$$\mathcal{D}[\![\Xi]\!] := \{\rho \colon \mathit{TypeVar} \rightharpoonup \mathit{Rel} \mid \operatorname{dom}\rho = \Xi\}$$

for any finite set Ξ of type variables.

(b) ***Expression interpretation.*** Let us first try to adapt our definition of the expression interpretation for logical predicates in §4.1.1(b): Given a type τ, the expression interpretation $\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$ should now be a set of pairs (e_1, e_2) of expressions with the property that if somehow $(e_1, e_2) \longrightarrow^* (e_1', e_2')$ with (e_1', e_2') irreducible, then (e_1', e_2') lies in $\mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho$. Of course we interpret the relation '$\longrightarrow$' as acting elementwise on the pair (e_1, e_2), so the assumption means that $e_1 \longrightarrow^* e_1'$ and $e_2 \longrightarrow^* e_2'$. Similarly we interpret 'irreducible' to mean that both e_1' and e_2' are irreducible.

Rephrasing, (e_1, e_2) lies in $\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$ if whenever e_1' and e_2' are irreducible and $e_i \longrightarrow^* e_i'$, then (e_1', e_2') lies in $\mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho$. This indeed works as a definition.

However, we could also define the expression interpretation as follows: Recall that we in §4.1.1(b) discussed an alternative definition of the expression interpretation in the case of logical predicates. Namely, instead of requiring that *whenever* e_1 and e_2 reduce to irreducible expressions e_1' and e_2' these should lie in the correct value interpretation, we require that there *exists* an element of the correct value interpretation to which e_1 and e_2 reduce.

We adopt the second approach here. That is, we define the expression interpretation by

$$\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho := \left\{ (e_1, e_2) \in \mathit{ClExp}^2 \;\middle|\; \begin{array}{l} \exists (v_1, v_2) \in \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho : \\ e_1 \longrightarrow^* v_1 \text{ and } e_2 \longrightarrow^* v_2 \end{array} \right\}.$$

In §5.5 we discuss the nuances between the two definitions in more detail.

(c) ***Value interpretation.*** We finally define the value interpretation. As in the case of logical predicates we define the value interpretation of each sort of type individually. In the case of the unit type this is trivial:

$$\mathcal{V}[\![\Xi \triangleright \mathtt{1}]\!]_\rho := \{(\mathtt{1}, \mathtt{1})\}.$$

For product types we have in mind that two pairs $\langle v_1, v_2\rangle$ and $\langle v_1', v_2'\rangle$ are equivalent if the entries are pairwise equivalent, i.e., if v_1 and v_1' are equivalent, and if v_2 and v_2' are equivalent. Hence we obtain

$$\mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho := \left\{ \big(\langle v_1, v_2\rangle, \langle v_1', v_2'\rangle\big) \;\middle|\; \begin{array}{l} (v_1, v_1') \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho, \\ (v_2, v_2') \in \mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho \end{array} \right\}.$$

Sum types are perhaps slightly less intuitive. A value of a sum type $\tau_1 + \tau_2$ is either on the form $\iota_1\, v$ or $\iota_2\, v$, and while such two values are of course not *equal*, they are also not supposed to be equivalent: For recall that a sum type is supposed to be a sort of disjoint union[1] whose 'tags' are the injections ι_1 and ι_2, we require that $\iota_1\, v$ and $\iota_2\, v$ are not equivalent. Thus we define

[1] More properly a coproduct, but in the analogy with disjoint unions it is more apparent that $\iota_1\, v$ and $\iota_2\, v$ should not be equivalent.

$$\begin{aligned} \mathcal{V}[\![\Xi \triangleright \tau_1 + \tau_2]\!]_\rho := &\{(\iota_1\, v, \iota_1\, v') \mid (v, v') \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho\} \\ &\cup \{(\iota_2\, v, \iota_2\, v') \mid (v, v') \in \mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho\}. \end{aligned}$$

Notice that the union is indeed disjoint.

In the case of function types we seemingly have more freedom to choose when two values of the same function type are equivalent. Consider for instance the expressions

$$\lambda x.\mathtt{1} \quad \text{and} \quad \lambda x.\pi_1 \langle \mathtt{1}, \mathtt{1}\rangle.$$

Clearly these are *extensionally* equal since whenever they are applied to the same value they both eventually reduce to 1. But they are not obviously *intensionally* equal, in that they describe functions that have different meanings or 'senses'. To see this distinction more clearly, note that the predicates '$n > 2$' and 'the equation $a^n + b^n = c^n$ has no positive integer solution' have the same extension (i.e., truth value) when considered as funtions on $\mathbb{N}$, but that this is highly nontrivial.

Furthermore, if we think of function expressions as representing algorithms, then extensional equality of two such expressions corresponds to the two represented algorithms being equivalent in the sense that they solve the same algorithmic problem. But notice that extensional equality cannot capture the computational complexity of these algorithms.

If we agree to only consider the *result* of computations, then we may adopt an extensional view of equality of function expressions[2]. For two function expressions $\lambda x.e_1$ and $\lambda x.e_2$ of type $\tau_1 \to \tau_2$ to be equivalent, they thus must assume the same value on each input. That is, if v is a (closed) value of type τ_1, the expressions $e_1[v/x]$ and $e_2[v/x]$ must be equal. In other words:

[2] This is of course the standard approach in mathematics. But consider for instance the following: The polynomials 0 and $x^2 - x$ in $\mathbb{F}_2[x]$ (where $\mathbb{F}_2$ is the field with two elements) are distinct polynomials, but the corresponding polynomial functions $x \mapsto 0$ and $x \mapsto x^2 - x$ are equal, since they are both identically zero on $\mathbb{F}_2$. Since functions in mathematics *by definition* are extensional (up to a choice of codomain), we thus cannot with no further considerations model polynomials using functions. Since functions in our language are not defined extensionally but instead by expressions, we get a legitimate choice of how to interpret them

$$\mathcal{V}[\![\Xi \triangleright \tau_1 \to \tau_2]\!]_\rho := \left\{ (\lambda x.e_1, \lambda x.e_2) \;\middle|\; \begin{array}{l} \forall (v_1, v_2) \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho : \\ (e_1[v_1/x], e_2[v_2/x]) \in \mathcal{E}[\![\Xi \triangleright \tau_2]\!]_\rho \end{array} \right\}.$$

Lastly consider universal types. Since an expression on the form $\Lambda\_.e$ is just a function whose argument is a type, the above considerations say that two such expressions $\Lambda\_.e_1$ and $\Lambda\_.e_2$ should be considered equivalent whenever e_1 and e_2 are when applied to the same type. Of course, in our language we do not apply such expressions to types, so we model this application of e_1 and e_2 to the 'same type' in a different way. If these are of type $\forall\alpha.\tau$, then we simply require that e_1 and e_2 are equivalent when α is interpreted as some element of *Rel*. We thus arrive that the interpretation

$$\mathcal{V}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho := \left\{ (\Lambda\_.e_1, \Lambda\_.e_2) \;\middle|\; \begin{array}{l} \forall R \in Rel: \\ (e_1, e_2) \in \mathcal{E}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha\mapsto R]} \end{array} \right\}.$$

(d) ***Context interpretation.*** Before defining logical equivalence we must first consider how to interpret type contexts. Here we simply copy the definition from §4.1.1(d), extended to pairs of values:

$$\mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho := \{\gamma\colon \operatorname{dom}\Gamma \to ClVal^2 \mid \forall x \in \operatorname{dom}\Gamma\colon \gamma(x) \in \mathcal{V}[\![\Xi \triangleright \Gamma(x)]\!]_\rho\}.$$

For $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$ we write γ_1 and γ_2 for the coordinate functions of γ, so that $\gamma(x) = (\gamma_1(x), \gamma_2(x))$ for $x \in \operatorname{dom}\Gamma$.

5.3.2. Logical equivalence

We are now in a position to define the logical relation.

5.3.1 • Definition: ***Logical equivalence.***
Let Ξ be a finite set of type variables, Γ a type context, and τ a type. Then two expressions e_1 and e_2 are ***logically equivalent*** with type τ with respect to Ξ and Γ, written

$$\Xi \mid \Gamma \vdash e_1 \cong^{log} e_2 : \tau,$$

if $\Xi \mid \Gamma \vdash e_1 : \tau$ and $\Xi \mid \Gamma \vdash e_2 : \tau$ and

$$\forall \rho \in \mathcal{D}[\![\Xi]\!], \gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho\colon (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho. \qquad \blacktriangle$$

Logical equivalence is indeed an equivalence relation, but we postpone the argument to §5.5. We now intend to show that logical equivalence implies contextual equivalence, and by Theorem 5.2.9 it suffices to show that logical equivalence is compatible and adequate.

5.4 ◇ Compatibility and adequacy

5.4.1 • Lemma: ***Compatibility.***
Logical equivalence $\cong^{log}$ satisfies the compatibility rules in §B.4.

Proof. The proof of this claim is very similar to that of the corresponding claim Lemma 4.2.2 for the logical predicate of Chapter 4, with the minor complication that our definition of the expression interpretation is slightly different. This only turns out to make the proofs of each case easiler. We give some representative examples to illustrate how to modify the arguments to apply in the binary case.

R-VAR: Assume that $\Xi \vdash \Gamma$ and that $\Gamma(x) = \tau$. It then follows that $\Xi \mid \Gamma \vdash x : \tau$ by T-VAR. For $\rho \in \mathcal{D}[\![\Xi]\!]$ and $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$ we must show that $(\gamma_1(x), \gamma_2(x)) \in \mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$. But notice that $(\gamma_1(x), \gamma_2(x)) = \gamma(x)$, which lies in $\mathcal{V}[\![\Xi \triangleright \Gamma(x)]\!]_\rho$ by definition of the context interpretation. Since $\Gamma(x) = \tau$, it also lies in[3] $\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho$.

[3]By a version of Lemma 4.1.4(i) for binary logical relations.

R-PAIR: Assume that $\Xi \mid \Gamma \vdash e_i \cong^{log} e_i' : \tau_i$ for $i \in \{1,2\}$. Then T-PAIR implies that $\Xi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$ and $\Xi \mid \Gamma \vdash \langle e_1', e_2' \rangle : \tau_1 \times \tau_2$. Next notice that

$$\big(\gamma_1(\langle e_1, e_2 \rangle), \gamma_2(\langle e_1', e_2' \rangle)\big) = \big(\langle \gamma_1(e_1), \gamma_1(e_2) \rangle, \langle \gamma_2(e_1'), \gamma_2(e_2') \rangle\big),$$

and that we by induction have

$$(\gamma_1(e_1), \gamma_2(e_1')) \in \mathcal{E}[\![\Xi \triangleright \tau_1]\!]_\rho \quad \text{and} \quad (\gamma_1(e_2), \gamma_2(e_2')) \in \mathcal{E}[\![\Xi \triangleright \tau_1]\!]_\rho.$$

Hence there are pairs $(v_1, v_1') \in \mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho$ and $(v_2, v_2') \in \mathcal{V}[\![\Xi \triangleright \tau_2]\!]_\rho$ such that

$$\gamma_1(e_1) \to^* v_1, \quad \gamma_2(e_1') \to^* v_1', \quad \gamma_1(e_2) \to^* v_2 \quad \text{and} \quad \gamma_2(e_2') \to^* v_2'.$$

But then it follows that[4]

$$\langle \gamma_1(e_1), \gamma_1(e_2) \rangle \to^* \langle v_1, v_2 \rangle \quad \text{and} \quad \langle \gamma_2(e_1'), \gamma_2(e_2') \rangle \to^* \langle v_1', v_2' \rangle$$

by several applications of E-HEAD. But these pairs of values are also values and hence $(\langle v_1, v_2 \rangle, \langle v_1', v_2' \rangle)$ lies in $\mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$ as required.

[4]This is where our new definition of the expression interpretation is useful, since we avoid technical results like those of Lemma 4.2.1. Strictly speaking this conclusion follows by induction on the lengths of each reduction, but it should be obvious how to make this precise.

R-PROJ$_1$: Assume that $\Xi \mid \Gamma \vdash e \cong^{log} e' : \tau_1 \times \tau_2$, and consider $\rho \in \mathcal{D}[\![\Xi]\!]$ and $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$. By induction there is an element $(\langle v_1, v_2 \rangle, \langle v_1', v_2' \rangle)$ of $\mathcal{V}[\![\Xi \triangleright \tau_1 \times \tau_2]\!]_\rho$ such that $\gamma_1(e) \to^* \langle v_1, v_2 \rangle$ and $\gamma_2(e') \to^* \langle v_1', v_2' \rangle$. By E-PROJ$_1$ it follows that $\pi_1\, \gamma_1(e) \to^* v_1$ and $\pi_1\, \gamma_1(e) \to^* v_1'$, and since (v_1, v_1') lies in $\mathcal{V}[\![\Xi \triangleright \tau_1]\!]_\rho$, the claim follows.

R-TLAM: Assume that $\Xi, \alpha \mid \Gamma \vdash e \cong^{log} e' : \tau$ and that $\alpha \notin \mathrm{FV}_{Type}(\Gamma)$, and consider $\rho \in \mathcal{D}[\![\Xi]\!]$ and $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$. We then must show that

$$(\Lambda\_.\gamma_1(e), \Lambda\_.\gamma_2(e')) = (\gamma_1(\Lambda\_.e), \gamma_2(\Lambda\_.e')) \in \mathcal{E}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho.$$

Let $R \in Rel$ and note that γ also lies in[5] $\mathcal{G}[\![\Xi, \alpha \triangleright \Gamma]\!]_{\rho[\alpha \mapsto R]}$. By induction $(\gamma_1(e), \gamma_2(e'))$ thus lies in $\mathcal{E}[\![\Xi, \alpha \triangleright \forall\alpha.\tau]\!]_{\rho[\alpha \mapsto R]}$, so by definition of the value interpretation of universal types $(\Lambda\_.\gamma_1(e), \Lambda\_.\gamma_2(e'))$ lies in $\mathcal{V}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho$, and hence also in[6] $\mathcal{E}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho$ as required.

[5]By a version of Lemma 4.1.5 for binary relations.

[6]By a version of Lemma 4.1.4(i) for binary relations.

R-TAPP: Assume that $\Xi \mid \Gamma \vdash e \cong^{log} e' : \forall\alpha.\tau$ and that $\Xi \vdash \tau'$. Considerin $\rho \in \mathcal{D}[\![\Xi]\!]$ and $\gamma \in \mathcal{G}[\![\Xi \triangleright \Gamma]\!]_\rho$ we show that

$$(\gamma_1(e)\_, \gamma_2(e')\_) = (\gamma_1(e\_), \gamma_2(e'\_)) \in \mathcal{E}[\![\Xi \triangleright \tau[\tau'/\alpha]]\!]_\rho.$$

By induction $\gamma_1(e)$ and $\gamma_2(e')$ reduce to values $v = \Lambda\_.e_1$ and $v' = \Lambda\_.e_1'$ such that $(v, v') \in \mathcal{V}[\![\Xi \triangleright \forall\alpha.\tau]\!]_\rho$. It follows by E-TAPP-TLAM that $\gamma_1(e)\_ \to^* e_1$ and $\gamma_2(e')\_ \to^* e_1'$, where (e_1, e_1') lies in $\mathcal{E}[\![\Xi, \alpha \triangleright \Gamma]\!]_{\rho[\alpha \mapsto T]}$. Next, e_1 and e_1' reduce to values v_1 and v_1' such that (v_1, v_1') lies in $\mathcal{V}[\![\Xi, \alpha \triangleright \tau]\!]_{\rho[\alpha \mapsto T]}$ and hence in[7] $\mathcal{V}[\![\Xi \triangleright \Gamma]\!]_\rho$. ■

[7]By a version of Lemma 4.1.6 for binary relations.

5.4.2 • **Lemma:** ***Adequacy.***
Logical equivalence $\cong^{log}$ is adequate.

Proof. Assume that $\vdash e_1 \cong^{log} e_2 : \mathsf{Bool}$. Since e_1 and e_2 are then closed, it follows that $(e_1, e_2) \in \mathcal{E}[\![\mathsf{Bool}]\!]$. Hence there exists a pair $(v_1, v_2) \in \mathcal{V}[\![\mathsf{Bool}]\!]$ such that $e_1 \rightarrow^* v_1$ and $e_2 \rightarrow^* v_2$. But since $\mathcal{V}[\![\mathsf{Bool}]\!]$ is the set $\{(\mathsf{false}, \mathsf{false}), (\mathsf{true}, \mathsf{true})\}$, this implies that $v_1 = v_2$.

To show that $e_1 \simeq e_2$, assume that v is a value with $e_1 \rightarrow^* v$. By Corollary 2.4.11 it follows that $v = v_1 = v_2$. The above then implies that also $e_1 \rightarrow^* v$ as desired. Hence $e_1 \rightarrow^* v$ implies $e_2 \rightarrow^* v$, and the converse follows similarly (or by symmetry). ■

Hence we arrive at the main theorem of this section:

5.4.3 • **Theorem:** ***Adequate congruence.***
Logical equivalence $\cong^{log}$ is an adequate congruence. In particular,

$$\Xi \mid \Gamma \vdash e_1 \cong^{log} e_2 : \tau \quad \textit{implies} \quad \Xi \mid \Gamma \vdash e_1 \cong^{ctx} e_2 : \tau. \qquad ■$$

5.5 ◇ ∀ vs. ∃

We have already mentioned that there are (at least) two different ways of defining the expression interpretation: Our official definition for binary logical relations is, as we recall,

$$\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho = \left\{ (e_1, e_2) \in ClExp^2 \;\middle|\; \begin{array}{l} \exists (v_1, v_2) \in \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho : \\ e_1 \rightarrow^* v_1 \text{ and } e_2 \rightarrow^* v_2 \end{array} \right\}.$$

We could also have adapted the definition from §4.1.1(b) directly and let

$$\mathcal{E}[\![\Xi \triangleright \tau]\!]_\rho = \left\{ (e_1, e_2) \in ClExp^2 \;\middle|\; \begin{array}{l} \forall e_1', e_2' \in Irr : \\ e_i \rightarrow^* e_i' \Rightarrow (e_1', e_2') \in \mathcal{V}[\![\Xi \triangleright \tau]\!]_\rho \end{array} \right\}.$$

Each of these definitions give rise to a different version of logical equivalence. Let us denote these by $\cong_\exists$ and $\cong_\forall$, respectively. Now notice the following:

- The relation $\cong_\exists$ is reflexive since it is compatible[8]. We can also give a more direct argument if the reduction is normalising: For then if e is a complete program, there is a boolean value v such that $e \rightarrow^* v$, and hence $(v, v) \in \mathcal{V}[\![\mathsf{Bool}]\!]$.

  If instead the reduction is Church–Rosser, then $\cong_\exists$ is transitive. For then if $\vdash e_1 \cong_\exists e_2 : \mathsf{Bool}$ and $\vdash e_2 \cong_\exists e_3 : \mathsf{Bool}$, then $e_1 \rightarrow^* v_1$ and $e_2 \rightarrow^* v_2$, and also $e_2 \rightarrow^* v_2'$ and $e_3 \rightarrow^* v_3'$, and by the Church–Rosser property we have $v_2 = v_2'$.

  Finally, $\cong_\exists$ is always symmetric.

[8] Cf. the discussion immediately following Definition 5.2.4.

- The relation $\cong_\forall$ is always transitive and symmetric, and again it is automatically reflexive. If the reduction is Church–Rosser, then reflexivity is immediate: If $e \to^* e_1$ and $e \to^* e_2$ with e_i irreducible, we have $e_1 = e_2$ and so $(e_1, e_2) \in \mathcal{V}[\![\mathsf{Bool}]\!]$.

Similarly we have two different definitions of Kleene equality. For complete programs e_1 and e_2, say that these are ∀-Kleene equivalent, denoted $e_1 \simeq_\forall e_2$, if they satisfy Definition 5.2.3, i.e., if *for all* values v of type Bool, $e_1 \to^* v$ if and only if $e_2 \to^* v$. Furthermore say that e_1 and e_2 are ∃-Kleene equivalent, denoted $e_1 \simeq_\exists e_2$, if *there exists* a value v of type Bool such that $e_1 \to^* v$ and $e_2 \to^* v$.

- ∀-Kleene equality is always an equivalence relation.
- ∃-Kleene equality is always symmetric. If the reduction is normalising then $\simeq_\exists$ is reflexive, and if the reduction is Church–Rosser $\simeq_\exists$ is transitive.

We next consider the relationship between the different variations of logical equivalence and Kleene equality:

- Assuming that the reduction is normalising, if $\vdash e_1 \cong_\forall e_2 : \mathsf{Bool}$ then $e_1 \simeq_\forall e_2$. For assume that $e_1 \to^* v$. By normalisation e_2 reduces to an irreducible e_2', so $(v, e_2') \in \mathcal{V}[\![\mathsf{Bool}]\!]$, implying that $v = e_2'$.

  Notice that if the reduction is not normalising, then there is no reason to expect that e_2 has a normal form.
- Assuming that the reduction is normalising, if $\vdash e_1 \cong_\forall e_2 : \mathsf{Bool}$ then $e_1 \simeq_\exists e_2$. For then $e_1 \to^* e_1'$ and $e_2 \to^* e_2'$, and similar to above we have $e_1' = e_2'$.
- Assuming that the reduction is Church–Rosser, if $\vdash e_1 \cong_\exists e_2 : \mathsf{Bool}$ then $e_1 \simeq_\forall e_2$. This is the content of Lemma 5.4.2.
- If $\vdash e_1 \cong_\exists e_2 : \mathsf{Bool}$ then $e_1 \simeq_\exists e_2$. This is also clear since e_1 and e_2 have a common normal form as above.

In short, since we have proved that the reduction is deterministic and hence Church–Rosser (cf. Theorem 2.4.10), we know that $\cong_\forall$, $\cong_\exists$ and $\simeq_\forall$ are equivalence relations, that $\simeq_\forall$ is symmetric and transitive, and that $\cong_\exists$ implies $\simeq_\forall$ and $\simeq_\exists$.

Order Theory A

In this appendix we give an overview of the order theory necessary to study fixed-points of monotone maps.

A.1 ⋄ Objects: Partial orders and lattices

We begin by recalling the definitions and basic properties of various kinds of ordered sets.

A.1.1. Ordered sets

A.1.1 • Definition: ***Orders.***
A binary relation $\leq$ on a set P is called a ***preorder*** if $\leq$ is

(a) reflexive, $x \leq x$ for all $x \in P$, and

(b) transitive, $x \leq y$ and $y \leq z$ implies $x \leq z$ for all $x, y, z \in P$.

If furthermore $\leq$ is

(c) antisymmetric, $x \leq y$ and $y \leq x$ implies $x = y$ for all $x, y \in P$,

then $\leq$ is called a ***partial order***, and the pair $(P, \leq)$ is also called a ***poset***. If $\leq$ is also

(d) strongly connected, $x \leq y$ or $y \leq x$ for all $x, y \in P$,

then $\leq$ is a ***total order***. ▲

Notice that any subset of a preordered set is also preordered with the inherited order. As usual we intentionally conflate a preordered set $(P, \leq)$ and the underlying set P when this does not lead to confusion. We define the ***dual order*** $\leq^{op}$ on P by letting $x \leq^{op} y$ if and only if $y \leq x$. The resulting preordered set $(P, \leq^{op})$ is also simply denoted P^{op} and is called the ***dual*** of $(P, \leq)$.

If both $x \leq y$ and $y \leq x$ then x and y are ***equivalent*** and we write $x \equiv y$. By quotienting out by $\equiv$ any preordered set becomes a poset.

If $x \in P$ and $A \subseteq P$, then x is an ***upper bound*** of A if $a \leq x$ for all $a \in A$. In this case A is said to be ***bounded above***. If also $x \in A$ then x is a ***greatest element*** of A. If A has a single greatest element, then this element is the ***maximum*** of A. Furthermore, if $x \in A$ has the property that $x \leq a$ implies $a \leq x$ for all $a \in A$, then x is a ***maximal element*** of A. Every greatest element is thus maximal. In case $\leq$ is a partial order, A has at most one greatest element which is its maximum.

For $A \subseteq P$ we write

$$A^{\uparrow} := \{y \in P \mid \exists x \in A \colon x \leq y\},$$

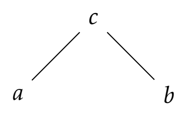

Figure A.1. A so-called ***Hasse diagram*** for the poset $\{a, b, c\}$ in which $a < c$ and $b < c$, and no other relations hold. This poset has a maximum c and two distinct minimal elements a and b, and hence no minimum.

The subset $a^{\uparrow} = \{a, c\}$ is upward closed but not downward closed.

and we let $x^\uparrow := \{x\}^\uparrow$ for $x \in P$, so that $A^\uparrow = \bigcup_{x \in A} x^\uparrow$. A subset A of P is said to be ***upward closed*** or an ***upper set*** if $A = A^\uparrow$. An upper set is ***principal*** if it is on the form $x^\uparrow$ for some $x \in P$.

The above notions have obvious dual counterparts. Clearly A is upward closed if and only if $P \setminus A$ is downward closed, and vice versa. There is no such correspondence between *principal* upper and lower sets.

Suppose now that $\leq$ is a partial order. If P itself has a maximum, then this is denoted $\top$ or $\top_P$ and called the ***top element*** of P, and P is said to be ***topped***. If P has a minimum, then this is denoted $\bot$ or $\bot_P$ and called the ***bottom element*** of P, and P is called ***pointed***. If P is both topped and pointed it is said to be ***bounded***.

The following result is easy to prove but shows that the collection of downward (or upward) closed subsets of a poset characterise the order:

A.1.2 • Lemma. *Let P be a poset, and let $x, y \in P$. The following are equivalent:*

(i) $x \leq y$.

(ii) $x^\uparrow \subseteq y^\uparrow$.

(iii) *For all upward closed $A \subseteq P$, $x \in A$ implies $y \in A$.* ■

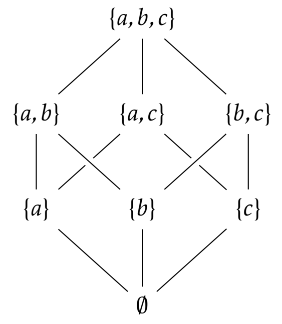

Figure A.2. Hasse diagram for the power set $\mathcal{P}(\{a,b,c\})$. Power sets are the most important class of (complete) lattices. In fact, they are ***Boolean algebras*** since every element has a complement.

A.1.2. Lattices

If a set $A \subseteq P$ has a *least* upper bound x, then x is called a ***supremum*** or ***join*** of A. A join of A is determined up to equivalence if it exists, and we denote any such join by $\bigvee A$. We similarly define an ***infimum*** or ***meet*** of A, denoted $\bigwedge A$, to be a greatest lower bound of A, if it exists. A join of a one- or two-element subset $\{x, y\}$ of P is denoted $x \vee y$, and we similarly write $x \wedge y$ for one of its meet.

Clearly two elements x and y has a join (resp. meet) in P just when they have a meet (resp. join) in the dual P^{op}.

A.1.3 • Definition: *Lattices.*
Let L be a poset. If $x \vee y$ and $x \wedge y$ exist for all $x, y \in L$, then L is called a ***lattice***.

There is also an algebraic theory of lattices in which the operations $\vee$ and $\wedge$ are primary, and in which the order is defined using these operations (cf. Davey and Priestley 2002, Chapter 2). Both approaches are equivalent.

In standard first-order logic and universal algebra we usually require the underlying set of a structure or algebra to be nonempty (relaxing this assumption gets us into ***free logic*** territory, cf. Priest 2008, Chapter 13). However, in algebra it is sometimes useful, or at least more natural, to allow domains to be empty: Otherwise the corresponding algebraic categories often have no initial object, and parallel morphisms with disjoint images have no equaliser.

Note also that authors differ on whether lattices must be nonempty, and even whether they must be bounded (in which case they cannot be empty): Schröder (2016, Definition 8.1) allows empty lattices, Davey and Priestley (2002, Definition 2.4) do not, and Johnstone (1982, §1.4) requires lattices to be bounded.

A.1.3. Completeness in posets

A preordered set P is ***complete*** if every subset of P has a meet. If on the other hand P has all joins, then we might have called P 'cocomplete', if not for the fact that these properties are equivalent, as a standard argument shows.

Let $\mathcal{C}$ be a preorder category, i.e., a category with at most one arrow between a pair of objects. Then any pair of parallel arrows in $\mathcal{C}$ have an equaliser, so for $\mathcal{C}$ to be complete it suffices for it to have all products (cf. Leinster 2014, Proposition 5.1.26(a) or any book on category theory). But products are precisely meets, so our terminology for posets is in agreement with the terminology from category theory.

Instead we are interested in preordered sets that do not have all joins, but that have 'enough' joins, and specifically joins of particular kinds of subsets:

A.1.4 • Definition. Let P be a poset, let $C \subseteq P$, and let κ be a cardinal number. We say that

(a) C is a ***chain*** if it is totally ordered.

(b) C is a ***κ-chain*** if it is a chain and $|C| \leq \kappa$.

(c) C is ***consistent*** if every finite subset of C has an upper bound in P.

(d) C is ***directed*** if every finite subset of C has an upper bound in C.

If C has a join in P, then C is called ***convergent***. ▲

A directed set is thus clearly consistent. Furthermore, if P is topped then every subset is consistent, so this notion is only interesting if P is not topped. If a set $\{x_1, \ldots, x_n\}$ is consistent then we also say that the x_i are consistent.

By induction directedness of D is equivalent to the property that, for every *pair* of elements $x, y \in D$, there exists a $z \in D$ with $x \leq z$ and $y \leq z$, as long as D is nonempty. This is the usual definition of directedness, but we prefer the 'unbiased' version (cf. nLab authors 2024). Interestingly, some authors explicitly require both the existence of upper bound of finite subsets *and* nonemptyness (cf. Gierz et al. 2003, Definition O-1.1), which of course is redundant.

Notice that a directed set D is automatically nonempty, since the empty set is a finite subset of D. If D is a directed set whose join exists, we often write $\bigsqcup D := \bigvee D$ instead. Another common notation is $\bigvee^{\uparrow} D$. We further note that the image of a chain, κ-chain, consistent or directed set under a monotone map also has this property.

For each type of subset of a poset, we may study special kinds of posets in which every such subset has a join. In the case of directed sets we in particular get the following:

A.1.5 • Definition. Let F be a property of subsets of posets. A poset P is ***F-cocomplete*** if every subset of P with the property F is convergent.▲

In this setting what we call 'cocompleteness' is often simply called 'completeness', but we follow Crole (1993) and prefer the category theory-friendly nomenclature.

For the most important choices of F we use slightly altered terminology:

- If F is 'bounded above', then P is ***bounded cocomplete***. If P is nonempty then the empty set has an upper bound, and P is automatically pointed. Some authors (e.g. Goubault-Larrecq 2013, Definition 5.7.3) build pointedness into their definition of bounded cocompleteness, while others (e.g. Gierz et al. 2003, Definition O-2.1(v)) do not but instead build nonemptyness into their definition of posets (cf. Gierz et al. 2003, Definition O-1.6). In Definition A.1.9 we define the class of posets of interest to us, and these posets will in either case be nonempty.

- Similarly, if F is 'consistent' then P is ***consistent cocomplete***. Some authors also use the term ***coherent*** for P in this case (e.g. Barendregt 1984, Definition 1.2.28).
- Most importantly, if F is 'directed', then P is ***directed cocomplete*** and is called a ***dcpo***. We obtain analogous properties if instead F is 'ω-chain' or 'chain', in which case we respectively call P an ***ω-cpo*** or a ***ccpo***. We will focus on dcpo's in the sequel, but many results have analogous statements for the other types of complete posets. If any of the above are pointed, then we add an extra 'p' to their abbreviations, so that we get ***ω-cppo***, ***ccppo*** and ***dcppo***.

A.1.6 • Lemma. *Let P be a poset, and consider the following properties:*

(i) *P is consistent cocomplete.*

(ii) *P is bounded cocomplete.*

(iii) *$\bigwedge A$ exists for all nonempty $A \subseteq P$.*

Then the following implications hold:

$$(i) \Rightarrow (ii) \Leftrightarrow (iii).$$

If P is a dcpo then the implication (ii) $\Rightarrow$ (i) also holds.

Proof. The equivalence of (ii) and (iii) is standard. Assuming (i), if $A \subseteq P$ is bounded above then it is automatically consistent. Finally, if P is a dcpo and (ii) holds, let $A \subseteq P$ be consistent and let $D := \{\bigvee F \mid F \subseteq_\omega A\}$. Then D is directed and it is easy to show that $\bigvee A = \bigsqcup D$. ∎

A.1.4. Algebraic posets and Scott domains

Finiteness also arises from the 'way-below' relation: Given $x, y \in P$, if $y \leq \bigsqcup D$ implies that $x \leq d$ for some $d \in D$, then x is ***way below*** y, written $x \ll y$. Then x is finite if and only if $x \ll x$. This notion leads to the theory of ***continuous posets***, cf. Goubault-Larrecq (2013, §5.1), a generalisation of algebraic posets.

A.1.7 • Definition: *Finiteness.*
Let P be a poset. An element $x \in P$ is ***finite*** if for all convergent directed $D \subseteq P$, if $x \leq \bigsqcup D$ then there is a $d \in D$ with $x \leq d$. The set of finite elements of P is denoted $F(P)$. ▲

Note that if x and y are finite, then $x \vee y$ is also finite if it exists. Furthermore, if $x, y \in P$ with x finite, then we write $x \leq_\omega y$. Since the finite elements of a power set are precisely those that are finite as sets, this notation is consistent with our notation '$\subseteq_\omega$' for finite subsets.

A.1.8 • Definition: *Algebraic posets.*
A poset P is ***algebraic*** if

$$x = \bigsqcup \{y \in F(P) \mid y \leq x\}$$

for all $x \in P$. ▲

Implicit in this definition is that the sets $\{y \in F(P) \mid y \leq x\}$ are directed.

A.1.9 • Definition: *Scott domains.*
If P is a dcppo that is algebraic and bounded cocomplete, then P is called a ***Scott domain***. ▲

By Lemma A.1.6 a Scott domain is automatically consistent complete.

A.1.10 • Examples.

(A) The lattice of subalgebras of an algebra (in the sense of universal algebra) is a complete algebraic lattice. For a concrete example, consider the group $(\mathbb{Z}, +)$: Its subgroups are $d\mathbb{Z}$ for $d \in \mathbb{N}$ (cf. Aluffi 2009, Proposition 6.9), and its lattice of subgroups is isomorphic to the dual $(\mathbb{N}, |)^{op}$ of the naturals ordered by divisibility. This is depicted in Figure A.3.

(B) Any power set is also a complete algebraic lattice.

(C) If X and Y are sets, then the set $(X \rightharpoonup Y)$ is a Scott domain: The finite elements in $(X \rightharpoonup Y)$ are those partial maps with finite domain[1], and it is thus clear that $(X \rightharpoonup Y)$ is algebraic. It is pointed with minimum $\perp$, and it is easy to see that it is a dcpo. Finally, if $F \subseteq (X \rightharpoonup Y)$ is bounded above, then taking the union of the graphs of the functions in F yields the graph of partial map. ⌟

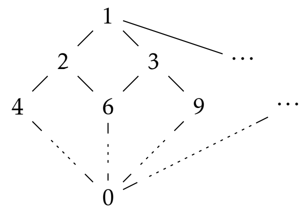

Figure A.3. Hasse diagram for the lattice of subgroups of $\mathbb{Z}$, where a node d represents the subgroup $d\mathbb{Z}$. Note that $1\mathbb{Z} = \mathbb{Z}$ and $0\mathbb{Z} = \{0\}$.

[1] If $f \in (X \rightharpoonup Y)$ is a finite element, let D be the set of the finite partial maps d with $d \leq f$. Then D is directed and $f = \bigsqcup D$. The converse follows by induction on $|\mathrm{dom}\, f|$.

A.1.5. Topology

There are several different ways to equip an ordered set P with a topology that somehow respect the ordering. Before surveying the options, consider the opposite problem, that of equipping a topological space with an ordering.

If X is a topological space, define the ***specialisation preorder*** $\preceq$ on X by the condition that $x \preceq y$ if and only if every neighbourhood of x is also a neighbourhood of y, or equivalently if $x \in \overline{\{y\}}$. The preordered set $(X, \preceq)$ is sometimes denoted ΩX.

This is a very natural ordering: For instance, X is T_0 if and only if $\preceq$ is a partial order, and X is T_1 is if and only if $\preceq$ is equality. More pertinent for us, every open set is upward closed in the specialisation preorder, and similarly every closed set is downward closed.

An obvious question is how the original order $\leq$ and the specialisation preorder $\preceq$ relate to each other. We have a useful criterion which the reader can readily check:

A.1.11 • Lemma. *Let $(P, \leq)$ be a preordered set equipped with a topology, and let $\preceq$ be the specialisation preorder.*

(i) *If every open set is upward closed with respect to $\leq$, then $\leq \subseteq \preceq$.*

(ii) *If every principal lower set with respect to $\leq$ is closed, then $\preceq \subseteq \leq$.* ■

Given the set $S = \{0,1\}$ of truth values ordered by $0 < 1$, the space $\mathrm{A}S$ is called the ***Sierpiński space***. It is well known that the functor $\mathcal{O}\colon \mathbf{Top}^{op} \to \mathbf{Set}$ sending a space to its topology is represented by S, so the restriction of $\mathcal{O}$ to the category of ordered sets is also represented by S.

(a) *Alexandroff topology.* Given the above, an obvious way in which to equip a preordered set $(P,\leq)$ with a topology is just to let the open sets be the upward closed subsets of P. This is called the ***Alexandroff topology*** (also called the ***specialisation topology***) on P, and is denoted $\alpha(P)$, and the topological space $(P,\alpha(P))$ is denoted $\mathrm{A}P$ (note that the 'A' is a capital 'α').

The specialisation preorder on $\mathrm{A}P$ is precisely the original ordering on P (i.e., $\Omega\mathrm{A}P = P$)[2], and it is easy to see that $\alpha(P)$ is the finest topology on P with this property (cf. Goubault-Larrecq 2013, Proposition 4.2.11).

Furthermore, since the upward closed subsets of P characterise its order by Lemma A.1.2, we may recover the order from the topology.

[2]In fact, one can show that there is an adjunction

$$\begin{tikzcd} \mathbf{Pre} \arrow[r, bend left, "\mathrm{A}"] \arrow[r, phantom, "\perp"] & \mathbf{Top} \arrow[l, bend left, "\Omega"] \end{tikzcd}$$

where **Pre** is the category of preordered sets and monotone maps (cf. §A.2.1). Note that Ω does not have a right adjoint since it does not preserve coequalisers: In both categories coequalisers are quotients, but while a quotient of a T_1-space is not necessarily T_1, a quotient of a discrete preordered set is still discrete.

(b) *Upper topology.* There is also a *coarsest* topology on P that has the ordering on P as its specialisation preorder. This is called the ***upper topology*** on P and is denoted $\nu(P)$. The space $(P,\nu(P))$ is denoted $\mathrm{N}P$. It is easy to check that $\nu(P)$ is indeed the coarsest such topology, once one has realised that $\overline{\{y\}} = y^{\downarrow}$ with respect to the specialisation preorder.

More concretely, $\nu(P)$ has a subbasis consisting the sets $P \setminus x^{\downarrow}$ for $x \in P$, or equivalently a basis consisting of the sets $P \setminus A^{\downarrow}$ for finite subsets $A \subseteq P$ (cf. Goubault-Larrecq 2013, Proposition 4.2.12).

(c) *Scott topology.* Neither the Alexandroff nor the upper topology is of much interest to us, but they help motivate the ***Scott topology***: We have already seen that directed subsets of posets have important properties, so we attempt to modify the Alexandroff topology to conform better to the directed subsets of P.

Equip P with a topology. Following Goubault-Larrecq (2013), let us think of open sets of P as 'tests', such that an open set U corresponds to the test 'given $x \in P$, does $x \in U$?'. We further think of the ordering on P as an *information ordering*, in the sense that if $x \leq y$, then y contains more information than x. This then implies that U is upward closed, since if it is possible to determine that x passes the test U, then it is also possible to determine that y does because it contains all the information in x.

We next (informally) say that U is 'computable' if there is a computable partial function $f : P \rightharpoonup \{0,1\}$ with the property that $f(x) = 1$ if and only if $x \in U$. Thinking of f as a program π, it proceeds by considering 'finite' parts $x_0, x_1, x_2, \ldots$ of x such that the sequence $(x_n)_{n\in\mathbb{N}}$ 'approximates' x. Since x_n is supposed to be a better approximation of x than x_m if $m \leq n$, we suppose that the sequence $(x_n)_{n\in\mathbb{N}}$ is increasing. We interpret the idea that it approximates x as $x = \bigvee_{n\in\mathbb{N}} x_n$.

Given a test U and the corresponding program π, if π gets x as input and outputs 1, then it must have concluded that $x \in U$ on the basis of some approximation x_n. But then x_n must itself pass the test U since x_n is precisely the part of x that π has used to conclude that $x \in U$. In other words, $x_n \in U$.

This implies that U should have the property that if $(x_n)_{n\in\mathbb{N}}$ is an increasing sequence whose join x lies in U, then there is an $n \in \mathbb{N}$ such that x_n already lies in U. Or put another way, if C is a nonempty ω-chain with $\bigvee C \in U$, then $C \cap U \neq \emptyset$.

However, it turns out that using ω-chains leads to a theory that has some bizarre properties (cf. Abramsky and Jung 1994, §2.2.4). Hence it is customary to instead use *directed sets*, even though these do not exactly model the way in which a program executes. Thus we arrive at the following definition:

A.1.12 • Definition. Let P be a poset. A subset $U \subseteq P$ is said to be ***inaccessible*** if $\bigsqcup D \in U$ implies $D \cap U \neq \emptyset$ for all convergent directed sets $D \subseteq P$. ▲

The ***Scott topology*** on P then consists of all the inaccesible upper sets. We denote it $\sigma(P)$, and the space $(P, \sigma(P))$ is denoted ΣP. This also has the property that the specialisation preorder on ΣP is precisely the original ordering on P, implying that $\sigma(P)$ lies (usually strictly) between $\alpha(P)$ and $\nu(P)$: simply notice that every principal lower set is Scott closed (cf. Goubault-Larrecq 2013, Proposition 4.2.18).

(d) *Pointwise convergence.* If $x \in P$ is finite, then we easily check that $x^{\uparrow}$ is inaccessible and hence Scott open. Hence the principal upper sets in P constitute subcollection of the Scott topology. Letting these induce a topology we get the following:

In the complete lattice $[0,1]$ with the usual order, only 0 is finite. On the other hand, it is easy to see that $x^{\uparrow}$ is accessible for all $x \in (0,1]$.

Or a less trivial example: In the complete lattice C of closed subsets of $\mathbb{R}$, where meet is intersection and join is union followed by closure, infinite closed sets are not finite in C. For instance,

$$\bigsqcup_{n\in\mathbb{N}_+} [\tfrac{1}{n}, 1] = [0,1],$$

so $[0,1]$ is not finite. Hence $[0,1]^{\uparrow}$ is accessible.

A.1.13 • Definition: *Topology of pointwise convergence.* Let P be a poset. The topology on P generated by

$$\{x^{\uparrow} \mid x \in F(P)\}$$

is called the ***topology of pointwise convergence***. ▲

That is, $\{x^{\uparrow} \mid x \in F(P)\}$ is a subbasis for the topology. It gets its name from its main application in domains of partial maps, as we return to in §A.3.

The topology of pointwise convergence is generally strictly coarser than the Scott topology[3], but in many important posets the two coincide:

[3] In $[0,1]$ the only finite element is 0, so the topology of pointwise convergence is trivial. On the other hand, any subset $(x,1]$ is inaccessible. In general, sets on the form $x^{\uparrow} \setminus \{x\}$ provide counterexamples in many dcpo's.

A.1.14 • Proposition. *If P is an algebraic poset, then the Scott topology and the topology of pointwise convergence on P coincide.*

Proof. Let U be a Scott open subset of P, and let $x \in U$. Since P is algebraic we have $x = \bigsqcup\{y \in F(P) \mid y \leq x\}$, and since U is inaccessible there is a $y \leq_{\omega} x$ in U. Finally, since U is upward closed we have $y^{\uparrow} \subseteq U$, so in total

$$U = \bigcup_{y\in U\cap F(P)} y^{\uparrow}.$$

Thus U is also open in the topology of pointwise convergence. ■

A.2 ⋄ Arrows

A.2.1. Maps between ordered sets

A.2.1 • Definition. Let P and Q be preordered sets. A map $f\colon P \to Q$ is said to be ***monotone*** if $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in P$. ▲

Clearly the composition of two monotone maps is monotone, and the identity map is monotone, so this yields a category **Pre** of preordered sets and monotone maps.

We have the following characterisation of monotonicity which the reader can readily check:

A.2.2 • Proposition. *Let P and Q be preordered sets and let $f\colon P \to Q$ be a function. Then the following are equivalent:*

(i) *f is monotone.*

(ii) *If $B \subseteq Q$ is upward closed, then $f^{-1}[B]$ is also upward closed.*

(iii) *If $B \subseteq Q$ is downward closed, then $f^{-1}[B]$ is also downward closed.* ■

A.2.2. Continuity

If P and Q are preordered sets, then it follows directly from Proposition A.2.2 that a map $f\colon P \to Q$ is monotone if and only if it is continuous with respect to the Alexandroff topology. If f instead is continuous with respect to the Scott topology, then f is called ***Scott continuous***.

A.2.3 • Lemma. *If $f\colon P \to Q$ is Scott continuous, then f is monotone and hence continuous with respect to the Alexandroff topology.*

Proof. Let $x, y \in P$ with $x \leq y$. The set $f(y)^{\downarrow}$ is closed in Q, so its preimage $f^{-1}[f(y)^{\downarrow}]$ is closed in P and is in particular downward closed. Hence it contains x, and so $f(x) \in f(y)^{\downarrow}$, which means that $f(x) \leq f(y)$. ■

There is a different characterisation of continuous maps between posets. If $f\colon P \to Q$ is a map between posets, then we say that f ***preserves joins*** if whenever $\bigvee A$ exists in P for a subset $A \subseteq P$, then $\bigvee f[A]$ exists in Q and $f(\bigvee A) \equiv \bigvee f[A]$. We often consider slightly different properties where we require f to preserve certain properties of subsets as well as joins of subsets with these properties. For instance, we say that f ***preserves directed joins*** if, whenever $D \subseteq P$ is directed and $\bigsqcup D$ exists in P, then $f[D]$ is also directed, $\bigsqcup f[D]$ exists in Q and $f(\bigsqcup D) \equiv \bigsqcup f[D]$. We then have the following:

A function that preserves directed joins is also often called continuous, but we reserve this name for functions that are continuous in the topological sense.

A.2.4 • Proposition. *Let P and Q be preordered sets. A map $f\colon P \to Q$ preserves directed joins if and only if it is Scott continuous.*

Proof. First assume that f preserves directed joins, and let $F \subseteq Q$ be closed in the Scott topology. Assume that $D \subseteq f^{-1}[F]$ is convergent and directed. Then $f[D] \subseteq F$, so

$$f\left(\bigsqcup D\right) = \bigsqcup f[D] \subseteq F,$$

which implies that $\bigsqcup D \in f^{-1}[F]$. We easily see that f is monotone, so Proposition A.2.2 implies that $f^{-1}[F]$ is also downward closed, hence closed in the Scott topology.

Conversely assume that f is Scott continuous, and let $D \subseteq P$ be convergent and directed. By Lemma A.2.3 f is monotone, so $f(\bigsqcup D)$ is an upper bound of $f[D]$. If x is any upper bound of $f[D]$, then $f[D] \subseteq x^{\downarrow}$, implying that $D \subseteq f^{-1}[x^{\downarrow}]$. But since $x^{\downarrow}$ is Scott closed so is $f^{-1}[x^{\downarrow}]$, and hence $\bigsqcup D \in f^{-1}[x^{\downarrow}]$. This just means that $f(\bigsqcup D) \in x^{\downarrow}$, so $f(\bigsqcup D) \leq x$ as required. ∎

This result gives another interpretation of the Scott topology: Namely the one in which a set U is open just when the characteristic function $\chi_U : P \to S$ preserves directed joins. (Here $S = \{0, 1\}$ is the set of truth values with $0 < 1$, as defined above.)

A.2.3. Expansive maps

This section is only needed in the proof of Theorem A.4.10. Readers can on a first pass skip this section and return to it as needed.

A map $f : P \to P$ on a poset P is called ***expansive*** if $x \leq f(x)$ for all $x \in P$. The following lemma describes how we in some cases can restrict f to a particularly nice subset of P on which P is monotone.

A ***sub-ccpo*** Q of P is a subset of P such that if C is a chain in Q with join $\bigvee C \in P$, then $\bigvee C$ also lies in Q. Furthermore, Q is a ***sub-ccppo*** of P if the bottom element of P also lies in Q.

A.2.5 • Lemma. *Let P be a ccppo and $f : P \to P$ be expansive. If P_0 is the smallest f-invariant sub-ccppo of P, then P_0 is a chain and $f|_{P_0}$ is monotone. In particular, P_0 is bounded.*[4]

[4]This result is adapted from Davey and Priestley (2002, Exercise 8.20).

Proof. We will call an element $x \in P_0$ a 'roof' if $y < x$ implies $f(y) \leq x$ for all $y \in P_0$. For a roof x we consider the set

$$Z_x = \{y \in P_0 \mid y \leq x \text{ or } f(x) \leq y\}.$$

We claim that Z_x is an f-invariant ccppo, so let $y \in Z_x$. If $y \leq x$, then either $y = x$ in which case $f(x) \leq f(y)$, or else $y < x$ so that $f(y) \leq x$ since x is a roof. If instead $f(x) \leq y$, then since f is expansive we have $f(x) \leq f(y)$. Hence $f(y) \in Z_x$, so Z_x is f-invariant. Next let $C \subseteq Z_x$ be a chain. If $y \leq x$ for all $y \in x$, then we also have $\bigvee C \subseteq x$. If instead there is some $y \in C$ with $f(x) \leq y$, then we clearly have $f(x) \leq \bigvee C$. Thus Z_x is a ccppo, so by minimality of P_0 we have $P_0 = Z_x$.

Next we claim that every element in P_0 is a roof. Consider the set

$$Z = \{x \in P_0 \mid x \text{ is a roof}\}.$$

We show that Z is an f-invariant ccppo. If x is a roof and $y \in P_0$, then $y \in Z_x$ and so either $y \leq x$ or $f(x) \leq y$. If $y < f(x)$ then we must have $y \leq x \leq f(x)$, so that $f(x)$ is also a roof, and so Z is f-invariant. If $C \subseteq Z$ is a chain and $y < \bigvee C$, then there is some $x \in C$ with $y < x \leq \bigvee C$, so Z is also chain cocomplete. Again by minimality we have $P_0 = Z$.

Now notice that P_0 is a chain: If $x, y \in P_0$, then x is a roof and so $y \in Z_x$, which implies that either $y \leq x$ or $x \leq f(x) \leq y$. If $x < y$, then necessarily $f(x) \leq y \leq f(y)$, so f is monotone on P_0. Finally, since itself P_0 is a chain the join $\bigvee P_0$ lies in[5] P_0 and is its maximum. ■

[5]This is where we use that P is chain cocomplete and not just ω-chain cocomplete, since P_0 may be uncountable. The rest of the proof goes through with this weaker assumption (with 'ccppo' replaced with 'ω-cppo'), but we shall not need this fact.

A.3 ⋄ Posets of partial maps

Recall from Example A.1.10(C) that a set $(X \rightharpoonup Y)$ of partial maps is a Scott domain whose finite elements are the finite partial maps. Unless Y is a singleton (or X is empty), $(X \rightharpoonup Y)$ is not topped: If $x \in X$ and $y_1, y_2 \in Y$ are distinct, then any maps with $x \mapsto y_1$ and $x \mapsto y_2$ respectively are inconsistent.

A.3.1. Topology

Recall that on $(X \rightharpoonup Y)$ the Scott topology and topology of pointwise convergence coincide by Proposition A.1.14. In this context we can better explain the meaning of 'pointwise convergence': If $(f_i)_{i \in I}$ is a sequence in $(X \rightharpoonup Y)$ which converges to some $f \colon X \rightharpoonup Y$, then this means that for every $g \leq_\omega f$ there exists an $i_0 \in I$ such that $i \geq i_0$ implies $g \leq f_i$. Since such a map g is uniquely determined by its domain, this just says that for every $A \subseteq_\omega \operatorname{dom} f$ there is such an i with $f_i|_A = f|_A$ when $i \geq i_0$. That is, the functions in the sequence eventually agree with f on arbitrarily large, but finite, subsets of X.

Note that the name 'topology of pointwise convergence' is usually given to the product topology on the set $(X \to Y)$ of total maps, when Y is a topological space. In this case 'pointwise' means that a sequence $(f_n)_{n \in \mathbb{N}}$ (or more generally a net) in $(X \to Y)$ converges to a function f if and only if the sequences $(f_n(x))_{n \in \mathbb{N}}$ converge to $f(x)$ for all $x \in X$.

On the other hand, every partial map f can be approximated by finite maps, though unless we know more about the structure of X we cannot necessarily approximate f with a *sequence*. Let $\mathcal{A} = \mathcal{P}_\omega(\operatorname{dom} f)$ and let $f_A = f|_A$. Then $(f_A)_{A \in \mathcal{A}}$ is a net, and it is clear that it converges to f.

Let $*$ be some element disjoint from Y, and let $Y_* := Y \cup \{*\}$. Every $f \colon X \rightharpoonup Y$ gives rise to a total function $f_* \colon X \to Y_*$ by

$$f_*(x) = \begin{cases} f(x), & x \in \operatorname{dom} f, \\ *, & x \notin \operatorname{dom} f. \end{cases}$$

This map clearly induces a bijection $(X \rightharpoonup Y) \xrightarrow{\sim} (X \to Y_*)$ given by $f \mapsto f_*$. Furthermore, for $x \in X$ define $\epsilon_x \colon (X \rightharpoonup Y) \to Y_*$ by $\epsilon_x(f) = f_*(x)$. We then have the following result:

A.3.1 • Proposition. *Equip Y_* with the discrete topology. Then $(X \rightharpoonup Y)$ along with the maps $(\epsilon_x)_{x \in X}$ is an X-fold product of Y_*.*

Proof. It suffices to show that if Z is a topological space and $(f_x)_{x \in X}$ is a family of continuous maps $f_x \colon Z \to Y_*$, then there is a unique continuous

$F\colon Z \to (X \rightharpoonup Y)$ such that the diagram

$$\begin{tikzcd}
 & & Y_* \\
Z \arrow[rr, dashed, "F"] \arrow[urr, bend left, "f_x"] & & (X \rightharpoonup Y) \arrow[u, "\epsilon_x"']
\end{tikzcd}$$

commutes for all $x \in X$. Existence of such a set function F as well as uniqueness are clear since the diagram forces, for $z \in Z$,

$$F(z)_*(x) = (\epsilon_x \circ F)(z) = f_x(z),$$

so $F(z)$ is the partial function with domain $\{x \in X \mid f_x(z) \neq *\}$ given by $F(z)(x) = f_x(z)$.

To see that F is continuous at z, let $g \leq_\omega F(z)$. For $x \in \operatorname{dom} g$ let $U_x \subseteq Z$ be a neighbourhood of z such that $f_x(w) = f_x(z)$ for $w \in U_x$.[6] Define $U := \bigcap_{x \in \operatorname{dom} g} U_x$ and note that U is a neighbourhood of z since $\operatorname{dom} g$ is finite. For $w \in U$ and $x \in \operatorname{dom} g$ we thus have

$$F(w)(x) = f_x(w) = f_x(z) = F(z)(x),$$

and hence $g \leq F(w)$. Thus F is continuous. ■

[6]This neighbourhood exists since f_x is continuous and Y_* is discrete.

Thus we have shown that the topology of pointwise convergence as defined above is a special case of 'the' topology of pointwise convergence, i.e., the product topology.

A.3.2. Compactness

There is another characterisation of continuity to which we now turn. It will turn out that continuity is equivalent to monotonicity along with the following property:

A.3.2 • Definition: *Compactness.*
A map $F\colon (X \rightharpoonup Y) \to (Z \rightharpoonup W)$ is called ***compact*** if it has the following property: For every $f\colon X \rightharpoonup Y$ and $z \in \operatorname{dom} F(f)$, there is a finite partial map $f_0 \leq f$ such that $z \in \operatorname{dom} F(f_0)$ and $F(f_0)(z) = F(f)(z)$. ▲

Informally, to compute $F(f)(z)$ we first compute $F(f)$ and then evaluate this function at z. But if F is compact, then it suffices to look at an appropriately chosen *finite* partial map f_0 instead of f. Or put another way, each value $F(f)(z)$ depends only on the values of f at finitely many points.

Using the properties of the topology of pointwise convergence studied above, it is easy to show that continuity implies compactness: Assume that F is continuous, consider $f\colon X \rightharpoonup Y$ and let $(f_A)_{A \in \mathcal{A}}$ be the net from §A.3.1 that converges to f. For each $z \in Z$ the composition $\epsilon_z \circ F$ is continuous, so

$$F(f_A)(z) = (\epsilon_z \circ F)(f_A) \to (\epsilon_z \circ F)(f) = F(f)(z).$$

Since W is discrete there is an A such that equality holds.

However, it is just as easy, and more elementary, to prove equivalence of compactness (plus monotonicity) and continuity using the other characterisations of continuity:

A.3.3 • Proposition. *Let $F\colon (X \rightharpoonup Y) \to (Z \rightharpoonup W)$ be a map. Then F is monotone and compact if and only if it is continuous.*

The assumption of monotonicity can be replaced with a sort of 'converse' of compactness, see Moschovakis (2006, Proposition 6.23).

Proof. Assume that F is monotone and compact. We prove that F preserves directed joins.[7] Let $D \subseteq (X \rightharpoonup Y)$ be directed, and since $\bigsqcup F[D] \leq F(\bigsqcup D)$ by monotonicity, it suffices to prove the other inequality. Letting $f := \bigsqcup D$, it suffices to show that for $z \in Z$ there is a $d \in D$ with $F(f)(z) = F(d)(z)$. But compactness yields an $f_0 \leq_\omega f$ such that $F(f)(z) = F(f_0)(z)$, and since f_0 finite there is a $d \in D$ with $f_0 \leq d$ as desired.

[7]The proof of this implication is based on Moschovakis (2006, Lemma 6.29).

For the converse, if F preserves directed joins, then it is monotone. To prove compactness, let $f\colon X \rightharpoonup Y$ and $z \in \operatorname{dom} F(f)$, and let $D := \{d\colon X \rightharpoonup Y \mid d \leq_\omega f\}$ such that $f = \bigsqcup D$. Furthermore, there is some $d \in D$ with $z \in \operatorname{dom} F(d)$. Hence

$$F(f)(z) = F\Big(\bigsqcup D\Big)(z) = \Big(\bigsqcup F[D]\Big)(z) = F(d)(z),$$

so F is compact. ■

A.4 ◇ A survey of fixed-point theorems

A.4.1. Introduction to fixed-points

In this section we prove some of the central fixed-point theorems in order theory. Let us immediately say what a fixed-point is in the first place:

A.4.1 • Definition. Let $f\colon P \to P$ be a map on a poset P. An element $x \in P$ is said to be

(a) ***f-closed*** if $f(x) \leq x$,

(b) ***f-consistent*** if $x \leq f(x)$, and

(c) a ***fixed-point*** of f if $f(x) = x$.

If f has a least (resp. greatest) fixed-point, then we denote this by $\mu(f)$ (resp. $\nu(f)$). ▲

Functions can have closed or consistent elements without having fixed-points, but we do have the following (which comes from Gunter 1992, §10.1):

A.4.2 • Lemma. *Let $f : P \to P$ be a monotone map on a poset. If f has a least closed element, then this is also a fixed-point of f, and indeed the least fixed-point.*

Proof. Let $x^* \in P$ be the least f-closed element. Then $f(f(x^*)) \leq f(x^*)$ so $f(x^*)$ is also closed, and hence $x^* \leq f(x^*)$. Thus x^* is a fixed-point. ■

The converse of Lemma A.4.2 does not hold in general. For instance, define $f : \mathbb{N}^{op} \to \mathbb{N}^{op}$ by $f(0) = 0$ and $f(n) = n - 1$ for $n < 0$. Then there is no minimal f-closed element, but 0 is the least fixed-point of f. It is also not sufficient to assume that the poset has a minimum: Just add a copy of $\mathbb{N}$ below $\mathbb{N}^{op}$ (i.e., consider the linear sum $\mathbb{N} \oplus \mathbb{N}^{op}$) and define $f(k) = k + 1$ for $k \in \mathbb{N}$.

In this section we prove some of the central fixed-point theorems in order theory. In practice, the only such theorem we strictly speaking need is an elementary version of Kleene's fixed-point theorem (cf. Theorem A.4.12). This ensures the existence of least fixed-points of *continuous* functions on ccpo's. The reader not interested in generalisations can skip straight to §A.4.3(c).

While we will not need to consider fixed-points of discontinuous functions in our study of programming languages, Kleene's theorem has a very natural extension to discontinuous monotone functions on dcpo's. It has various applications in pure mathematics (for instance in topology and measure theory) and it helps shed some light on what is significant about the recursive definitions we make in computer science. Its proof, however, is fairly technical, and we discuss how to circumvent its usage when needed. We present two different paths to a proof of this theorem:

- One goes through the so-called Zermelo's fixed-point theorem (cf. Theorem A.4.10) and is fairly elementary but technical. The proof is also not particularly illuminating (at least in the author's opinion), and it yields a weaker statement than the alternative proof.
- The other avoids Zermelo's theorem but instead uses some non-trivial results from set theory. In particular we need the notion of ordinal numbers, transfinite recursion and induction, and Hartogs' theorem. The upshot is the ability to perform transfinite iteration, i.e., compose a function with itself α many times for any ordinal number α.

  For completeness we also give a proof of Zermelo's theorem using the theory of ordinals we develop below.

We signpost which results and proofs pertain to which path.

While Kleene's fixed-point theorem is necessary to define *functions* recursively[8], defining *sets* by recursion is relatively easy. In this case we only need Knaster–Tarski's fixed point theorem (cf. Theorem A.4.9) for monotone functions on complete lattices, which avoids the discussion of continuity altogether. Its proof is also simple, though it has the disadvantage that it yields a less constructive description of the fixed-points.

[8]Necessary in our presentation; there are of course ways of proving recursion theorems that are more elementary, but also (in the author's opinion) less illuminating.

We begin by covering the set-theoretical background necessary to follow the second path mentioned above. Readers only interested in taking the first path can skip the next section without loss of continuity.

A.4.2. Set-theoretical preliminaries

If $f : X \to X$ is a function, then it is easy to define what it means to compose f by itself a finite number of times: We simply let $f^0 = \mathrm{id}_X$ and $f^{n+1} = f \circ f^n$. If X is a topological space, then we may even ask whether the limit $\lim_{n\to\infty} f^n$ exists in some sense. But it is not clear that this construction can be extended to an *ordinal* number of times. Below we consider two scenarios in which this is possible.

If P is a pointed poset, then we say that a function $f : P \to P$ is ***transfinitely iterable*** if there for each ordinal α exists an element $f^\alpha(\bot) \in P$ such that

$$\begin{aligned} f^0(\bot) &= \bot, \\ f^\alpha(\bot) &= f(f^{\alpha-1}(\bot)) && \text{if } \alpha \text{ is a successor,} \\ f^\alpha(\bot) &= \bigvee_{\beta<\alpha} f^\beta(\bot) && \text{if } \alpha \text{ is a limit.} \end{aligned}$$

The class function given by $\alpha \mapsto f^\alpha(\bot)$ is sometimes called the ***transfinite orbit*** of f. If there is an ordinal α such that $f^\alpha(\bot) = f^\beta(\bot)$ for all $\beta > \alpha$, then there is a least such α, and this is called the ***closure ordinal*** of f.

The definition of closure ordinals is adapted from Terese (2003, Definition A.3.9) where P is a power set. The special case $P = \mathcal{P}(\mathbb{N})$ is studied in Aanderaa (1974, Definition 4).

A.4.3 • Remark. We prove below that every monotone function f on a ccppo P is transfinitely iterable, and that if f is continuous then its closure ordinal is at most ω (cf. Theorems A.4.12 and A.4.13). In the case $P = \mathcal{P}(\mathbb{N})$ one can show that the closure ordinal of f is countable (this is noted in Aanderaa 1974, Remark 1 without proof), but as we mention in Example A.4.5 this is not generally the case when the underlying set is uncountable. ⌟

A.4.4 • Example: Topologies.
Let X be a set, let $\mathcal{S}$ be a collection of subsets of X, and consider the inference rules

$$\begin{aligned} &\Rrightarrow S && \text{for } S \in \mathcal{S}, \\ \mathcal{U} &\Rrightarrow \bigcup \mathcal{U} && \text{for } \mathcal{U} \subseteq \mathcal{P}(X), \\ \mathcal{U} &\Rrightarrow \bigcap \mathcal{U} && \text{for } \mathcal{U} \subseteq \mathcal{P}_\omega(X). \end{aligned}$$

Let F be the represented generating function, and let $\mathcal{O}$ be its least fixed-point, i.e. the topology generated by $\mathcal{S}$. We claim that the closure ordinal of F is 3: For $F^1(\emptyset)$ contains $\mathcal{S}$, $F^2(\emptyset)$ contains all finite intersections of elements in $\mathcal{S}$, and $F^3(\emptyset)$ contains all unions of those intersections. It is clear that 3 is minimal.

Consider instead the generating function G generated by the same rules, except that the rules $\mathcal{U} \Rrightarrow \bigcap \mathcal{U}$ are replaced by their 'biased' versions, namely $\Rrightarrow X$ and $U, V \Rrightarrow U \cap V$ for all $U, V \subseteq X$. We claim that the closure ordinal of G is at most $\omega + 1$, and that this bound is optimal.

Notice that intersections of n sets from $\mathcal{S}$ lie in $G^n(\emptyset)$, and hence all finite intersections are contained in $G^\omega(\emptyset)$. Applying G a final time yields the union of these intersections, so $\mathcal{O} = G^{\omega+1}(\emptyset)$.

For optimality we first claim that if $n \geq 1$, then every set in $G^n(\emptyset)$ is on the form

$$\bigcup_{i\in I}(U_{i,1} \cap \cdots \cap U_{i,2^{n-1}})$$

for some index set I and some $U_{i,j} \in \mathcal{S} \cup \{X\}$ (the case $I = \emptyset$ yielding the empty set). This is clear for $n = 1$. If $n > 1$, notice that the class of these sets is closed under unions, so it suffices to consider binary intersections. If

$$\bigcup_{i\in I}(U_{i,1} \cap \cdots \cap U_{i,2^{n-1}}) \quad \text{and} \quad \bigcup_{j\in J}(V_{j,1} \cap \cdots \cap V_{j,2^{n-1}})$$

are sets in $G^n(\emptyset)$, then their intersection is

$$\bigcup_{(i,j)\in I\times J} (U_{i,1} \cap \cdots \cap U_{i,2^{n-1}} \cap V_{j,1} \cap \cdots \cap V_{j,2^{n-1}}).$$

And each element of the union is the intersection of 2^n sets from $\mathcal{S}$ as claimed.

Consider now the following example: Let $p_1, p_2, \ldots$ be an enumeration of the primes, equip $\mathbb{Z}$ with the topology generated by the sets $p_i\mathbb{Z}$, and equip $X = \coprod_{n\in\mathbb{N}} \mathbb{Z}$ with the corresponding disjoint union topology. Let $U_n = \bigcap_{i=1}^n p_i\mathbb{Z}$ and consider the set $U = \coprod_{n\in\mathbb{N}} U_n$ in U. Then U is open in X, but it does not lie in $F^n(\emptyset)$ for any finite n since its summand U_{2^n} is the intersection of too many subbasic sets. ⌟

A.4.5 • Example: σ-algebras.
Let X be a set, and let $\mathcal{D}$ be a collection of subsets of X. Consider the inference rules (with potentially infinitely many antecedents)

$$\begin{aligned} &\Rightarrow X, && \\ &\Rightarrow D && \text{for } D \in \mathcal{D}, \\ A &\Rightarrow X \setminus A && \text{for } A \subseteq X, \\ \mathcal{A} &\Rightarrow \bigcup \mathcal{A} && \text{for } \mathcal{A} \subseteq \mathcal{P}_{\omega_1}(X). \end{aligned}$$

The function F generated by these rules then has at its least fixed-point the σ-algebra $\sigma(\mathcal{D})$ generated by $\mathcal{D}$. Assuming the axiom of countable choice (ACC), it is not difficult to show by transfinite induction that the closure ordinal of F is at most ω_1 (cf. Folland 2007, Proposition 1.23). In fact, if X is an uncountable Polish space and $\mathcal{D}$ is its topology—so that $\sigma(\mathcal{D})$ becomes the Borel algebra on X—one can show, again assuming ACC, that the closure ordinal of F is indeed ω_1, though this is slightly technical (cf. Kechris 1995, Theorem 22.4).

When X is a separable metric space and $\mathcal{D}$ its topology, we call the closure ordinal of F the ***Baire order*** of X. The above thus means that under ACC, all uncountable Polish spaces have Baire order ω_1.

If some choice axiom is not assumed, then things work out less nicely: It is both consistent with ZF that the Baire order of the Cantor space is 4, and that it is ω_2 (cf. Miller 2008). ⌟

The discussion thus progresses in two steps: First we study when functions are transfinitely iterable, and later in the context of fixed-points we study when functions have closure ordinals and what they are.

(a) *Monotone functions.* We first prove that transfinite composition is possible for monotone functions on ccppo's.

A.4.6 • Lemma: *Iteration I.*
Let P be a ccppo and let $f\colon P \to P$ be monotone. For each ordinal α there exists a map $f^\alpha\colon P \to P$ such that

$$\begin{aligned} &f^0(x) = x, \\ &f^\alpha(x) = f(f^{\alpha-1}(x)) && \textit{if } \alpha \textit{ is a successor}, \\ &f^\alpha(x) = \bigvee_{\beta<\alpha} f^\beta(x) && \textit{if } \alpha \textit{ is a limit}, \end{aligned}$$

for all $x \in P$. Furthermore, all f^α are monotone. In particular, f is transfinitely iterable.

Proof. Fix some $x \in P$. We define a binary operation $\Phi = \Phi_x$ taking as arguments an ordinal α and a function $g\colon \alpha \to P$: We first let $\Phi(g,0) = x$. If α is a successor, then we let $\Phi(g,\alpha) = f(g(\alpha-1))$. If α is a limit ordinal and g is monotone, then we let $\Phi(g,\alpha) = \bigvee_{\beta<\alpha} g(\beta)$. (Note that we indeed take the join of a chain since α is a chain and g is assumed monotone.) Finally, if α is a limit ordinal but g is not monotone, then we let $\Phi(g,\alpha)$ be some arbitrary element of P (we will not need to consider such g). By transfinite recursion there thus exists a unary (definite) operation $\Pi = \Pi_x$ such that

$$\Pi(\alpha) = \Phi(\Pi|_\alpha, \alpha)$$

for all α.

We prove by transfinite induction that $\Pi|_\alpha$ is monotone for all α. For $\alpha = 0$ this is obvious, so assume that $\Pi|_\xi$ is monotone for all $\xi < \alpha$ and let $\beta < \gamma < \alpha$. We consider three cases:

β and γ are successors: Then $\Pi|_\gamma$ is monotone, and since f is also monotone we have

$$\Pi(\beta) = f(\Pi(\beta-1)) \leq f(\Pi(\gamma-1)) = \Pi(\gamma).$$

β is a limit, γ is a successor: Since β is the union of all *successor* ordinals smaller than it[9], it suffices to show that $\Pi(\xi) \leq \Pi(\gamma)$ if $\xi < \beta$ is a successor ordinal. But since $\Pi|_\gamma$ is monotone, we similarly to above find that

$$\Pi(\xi) = f(\Pi(\xi-1)) \leq f(\Pi(\gamma-1)) = \Pi(\gamma).$$

[9]This follows since if $\xi < \beta$, then also $\xi + 1 < \beta$.

β is a successor, γ is a limit: Let ξ be a successor ordinal with $\beta \leq \xi < \gamma$. Since $\Pi|_\gamma$ is monotone, we again have

$$\Pi(\beta) = f(\Pi(\beta - 1)) \leq f(\Pi(\xi - 1)) = \Pi(\xi),$$

which implies that

$$\Pi(\beta) \leq \bigvee_{\xi<\gamma} \Pi(\xi) = \Pi(\gamma).$$

Hence $\Pi|_\alpha$ is monotone, and we thus always have $\Pi(\alpha) = \bigvee_{\beta<\alpha} \Pi(\beta)$ when α is a limit ordinal. Writing $f^\alpha(x) := \Pi_x(\alpha)$ we thus obtain a map $f^\alpha \colon P \to P$, and we have

$$\begin{aligned} &f^0(x) = x, \\ &f^\alpha(x) = f(f^{\alpha-1}(x)) && \text{if } \alpha \text{ is a successor,} \\ &f^\alpha(x) = \bigvee_{\beta<\alpha} f^\beta(x) && \text{if } \alpha \text{ is a limit,} \end{aligned}$$

for all $x \in P$ and ordinals α.

Next we show that the map f^α is monotone for all ordinals α, i.e. that if $x, y \in P$ with $x \leq y$, then $f^\alpha(x) \leq f^\alpha(y)$. If $\alpha = 0$, then this is obvious, so assume that it holds for all ordinals $\beta < \alpha$. If α is a successor, then

$$f^\alpha(x) = f(f^{\alpha-1}(x)) \leq f(f^{\alpha-1}(y)) = f^\alpha(y).$$

If instead α is a limit, then

$$f^\beta(x) \leq f^\beta(y) \leq \bigvee_{\beta<\alpha} f^\beta(y) = f^\alpha(y)$$

for all $\beta < \alpha$, and taking the join on the left-hand side we again get $f^\alpha(x) \leq f^\alpha(y)$ as desired. ■

(b) ***Expansive functions.*** We obtain the second iteration lemma as a corollary of the first, but we note that it can also be proved from first principles by an argument very similar to that in the proof of the first lemma. We refer to Moschovakis (2006, Lemma 7.25) for such a proof. This result will only be used in the proof of Zermelo's fixed-point theorem (cf. Theorem A.4.10) which is redundant in the presence of the Kleene's general fixed-point theorem (cf. Theorem A.4.13), and it can thus be skipped without loss of continuity.

A.4.7 • Lemma: ***Iteration II.***
Let P be a ccppo and let $f \colon P \to P$ be expansive. Then f is transfinitely iterable. Furthermore, if $\alpha \leq \beta$ then $f^\alpha(\bot) \leq f^\beta(\bot)$.

Proof. By Lemma A.2.5 f restricts to a monotone map on some sub-ccppo P_0 of P. And P_0 contains $\bot$, so Lemma A.4.6 yields the existence part of the lemma. The last claim follows easily by induction in β: If β is a successor, then it follows since f is expansive, and if β is a limit, then it follows by the definition of f^α. ■

(c) *Hartogs' theorem.* The final result we need from set theory is the following. It is a standard result whose proof is less elementary than that of the above lemmas, so we simply refer to the literature.

A.4.8 • Theorem: *Hartogs' theorem.*
For every set X there is an ordinal α such that there is no injection $\alpha \hookrightarrow X$. In particular, there is no greatest ordinal.

Proof. See Moschovakis (2006, Theorem 7.34 and Exercise 12.13) or Goldrei (1996, Theorem 8.18). ■

A.4.3. Fixed-point theorems

(a) *Knaster–Tarski.* The fixed-point theorem which is most elementary is the one by Knaster and Tarski for a monotone map f on a complete lattice L. Its proof is simple, and while it gives both the existence of a least and a greatest fixed-point and characterises these among the f-closed and -consistent elements of L, it provides no computationally relevant description of these fixed-points. But despite its simplicity, it is very useful in pure mathematics.

The proof of Theorem A.4.9 is based on Davey and Priestley (2002, Theorem 2.35).

A.4.9 • Theorem: *Knaster–Tarski's fixed-point theorem.*
If L is a complete lattice and $f : L \to L$ is monotone, then f has a least and a greatest fixed-point, and these are given by

$$\mu(f) = \bigwedge\{x \in L \mid f(x) \leq x\} \quad \text{and} \quad \nu(f) = \bigvee\{x \in L \mid x \leq f(x)\}.$$

In particular, $\mu(f)$ is the smallest f-closed element and $\nu(f)$ is the greatest f-consistent element in L.

Proof. Denote the meet above by x^*. If x is f-closed, then $x^* \leq x$, so $f(x^*) \leq f(x) \leq x$. Taking the meet of x we get $f(x^*) \leq x^*$, so x^* is closed. It follows that $f(f(x^*)) \leq f(x^*)$, so $f(x^*)$ is also closed, and so $x^* \leq f(x^*)$. Hence x^* is a fixed-point. Since every other fixed-point is in particular closed, x^* is the least fixed-point. ■

(b) *Zermelo.* A definition we will need in the proof: The ***lift*** of a poset P is the poset $P_\perp$ with underlying set $P \cup \{\perp\}$ (where $\perp \notin P$), and $\perp < x$ for all $x \in P$.

A.4.10 • Theorem: *Zermelo's fixed-point theorem.*
If P is a ccpo and $F : P \to P$ is expansive, then F has a fixed-point.

Proof without ordinals. Consider instead the lift $P_\perp$, which is a ccppo. Extend F to $P_\perp$ by letting $F(\perp)$ be some element of P (that is, $F(\perp) \neq \perp$). Let P_0 be the smallest F-invariant sub-ccppo of $P_\perp$. Lemma A.2.5 then implies that P_0 has a greatest element $\top$, and $\top \leq F(\top)$ since F is expansive. But $F(\top) \in P_0$ by invariance, so $\top$ is a fixed-point of F. Notice that since $F(\perp) \neq \perp$, $\perp$ is not a fixed-point of F, and so F has a fixed-point lying in P, proving the original claim. ■

Proof using ordinals. Again consider $P_\perp$ and extend F similarly. Let α be an ordinal such that there is no injection $\alpha \hookrightarrow P_\perp$. Lemma A.4.7 then implies that there are two distinct ordinals β and γ such that $F^\beta(\perp) = F^\gamma(\perp)$. Assuming without loss of generality that $\beta < \gamma$, monotonicity implies that

$$F^\beta(\perp) \leq F^{\beta+1}(\perp) \leq F^\gamma(\perp),$$

so $F^\beta(\perp)$ is a fixed-point of F. Again it cannot equal $\perp$, so it lies in P. ■

A.4.11 • *Remark.* Our proof of Theorem A.4.10 is based on Davey and Priestley (2002, Exercise 8.20). The authors give this result the name 'CPO Fixpoint Theorem III', and they add the hypothesis that P be a dcppo, but also claim that then F has a *minimal* fixed-point. But this is false: Consider for instance the poset $P = (\omega + 1)^{op}$, i.e., the set $\omega \cup \{\omega\}$ equipped with the ordering $\preceq$ given by

$$\omega \prec \cdots \prec n \prec n-1 \prec \cdots \prec 1 \prec 0.$$

This is clearly a dcppo since every nonempty subset even has a maximum. Define $F\colon P \to P$ by letting $F(\omega) = 0$ and $F(n) = n$ for $n \in \omega$. Then F is obviously expansive, but every natural number, of which there is no minimal with respect to $\preceq$, is a fixed-point. In the context of the proof below, Davey and Priestley claim that the smallest F-invariant sub-dcppo of P, here $\{\omega, 0\}$, has a greatest element, here 0, and that this is a minimal fixed-point of F. But this is false, since e.g. 1 is also a fixed-point.

The above counterexample is taken from Hansen (2023). ⌟

(c) ***Kleene.*** We finally arrive at the most (in a sense only) important fixed-point theorems, namely that due to Kleene as well as a generalisation. For readers that have skipped to here from §A.4.1 and that are not interested in generalisations it suffices to note Theorem A.4.12 below. Other readers can also study Theorem A.4.13, but note that this is not a strict generalisation of the former since it assumes that the poset in question is a ccppo and not just an ω-cppo.

Theorem A.4.12 is an immediate generalisation of Davey and Priestley (2002, Theorem 8.15) from dcppo's to ω-cppo's.

A.4.12 • **Theorem:** ***Kleene's fixed-point theorem I.***
If P is an ω-cppo and $f\colon P \to P$ is **ω-continuous**[10]*, then f has a least fixed-point given by*

$$\mu(f) = \bigvee_{n \in \mathbb{N}} f^n(\perp).$$

Furthermore, $\mu(f)$ is the least f-closed element in L

[10]Meaning that f preserves joins of ω-chains.

Notice that this just means that $\mu(f) = f^\omega(\perp)$.

Proof. First notice that, since f is continuous,

$$f\left(\bigvee_{n \in \mathbb{N}} f^n(\perp)\right) = \bigvee_{n \in \mathbb{N}} f^{n+1}(\perp) = \bigvee_{n \in \mathbb{N}^+} f^n(\perp) = \bigvee_{n \in \mathbb{N}} f^n(\perp),$$

where we use that $f^0(\bot) = \bot$. Hence $\bigvee_{n\in\mathbb{N}} f^n(\bot)$ is indeed a fixed-point of f. If $x \in P$ is f-closed, then $\bot \leq x$, and hence $f^n(\bot) \leq f^n(x) \leq x$ since f is monotone. Taking the join on the left-hand side yields $\bigvee_{n\in\mathbb{N}} f^n(\bot) \leq x$ as desired. ∎

In the case where f is ω-continuous, it is thus easy to show that f has a fixed-point and even give a fairly explicit formula for it. Furthermore, if P is a powerset and the elements $f^n(\bot)$ are thus sets, the theorem says that an element y lies in $\mu(f)$ if and only if it lies in some $f^n(\bot)$.

If f is not continuous, we are not so lucky. However, since $\omega = (\mathbb{N}, \leq)$ is an ordinal we may be inspired to consider the transfinite orbit $\alpha \mapsto f^\alpha(\bot)$ of f and attempt to find a fixed-point among its image. This is indeed possible, but since we cannot be sure that α is countable, we must assume that P is chain-complete.

We again present two proofs, one going through Zermelo's fixed-point theorem that is based on Moschovakis (2006, Theorem 7.36), and one using ordinals that is based on Davey and Priestley (2002, Exercise 8.19). Since the statement of the theorem itself mentions ordinals, the first of the two proofs of course only gives us the existence of a fixed-point and not the formula for it we display below.

There is also a proof of Theorem A.4.13 that avoids both ordinals and Zermelo's theorem that is due to Pataraia. This instead uses Knaster–Tarski's fixed-point theorem. However, as far as the author can tell, it requires P to be directed cocomplete, and as with the proof going through Zermelo's theorem it does not express the fixed-point in terms of transfinite orbits. (Cf. Davey and Priestley 2002, Theorem 8.22.)

A.4.13 • Theorem: ***Kleene's fixed-point theorem II.***
Let P be a ccppo and let $f\colon P \to P$ be monotone. Let

$$\begin{aligned} f^0(\bot) &= \bot, \\ f^\alpha(\bot) &= f(f^{\alpha-1}(\bot)) && \textit{if } \alpha \textit{ is a successor}, \\ f^\alpha(\bot) &= \bigvee_{\beta<\alpha} f^\beta(\bot) && \textit{if } \alpha \textit{ is a limit}, \end{aligned}$$

for all ordinals α. Then $\mu(f) = f^\alpha(\bot)$ for some ordinal α, and $\mu(f)$ is the least f-closed element in P.

Proof without ordinals. Let Q be the subset of P consisting of elements x that are f-consistent, and which satisfy $x \leq y$ for all f-closed elements $y \in P$. We claim that Q is an f-invariant sub-ccppo of P, so let first $x \in Q$. Then $x \leq f(x)$, so $f(x) \leq f(f(x))$ by monotonicity. For closed y we similarly have $f(x) \leq f(y) \leq y$, so $f(x) \in Q$. Next let $C \subseteq Q$ be a chain. For each $x \in C$ we have

$$x \leq f(x) \leq f\Bigl(\bigvee C\Bigr),$$

implying that $\bigvee C \leq f(\bigvee C)$. If $y \in P$ closed, then it is an upper bound of C, and so $\bigvee C \leq y$. Hence $\bigvee C \in Q$, so Q is indeed chain-complete.

In total, f restricts to an expansive map on the ccppo Q, so Theorem A.4.10 yields a fixed-point $x^* \in Q$ of f. By definition of Q, x^* is smaller than all f-closed elements in P, so it is also the least fixed-point of f. ∎

Proof using ordinals. By Hartogs' theorem (cf. Theorem A.4.8) there is some ordinal α such that there is no injection $\alpha \hookrightarrow P$. On the other hand, $\beta \mapsto f^\beta(\bot)$ is a function $\alpha \to P$, so this cannot be injective. Hence there are distinct ordinals $\beta, \gamma < \alpha$ with $f^\beta(\bot) = f^\gamma(\bot)$. Because α is totally ordered we may assume without loss of generality that $\beta < \gamma$. Since the map $\beta \mapsto f^\beta(\bot)$ is monotone and $\beta + 1 \leq \gamma$, this implies that

$$f^\beta(\bot) \leq f^{\beta+1}(\bot) \leq f^\gamma(\bot),$$

and hence

$$f(f^\beta(\bot)) = f^{\beta+1}(\bot) = f^\beta(\bot).$$

Thus $f^\beta(\bot)$ is a fixed-point of f.

To show that f has a *least* fixed-point, let $x \in P$ is f-closed. It is then clear by induction that $f^\alpha(x) \leq x$ for any α: If α is a successor, then

$$f^\alpha(x) = f(f^{\alpha-1}(x)) \leq f(x) \leq x,$$

and if α is a limit, then $f^\beta(x) \leq x$ for all $\beta < \alpha$, and taking the join on the left-hand side yields $f^\alpha(x) \leq x$. Hence if α is such that $f^\alpha(\bot)$ is a fixed-point, then $\bot \leq x$, and so

$$f^\alpha(\bot) \leq f^\alpha(x) \leq x,$$

so $f^\alpha(\bot)$ is indeed the least fixed-point of f, as well as the least f-closed element in P. ■

Syntax and Semantics B

B.1 ◇ Syntax

$$
\begin{array}{llcll}
 & x, f & \in & \mathit{Var} & \\
 & l & \in & \mathit{Loc} & \\
 & \alpha & \in & \mathit{TypeVar} & \\
\mathit{Exp} & e & ::= & 1 & \text{(unit value)} \\
 & & \mid & x & \text{(variables)} \\
 & & \mid & \langle e, e \rangle \mid \pi_1\, e \mid \pi_2\, e & \text{(products)} \\
 & & \mid & \iota_1\, e \mid \iota_2\, e \mid \mathsf{match}(e, x, e, e) & \text{(sums)} \\
 & & \mid & \lambda x.e \mid \mathsf{rec}\, f(x) := e \mid e\, e & \text{(functions)} \\
 & & \mid & \Lambda\_.e \mid e\, \_ \mid \mathsf{pack}\, e \mid \mathsf{unpack}(e, x, e) & \text{(polymorphism)} \\
 & & \mid & \mathsf{fold}\, e \mid \mathsf{unfold}\, e & \text{(recursion)} \\
 & & \mid & l \mid \mathsf{ref}\, e \mid e := e \mid {!}\, e & \text{(references)} \\
\mathit{Val} & v & ::= & 1 & \text{(unit value)} \\
 & & \mid & \langle v, v \rangle & \text{(value pairs)} \\
 & & \mid & \iota_1\, v \mid \iota_2\, v & \text{(sum values)} \\
 & & \mid & \lambda x.e \mid \mathsf{rec}\, f(x) := e & \text{(function values)} \\
 & & \mid & \Lambda\_.e \mid \mathsf{pack}\, v & \text{(polymorphic values)} \\
 & & \mid & \mathsf{fold}\, v & \text{(recursive values)} \\
 & & \mid & l & \text{(reference values)} \\
\mathit{Type} & \tau & ::= & 1 & \text{(unit type)} \\
 & & \mid & \alpha & \text{(type variables)} \\
 & & \mid & \tau \times \tau & \text{(product types)} \\
 & & \mid & \tau + \tau & \text{(sum types)} \\
 & & \mid & \tau \to \tau & \text{(function types)} \\
 & & \mid & \forall \alpha.\tau \mid \exists \alpha.\tau & \text{(polymorphic types)} \\
 & & \mid & \mu \alpha.\tau & \text{(recursive types)} \\
 & & \mid & \mathsf{Ref}\, \tau & \text{(reference types)} \\
\mathit{ECtx} & E & ::= & - & \text{(hole)} \\
 & & \mid & \langle E, e \rangle \mid \langle v, E \rangle \mid \pi_1\, E \mid \pi_2\, E & \text{(products)} \\
 & & \mid & \iota_1\, E \mid \iota_2\, E \mid \mathsf{match}(E, x, e, e) & \text{(sums)} \\
 & & \mid & E\, e \mid v\, E & \text{(function application)} \\
 & & \mid & E\, \_ \mid \mathsf{pack}\, E \mid \mathsf{unpack}(E, x, e) & \text{(polymorphism)} \\
 & & \mid & \mathsf{fold}\, E \mid \mathsf{unfold}\, E & \text{(recursion)} \\
 & & \mid & l \mid \mathsf{ref}\, E \mid E := e \mid v := E \mid {!}\, E & \text{(references)}
\end{array}
$$

B.2 ⋄ Static semantics

B.2.1. Basic features

T-VAR
$$\frac{\Xi \vdash \Gamma \qquad \Xi \vdash \Sigma \qquad \Gamma(x) = \tau}{\Xi \mid \Gamma \mid \Sigma \vdash x : \tau}$$

T-UNIT
$$\frac{\Xi \vdash \Gamma \qquad \Xi \vdash \Sigma}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{1} : \mathsf{1}}$$

T-PAIR
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \qquad \Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

T-PROJ1
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau_1 \times \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \pi_1\, e : \tau_1}$$

T-PROJ2
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau_1 \times \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \pi_2\, e : \tau_2}$$

T-INJ1
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau_1}{\Xi \mid \Gamma \mid \Sigma \vdash \iota_1\, e : \tau_1 + \tau_2}$$

T-INJ2
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \iota_2\, e : \tau_1 + \tau_2}$$

T-MATCH
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau_1 + \tau_2 \qquad \Xi \mid \Gamma, x : \tau_1 \mid \Sigma \vdash e_1 : \tau \qquad \Xi \mid \Gamma, x : \tau_2 \mid \Sigma \vdash e_2 : \tau}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{match}(e, x, e_1, e_2) : \tau}$$

T-LAM
$$\frac{\Xi \mid \Gamma, x : \tau_1 \mid \Sigma \vdash e : \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

T-APP
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \qquad \Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_1 \to \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash e_2\, e_1 : \tau_2}$$

B.2.2. Polymorphism

T-TLAM
$$\frac{\Xi, \alpha \mid \Gamma \mid \Sigma \vdash e : \tau \qquad \alpha \notin \mathrm{FV}_{\mathit{Type}}(\Gamma) \qquad \alpha \notin \mathrm{FV}_{\mathit{Type}}(\Sigma)}{\Xi \mid \Gamma \mid \Sigma \vdash \Lambda\_.e : \forall \alpha.\tau}$$

T-TAPP
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \forall \alpha.\tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \mid \Sigma \vdash e\,\_ : \tau[\tau'/\alpha]}$$

T-PACK
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau[\tau'/\alpha]}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\, e : \exists \alpha.\tau}$$

T-UNPACK
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \exists \alpha.\tau \qquad \Xi, \alpha \mid \Gamma, x : \tau \mid \Sigma \vdash e_2 : \tau'}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{unpack}(e_1, x, e_2) : \tau'}$$

B.2.3. Recursive functions and types

T-REC
$$\frac{\Xi \mid \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \mid \Sigma \vdash e : \tau_2}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{rec}\, f(x) := e : \tau_1 \to \tau_2}$$

T-FOLD
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{fold}\, e : \mu\alpha.\tau}$$

T-UNFOLD
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \mu\alpha.\tau}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{unfold}\, e : \tau[\mu\alpha.\tau/\alpha]}$$

B.2.4. References

T-LOC
$$\frac{\Xi \vdash \Gamma \qquad \Xi \vdash \Sigma \qquad l \in \operatorname{dom} \Sigma}{\Xi \mid \Gamma \mid \Sigma \vdash l : \mathsf{Ref}\, \Sigma(l)}$$

T-ALLOC
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \tau}{\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{ref}\, e : \mathsf{Ref}\, \tau}$$

T-STORE
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \mathsf{Ref}\, \tau \qquad \Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau}{\Xi \mid \Gamma \mid \Sigma \vdash e_1 := e_2 : \mathsf{1}}$$

T-LOAD
$$\frac{\Xi \mid \Gamma \mid \Sigma \vdash e : \mathsf{Ref}\, \tau}{\Xi \mid \Gamma \mid \Sigma \vdash\, !e : \tau}$$

B.3 ⋄ Dynamic semantics

B.3.1. Pure head reductions

E-PROJ1
$$\frac{}{\pi_1 \langle v_1, v_2 \rangle \rightarrow_p v_1}$$

E-PROJ2
$$\frac{}{\pi_2 \langle v_1, v_2 \rangle \rightarrow_p v_2}$$

E-MATCH-INJ1
$$\frac{}{\mathsf{match}(\iota_1\, v, x, e_1, e_2) \rightarrow_p e_1[v/x]}$$

E-MATCH-INJ2
$$\frac{}{\mathsf{match}(\iota_2\, v, x, e_1, e_2) \rightarrow_p e_2[v/x]}$$

E-LAM-APP
$$\frac{}{(\lambda x.e)\, v \rightarrow_p e[v/x]}$$

E-REC-APP
$$\frac{}{(\mathsf{rec}\, f\,(x) := e)\, v \rightarrow_p e[(\mathsf{rec}\, f\,(x) := e)/f][v/x]}$$

E-TAPP-TLAM
$$\frac{}{(\Lambda \_.e)\, \_ \rightarrow_p e}$$

E-UNPACK-PACK
$$\frac{}{\mathsf{unpack}(\mathsf{pack}\, v, x, e) \rightarrow_p e[v/x]}$$

E-UNFOLD-FOLD
$$\frac{}{\mathsf{unfold}\,(\mathsf{fold}\, v) \rightarrow_p v}$$

B.3.2. Impure head reductions

E-PURE
$$\frac{e \rightarrow_p e'}{(\sigma, e) \rightarrow_h (\sigma, e')}$$

E-ALLOC
$$\frac{l \notin \mathrm{dom}\, \sigma}{(\sigma, \mathsf{ref}\, v) \rightarrow_h (\sigma[l \mapsto v], l)}$$

E-STORE
$$\frac{l \in \mathrm{dom}\, \sigma}{(\sigma, l := v) \rightarrow_h (\sigma[l \mapsto v], \mathsf{1})}$$

E-LOAD
$$\frac{\sigma(l) = v}{(\sigma, !\, l) \rightarrow_h (\sigma, v)}$$

B.3.3. Evaluation contexts

E-HEAD
$$\frac{(\sigma, e) \rightarrow_h (\sigma', e')}{(\sigma, E[e]) \rightarrow (\sigma', E[e'])}$$

B.4 ⋄ Compatibility rules

R-VAR
$$\frac{\Xi \vdash \Gamma \qquad \Gamma(x) = \tau}{\Xi \mid \Gamma \vdash x \mathrel{\mathcal{R}} x : \tau}$$

R-UNIT
$$\frac{}{\Xi \mid \Gamma \vdash \mathsf{1} \mathrel{\mathcal{R}} \mathsf{1} : \mathsf{1}}$$

R-PAIR
$$\frac{\Xi \mid \Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1' : \tau_1 \qquad \Xi \mid \Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2' : \tau_2}{\Xi \mid \Gamma \vdash \langle e_1, e_2 \rangle \mathrel{\mathcal{R}} \langle e_1', e_2' \rangle : \tau_1 \times \tau_2}$$

R-PROJ1
$$\frac{\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau_1 \times \tau_2}{\Xi \mid \Gamma \vdash \pi_1\, e \mathrel{\mathcal{R}} \pi_1\, e' : \tau_1}$$

R-PROJ2
$$\frac{\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau_1 \times \tau_2}{\Xi \mid \Gamma \vdash \pi_2\, e \mathrel{\mathcal{R}} \pi_2\, e' : \tau_2}$$

R-INJ1
$$\frac{\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau_1}{\Xi \mid \Gamma \vdash \iota_1\, e \mathrel{\mathcal{R}} \iota_2\, e' : \tau_1 + \tau_2}$$

R-INJ2
$$\frac{\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau_2}{\Xi \mid \Gamma \vdash \iota_2\, e \mathrel{\mathcal{R}} \iota_2\, e' : \tau_1 + \tau_2}$$

R-MATCH
$$\frac{\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau_1 + \tau_2 \qquad \Xi \mid \Gamma, x : \tau_1 \vdash e_1 \mathrel{\mathcal{R}} e_1' : \tau \qquad \Xi \mid \Gamma, x : \tau_2 \vdash e_2 \mathrel{\mathcal{R}} e_2' : \tau}{\Xi \mid \Gamma \vdash \mathsf{match}(e, x, e_1, e_2) \mathrel{\mathcal{R}} \mathsf{match}(e', x, e_1', e_2') : \tau}$$

R-LAM
$$\frac{\Xi \mid \Gamma, x : \tau_1 \vdash e \mathrel{\mathcal{R}} e' : \tau_2}{\Xi \mid \Gamma \vdash \lambda x.e \mathrel{\mathcal{R}} \lambda x.e' : \tau_1 \to \tau_2}$$

R-APP
$$\frac{\Xi \mid \Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1' : \tau_1 \qquad \Xi \mid \Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2' : \tau_1 \to \tau_2}{\Xi \mid \Gamma \vdash e_2\, e_1 \mathrel{\mathcal{R}} e_2'\, e_1' : \tau_2}$$

R-TLAM
$$\frac{\Xi, \alpha \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \tau \qquad \alpha \notin \mathrm{FV}_{Type}(\Gamma)}{\Xi \mid \Gamma \vdash \Lambda\_.e \mathrel{\mathcal{R}} \Lambda\_.e' : \forall \alpha.\tau}$$

R-TAPP
$$\frac{\Xi \mid \Gamma \vdash e \mathrel{\mathcal{R}} e' : \forall \alpha.\tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \vdash e\,\_ \mathrel{\mathcal{R}} e'\,\_ : \tau[\tau'/\alpha]}$$

Bibliography

Aanderaa, Stål (1974). 'Inductive Definitions and Their Closure Ordinals'. In: *Generalized Recursion Theory*. Ed. by J.E. Fenstad and P.G. Hinman. Vol. 79. Studies in Logic and the Foundations of Mathematics. Elsevier, pp. 207–220. DOI: 10.1016/S0049-237X(08)70589-5.

Abramsky, Samson and Achim Jung (1994). 'Domain theory'. In: *Handbook of Logic in Computer Science*. Ed. by S. Abramsky, Dov M. Gabbay and T. S. E. Maibaum. 1st ed. Vol. 3. Oxford University Press. Chap. 1, pp. 1–168.

Aluffi, Paolo (2009). *Algebra. Chapter 0*. 1st ed. American Mathematical Society. xxi + 713 pp. ISBN: 978-0-8218-4781-7.

Awodey, Steve (2010). *Category Theory*. 2nd ed. Oxford University Press. xv + 311 pp. ISBN: 978-0-19-958736-0.

Baader, Franz and Tobias Nipkow (1998). *Term Rewriting and All That*. 1st ed. Cambridge University Press. xii + 301 pp. ISBN: 0-521-45520-0.

Barendregt, H. P. (1984). *The Lambda Calculus. Its Syntax and Semantics*. revised. Elsevier. xv + 621 pp. ISBN: 0-444-86748-1.

Birkedal, Lars, Aleš Bizjak and Jan Schwinghammer (Oct. 2013). 'Step-Indexed Relational Reasoning for Countable Nondeterminism'. In: *Logical Methods in Computer Science* Volume 9, Issue 4. DOI: 10.2168/LMCS-9(4:4)2013.

Crole, Roy L. (1993). *Categories for Types*. 1st ed. Cambridge University Press. xvii + 335 pp. ISBN: 0-521-45092-6.

Curry, Haskell B., Robert Feys and William Craig (1958). *Combinatory Logic*. Revised ed. Vol. 1. North-Holland Publishing Company. xvi + 417 pp.

Davey, B. A. and H. A. Priestley (2002). *Introduction to Lattices and Order*. 2nd ed. Cambridge University Press. xii + 298 pp. ISBN: 978-0-521-78451-1.

Folland, Gerald B. (2007). *Real Analysis: Modern Techniques and Their Applications*. 2nd ed. Wiley. xiv + 386 pp. ISBN: 0-471-31716-0.

Gierz, G. et al. (2003). *Continuous Lattices and Domains*. 1st ed. Cambridge University Press. xxxvi + 591 pp. ISBN: 978-0-511-06356-5. DOI: 10.1017/CBO9780511542725.

Goldrei, Derek (1996). *Classic Set Theory*. 1st ed. Chapman & Hall. viii + 502 pp. ISBN: 0-412-60610-0.

Goubault-Larrecq, Jean (2013). *Non-Hausdorff Topology and Domain Theory. Selected Topics in Point-Set Topology*. 1st ed. Cambridge University Press. vi + 491 pp. ISBN: 978-1-107-03413-6.

Gunter, Carl A. (1992). *Semantics of Programming Languages. Structures and Techniques*. 1st ed. The MIT Press. xviii + 419 pp. ISBN: 0-262-07143-6.

Hansen, Danny Nygård (2023). 'Minimality in Zermelo's fixed-point theorem'. URL: https://math.stackexchange.com/q/4779743 (visited on 03/10/2023).

Harper, Robert (2016). *Practical Foundations for Programming Languages*. 2nd ed. Cambridge University Press. xviii + 494 pp. ISBN: 978-1-107-15030-0.

Hindley, J. Roger (1997). *Basic Simple Type Theory*. 1st ed. Cambridge University Press. xi + 186 pp. ISBN: 0-521-46518-4.

Johnstone, Peter T. (1982). *Stone Spaces*. 1st ed. Cambridge University Press. xxi + 370 pp. ISBN: 0-521-23893-5.

Kechris, Alexander S. (1995). *Classical Descriptive Set Theory*. 1st ed. Springer. xviii + 402 pp. ISBN: 0-387-94374-9.

Leinster, Tom (2014). *Basic Category Theory*. 1st ed. Cambridge University Press. viii + 183 pp. ISBN: 978-1-107-04424-1. DOI: 10.48550/arXiv.1612.09375.

Miller, Arnold W. (2008). 'Long Borel Hierarchies'. In: *Mathematical Logic Quarterly* 54.3, pp. 307–322. DOI: 10.1002/malq.200710044.

Moschovakis, Yiannis (2006). *Notes on Set Theory*. 2nd ed. Springer. xii + 276 pp. ISBN: 0-387-28722-1.

nLab authors (Mar. 2024). *biased definition*. https://ncatlab.org/nlab/show/biased+definition. Revision 9.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st ed. The MIT Press. xxi + 623 pp. ISBN: 0-262-16209-1.

Pitts, Andrew M. (2005). 'Typed Operational Reasoning'. In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. 1st ed. The MIT Press. Chap. 7, pp. 245–289.

— (2013). *Nominal Sets. Names and Symmetry in Computer Science*. 1st ed. Cambridge University Press. xiii + 276 pp. ISBN: 978-1-107-01778-8.

Priest, Graham (2008). *An Introduction to Non-Classical Logic. From If to Is*. 2nd ed. Cambridge University Press. xxxii + 613 pp. ISBN: 978-0-511-39361-7.

Schröder, Bernd (2016). *Ordered Sets. An Introduction with Connections from Combinatorics to Topology*. 2nd ed. Birkhäuser. xvi + 420 pp. ISBN: 978-3-319-29786-6. DOI: 10.1007/978-3-319-29788-0.

Skorstengaard, Lau (2019). *An Introduction to Logical Relations*. https://cs.au.dk/~birke/papers/AnIntroductionToLogicalRelations.pdf.

Terese (2003). *Term Rewriting Systems*. 1st ed. Cambridge University Press. xxii + 884 pp. ISBN: 0-521-39115-6.

Index of Notation

Index