COMP31311 Giving Meaning to Programs

Joe Razavi
Joseph.Razavi@manchester.ac.uk

Andrea Schalk
A.Schalk@manchester.ac.uk

25th September 2022

Contents

Overview

Introduction

There's an old joke that history is '*just one damn thing after another!*' But we can hardly laugh: most of us 'understand' a program by imagining what happens, step-by-step, when it runs. One damn thing after another. This is error-prone and unenlightening. It also seems contrary to the promises made by programming languages. They have *abstractions* like 'functions' and 'objects', which promise us that we can ignore the internal behaviour, and treat parts of the program as having a high level *meaning*. These meanings are often left informal, and the result is that in complicated situations, we're forced to resort to looking inside the abstraction again and running the program in our heads. How humiliating! What we should do instead is to build a mathematical theory of program meanings, one in which we can *prove* that the abstract meaning really captures everything relevant about how the program behaves.

This can be done for real programming languages, but the resulting theory has lots of complicated details. Instead we will work with simplified programming languages which capture the essential nature of the features we study. We choose functions as a high-level feature to examine. This is because for functions the intended mathematical interpretation is well understood, and the theory which links the programming feature to the mathematics is stable. Modelling features like 'objects' is a subject of ongoing research, but the techniques we look at in this course form the basis of much of this work.

The structure of the course is as follows. First, to study functions as a programming language feature, we need to give a step-by-step account of how a functional language computes. We do this in Chapter 1, where we study the untyped λ-calculus. We find that this language contains some programs with unexpected behaviour, which we rule out in Chapter 2, by introducing a type system, resulting in the simply typed λ-calculus. In a sense, these two systems represent the 'reality' of programming with functions, because the step-by-step semantics tells us that the computation can actually be done. Therefore, we spend a significant amount of effort studying the properties of these systems.

With this foundation in place, we introduce the crucial notion of contextual equivalence, which lets us describe whether a proposed abstract 'meaning' for a program is correct. We then compare the programs of the simply typed λ-calculus to mathematical functions, and show that they correspond well: programs in the simply typed λ-calculus can safely be thought of a functions. But things are not as nice as they might seem: we also find that not many functions can actually be written in this language!

To rectify this problem, in Chapter 3, we add some features to the calculus to obtain a realistic (if still simple) programming language: PCF. This language has

recursion, and, hence, non-termination. This creates interesting problems for a mathematical semantics, and we look at the subtle theory built to solve them.

The material we cover on this unit is well-established, but our presentation is original. We give references of titles that have influenced in particular.

Bibliography

[Bar92] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992. URL: https://home.ttic.edu/~dreyer/course/papers/barendregt.pdf.

[BDS13] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. URL: https://ftp.science.ru.nl/CSI/CompMath.Found/I.pdf.

[Buc21] Antonion Bucciarelli. Preuves et pogrammes: outils classiques. See https://www.irif.fr/~buccia/COURS/PPOC/SEANCES/ for additional notes, 2021. URL: https://www.irif.fr/~buccia/COURS/PPOC/SEANCES/s04.pdf.

[DP01] Kosta Dosen and Zoran Petric. The typed bohm theorem. *Electron. Notes Theor. Comput. Sci.*, 50(2):117–129, 2001. URL: http://www.mi.sanu.ac.rs/~kosta/KRIT1.pd.

[Jun14] A. Jung. Teaching denotational semantics. *ACM SIGLOG News*, 1(2):25–37, 2014. With an introduction by Mike Mislove. URL: https://www.cs.bham.ac.uk/~axj/pub/papers/Jung-2014-Teaching-denotational-semantics.pdf.

[KMY14] Yuichi Komori, Naosuke Matsuda, and Fumika Yamakawa. A simplified proof of the Church-Rosser Theorem. *Studia Logica volume*, 102:175–183, 2014. URL: https://link.springer.com/article/10.1007/s11225-013-9470-y.

[Loa98] Ralph Loader. Notes on simply typed lambda calculus. Technical report, The University of Edinburgh, 1998. URL: https://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/.

[Pfe18] Frank Pfenning. Lecture notes on subject reduction. See https://www.cs.cmu.edu/~fp/courses/15814-f19/lectures/ for additional notes, 2018. URL: https://www.cs.cmu.edu/~fp/courses/15814-f19/lectures/05-subred.pdf.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. URL: http://ropas.snu.ac.kr/~{}kwang/520/pierce_book.pdf.

[Sel08] Peter Selinger. Lecture notes on the lambda calculus. Chapters 1–3, 2008. URL: https://arxiv.org/pdf/0804.3434.

[Str06] Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming*. World Scientific Publishing Co, USA, 2006. URL: https://www.semanticscholar.org/paper/Domain-theoretic-foundations-of-functional-Streicher/730a2c4fe57e22dfcf62179795d877708c176280.

[Tak89] Masako Takahashi. Parallel reductions in lambda-calculus. *Journal of Symbolic Computation*, 7:113–123, 1989. URL: https://www.sciencedirect.com/science/article/pii/S0747717189800458.

These notes

When trying to develop the idea that it is possible (and desirable) to give a formal meaning to programs one has to make some decisions regarding which setting to use. The choices we made such as to reason about programs that cover computations on natural numbers and to look at three increasingly sophisticated languages are the standard choices made by most authors.

As is not unusual with academic authors when reviewing the available literature we decided we did not like some of the design choices made and we are therefore providing an original account. This account is more mathematical than we had envisioned, but our intention is not to get bogged down by studying a great number of proofs in detail. Instead we think we owe students the evidence that what we are talking about can be made mathematically precise.[1]

When it comes to the **assessments** for this unit, we expect students to be able to carry out the following:

- Solve problems concerned with specific terms, which might require manipulating these terms, applying operations to them, or reasoning about them. In many cases there will be exercises in the notes with a similar task, but we might also ask students to apply the key results in new ways. Most of the recommended exercises in the notes fall into this category.
- For the coursework only, give a formal proof of some results. Those results have been chosen not to require overly lengthy, or complicated, proofs. Students who want to practice this before the coursework is due are directed to do so with exercises in yellow boxes, or with proofs that have yellow boxes in the text.

We have tried to make it easier for the reader to study these notes at varying levels by adopting the following conventions:

- The key results (and in some cases corollaries) whose statements are particularly useful when it comes to answering exam (or coursework) questions have a **red** box around them.
- Other results, many of which are required along the way to establishing results in the previous category, appear in **blue** boxes. That way the reader

[1] We have made one strategic decision to accept a fact about the nature of α-equivalence without giving a proof.

may either concentrate on the main insights, or follow the development of how those results may be achieved via a number of steps.

- The majority of proofs have been pushed into appendices. See below into those boxed in grey and those boxed in yellow. The document has working hyperlinks allowing the reader to jump straight to the proof. While we make proofs to recommended and 'yellow' exercises available with some delay we will provide a complete document at the end that allows this functionality everywhere.

We further apply colour-coding to exercises and proofs:

- Recommended exercises appear in **orange**, and it is vital that students solve all those exercises, ideally in line with the schedule suggested on Blackboard.
- **Yellow** boxes appear around some exercises and some proofs in the main text. These are concerned with proofs that are not too technical. In many cases the exercises come with comments regarding what their purpose is, these are given in *italics*. Solving (some of) these exercises is useful to prepare for the two coursework exercises, and will also help the interested reader understand how one proves properties for our systems.
- There are a fair number of technical proofs in the notes which we supply here to give a completely rigorous account. They come with **grey** boxes and we expect that few students will want to look at these in detail.

In terms of **practical work** students should aim to do the following:

- Get some practice with manipulating terms in the weekly workshops;
- solve the exercises in **orange** boxes interspersed with the text as they progress through the material—we very strongly recommend following the schedule suggested on the Blackboard page;
- try at least some of the proofs and exercises in **yellow** boxes (these will help with the part of the coursework concerned with giving formal proofs);
- solve exercises from Chapter 5 for revision.

We do intend to develop these notes further by taking student feedback into account and we hope that students will be happy to give that, either by emailing us or by posting (anonymously if they prefer) on the Blackboard discussion board.

We have already made a number of changes based on previous student feedback:

- We now make technical proofs available with the main material while delaying publication of solutions to recommended exercises and 'yellow' items.
- Colour coding results so that students have more guidance in which of these they should know (and be able to apply) is something we have devised based on students telling us they found the number of propositions overwhelming.
- We are supplying further exercises for revision purposes since students told us they found it difficult to find sufficient material for practising for the exam.

- We have changed the nature of the synchronous sessions.

Acknowledgements. We would like to thank the many authors whose accounts have helped us create this one. We are grateful to Francisco Lobo for providing LaTeX infrastructure for us, including the package that allows us to provide solutions separately. We are also grateful to students whose feedback helps us improve the notes, in particular Vlad Sirbu, Laurence Warne and Tomas Savickas.

Chapter 1

The Untyped λ-Calculus

1.1 Computation out of the Spirit of Logic

Let's start with some 'history'—and a little bit of myth. In the 1920's, before our science existed, the logician Alonzo Church was trying to provide a logical foundation for mathematics.

Most foundations of mathematics assumed a notion of *substituting a term for a free variable*. For example, to prove $\exists x.P(x)$ we are supposed to provide a term t and prove $P(t)$—where $P(x)$ is supposed to be a formula with a free variable x, and $P(t)$ means P but with t substituted wherever x occurs. Similarly, in school one writes $f(x)$ to indicate a 'function' in the sense of an expression with a free variable x. Applying this function to get, say $f(t)$ again means substituting t in f wherever x occurs.

Church wanted to specify exactly how substitution worked, step by step, and to get rid of informal talk of 'expressions with free variables', at least for complete expressions: though parts of an expression may have free variables, the finished whole must explicitly tell us where terms can be substituted in. This would then allow the process of substitution to be described precisely, without appealing to intuition.

A computer scientist's heart sings to hear of things being made explicit: indeed, it is out of this trend in formal logic towards precise specification that our own subject came into being. After all, if the core activity involved in calculation is expanding or simplifying expressions by substituting terms for free variables, then a precise calculus of free variables and substitutions is a language for specifying calculations—a sort of programming language. In these notes we use Church's calculus as a way of capturing important behaviour of programs and as a basis for further investigation—we add types to improve our system's behaviour in Chapter 2, which has unintended consequences that are addressed in Chapter 3.

Other logicians like Russel and Frege had already been worrying about expressions with free variables. To indicate that the x in, for instance, $e^x + 5$ is supposed to be something we can substitute a term for, they would write $\hat{x}.(e^x + 5)$ so that x is no longer a free variable. Instead it is 'bound' by $\hat{x}.$, just like it is bound by the summation operation in $\sum_{x=1}^{10}(e^x + 5)$. It is now clear, for instance, that x is not a special constant (unlike e) but an arbitrary name, and the expressions $\hat{y}.(e^y + 5)$ and $\sum_{y=1}^{10}(e^y + 5)$ mean the same thing as the versions with x in.

Church's typewriter couldn't do $\hat{x}$, but he still wanted something pointy to remind him of that notation. Since he had Greek letters available, he decided to write $\hat{x}.T$ as $\lambda x.T$ which is now the familiar notation. He decided to explicate the

operation of substitution by defining a formal calculus of these λ-terms. To do so, he would define the set of λ-terms by recursion, and then substitution could be defined recursively, too.

1.2 A Calculus of Substitutions

To define a calculus to study the operation of substitution, we need two things: a way of writing a term with a variable in it, and a way of writing an application of one term to another, since that is how we indicate that substitution is supposed to happen in a mathematical expression. discussed above, and for the latter we put the two terms next to each other following the usual mathematical notation.

1.2.1 Variables and fresh variables

We first have to pause and consider what variables might be, and in particular what we expect to do with them. We assume that we have an infinite set Vars of variables, so that we know we can always find a variable that hasn't been mentioned before. To make this precise we use the fact that for an infinite set we may find an injective function from Vars to itself which is not surjective. We can use such a function to build a more useful function which gives us a variable we haven't mentioned yet. We pause for a moment to think about the source and target for such a function: We give it some variables that we have used, and it should give us back a variable that hasn't been mentioned, so it should be typed as

$$\mathrm{freshv} : \mathcal{P}_{\mathsf{fin}}(\mathsf{Vars}) \longrightarrow \mathsf{Vars},$$

where $\mathcal{P}_{\mathsf{fin}}$ is the set of *finite* subsets of a given set. It should further have the property that

$$\mathrm{freshv}\, S \notin S,$$

which ensures that we really have been given a *fresh* variable that hasn't been used before. We need this infrastructure below.

What exactly variables *are* is considered a low-level detail. For example, one could implement the calculus presented here by using a set of variables of the form x_n for all $n \in \mathbb{N}$, and then we could define, for instance

$$\mathrm{freshv}\, S = x_{(max\{n | x_n \in S\}+1)}.$$

But in these notes we don't want to be bogged down by worrying exactly how providing this infrastructure is achieved, and we freely use letters, primes, subscripts, etc to denote variables.

1.2.2 Terms

With the issue of variables dealt with, we can move on to the main definition.

Tip

In these notes there are a large number of *recursive* definitions. These are introduced in Chapter 5 of the COMP111 notes. Such definitions have one or more **base cases**, and one or more **step cases**. In our formal definitions we abbreviate 'base case' with 'bc', and 'step case' with 'sc'. We attach a name to these cases, which may then be used as a justification in formal arguments.

Typically our recursive definitions fall into two cases:

- We define a set, or a structure, such as the set of λ-terms. The base cases tell us how we may create terms from nothing, and the step cases how we may build new terms from previously created ones.[1]
- We define a function whose source is a recursively defined set or structure. In order to ensure that the function is defined for all possible inputs we ensure that we make case distinctions that match the definition of the structure that provides its inputs.

Definition 1: λ-terms

The set of **λ-terms** $\Lambda\mathsf{Trm}$ is defined by recursion as follows.

bcVar For $x \in \mathsf{Vars}$ we have $x \in \Lambda\mathsf{Trm}$.

scAbs If $x \in \mathsf{Vars}$ and $t \in \Lambda\mathsf{Trm}$ then $(\lambda x.\, t) \in \Lambda\mathsf{Trm}$.

scApp If $t, u \in \Lambda\mathsf{Trm}$ then $(tu) \in \Lambda\mathsf{Trm}$.

The two ways of creating new λ-terms from existing ones have names:

- Forming a **λ-abstraction** is the process of taking an existing term and prefixing that with a λ and a variable.
- Taking two λ-terms and putting them next to each other is known as **application**.

The justification for these names becomes clearer as we discuss the intended meaning of terms below.

Definition 2: subterm

We define the **subterm relation** between λ-terms as follows:

bcVar The λ-term x, where x is a variable, is a subterm of itself.

scAbs If $x \in \mathsf{Vars}$ and $t \in \Lambda\mathsf{Trm}$ then the subterms of $\lambda x.\, t$ are itself, t and all its subterms.

scApp If $t, t' \in \Lambda\mathsf{Trm}$ then the subterms of tt' are itself, t, t' and all their subterms.

We say that t' is a **proper subterm** of t provided that it is a subterm and t and t' are not equal.

According to our definition, as we build up a λ-term we should always bracket the subterms of a λ-term. It is better to have a convention which allows us to drop them for readability. We adopt the following conventions:

- In a λ-abstraction everything to its right (inside any brackets it is in) is considered part of the abstraction.
- When there is a sequence of applications, the leftmost one happens first.

Note that while this convention *allows* us to drop certain brackets, we may also choose to keep them if they aid readability.

[1] You may want to compare this with the definition of a regular expression, which is an example of this kind of thing.

Example 1.1. An example of a λ-term is $\lambda f.\, \lambda x.\, f(f(f\,x))$.

Example 1.2. The λ-term

$$(\lambda x.\, \lambda y.\, x)(\lambda x.\, x)\lambda x.\, x$$

becomes

$$[(\lambda x.\, [\lambda y.\, x])(\lambda x.\, x)](\lambda x.\, x)$$

where the 'implicit' brackets have been indicated with square brackets.

The intuition is that a λ-abstraction represents a term with a variable which can be substituted for.[2] A useful analogy is that λ-terms are a bit like functions.

Example 1.3. Take for example the λ-term $\lambda x.\, x$. We see below that when we apply this term to what we may think of as an argument we substitute something for x, and we get back whatever we put in.

This is a little like the behaviour of an identity function

$$\begin{array}{ccc} S & \longrightarrow & S \\ x & \longmapsto & x \end{array}$$

on a set S: If we apply this function to any element of S, it returns that element.

We sometimes may refer to a λ-term with a name that suggests this 'function-like' behaviour, and might call $\lambda x.\, x$ 'the identity term'. However, this intuition is not immediately meaningful for all such terms: look below to see some of the weird ones, such as Example 1.28!

On the other hand functions give a good intuition when it comes to understanding substitution.

An application is intended to be such a term supplied with the argument we want to substitute into that term. Note, however, that the left hand side of an application can be *any term*, it doesn't have to be an abstraction. The intuition behind this is that sometimes we might have to compute which function we want to apply before we are able to apply it. Concrete examples of this situation occur in Section 2.8, in particular in Section 2.8.2 when we mathematically define the *meaning* of λ-terms.

At the level of intuition we can already see one crucial principle: the variable 'bound' by a λ-abstraction is arbitrary, and so the meaning and behaviour of a λ-term should be invariant under renaming of bound variables.

Example 1.4. Again the analogy with functions works well. The following two descriptions define the same function:

$$\begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \longmapsto & x^2 + \sin x \end{array} \qquad \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ y & \longmapsto & y^2 + \sin y. \end{array}$$

[2] We do define this substitution operation below, but we have to build up to that.

This is because we calculate the result of applying a function to a particular input by substituting that input for the variable in the expression defining the function. To find out what the function above does with input π, we form

$$\pi^2 + \sin \pi,$$

and because we now have numbers we can calculate a final result. Because we carry out calculations via substitution it does not matter whether the variable we use is called x or y.

Example 1.5. Another area that provides examples for this idea is that of formulae in propositional logic. Here the 'for all' ($\forall$) and 'there exists' ($\exists$) quantifiers are tied to variables in a similar way.

We expect $\forall x.\, x$ and $\forall y.\, y$ to have the same meaning in every interpretation.

Exercise 1. *This exercise is suitable for students who want to reacquaint themselves with partial orders (see Definition 52), which are used significantly in the semantics of our third system.*

Show that the subterm relation is a partial order for λ-terms.

See page 265 for a solution.

1.2.3 Free and bound variables

We formalize this idea shortly: first we must finally define what we mean be free and bound variables—the concept which was so prominent in Church's motivations.

Definition 3: bound variables

For $t \in \Lambda\mathsf{Trm}$ we define the set of **bound variables of** t, $\mathrm{bv}(t)$, by recursion as follows:

bcVarbv $\mathrm{bv}\, x = \emptyset$ if $x \in \mathsf{Vars}$,

scAbsbv $\mathrm{bv}(\lambda x.\, t) = \mathrm{bv}\, t \cup \{x\}$,

scAppbv $\mathrm{bv}(tt') = \mathrm{bv}\, t \cup \mathrm{bv}\, t'$.

Example 1.6. The idea of bound variables makes sense for functions as well. Assume we have a definition of a function in two variables, via $f(x, y) = \ldots$. The expression defining the function may contain additional variables—often called *parameters* or viewed as *constants* but if we just look at the expression it is not clear that there is any difference between these.

Assume our function is given by the expression

$$f(x, y) = x^2 + z \sin y + xz.$$

If we only look at $x^2 + z \sin y + xz$ we have no way of distinguishing between x, y and z. But we know from the left hand side of the definition that there

are two arguments, x and y, to these function, and that in order to calculate the value of our function at some input we must

- substitute the first input for x
- substitute the second input for y
- either leave z in the resulting expression, or find out from the context whether we know which number z is meant to stand for.

The variables x and y are 'bound' in this expression because we need to be able to substitute actual inputs for them.

We could define exactly the same function as

$$f(v, w) = v^2 + z \sin w + vz,$$

and it is this idea that we can rename bound variables without affecting what happens when we apply the function to inputs that is analogous with the idea that the behaviour of a λ-term should be invariant under renaming these.

Example 1.7. The bound variables in

$$\forall y.\, \exists x.\, f(x, z) = y$$

are y and x. In an interpretation to investigate the meaning of this formula we expect to use elements of a domain of interpretation to substitute for x and y; and we have to check whether a suitable value of x exists for every value of y.

Example 1.8. The bound variables in $\lambda f.\, \lambda x.\, f(f\,x)$ can be calculated as follows:

$$\begin{aligned}
\mathrm{bv}(\lambda f.\, \lambda x.\, f(f\,x)) &= \mathrm{bv}(\lambda x.\, f(f\,x)) \cup \{f\} && \text{scAbsbv}\\
&= \mathrm{bv}(f(f\,x)) \cup \{x\} \cup \{f\} && \text{scAbsbv}\\
&= \mathrm{bv}\, f \cup \mathrm{bv}(f\,x) \cup \{x, f\} && \text{scAppbv}\\
&= \emptyset \cup \mathrm{bv}\, f \cup \mathrm{bv}\, x \cup \{x, f\} && \text{bcVarbv and scAppbv}\\
&= \emptyset \cup \emptyset \cup \emptyset \cup \{x, f\} && \text{bcVarbv}\\
&= \{x, f\} && \emptyset \text{ unit for } \cup
\end{aligned}$$

Definition 4: free variables

For $t \in \Lambda\mathsf{Trm}$ the set of free variables **of** t, $\mathrm{fv}(t)$ is defined by recursion as follows:

bcVarfv $\mathrm{fv}\, x = \{x\}$ if $x \in \mathsf{Vars}$,

scAbsfv $\mathrm{fv}(\lambda x.\, t) = \mathrm{fv}\, t \setminus \{x\}$,

scAppfv $\mathrm{fv}(tt') = \mathrm{fv}\, t \cup \mathrm{fv}\, t'$.

Example 1.9. In Example 1.6 we have already seen an expression defining a function which contains a free variable, namely z. In mathematics a letter appearing in an expression defining a function is

- either one of the inputs, that is a bound variable, or
- a free variable that we typically think of as a parameter or a constant.

Definition 5: variables

We define the **set of variables of a term** t to be $\mathrm{vars}(t) = \mathrm{fv}(t) \cup \mathrm{bv}(t)$.

Exercise 2. *This exercise is good practice for producing a proof by induction and using the formal definitions of free and bound variables. An example of a proof by induction is given in Proposition 1.1.*

We might instead choose to define the set of all variables of a λ-term as follows:

bcVarv′ $\mathrm{vars}'\, x = \{x\}$

bcAbsv′ $\mathrm{vars}'(\lambda x.\, t) = \{x\} \cup \mathrm{vars}'\, t$

bcAppv′ $\mathrm{vars}'(tt') = \mathrm{vars}'\, t \cup \mathrm{vars}'\, t'$.

Show that the two functions agree, that is, for all λ-terms t we have that $\mathrm{vars}\, t = \mathrm{vars}'\, t$.

A solution may be found on page 265.

There is a sense in which the definitions of free and bound variables are dual to each other: a specific occurrence of a variable is free if and only if it is not bound. However, the sets are not always disjoint, as the following exercise establishes.

We give a simple example of a proof by induction for λ-terms.

Proposition 1.1

If t' is a subterm of t then $\mathrm{bv}\, t' \subseteq \mathrm{bv}\, t$.

Proof. We carry out the proof by induction over the structure of the term t'. The induction hypothesis is that the property holds for proper subterms of the current one, that is if the current term is t and t' is a proper subterm of t, then for all proper subterms t'' of t' we have that $\mathrm{bv}\, t'' \subseteq \mathrm{bv}\, t'$..

bcVar If t is a variable then the only subterms it has are itself, and we have $\mathrm{bv}\, t \subseteq \mathrm{bv}\, t$.

scAbs If t is a λ-abstraction, say $t = \lambda x.\, u$ then unless $t = t'$, in which case the claim is trivial, t' must be a subterm of u and by the induction hypothesis we know that

$$\mathrm{bv}\, t' \subseteq \mathrm{bv}\, u \subseteq \mathrm{bv}\, u \cup \{x\} = \mathrm{bv}\, t.$$

scApp If t is an application, say $t = rs$ then unless $t = t'$, in which case the claim is trivial, t' must be a subterm of r, or a subterm of s, and so by

the induction hypothesis we have $\operatorname{bv} t' \subseteq \operatorname{bv} r$ or $\operatorname{bv} t' \subseteq \operatorname{bv} s$, which implies

$$\operatorname{bv} t' \subseteq \operatorname{bv} r \cup \operatorname{bv} s = \operatorname{bv} t.$$

RExercise 3. *This is an exercise inviting students to think about the nature of free and bound variables, and to practice proofs by induction.*

Carry out the following tasks for variables appearing in some term.

(a) Explain why the analogue of the statement from Proposition 1.1 for free variables does not hold. If t is a subterm of t', can you say anything about the relationship between $\operatorname{fv} t$ and $\operatorname{fv} t'$?

(b) Find a term $t \in \Lambda\mathsf{Trm}$ such that $\operatorname{bv}(t) \cap \operatorname{fv}(t) \neq \emptyset$. Conclude that there may be overlap between the free and bound variables of λ-terms.

(c) Show that if t is a subterm of t' then $\operatorname{vars} t \subseteq \operatorname{vars} t'$. *Hint: You may want to use the alternative definitions of* vars *from the previous exercise.*

Solutions appear on page 266.

1.2.4 Renaming and α-equivalence

We aim to define the crucial equivalence relation, traditionally known as α-*equivalence*, see Definition 7, which any meaningful operation defined on λ-terms must respect in the sense that it should give the same output for inputs that are α-equivalent, just as any theory of functions cannot make any distinctions based on which variable names are used in the definition of the function, see Example 1.6.

We say that two terms are α-equivalent if we can get one from the other just by renaming bound variables. Defining this formally turns out to be a little trickier than expected (see Definition 7), and indeed, one might argue that it is surprising that most of the time we can get away in mathematics or logic without providing such a definition! We first define the following operation of renaming variables, which simply replaces a variable name with another everywhere within a term.

Definition 6: renaming

Let $x, z \in \mathsf{Vars}$ we define the **renaming operation** $\operatorname{ren}_x^z : \Lambda\mathsf{Trm} \longrightarrow \Lambda\mathsf{Trm}$ by recursion as follows:

bcVarren In the base case we have

$$\operatorname{ren}_x^z y = \begin{cases} z & y = x \\ y & \text{else,} \end{cases}$$

scAbsren $\operatorname{ren}_x^z(\lambda y.\, t) = \lambda(\operatorname{ren}_x^z y).\, (\operatorname{ren}_x^z t)$,

scAppren $\operatorname{ren}_x^z(tt') = (\operatorname{ren}_x^z t)(\operatorname{ren}_x^z t')$.

Note that this renaming function does *not* map λ-terms which are the same up to renaming bound variables to the same output.

Example 1.10. We give an example to illustrate how renaming works.

$$\begin{aligned}
\mathrm{ren}^z_x(\lambda x.\, y\lambda x.\, y) &= \lambda(\mathrm{ren}^z_x\, x).\ \mathrm{ren}^z_x(y\lambda x.\, y) && \text{scAbsren}\\
&= \lambda z.\ \mathrm{ren}^z_x\, y\, \mathrm{ren}^z_x(\lambda x.\, y) && \text{bcVarren and scAppren}\\
&= \lambda z.\, y\lambda(\mathrm{ren}^z_x\, x).\ \mathrm{ren}^z_x\, y && \text{bcVarren and scAbsren}\\
&= \lambda z.\, y\lambda z.\, y && \text{bcVarren}
\end{aligned}$$

RExercise 4. *This exercise aims to help students understand how renaming works in contrast with capture-avoiding substitution as defined below.*

Give two terms that are the same apart from renaming one bound variable, such that applying the ren function gives two terms that are no longer connected in this way.

A solution appears on page 267.

We begin by establishing simple properties of the renaming operation. Subsequently we may use some of these without making reference to this result.

Proposition 1.2

The following statements hold for all λ-terms t and $x, y, z \in \mathsf{Vars}$ for the ren function.

(a) $\mathrm{ren}^x_x\, t = t$.

(b) If $y \notin \mathrm{vars}\, t$ then $\mathrm{ren}^z_y\, \mathrm{ren}^y_x\, t = \mathrm{ren}^z_x\, t$.

(c) If $y \notin \mathrm{vars}\, t$ then $\mathrm{ren}^x_y\, \mathrm{ren}^y_x\, t = t$.

(d) If x, x', y and y' are variables such that

- $x \neq y$
- $x \neq y'$ and
- $y \neq x'$

then $\mathrm{ren}^{x'}_x\, \mathrm{ren}^{y'}_y\, t = \mathrm{ren}^{y'}_y\, \mathrm{ren}^{x'}_x\, t$.

(e) If $y \notin \mathrm{vars}\, t$ then $\mathrm{ren}^x_y\, t = t$.

Proof. See page 206.

Our next object is to look at what we may say about (free) variables in a term after renaming has taken place.

Proposition 1.3

The following statements hold for all λ-terms t and variables x and y, where $y \notin \mathrm{vars}\, t$.

(a) If $x \neq y$ then $\mathrm{vars}(\mathrm{ren}^y_x\, t) \subseteq (\mathrm{vars}\, t \cup \{y\}) \setminus \{x\}$. Note in particular that $x \notin \mathrm{vars}(\mathrm{ren}^y_x\, t)$.

(b) We have $x \notin \mathrm{fv}(\lambda y.\ \mathrm{ren}_x^y\, t)$.

(c) We have

$$\mathrm{fv}(\mathrm{ren}_x^y\, t) = \begin{cases} \mathrm{fv}\, t & x \notin \mathrm{fv}\, t \\ (\mathrm{fv}\, t \setminus \{x\}) \cup \{y\} & \text{else.} \end{cases}$$

Proof. See page 208.

We are finally in a position to define α-equivalence. We repeatedly have to talk about 'fresh variables' for some given terms. We say that the variable w is **fresh for the terms** $t_1, t_2, \ldots, t_n$ if and only if

$$w \notin \mathrm{vars}\, t_1 \cup \mathrm{vars}\, t_2 \cdots \mathrm{vars}\, t_n.$$

Intuitively, two terms are equivalent if they differ only by the names of bound variables. Another way to put this is that we can make the terms equal by renaming their bound variables to any suitably fresh variables. Of course, formalizing this involves making precise what we mean by 'suitably fresh' – but there is something more subtle to attend to. The word 'any' is ambiguous: sometimes it should be formalize by 'there exists', and sometimes by 'for all'. In this case we really *don't care* which variables get used, so we might hope that it doesn't matter whether we define α-equivalence by demanding that certain fresh variables exist, or demanding a condition involving all fresh variables. This is an unusal situation, but for α-equivalence it turns out to be correct.

Tip

So far all the statements from the exercises can be shown using a straight-forward structural induction, where the step cases assume that we have the property for immediate subterms of the current term. As our theory develops we have to use stronger induction hypotheses. The reader may want to look out for these. The proof of the following theorem gives one straight-forward induction and one that is quite complicated, so looking at other proofs may be a better idea.

Proposition 1.4

Consider the two relations on λ-terms defined as follows:

bcVar$\sim_\exists$ Both terms are variables, and they are identical.

bcVar$\sim_\forall$ Both terms are variables, and they are identical.

scAbs$\sim_\exists$ $\lambda x.\, t \sim_\exists \lambda x'.\, t'$ if and only if **there exists** a variable w which is fresh for $\lambda x.\, t$ and $\lambda x'.\, t'$ and we have $\mathrm{ren}_x^w\, t \sim_\exists \mathrm{ren}_{x'}^w\, t'$.

scAbs$\sim_\forall$ $\lambda x.\, t \sim_\forall \lambda x'.\, t'$ if and only if **for all** variables w which are fresh for $\lambda x.\, t$ and $\lambda x'.\, t'$ we have $\mathrm{ren}_x^w\, t \sim_\forall \mathrm{ren}_{x'}^w\, t'$.

scApp$\sim_\exists$ $tu \sim_\exists t'u'$ if and only if $t \sim_\exists t'$ and $u \sim_\exists u'$.

scApp$\sim_\forall$ $tu \sim_\forall t'u'$ if and only if $t \sim_\forall t'$ and $u \sim_\forall u'$.

Then for all terms t and t', we have $t \sim_\exists t'$ if and only if $t \sim_\forall t'$,

Proof. See page 211.

The previous result gives us that the two relations defined there agree. This is the relation we want to use to talk about terms that only differ in the names used for bound variables.[3]

Definition 7: α-equivalence

Two λ-terms are **α-equivalent** if and only if they are related by the unique relation described by Proposition 1.4, for which we use the symbol $\sim_\alpha$.

The following is immediate from the definition of α-equivalence in light of Proposition 1.4.

Corollary 1.5

Given terms of the form $\lambda x.\, t$ and $\lambda x'.\, t'$, we have the following:

- If there exists a variable w which is fresh for $\lambda x.\, t$ and $\lambda x'.\, t'$ such that $\mathrm{ren}^w_x\, t \sim_\alpha \mathrm{ren}^w_{x'}\, t'$, then $\lambda x.\, t \sim_\alpha \lambda x'.\, t'$.
- If $\lambda x.\, t \sim_\alpha \lambda x'.\, t'$ then for all variables w which are fresh for $\lambda x.\, t$ and $\lambda x'.\, t'$ we have $\mathrm{ren}^w_x\, t \sim_\alpha \mathrm{ren}^w_{x'}\, t'$.

Above we have used the notion of α-equivalence informally, and readers may want to go back over the examples given there.

RExercise 5. *The aim of this exercise is for students to investigate the formal definition of α-equivalence.*

Show that the following terms are α-equivalent:

$$\lambda x.\, \lambda y.\, xy \qquad \text{and} \qquad \lambda y.\, \lambda x.\, yx.$$

A solution may be found on page 267.

One might wonder whether in the definition of α-equivalence in the case of abstraction we choose a particular variable to check whether the second argument of the abstractions become α-equivalent after renaming. The following exercise shows that the choice does not matter.

Tip

The way α-equivalence is defined makes it clear that two terms can only be α-equivalent if their parse trees have the same shape, and indeed, the only way in which they may differ is in the use of names of some of the bound variables. We make use of this numerous times when we prove statements which are implications predicated on two terms being α-equivalent.

[3] Please note that in the Blackboard video we give a slightly different, but equivalent, definition of α-equivalence, making use of the function freshv, which returns a particular fresh variable.

Below we introduce operations that make the untyped λ-calculus into a programming language, and which allow us to prove properties of that language. As emphasized above we expect these operations to behave sensibly with respect to α-equivalence. This requires us to collect a number of statements that connect α-equivalence and the renaming operation. These may appear tedious or excessive, but in order to ensure that our system behaves as described there is no alternative, and we owe it to the reader to make good on our claim that we provide a rigorous account of our chosen topics.

Proposition 1.6

Show the following for λ-terms t and t'.

(a) If $t = t'$ then $t \sim_\alpha t'$.

(b) If $t \sim_\alpha t'$ then $\operatorname{fv} t = \operatorname{fv} t'$.

(c) If z and w are variables and

$$\lambda w.\, t \sim_\alpha \lambda z.\, t \qquad \text{as well as} \qquad \lambda w.\, t' \sim_\alpha \lambda z.\, t'$$

then

$$\lambda w.\, tt' \sim_\alpha \lambda z.\, tt'.$$

Proof. *This exercise is concerned with proving properties for α-equivalence that are not too technical.*

See page 268.

Proposition 1.7

The following statements hold for λ-terms t and t' and variables x and x'.

(a) If $x \notin \operatorname{fv} t$ and z is a variable with $z \notin \operatorname{vars} t$ then $t \sim_\alpha \operatorname{ren}_x^z t$.

(b) If $t \sim_\alpha t'$ then for every variable z with $z \notin \operatorname{vars} t \cup \operatorname{vars} t'$ we have $\operatorname{ren}_x^z t \sim_\alpha \operatorname{ren}_x^z t'$.

(c) If z is a variable with $z \notin \operatorname{vars}(\lambda x.\, t)$ then $\lambda x.\, t \sim_\alpha \lambda z.\, \operatorname{ren}_x^z t = \operatorname{ren}_x^z(\lambda x.\, t)$.

(d) If $t \sim_\alpha t'$ then $\lambda x.\, t \sim_\alpha \lambda x.\, t'$.

Proof. See page 214.

Proposition 1.8

We have that α-equivalence is an equivalence relation.

Proof. *This is a longish proof but it is not technically intricate. See Section 4.2.1 to remind yourself of the required properties.*

See page 268.

The α-equivalence relation is also well-behaved with respect to subterms.

Proposition 1.9

Let u and t be λ-terms such that u is a subterm of t. If u' is α-equivalent to u then the term t' arising from replacing any occurrence of u in t by u' is α-equivalent to t.

Proof. This is a proof by induction over the structure of the term t. If $u = t$ then there is nothing to show, so below we assume that u is a proper subterm of t.

bcVar If t is a variable then it has no proper subterms so there is nothing to show here.

scAbs If t is an abstraction, say $\lambda x.\, r$, then u has to be a subterm of r, and replacing some occurrence of u gives a term r'. We know by the induction hypothesis that $r \sim_\alpha r'$, and so by Proposition 1.7 we have $\lambda x.\, r \sim_\alpha \lambda x.\, r'$.

scApp If t is an application, say rs, then u has to be a subterm of r or a subterm of s. Hence replacing a copy of u by u' happens either in r, to give a term r', or in s, to give a term s'. By the induction hypothesis we have $r \sim_\alpha r'$ or $s \sim_\alpha s'$, and so by scAppα we have $t' = r's \sim_\alpha rs = t$ or $t' = rs' \sim_\alpha rs = t$.

Much of what we do in the remainder of this section is concerned with ensuring that the various operations we describe work well with α-equivalence. We require that for α-equivalent terms t and t' and an operation op on λ-terms we have that $\mathrm{op}\, t$ and $\mathrm{op}\, t'$ are also α-equivalent, that is

$$t \sim_\alpha t' \qquad \text{implies} \qquad \mathrm{op}\, t \sim_\alpha \mathrm{op}\, t'.$$

1.2.5 Substitution

We are finally ready to define the notion of substitution.

It may not be obvious at this stage, but substitution in the way we use the term is not the same is using the renaming operation!

One might think of substitution as renaming done in way that preserves α-equivalence. This leads to various subtleties. The first of these is the phenomenon of 'shadowing'. This refers to a situation where some instances of the same variable occur freely while some are bound in some λ-term.

Example 1.11. Consider the λ-term $x\lambda x.\, x$. It looks like the free variable x should be an operation on functions, and then the term is the application of this function to the identity function. For instance, if x is replaced by (via substitution, for example) $\lambda f.\, \lambda y.\, f\, y$, then we expect the above to be the application of this to the identity term, that is $(\lambda f.\, \lambda y.\, f\, y)\lambda x.\, x$. However, if we were naively to replace every copy of x with that term we would get the

wrong answer (perhaps even a syntactically meaningless answer, depending on what you try).

This example tells us that substitution is something which operates on free occurrences of a variable only.

A more subtle difficulty arises from the need to respect α-equivalence.

Example 1.12. Consider the terms $\lambda x.\, y$ and $\lambda x'.\, y$ which are α-equivalent and have a free variable y. Suppose we substitute x for y. If we do this naively, the first becomes $\lambda x.\, x$ which is the identity term (and has no free variables!), but the second is $\lambda x'.\, x$ which has a free variable x which is the value it always returns. These terms are no-longer α-equivalent! We say that in the first case, the x has been 'captured' by the λ-abstraction $\lambda x.\,(\ldots)$.

Example 1.13. Alternatively, assume we have two descriptions of the same function,

$$\begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \longmapsto & x \cdot y \end{array} \qquad \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x' & \longmapsto & x' \cdot y, \end{array}$$

If we would like to rename the variable defining the function to y for the left hand description then we get problems if we do so naively:

$$\begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ y & \longmapsto & y \cdot y = y^2 \end{array} \qquad \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x' & \longmapsto & x' \cdot y, \end{array}$$

and now our two descriptions no longer define the same function.

What we need to define is the famous *capture-avoiding substitution*, which watches out when substituting into a λ-abstraction, and renames the bound variable if it occurs inside the term we are substituting in.

So far we have only worried about substituting one variable for another, but we need to consider the more general case where a term is substituted for a variable.

Example 1.14. Assume we have a function $f : \mathbb{R} \longrightarrow \mathbb{R}$ given by

$$f x = x^2 + \log(x+1)^2.$$

Now assume we have a further function, say $g : \mathbb{R} \longrightarrow \mathbb{R}$ given by $gx = xe^x + x$.

If we want to calculate he results of $f(gx)$ we have to substitute the expression that defines g for the variable x in the expression that defines f, so that we obtain

$$f(gx) = (xe^x + x)^2 + \log(xe^x + x + 1).$$

So substituting general terms is a useful operation when we want to calculate the values of functions, and we see in Sections 1.2.5 and 1.2.6 below how this is used when computing with λ-terms.

Example 1.15. We look at a variation of the previous example. Now assume that

$$\begin{aligned} f : \mathbb{R} &\longrightarrow \mathbb{R} & \qquad g : R &\longrightarrow R \\ x &\longmapsto x^2 + \log(x+1)^2 & \qquad y &\longmapsto x \cdot e^y, \end{aligned}$$

where x is a parameter for the second function. If we want to calculate what happens to an input if we first apply the function g, and then f, then we may do so by replacing x in the expression defining f by the output of g. However, if we do this naively, we obtain the assignment

$$x \longmapsto (x \cdot e^y)^2 + \log(x \cdot e^y + 1)^2,$$

and that is certainly not what we intended. The variable for which g gives us a function is y, but we can't just replace that y by x since x occurs as a parameter in the definition of g. In turn, f might have a parameter called y in its definition! So the safest thing to do is to use a fresh variable name to describe the assignment we want to define and use

$$z \longmapsto (x \cdot e^z)^2 + \log(x \cdot e^z + 1)^2.$$

We need to define substitution in such a way so as to avoid the pitfalls mentioned above, and the machinery developed so far is largely to support this, and so the following definition.[4]

Definition 8: capture-avoiding substitution

Given λ-terms t and a, and a variable x, we define the **capture avoiding substitution** of a for x in t, denoted $t[a/v]$, by recursion on t as follows.

bcVar[] In the base case we have $t = y$ for some variable y and

$$y[a/x] = \begin{cases} a & y = x \\ y & \text{else.} \end{cases}$$

scAbs[] Here $t = \lambda y.\, u$ and for $w = \mathrm{freshv}(\mathrm{vars}\, t \cup \mathrm{vars}\, a \cup \{x\})$ we set

$$(\lambda y.\, u)[a/x] = \lambda w.\, (\mathrm{ren}^w_y\, u)[a/x].$$

scApp[] Here $t = rs$ and $(rs)[a/x] = (r[a/x])(s[a/x])$.

The definition above looks complicated, and indeed it is a significant discovery. The base case and the case of application are fairly straightforward, and all the subtleties are in the case of λ-abstraction. We briefly discuss that definition.

- We finally makes use of the assumption that the set of variables is infinite, to make sure that we change the variable we use for the outermost λ-abstraction

[4] This definition is not quite the one we give in the Blackboard video—there we make a case distinction in the abstraction case. Up to α-equivalence, it does not matter which of these decisions we choose. The one we give here is more convenient when it comes to proving properties of substitution.

is not free in the term we are substituting in (and it also makes sure that it does not change the meaning of the original by using a variable that already occurs in t).

- Example 1.12 shows that we should not carry out the substitution if we would do so for a variable bound by the outermost λ-abstraction. Example 1.12 shows that in that situation we should not carry out the substitution. By renaming the abstracted variable into a fresh one throughout the term, we make sure that this substitution does not occur. We could instead have changed our definition to say that if $x = y$, and so the term into which we aim to substitute is $\lambda x.\, u$ we should just return that term without any changes. However, if we have to prove something that involves the substitution operation we typically have to make case distinctions that mirror those of this definition, and so having fewer of these makes such proofs shorter. We show in Proposition 1.10 that up to α-equivalence, it does not matter which of these options we choose: Since $x \notin \mathrm{fv}(\lambda x.\, u)$ that result establishes that

$$(\lambda x.\, u)[a/x] \sim_\alpha \lambda x.\, u.$$

- We could also have avoided the renaming provided that $y \neq x$ as well as $y \notin \mathrm{fv}\, a$, so as to avoid changing the term when it is not necessary. Again, it makes proofs shorter if we avoid this case distinction.

Example 1.16. We revisit the two terms from Example 1.12 to see what happens if instead of applying the renaming function we apply the substitution one. We assume that y, x and x' are distinct variables. Assume that $w = \mathrm{freshv}\{y, x\}$. Then

$$\begin{aligned} (\lambda x.\, y)[x/y] &= \lambda w.\, (\mathrm{ren}^{w}_{x}\ y)[x/y] && \text{scAbs[ ]} \\ &= \lambda w.\, y[x/y] && \text{scAbsren} \\ &= \lambda w.\, x && \text{bcVar[ ]} \end{aligned}$$

while

$$\begin{aligned} (\lambda x'.\, y)[x/y] &= \lambda w.\, (\mathrm{ren}^{w}_{x'}\ y)[x/y] && \text{scAbs[ ]} \\ &= \lambda w.\, y[x/y] && \text{scAbsren} \\ &= \lambda w.\, x && \text{bcVar[ ]} \end{aligned}$$

We can see that we are left with two terms that are still α-equivalent.

We collect some facts about substitution. Again we require these to establish properties about our systems viewed as programming languages.

Proposition 1.10

The following statements hold for λ-terms t and a and variable x.

(a) $\mathrm{fv}(t[a/x]) \subseteq (\mathrm{fv}\, t \setminus \{x\}) \cup \mathrm{fv}\, a$.

(b) If $x \notin \mathrm{fv}\, t$ then $t[a/x] \sim_\alpha t$.

(c) If z and z' are fresh for t, a and x and y is an arbitrary variable then

$$\mathrm{ren}_z^{z'}((\mathrm{ren}_y^z u)[a/x]) \sim_\alpha (\mathrm{ren}_y^{z'} t)[a/x].$$

(d) If z and y are variables with $z \notin \mathrm{vars}(\lambda y.\, t) \cup \mathrm{vars}\, a \cup \{x\}$ then

$$(\lambda y.\, t)[a/x] \sim_\alpha \lambda z.\, (\mathrm{ren}_y^z t)[a/x].$$

(e) If z and y are variables with $z \notin \mathrm{vars}\, t \cup \mathrm{vars}\, a$ then

$$\mathrm{ren}_y^z(t[a/x]) \sim_\alpha (\mathrm{ren}_y^z t)[\mathrm{ren}_y^z a/\, \mathrm{ren}_y^z x].$$

(f) If z is a variable with $z \notin \mathrm{vars}\, t \cup \mathrm{vars}\, a$ then

$$(\mathrm{ren}_x^z t)[a/z] \sim_\alpha t[a/x].$$

Proof. See page 215.

We should emphasize that we think of this renaming as a low-level detail, and that in our examples we feel free to implicitly use α-equivalence everywhere to name variables in a more human-readable way.

When carrying out substitution for a given term we need to know which function freshv is used to provide us with fresh variable names as needed. For looking at such examples it may be easier instead to carry out β-reduction up to α-equivalence, and we say more about this at the end of this section.

Example 1.17. Suppose our low-level variables are numbers in boxes, and freshv adds 1 to the maximum of its arguments. Then we would have for example

$$(\lambda\boxed{1}.\,\boxed{2})[\boxed{1}/\boxed{2}] = \lambda\boxed{3}.\,\boxed{1}.$$

If we want to calculate what $\lambda x.\, y[x/y]$ is we see that we need to know

$$\mathrm{freshv}(\mathrm{vars}(\lambda x.\, y) \cup \{x\} \cup \{y\}) = \mathrm{freshv}\{x, y\}.$$

If we name that variable w we may then write

$$(\lambda x.\, y)[x/y] = \lambda w.\, x$$

without having to look into exactly how w is calculated from $\{x, y\}$.

An important property of substitution is that if we substitute α-equivalent terms into α-equivalent terms, we obtain terms that are again α-equivalent.

Proposition 1.11

Let t, t', a and a' be λ-terms. If $t \sim_\alpha t'$ and $a \sim_\alpha a'$ then for all variables x we have that $t[a/x] \sim_\alpha t'[a'/x]$.

In particular if $\lambda x.\, t \sim_\alpha \lambda x'.\, t'$ then

$$t[a/x] \sim_\alpha t'[a/x'].$$

Proof. *This is a proof by induction over the structure of one of the terms. The base case and the application case are straightforward, and for the abstraction case one may use previously established results to give a succinct argument for that case. The second statement follows quite easily from statements we have shown.*

See page 269.

We ask ourselves what happens if we substitute twice.

Example 1.18. Consider a term where we want to substitute twice, that is a term of the form $t[b/y][a/x]$. We would like to think about whether there is a way in which we could change the order in which the substitutions are carried out. We look at a small example to better understand the situation, using the following:

$$(yx)[\lambda z.\, zx/y][\lambda z'.\, z'/x]$$

We carry out the first substitution, which gives

$$((\lambda z.\, zx)x)[\lambda z'.\, z'/x].$$

For simplicity's sake we assume that all the variables are distinct. For w the appropriate variable produced by the freshv function the second substitution gives (after carrying out the appropriate renaming operation)

$$(\lambda w.\, w(\lambda z'.\, z'))(\lambda z'.\, z').$$

If instead we first carry out the second substitution of $a = \lambda z'.\, z'$ for x in the given term $'lty.x$ the result is

$$y(\lambda z'.\, z').$$

We can see that in order to obtain the same result as before (up to α-equivalence) it is not sufficient to replace y by $b = \lambda z.\, zx$—we also have to replace the x that appears in that formula by $a = \lambda z'.\, z'$. In other words, we have to carry out $t[a/x][b[a/x]/y]$. This observation motivates the formula given in the following exercise.

Based on the ideas introduced in the previous example we are able to formulate two properties that allow us to cope with substituting twice into the same term.

Proposition 1.12

If t, a and b are λ-terms and x and y are distinct variables such that $y \notin \mathrm{fv}\, a$ then

(a) $t[b[a/x]/x] \sim_\alpha (t[b/x])[a/x]$.

(b) $t[a/x][(b[a/x])/y] \sim_\alpha (t[b/y])[a/x]$.

Proof. See page 224.

1.2.6 β-Reduction

In any programming language, once a program has been written something has to happen to ensure that the program actually *does* something. It may have to be compiled, or interpreted, and following the instructions laid down something *happens*. For example, in an imperative language (such as the While language) the values stored in some memory locations change.

We may think of this as the *dynamic* behaviour associated with a program. There is an interesting question of how this is actually defined for a given programming language, and indeed the history of the subject has a few cases where different versions of what was meant to be the same language show different behaviours.

On this course unit we want to be able to reason rigorously about the behaviour of programs by way of capturing their meaning, so we need to give a rigorous definition of these 'dynamics'. In order to finally get there we had to define the crucial operation of substitution, and that required the machinery built up so far.

As suggested previously the dynamic behaviour we use is based on that notion of substitution, and this also gives an explicit description of the the process assumed when describing function application or proofs involving quantifiers. We may think of this as turning our set of terms into a programming language.

Definition 9: β-reduction

We define the relation of **β-reduction**, denoted by $\xrightarrow{\beta}$, on λ-terms as follows.

bcVarβ $(\lambda x.\, t)a \xrightarrow{\beta} t[a/x]$.

scAbsβ If $t \xrightarrow{\beta} t'$ then $\lambda x.\, t \xrightarrow{\beta} \lambda x.\, t'$.

scAppβ if $t \xrightarrow{\beta} t'$ then $tu \xrightarrow{\beta} t'u$ and if $u \xrightarrow{\beta} u'$ then $tu \xrightarrow{\beta} tu'$.

We see in Section 1.3 how β-reduction allows us to program in the λ-calculus. For now we observe that

- the base case tells us to use substitution to 'apply' a λ-term to an 'argument' and
- the remaining cases say that we are allowed to apply reduction to such situations that occur 'inside' a λ-term.

We develop terminology that allows us to talk about these two cases. A **redex** is a term that matches the bcVarβ case, while a term that matches one of the other two cases is a *term that contains a redex*. See Example 1.20 for a situation where both arise.

We write

$$\xrightarrow{\beta}\!\!\!\!\!\rightarrow$$

for the reflexive transitive closure of $\xrightarrow{\beta}$. When we use the term *β-reduction* we sometimes mean this closure rather than the original relation. When we wish to make it clear which we mean, we explicitly say '*β-reduces in one step*' or '*β-reduces in many steps*'.

Example 1.19. Recall Example 1.4 suggesting that when we apply a function to a specific input we calculate the output by substituting the input for the variable(s) in the expression specifying the function's behaviour. We note that the input could be something more complicated than a number—it could be the output of another function, not yet evaluated but written as an arithmetic expression.

We can see this behaviour for β-reduction in the case where the λ-term in question is an abstraction:

$$(\lambda x.\,(xx)(yx))(\lambda z.\,zz) \xrightarrow{\beta} ((xx)(yx))[\lambda z.\,zz/x] = ((\lambda z.\,zz)(\lambda z.\,zz))(y(\lambda z.\,zz))$$

We have replaced each occurrence of the the bound variable x with the 'argument' $\lambda z.\,zz$.

Example 1.20. We look at a more complicated example. Consider the λ-term

$$((\lambda x.\,x)\lambda y.\,yy)z.$$

If we look at the rules as given, it's not immediately clear how to apply them, and it is tempting to think of z as the argument that we have to do something with. However, a second glance at the definition shows that

- we have to find an instance of a λ-abstraction and
- there has to be a term this abstraction is applied to.

Our term is a triple application, and so the only λ-abstraction that is applied to anything is the first one.

Hence the only possible step we may take is the first of the rules for **scApp**β: Noting that

$$(\lambda x.\,x)\lambda y.\,yy \xrightarrow{\beta} \lambda y.\,yy$$

we may reduce the overall term

$$((\lambda x.\,x)\lambda y.\,yy)z \xrightarrow{\beta} (\lambda y.\,yy)z,$$

replacing the second x that appears in the term by $\lambda y.\,yy$. So the term we started with *contains a redex*, and reducing that term results in a redex, which means that we may use bcVarβ and continue with

$$(\lambda y.\,yy)z \xrightarrow{\beta} zz.$$

RExercise 6. Carry out three steps of β-reduction for the following term

$$((\lambda x.\,\lambda y.\,z(xy))\lambda y'.\,y'x)((\lambda x.\,x)(yy')).$$

Hint: You may want to start by putting in extra brackets if you find that helpful, and then you need to find a redex in the term.

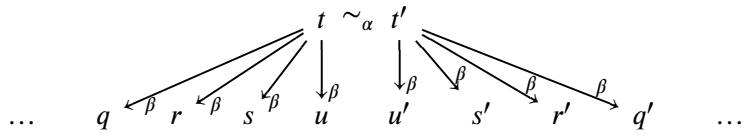
See 270 for a solution.

Further examples of β-reductions are given in the following sections, see Examples 1.22, 1.23, 1.26 and 1.28. Note that there is nothing that limits the number

of redexes that may be contained in a term—the reader interested in this situation may want to look ahead to Example 1.32

1.2.7 Properties of β-reduction

A persistent thread of our narrative is that everything we define should be compatible with the idea that we don't want to distinguish between α-equivalent terms as far as their behaviour is concerned. This is certainly true for β-reduction, but since this is not an operation on λ-terms but a *relation* we need to think about what we mean by this. If we have two α-equivalent terms t and t', they might β-reduce in a number of ways, that is



What should we demand in this situation? The answer may be found in Proposition 1.16, but to establish that we also have to worry about the interactions between α-equivalence and renaming, as well as α-equivalence and substitution.

We show first of all that if a subterm of a given term can do a reduction then there is a matching reduction in that term.

Proposition 1.13

If u is subterm of t and $u \xrightarrow{\beta} u'$ then there is a term t' such that u' is a subterm of t' and $t \xrightarrow{\beta} t'$.

Further if u is a subterm of t and $u \xrightarrow{\beta}{}^{*} u'$ in n steps then there is a term t' such that u' is a subterm of t' and $t \xrightarrow{\beta}{}^{*} t'$ in n steps.

Proof. *This exercise invites students to think about how reductions of subterms relate to reductions of the given term.*

See page 271.

When we prove a statement where we assume that we have terms t and t' with $t \xrightarrow{\beta} t'$, we sometimes say that this is 'by induction over why t β-reduces to t'.' What this means is that we carry out a structural induction with additional case distinctions since we typically need to know more about the structure of a term to track how t' comes about, and knowing that t β-reduces gives us some information about the structure of the term (it can't be a variable, for example). It is important still to ensure that all cases are covered, unless we can rule them out by what we know about the structure of the term.

As usual we have to establish a connection between renaming and the new concept.

Proposition 1.14

If $t \xrightarrow{\beta} t'$ and x and z are variables with $z \notin \operatorname{vars} t$ then there is a λ-term u with $\operatorname{ren}_x^z t \xrightarrow{\beta} u \sim_\alpha \operatorname{ren}_x^z t'$.

Proof. See page 227.

We show that β-reduction and substitution work well together.

Proposition 1.15

For all variables x and λ-terms t, t', a and a' we have

(a) if $t \xrightarrow{\beta} t'$ then $\operatorname{fv} t \supseteq \operatorname{fv} t'$,

(b) if $t \xrightarrow{\beta} t'$ then there is $u' \in \Lambda\mathsf{Trm}$ with $t[a/x] \xrightarrow{\beta} u' \sim_\alpha t'[a/x]$, and

(c) if $a \xrightarrow{\beta} a'$ then there is $u' \in \Lambda\mathsf{Trm}$ with $t[a/x] \xrightarrow{\beta} u' \sim_\alpha t[a'/x]$.

Proof. See page 228.

We further note that β-reduction works fine for α-equivalence in the sense that α-equivalent terms can essentially carry out the same reductions.

The following result tells us that if we have two α-equivalent terms, and one of them can carry out a reduction, then the other term can mirror it: If we have the information on the left, we get the information on the right.

$$\begin{array}{ccc} t & \sim_\alpha & u \\ {\scriptstyle\beta}\downarrow & & \\ t' & & \end{array} \qquad \text{there exists } u' \text{ with} \qquad \begin{array}{ccc} t & \sim_\alpha & u \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ t' & \sim_\alpha & u' \end{array}$$

Proposition 1.16

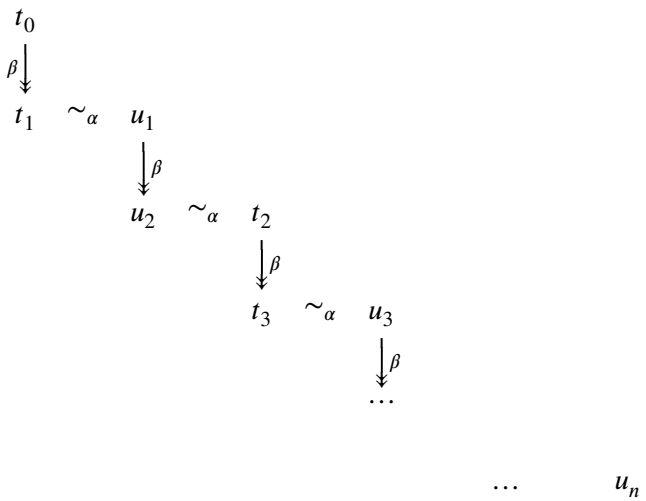
If t and u are α-equivalent λ-terms with $t \xrightarrow{\beta} t'$ then there exists u' with $u \xrightarrow{\beta} u' \sim_\alpha t'$.

Further if $t \sim_\alpha u$ and $t \xrightarrow{\beta} t'$ then there exists u' with $u \xrightarrow{\beta} u' \sim_\alpha t'$.

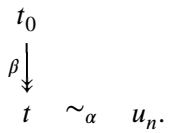
Proof. See page 230.

We have to reason about situations where we do not carry out just one reduction step, but a number of them, and sometimes we find ourselves in a situation that we have to reason about terms that are α-equivalent to the ones we reach,

that is situations like this:



The following results us that there is a way of carrying out a number of β-reductions to obtain a term that is α-equivalent to the final term u_n:



Corollary 1.17

Let $t_0, t_1, \dots t_n$ and $u_0, u_1, \dots, u_n$ be terms such that for $0 \leq i < n/2$ we have $t_{2i} \xrightarrow{\beta} u_{2i+1}$ as well as $u_{2i+1} \xrightarrow{\beta} t_{2i+2}$. and for all $0 \leq i \leq n$ we have $t_i \sim_\alpha u_i$. Then there exists a term t with $t_0 \xrightarrow{\beta} t \sim_\alpha u_n$.

Proof. *This proof is fairly straightforward if one can find the right perspective, but students might find it frustrating otherwise, in which case they shouldn't fret.*

See page 271.

When we carry out β-reduction for a given term we have to carry out substitution, which means we have to make reference to freshv, when it may be tempting instead to write down an α-equivalent term. When we do that we need to make it clear that we are no longer talking about β-reduction as we have defined it.

Example 1.21. Consider the term $(\lambda y.\, \lambda x.\, y)z$. We know that this β-reduces to

$$\lambda x.\, y[z/y],$$

and we can see that for

$$w = \text{freshv}(\text{vars}(\lambda x.\, y) \cup \{z\} \cup \{y\}) = \text{freshv}\{x, y, z\}$$

this is

$$\lambda w.\, z,$$

but it is rather tempting instead to write the term

$$\lambda x.\, z,$$

which is α-equivalent to the actual result.

The previous result tells us that when we interleave β-reduction with α-equivalence we get a relation with sensible properties.

We write

$$t \xrightarrow{\beta\alpha} u \qquad \text{if and only if} \qquad \text{there is } t' \text{ with } t \xrightarrow{\beta} t' \sim_\alpha u.$$

So in Example 1.21 we would write

$$(\lambda y.\, \lambda x.\, y)x \xrightarrow{\beta\alpha} \lambda x.\, z.$$

We only use this relation for specific examples—when we talk about terms generally we keep β-reduction and α-equivalence separate. We talk of $\beta\alpha$-reduction in this situation.

Note that the relation $\xrightarrow{\beta\alpha}$ is also well-behaved for subterms.

Corollary 1.18

If $u \xrightarrow{\beta\alpha} u'$ and u is a subterm of t then there exists a term t' such that $t \xrightarrow{\beta\alpha} t'$ and u' is a subterm of t'.

Proof. We have all the ingredients to show this. If $u \xrightarrow{\beta\alpha} u'$ we know that we can find a term r with $u \xrightarrow{\beta} r \sim_\alpha u'$. By Proposition 1.13 we know that we can find a term s such that r is a subterm of s and $t \xrightarrow{\beta} s$. By Proposition 1.9 we further know that if we replace an occurrence of r in s by the α-equivalent term u' we obtain a term t' that is α-equivalent to s. Hence we have that

$$t \xrightarrow{\beta} s \sim_\alpha t',$$

which means $t \xrightarrow{\beta\alpha} t'$.

Looking at examples given in the text the reader might wonder why having gone to such trouble to define substitution and β-reduction precisely, we now return to relying on the reader's intuitions in that we use α-equivalence informally, and without worrying about which particular variable might be returned by the function freshv. The reason is that we are taking the road Church made for us but *in the opposite direction*: whereas Church wanted to show that everyday mathematics could be turned into a precise syntactic system, we want to view this system as a programming language, and ask whether we can reason about it using everyday mathematics. Now we know that a precise specification of substitution which respects α-equivalence is possible, we make free use of α-equivalence to help us understand what it does.

A typical computation will involve more than one β-reduction and so we check that $\xrightarrow{\beta}$ inherits the properties which $\xrightarrow{\beta}$ has by definition or because of Proposition 1.15.

Proposition 1.19

For all variables x and y and λ-terms t, t', u, u', a and a' such that

$$t \xrightarrow{\beta}\!\!\!\!\!\rightarrow t', \qquad u \xrightarrow{\beta}\!\!\!\!\!\rightarrow u' \qquad \text{and} \qquad a \xrightarrow{\beta}\!\!\!\!\!\rightarrow a',$$

we have

(a) $\lambda y.\, t \xrightarrow{\beta}\!\!\!\!\!\rightarrow \lambda y.\, t'$,

(b) $tu \xrightarrow{\beta}\!\!\!\!\!\rightarrow t'u$,

(c) $tu \xrightarrow{\beta}\!\!\!\!\!\rightarrow tu'$,

(d) $\mathrm{fv}\, t \supseteq \mathrm{fv}\, t'$ and

(e) there is a λ-term t'' with $t[a/x] \xrightarrow{\beta}\!\!\!\!\!\rightarrow t'' \sim_\alpha t'[a'/x]$.

Proof. See page 232.

1.3 How to think of example terms

So far we have been having a good time learning about the history of logic, but our motivation for studying the λ-calculus was that it could be used as a functional programming language. Though this is *not* a course about writing programs in the λ-calculus, intellectually speaking we should check that we can write example terms with complicated behaviour and think of them as functional programs in some sense. This reassures us that the mathematical results we derive do have something to do with programming languages!

1.3.1 Computation via substitution

We have something in the λ-calculus which looks like it might play the role of functions. There are λ-terms which mimic the way we write function definitions as expressions with variables which can be substituted for. Then β-reduction gives these a meaning by explaining how and when to carry out substitution.

Example 1.22. Consider again the term which resembles the identity function, $\lambda x.\, x$ and let us think about what happens when we combine it with another term, say $\lambda f.\, \lambda x.\, f\, x$ via application. We get $(\lambda x.\, x)\lambda f.\, \lambda x.\, f\, x$, which before we had defined β-reduction was just a piece of syntax. But now we can see that it is of the same form as the first case of the definition of β-reduction: Working through the definition of β-reduction and substitution, we find

$$(\lambda x.\, x)\lambda f.\, \lambda x.\, f\, x \xrightarrow{\beta} \lambda f.\, \lambda x.\, f\, x$$

so for this term, β-reduction looks like a kind of 'simplification' which matches our expectation of what should happen when we apply the identity function to something. (Namely, that we get that thing back again!)

Example 1.23. Let's look at a slightly more complicated application,

$$(\lambda y.\,(g\,y)\,y)\lambda f.\,\lambda x.\,f\,x,$$

which contains a free variable g. The free variable makes it harder to read since we wonder 'who is g?', but if we remember that λ-terms are just pieces of syntax, this is a perfectly good λ-term and, indeed, a redex. We have

$$(\lambda y.\,(g\,y)\,y)\lambda f.\,\lambda x.\,f\,x \xrightarrow{\beta} (g(\lambda f.\,\lambda x.\,f\,x))\lambda f.\,\lambda x.\,f\,x$$

which shows that we make two copies of the input, as expected, and put them in the right place. So far, so good.

1.3.2 Using multiple arguments

We seem to have captured the crucial syntactic aspect of functions that applying them to an input means copying that input and pasting it in the function definition wherever the argument variable occurs there. Next, we might wonder what to do about functions which take several arguments.

Example 1.24. We look at an example of a function which takes two arguments which we would like to be able to express in a functional programming language.

Assume that $\mathcal{F}$ is a set of functions from some set S to some set T. We may then define a function

$$\begin{aligned} \mathcal{F} \times S &\longrightarrow T \\ (f, s) &\longmapsto f\,s, \end{aligned}$$

which takes as its input a function from $\mathcal{F}$ and an element of s and returns as its output the result of applying f to the argument s. We call this the *application function.*

If we want to find a λ-term that behaves like the application function from Example 1.24 we have to solve the problem of how to talk about pairs. If we had a λ-calculus with a notion of pairs built in, we might mimic the way we would write this in ordinary maths, $(f, x) \mapsto f\,x$ and write this as $\lambda(f, x).\,f\,x$. However, the λ-calculus we have defined does *not* have such a notion.

Does this mean we can only write one-argument functions in the λ-calculus? In a strict sense, the answer is 'yes', but in fact it is fairly straightforward to model functions of more than one argument. The key insight is that in the λ-calculus it is easy to write a λ-term which, when applied to an input, reduces to a term which can *also* be applied to an input. This means we can think of the original term as eventually applying to several inputs.

Example 1.25. Let's think about this in terms of ordinary functions, before worrying about λ-terms. Consider the division function

$$\begin{aligned} d : \mathbb{R} \times \mathbb{R} &\longrightarrow \mathbb{R} \\ (x, y) &\longmapsto y/x \end{aligned}$$

How can we express this function without using pairs, and instead using the ability to output a function? Let $\mathsf{Fun}(\mathbb{R}, \mathbb{R})$ be the set of functions from $\mathbb{R}$ to $\mathbb{R}$. Instead of writing the function d above, we can instead write a function $d' : \mathbb{R} \to \mathsf{Fun}(\mathbb{R}, \mathbb{R})$ which sends an input x to the function $d''_x : \mathbb{R} \to \mathbb{R}$ which divides its input by x. Concretely, for all $x \in \mathbb{R}$, we have

$$\begin{aligned} d''_x : \mathbb{R} &\longrightarrow \mathbb{R} \\ y &\longmapsto y/x \end{aligned}$$

and then we can define

$$\begin{aligned} d' : \mathbb{R} &\longrightarrow \mathsf{Fun}(\mathbb{R}, \mathbb{R}) \\ x &\longmapsto d''_x \end{aligned}$$

and then for all $w, z \in \mathbb{R}$ we have

$$(d'w)z = d''_w z = z/w = d(w, z)$$

so by using a function which outputs another function, we are able to write a function which accepts two inputs without using pairs.

Example 1.26. To see what this means for λ-terms, let's consider the λ-term which models the application operation, $\lambda f.\, \lambda x.\, fx$ and apply it to $\lambda y.\, y$. We have

$$(\lambda f.\, \lambda x.\, fx)\lambda y.\, y \xrightarrow{\beta\alpha} \lambda x.\, (\lambda y.\, y)x$$

so if we apply it to $\lambda y.\, y$ and then to something else—let's just take a free variable z–we have

$$\begin{aligned} ((\lambda f.\, \lambda x.\, fx)\lambda y.\, y)z &\xrightarrow{\beta\alpha} (\lambda x.\, (\lambda y.\, y)x)z \\ &\xrightarrow{\beta\alpha} (\lambda y.\, y)z \end{aligned}$$

which means we get the application we wanted. If we look in more detail at the result of just applying $\lambda f.\, \lambda x.\, fx$ to $\lambda y.\, y$ and reducing, we have $\lambda x.\, (\lambda y.\, y)x$. This is a model of the function which takes an input and applies $\lambda y.\, y$ to it. So it is like the application function but with the first argument 'fixed' to $\lambda y.\, y$.

RExercise 7. Write a λ-term which is supposed to take three arguments, f, g, and x, with the following intended interpretations. The argument f is supposed to be a two-argument term in the sense explained above. The argument g is supposed to take one argument, and we regard x as an arbitrary piece of data. Your term is supposed to return f with x supplied as its first argument, and the result of applying g to x as its second.

If we were allowed to use tuples, we would write the operation as

$$\lambda(f, g, x).\, f(x, gx)$$

How do we write this as a proper λ-term?

See 272 for a solution.

The above shows that we can model the simple operation of substituting the input to a function into its definition, and this seems to work as expected, including a way to represent functions which take multiple arguments.

1.3.3 Non-trivial behaviour

To come up with interesting examples of λ-terms to study, we find the following ideas useful.

- **Projections**. Now we know how to model multi-argument functions, we can look at the simplest examples: projection functions. These simply return one of their arguments. For instance, there are three projection functions with three arguments: $\lambda p.\, \lambda s.\, \lambda r.\, p$, $\lambda p.\, \lambda s.\, \lambda r.\, s$, and $\lambda p.\, \lambda s.\, \lambda r.\, r$. Try applying these to some terms to see what they do. Though their behaviour is simple, projection functions are nice building blocks for more complicated programs.
- **Returning multiple results**. One thing we can do with projection functions is take them as arguments. The simplest example of this is a term like $\lambda q.\, qxyz$. If we apply this term to the projection $\lambda p.\, \lambda s.\, \lambda r.\, p$, the result β-reduces to $(\lambda p.\, \lambda s.\, \lambda r.\, p)xyz$ which β-reduces to x. Similarly, applying it to the other projections can result in either y or z depending which projections we supply. So the term $\lambda q.\, qxyz$ acts like a tuple containing three values x, y and z. We could write examples which construct such terms, which is like returning multiple results.
- **Case selection**. Another way of reading the term $(\lambda p.\, \lambda s.\, \lambda r.\, p)xyz$, if we only consider what it does when given a three-input projection as an argument, is that it is a case statement which does something different depending on which one is it given. In this interpretation, x, y and z would be more complex terms encoding subsequent behaviour. This means the three projections with three inputs behave a bit like elements of an "enumeration" data structure, since we can define terms which have three possible behaviours, and select which to use by accepting a projection as input.

  A simpler case of this phenomenon is an "if-then-else" construct. Here we only need two values to represent 'true' and 'false', so we would use the two two-argument projections, $\lambda t.\, \lambda f.\, t$ and $\lambda t.\, \lambda f.\, f$.
- **Iteration**. Suppose we have two λ-terms s and z, and we would like to iterate s five times applied to z. We could write this as $sssssz$. We could abstract this situation by writing an "iterate five times" term $\lambda z.\, \lambda s.\, sssssz$. This "iterator" term is nice, because we can accept it as an argument, meaning that we can write terms which perform iteration a varying number of times. For instance consider the term $\lambda n.\, ntf$. If we pass this an iterator which repeats its argument n times, then the result will be f applied n times to t. This is like having a "for loop" which can depend on a variable.

Example 1.27 (Swapping). To investigate the idea of returning multiple arguments, we write a 'swapping' term which given a pair implemented as

discussed above, returns a similar pair but with the contents swapped. We follow the discussion above to extract the elements of the pair, and construct a new pair where these occur in the opposite order, which gives the term

$$\lambda p.\,(\lambda f.\,f\,[p(\lambda x.\,\lambda y.\,y)]\,[p(\lambda x.\,\lambda y.\,x)]).$$

RExercise 8. Write a λ-term which takes one input and returns the encoding of a pair containing two copies of he input.

A solution appears on page 272.

Example 1.28 (Paper, Scissors, Rock). An example of the 'case selection' behaviour described above can be given by trying to model the behaviour of the game "paper, scissors, rock". We would like to come up with three terms to represent the three outcomes, and a function which cycles between them.

Since we want three values, we use the three three-argument projections. If we want to write the element rock following the idea above, we might write this as the λ-term $\lambda p.\,\lambda s.\,\lambda r.\,r$. Similarly, we can model paper as $\lambda p.\,\lambda s.\,\lambda r.\,p$ and scissors as $\lambda p.\,\lambda s.\,\lambda r.\,s$. Now we could define the function f from above as follows.

$$\lambda x.\,\lambda y.\,(((xg)h)i)y$$

And the function beats above is encoded by the λ-term

$$\begin{aligned}\lambda x.\,(((x[&\lambda p.\,\lambda s.\,\lambda r.\,r])\\ &[\lambda p.\,\lambda s.\,\lambda r.\,p])\\ &[\lambda p.\,\lambda s.\,\lambda r.\,s]).\end{aligned}$$

Now to find out which element is beaten by scissors, we apply this to $\lambda p.\,\lambda s.\,\lambda r.\,s$ and we get

$$\begin{aligned}
& \lambda x.\,(((x[\lambda p.\,\lambda s.\,\lambda r.\,r])[\lambda p.\,\lambda s.\,\lambda r.\,p])[\lambda p.\,\lambda s.\,\lambda r.\,s])(\lambda p.\,\lambda s.\,\lambda r.\,s)\\
\xrightarrow{\beta\alpha}\ & (((\lambda p.\,\lambda s.\,\lambda r.\,s)[\lambda p.\,\lambda s.\,\lambda r.\,r])[\lambda p.\,\lambda s.\,\lambda r.\,p])[\lambda p.\,\lambda s.\,\lambda r.\,s]\\
\xrightarrow{\beta\alpha}\ & ((\lambda s.\,\lambda r.\,s)[\lambda p.\,\lambda s.\,\lambda r.\,p])[\lambda p.\,\lambda s.\,\lambda r.\,s]\\
\xrightarrow{\beta\alpha}\ & (\lambda r.\,[\lambda p.\,\lambda s.\,\lambda r.\,p])[\lambda p.\,\lambda s.\,\lambda r.\,s]\\
\xrightarrow{\beta\alpha}\ & [\lambda p.\,\lambda s.\,\lambda r.\,p]
\end{aligned}$$

which is our encoding of paper, as expected!

This shows that the apparently simple λ-calculus can encode non-trivial behaviours.

Example 1.29. Similar behaviour to that given in Example 1.28 can be represented by a different term which you may find neater:

$$\lambda x.\,\lambda p.\,\lambda s.\,\lambda r.\,xpsr$$

Try applying that to the same input as in Example 1.28.

RExercise 9. Since the booleans have two elements, if we want to model the behaviour of functions on booleans, we can try using the projections

- $\lambda t.\, \lambda f.\, t$ for 'true', and
- $\lambda t.\, \lambda f.\, f$ for 'false,.

also compare page 37.

Write a λ-term which computes the conjunction, also known as the logical 'and' function, for this encoding of the booleans.

Try to find a term which works written in the style of both Example 1.28 and the 'neater' Example 1.29..

See page 272 for a solution.

Next we look at how to encode some operations on 'iterator' terms.

Example 1.30 (Successor). While it might at first seem simplistic, it is useful to have an explicit λ-term which takes an iterator term, and returns a new one which does one more iteration. To do this, we want to 'insert an extra s'. For example $\lambda z.\, \lambda s.\, s(sz)$ must become $\lambda z.\, \lambda s.\, s(s(sz))$. One idea for how to do this is to feed the given term the expression sz as its base case. This idea needs improvement in two ways. If we take this literally and implement successor as

$$\lambda n.\, n(sz)$$

then if we apply this to $\lambda z.\, \lambda s.\, s(sz)$ and β-reduce as much as possible, we get

$$\lambda w.\, w(w(sz)).$$

First, this is no longer an encoding of a natural number because it only expects one argument. But worse, our definition was not invariant under α-equivalence, because we assumed that the variables involved in n were actually called s and z. But even if they are to begin with, capture-avoiding substitution will replace them with fresh variables, which in this case leaves us with s and z as free variables.

To get around this, we need to output something which begins with $\lambda z.\, \lambda s.\, \ldots$ as expected from an encoding of a number. Then we can safely pass z and s into n, because if s and z get renamed, n will receive the renamed ones to use. This means our final implementation is

$$\lambda n.\, (\lambda z.\, \lambda s.\, n[sz]s).$$

Example 1.31. We can now look at implementing a more sophisticated operation. For instance, suppose we want to write a term which when applied to an iterator, returns the iterator which does twice as many iterations. We take

inspiration from the recusive way of defining the doubling function:

$$d(n) = \begin{cases} 0 & \text{if } n = 0 \\ SSdn' & \text{if } n = Sn' \end{cases}$$

where S is the successor function. The base case here is just 0, which corresponds to $\lambda z.\, \lambda s.\, z$. Note that we have to return an encoded natural number, which is a term beginning $\lambda z.\, \lambda s.\, \ldots$, in either case. That means we should start our returned value with $\lambda z.\, \lambda s.$ and for the base case simply output z.

We need to think of the step case as an operation applied to the result of doubling the predecessor. If we use r to label this result, then the operation we apply in the step case is $\lambda r.\, ssr$. Overall, this means that we expect the doubling function to correspond to the λ-term

$$\lambda n.\, (\lambda z.\, \lambda s.\, nz[\lambda r.\, s(sr)])$$

RExercise 10. Write a λ-term which when applied to two iterators outputs an iterator which repeats its second argument as many times as the sum of the repetitions of the inputs. Be sure to test your term on a small input to make sure you get the result you expect!

A solution may be found on page 272.

RExercise 11. Suppose you are given a λ-term p which, when supplied with two iterators, returns the iterator which repeats its argument as many times as the sum of the number of repetitions of its input. In terms of p, write a λ-term which implements multiplication in the same sense.

A solution appears on page 273.

1.4 Confluence

In this section we consider the question in which sense computation in the λ-calculus gives us a well-defined answer.

1.4.1 Diverging computations

In the previous section, we saw that the λ-calculus can be used to write simple functional programs. However, it is not immediately obvious that the rewriting process is well-behaved enough that we can really think of λ-terms as mathematical functions. The first thing we might worry about is whether the 'functions' we write in this language have a single value. When we look at the rule for β-reducing an application, we see that there is sometimes a choice about what to do.

Example 1.32. For example, consider the term

$$((\lambda f.\, s\lambda x.\, f(fx))\lambda y.\, y)((\lambda z.\, z)w)$$

which is an application. In this case both sides of the application are of the form $(\lambda x.\, \ldots)a$.

This term is a redex and correspondingly the first case in the definition of β-reduction applies. We have

$$(\lambda f.\, \lambda x.\, f(fx))\lambda y.\, y \xrightarrow{\beta\alpha} \lambda x.\, (\lambda y.\, y)((\lambda y.\, y)x)$$

and

$$(\lambda z.\, z)w \xrightarrow{\beta} w$$

so using the two Corollary 1.18 we have both that

$$((\lambda f.\, \lambda x.\, f(fx))\lambda y.\, y)((\lambda z.\, z)w) \xrightarrow{\beta\alpha} (\lambda x.\, (\lambda y.\, y)((\lambda y.\, y)x))((\lambda z.\, z)w)$$

and

$$((\lambda f.\, \lambda x.\, f(fx))\lambda y.\, y)((\lambda z.\, z)w) \xrightarrow{\beta\alpha} ((\lambda f.\, \lambda x.\, f(fx))\lambda y.\, y)w.$$

The fact that we can rewrite one λ-term into two different terms is at first disturbing if we want to model functions, since a function must have only one answer. However, the case above seems harmless, because we also have

$$(\lambda x.\, (\lambda y.\, y)((\lambda y.\, y)x))((\lambda z.\, z)w) \xrightarrow{\beta\alpha} (\lambda x.\, (\lambda y.\, y)((\lambda y.\, y)x))w$$

as well as

$$((\lambda f.\, \lambda x.\, f(fx))\lambda y.\, y)w \xrightarrow{\beta\alpha} (\lambda x.\, (\lambda y.\, y)((\lambda y.\, y)x))w.$$

So in this case we can bring the two possible branches back together again.

$$\begin{array}{ccc} & ((\lambda f.\, \lambda x.\, f(fx))\lambda y.\, y)((\lambda z.\, z)w) & \\ {\scriptstyle\beta\alpha}\swarrow & & \searrow{\scriptstyle\beta\alpha} \\ (\lambda x.\, (\lambda y.\, y)((\lambda y.\, y)x))((\lambda z.\, z)w) & & ((\lambda f.\, \lambda x.\, f(fx))\lambda y.\, y)w \\ {\scriptstyle\beta\alpha}\searrow & & \swarrow{\scriptstyle\beta\alpha} \\ & (\lambda x.\, (\lambda y.\, y)((\lambda y.\, y)x))w & \end{array}$$

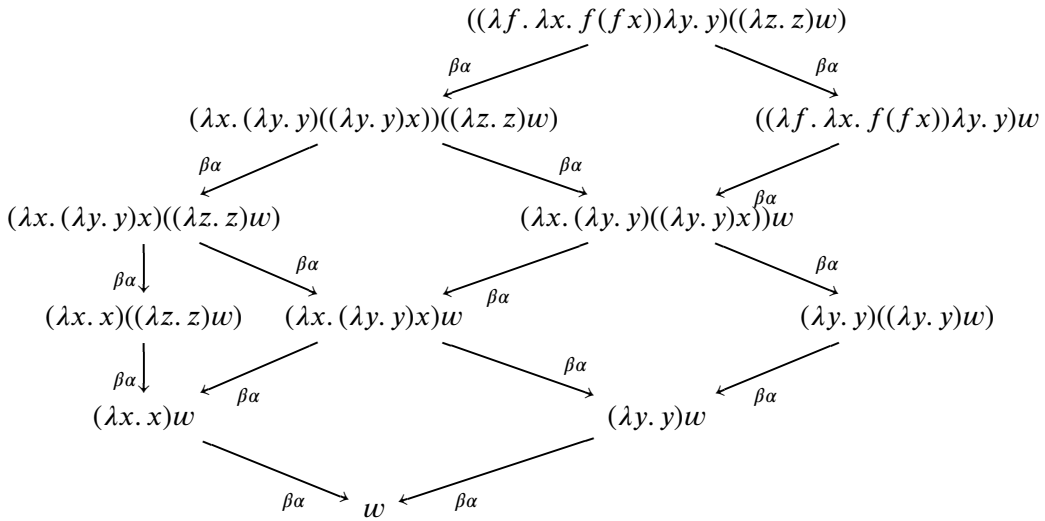
Note that since α-equivalence is reflexive, it is okay to write

$$t \xrightarrow{\beta\alpha} t' \qquad \text{or} \qquad t \xrightarrow{\beta\alpha}\!\!\!\!\rightarrow t'$$

even when we know that actually

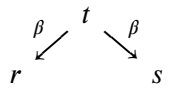
$$t \xrightarrow{\beta} t' \qquad \text{or} \qquad t \xrightarrow{\beta}\!\!\!\!\rightarrow t'.$$

We note in passing that we may look at the β-reductions (or $\beta\alpha$-reductions) that can be carried out from a given λ-term as giving us a graph, where the nodes are the λ-terms the given term can reduce to, and the edges come from one step of β-reduction (or $\beta\alpha$-reduction). The term from the previous example has further transitions, and we draw the corresponding graph on the following page.

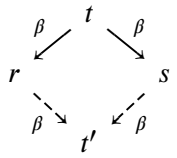

((λf. λx. f(fx))λy. y)((λz. z)w)
βα
βα
(λx. (λy. y)((λy. y)x))((λz. z)w)
((λf. λx. f(fx))λy. y)w
βα
βα
βα
(λx. (λy. y)x)((λz. z)w)
(λx. (λy. y)((λy. y)x))w
βα
βα
βα
βα
(λx. x)((λz. z)w)
(λx. (λy. y)x)w
(λy. y)((λy. y)w)
βα
βα
βα
βα
(λx. x)w
(λy. y)w
βα
βα
w

1.4.2 The diamond property

We might wonder if it is always the case that if we have



then there is a t' such that



This is an interesting property of a relation, and one often wants to know if a given relation satisfies it, especially for relations defined in computer science.

Definition 10: diamond property

Let S be a set and R be a binary relation on S. We say R has the **diamond property** if and only if for all $r, s, t \in S$, if

$$t \; R \; r \qquad \text{and} \qquad t \; R \; s$$

then

$$\text{there exists } t' \in S \qquad \text{such that} \qquad r \; R \; t' \text{ and } s \; R \; t'.$$

1.4.3 Confluence

So does β-reduction have the diamond property? The answer is no!

Example 1.33. Consider the following example. We have

$$(\lambda x.\, f\, x\, x)((\lambda y.\, y)u) \xrightarrow{\beta} (\lambda x.\, f\, x\, x)u$$

and

$$(\lambda x.\, f\, x\, x)((\lambda y.\, y)u) \xrightarrow{\beta} f((\lambda y.\, y)u)((\lambda y.\, y)u).$$

Now

$$(\lambda x.\, f\, x\, x)u \xrightarrow{\beta} f\, u\, u$$

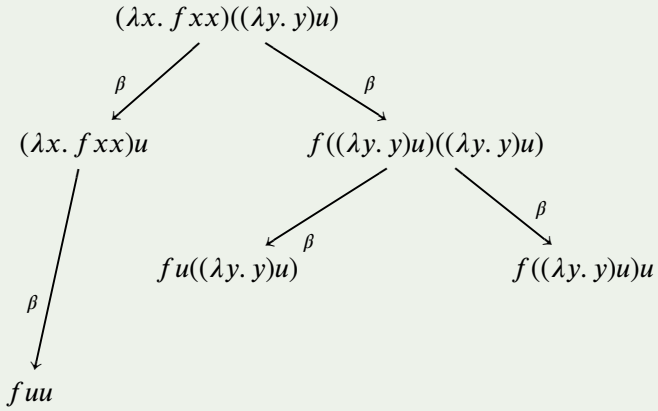
in one step, but in one step we can either do

$$f((\lambda y.\, y)u)((\lambda y.\, y)u) \xrightarrow{\beta} f\, u((\lambda y.\, y)u)$$

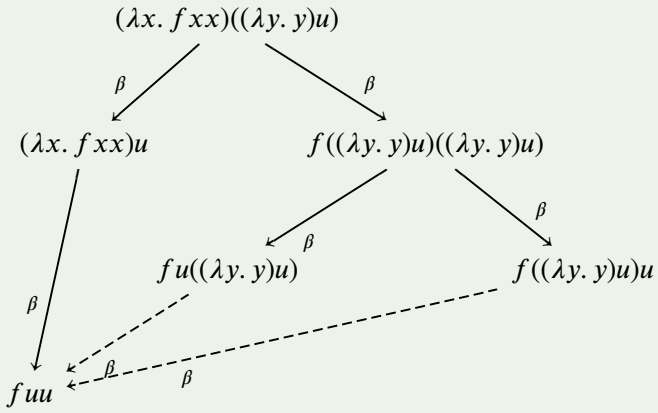
or

$$f((\lambda y.\, y)u)((\lambda y.\, y)u) \xrightarrow{\beta} f((\lambda y.\, y)u)u$$

but not both: β-reduction does not have the diamond property.



All is not lost: in this case it's clear that with just one more β-reduction we could make either of the two options above into $(f\,u)u$, so the two reductions we started with do eventually lead to the same 'final answer'.



It's clear from the previous example that a similar trick could force us to do any number of rewrites before we get to the same result from two diverging branches. So the best we can hope for is that β-reduction satisfies the following weaker property, which is still good enough for our purposes.

Note that we have to generalize here from making just one step along the relation to potentially making several.

For a binary relation R we use R^* to denote the reflexive transitive closure of R.

Definition 11: confluence

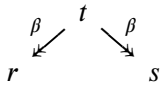
Let S be a set and R be a binary relation on S. We say R is **confluent** if and only if for all $r, s, t \in S$, if

$$t\ R^*\ r \qquad \text{and} \qquad t\ R^*\ s$$

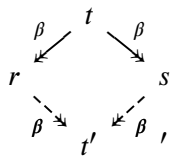
then

$$\text{there exists } t' \in S \qquad \text{such that} \qquad r\ R^*\ t' \text{ and } s\ R^*\ t'.$$

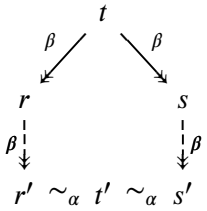
For β-reduction we may picture this property as follows: Whenever we have λ-terms t, r and s such that



then we can find t' such that



However, the situation is slightly more complicated: We only get what one might call *confluence up to α-equivalence* for β-reduction. In other words, what we are able to show is that in the situation above we can find λ-terms r', s' and t' such that the following holds.



1.4.4 The parallel reduct operation

How might we go about proving something like confluence, or rather, this version of it? One place to start would be to think about our counter-example to the diamond property. In that example, the fundamental reason why β-reduction could not bring the two derivations together in one step was this: one of the reductions we started off with copied a term which itself could be reduced, while the other reduced it before copying. The copied version could not then reduce in one step to match the other, because β-reduction can't do multiple things in parallel.

We might conjecture that this is 'the only thing that can go wrong'. The standard proof strategy which people were lead to by that conjecture was to define a new relation which is a kind of 'parallel reduction', which could reduce as many redexes in a single step as it wanted. One proves that there is a close relationship between parallel reduction steps and sequences of β-reduction steps, and then one shows that parallel reduction satisfies the diamond property.

This strategy works, but the first proofs were complicated. Over many decades, the proof was simplified until eventually it was not necessary to introduce parallel reduction at all. The crucial simplifying step was taken by Takahashi [Tak89] who defined the following operation which intuitively β-reduces 'all redexes in a term at once'.

We may think of the following as a function that takes as its input a λ-term and produced another λ-term as the output.

Definition 12: parallel reduct

The parallel reduct $\Diamond t$ of a λ-term t is defined by recursion on the structure of t as follows.

bcVar$\Diamond$ $\Diamond x = x$ if $x \in \mathsf{Vars}$

scAbs$\Diamond$ $\Diamond(\lambda x.\, t) = \lambda x.\, \Diamond t$

$$\mathbf{scApp}\Diamond\ \Diamond(tu) = \begin{cases} \Diamond t'[\Diamond u/x] & \text{if } t \text{ is of the form } \lambda x.\, t' \\ \Diamond t\, \Diamond u & \text{otherwise} \end{cases}$$

We write $\Diamond^n t$ to mean the result of applying the parallel reduct operation to t n times.

We may think of this operation as carrying out a number of β-reduction steps to arrive at its result.

It may be tempting to think of this operation as carrying out more reduction steps than it actually does. It is important to note while it addresses all the redexes in the given term, it does not necessarily carry out all possible reduction steps for any one subterm.

We give an example of how to calculate the parallel reduct of a term.

Example 1.34. We give an example of calculating the parallel reduct of a term. The reader might want to try this for a more interesting term to see the effect this operation can have.

$$\begin{aligned} \Diamond((\lambda x.\, x)y\lambda x.\, x) &= \Diamond((\lambda x.\, x)y)\,\Diamond(\lambda x.\, x) && \text{scApp}\,\Diamond \\ &= \Diamond x[\Diamond y/x]\lambda x.\ \Diamond x && \text{scApp}\Diamond,\ \text{scAbs}\Diamond \\ &= x[y/x]\lambda x.\, x && \text{bcVar}\,\Diamond \\ &= y\lambda x.\, x && \text{bcVar}[\,] \end{aligned}$$

1.4.5 Properties of the parallel reduct operation

We now prove that the parallel reduct is relevant to β-reduction, and has nice properties.

Tip

This might be your first time looking at a theorem whose proof has to be worked up to by proving lots of little results first. You might be happy reading this in the order it is written, digesting the proofs of the basic properties before moving on to see how they are used. But for most people it's easier to work *backwards*: a first glance over all the statements of the propositions will leave you wondering what the point of some of them is. It can be more pleasant to accept that they are true, and see what role they play in the property you care about. Once you see what they're good for, it's easier to go back and check that you really believe they're true!

The definition of parallel reduct involves a recursive step which carries out the translation inside a capture-avoiding substitution. In order to compare this operation to β-reduction, we need to make sure that the parallel reduct operation interacts well with the notion of free variables and capture avoiding substitution.

Now we are ready to study the parallel reduct. After checking that it behaves well with regard to the notion of free variables, we show that it has various properties when it interacts with $\xrightarrow{\beta}$ and $\xrightarrow[\beta]{}$. In general, we prefer to prove a property involving $\xrightarrow[\beta]{}$, but is is often easier at first to consider a single β-reduction step.

Proposition 1.20

For all λ-terms t we have $\operatorname{fv} t \supseteq \operatorname{fv} \Diamond t$.

Proof. *This is a straightforward proof by induction.*

See page 273.

Example 1.35. We note that the inclusion in the previous proposition may indeed be a strict one by giving an example.

$$\begin{aligned}
\operatorname{fv}(\Diamond(\lambda x.\, y)z) &= \operatorname{fv} \Diamond y[\Diamond z/x] && \text{scApp}\Diamond \\
&= \operatorname{fv} y[z/x] && \text{bcVar}\Diamond \\
&= \operatorname{fv} y && \text{bcVar}[\,] \\
&= \{y\} && \text{bcVarfv} \\
&\subsetneq \{y\} \cup \{z\} \\
&= \operatorname{fv}(\lambda x.\, y) \cup \operatorname{fv} z && \text{scAbsfv, vcVarfv} \\
&\operatorname{fv}((\lambda x.\, y)z) && \text{scAppfv}
\end{aligned}$$

In order to work with the parallel reduct we have to show that it is well-behaved with respect to α-equivalence, and in order to get that we need to study how it works when it comes to renaming.

Proposition 1.21

Let t be a λ-term and $z \in \mathsf{Vars} \setminus \operatorname{vars} t$. Then for every variable x we have $\Diamond \operatorname{ren}_x^z t \sim_\alpha \operatorname{ren}_x^z \Diamond t$.

Proof. See page 233.

We are able to argue about confluence only up to α-equivalence as explained above, and we have to make sure that the parallel reduct fits well with the point of view that we don't worry about the naming of bound variables.

Proposition 1.22

If t and t' are α-equivalent λ-terms then so are $\Diamond t$ and $\Diamond t'$.

Proof. See page 234.

1.4.6 A proof of confluence

We show that parallel reduct takes us further down a potential line of β-reductions, in other words, up to α-equivalence we can always β-reduce a term to its parallel reduct which helps to explain the name.

Proposition 1.23

For all λ-terms t there is a term u with $t \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} u \sim_\alpha \Diamond t$.

Proof. Proving this result is an assessed coursework exercise.

The two previous results together give us for α-equivalent terms t and u that we may find terms t' and u' with

$$\begin{array}{ccccccc} & & t & \sim_\alpha & u & & \\ & \swarrow^{\beta} & & & & \searrow^{\beta} & \\ t' & \sim_\alpha & \Diamond t & \sim_\alpha & \Diamond u & \sim_\alpha & u' \end{array}$$

We may immediately extend the previous result to an iteration of the parallel reduct.

Corollary 1.24

Let t be a λ-term. For $n \in \mathbb{N}$ then there is a term u with $t \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} u \sim_\alpha \Diamond^n t$.

In particular for $m \leq n$ in $\mathbb{N}$ there is a term u with $\Diamond^m t \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} u \sim_\alpha \Diamond^n t$.

Proof. *The proof required is quite short, but requires understanding how reduction interacts with α-equivalence.*

See page 274.

Next we show that, up to α-equivalence, parallel reduct is monotone with respect to $\xrightarrow{\beta}{\!\!\!\!\!\rightarrow}$.

Proposition 1.25

If t and a are λ-terms and x is a variable then there exists a λ-term t' with $\Diamond((\lambda x.\, t)a) \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} t' \sim_\alpha \Diamond(t[a/x])$.

Proof. See page 235.

Lemma 1.26

If t and t' are λ-terms with $t \xrightarrow{\beta}{\rightarrow} t'$ then there is a λ-term u with $\Diamond t \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} u \sim_\alpha \Diamond t'$.

Proof. See page 237.

Corollary 1.27

If t and t' are λ-terms with $t \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} t'$ then there is a λ-term u with $\Diamond t \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} u \sim_\alpha \Diamond t'$.

Proof. *Students might want to try this proof, in which case they will find Corollary 1.17 useful.*

See page 274.

Corollary 1.28

If t and t' are λ-terms such that if $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ t'$ then for all $n \in \mathbb{N}$ there is a λ-term u with $\Diamond^n t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ u \sim_\alpha \Diamond^n t'$.

Proof. See page 238.

Finally we show that any sequence of β-reductions can always be extended so that it ends, up to α-equivalence, at the iterated parallel reduct of its starting term as indicated by the diagram below.

$$\begin{array}{ccc} t & \xrightarrow{\beta} & t' \\ {\scriptstyle \beta\alpha}\downarrow & & \vdots\,{\scriptstyle \beta} \\ \Diamond t & \sim_\alpha & u \end{array}$$

Proposition 1.29

If t and t' are λ-terms with $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ t'$ then there is a λ-term u with $t' \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ u \sim_\alpha \Diamond t$.

Proof. We prove this statement by induction over the term t which allows us to trace why t reduces to t'. The induction hypothesis is that the claim holds for proper subterms.

bcVarβ If t is of the form $(\lambda x.\, u)a$ and $t' = u[a/x]$ then $\Diamond t = \Diamond u[\Diamond a/x]$, and so we want to show that $u[a/x]$ reduces to a term that is α-equivalent to $\Diamond u[\Diamond a/x]$. We know by Corollary 1.27 that there exist terms u' and a' with

$$u \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ u' \sim_\alpha \Diamond u \qquad \text{and} \qquad a \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ a' \sim_\alpha \Diamond a,$$

and by Proposition 1.11 that

$$u'[a'/x] \sim_\alpha \Diamond u[\Diamond a/x].$$

By Proposition 1.19 we know that we may find a term s such that

$$t' = u[a/x] \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ s \sim_\alpha u'[a'/x] \sim_\alpha \Diamond u[\Diamond a/x] = \Diamond t,$$

so s provides the required witness.

scAbsβ Assume that $t = \lambda x.\, r$ and $t' = \lambda x.\, r'$ where $r \xrightarrow{\beta}\ r'$. We have $\Diamond(\lambda x.\, r) = \lambda x.\, (\Diamond r)$ by scAbs$\Diamond$. By the induction hypothesis, we know $r' \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ r'' \sim_\alpha \Diamond r$. By Proposition 1.19 we have $\lambda x.\, r' \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ \lambda x.\, r''$, and by Proposition 1.7 we obtain $\lambda x.\, r'' \sim_\alpha \lambda x.\ \Diamond r$ and so altogether we get

$$t' = \lambda x.\, r' \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ \lambda x.\, r'' \sim_\alpha \lambda x.\ \Diamond r = \Diamond t$$

and xr'' is the require witness.

scAppβ If $t = rs$ and $t' = r's$, with $r \xrightarrow{\beta} r'$, then $\Diamond t = \Diamond r \, \Diamond s$. By the induction hypothesis we can find a term r'' with $r' \xrightarrow{\beta}\!\!\!\!\rightarrow r'' \sim_\alpha \Diamond r$, and we know from Proposition 1.23 that we can find a term s' with $s \xrightarrow{\beta}\!\!\!\!\rightarrow s' \sim_\alpha \Diamond s$. Now Proposition 1.19 tells us that $r's \xrightarrow{\beta}\!\!\!\!\rightarrow r''s'$ and from scAppα we obtain $r''s' \sim_\alpha \Diamond r \, \Diamond s$. Overall this means we have

$$t' = r's \xrightarrow{\beta}\!\!\!\!\rightarrow r''s' \sim_\alpha \Diamond r \Diamond s = \Diamond t,$$

so $r''s'$ is the required witness.

The case where instead of reducing r we reduce s is much the same.

Theorem 1.30

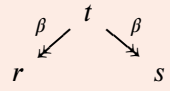
If t and t' are λ-terms with $t \xrightarrow{\beta}\!\!\!\!\rightarrow t'$ then there is $n \in \mathbb{N}$ and a λ-term u with $t' \xrightarrow{\beta}\!\!\!\!\rightarrow u \sim_\alpha \Diamond^n t$.

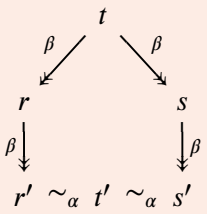
Proof. See page 239.

Using this theorem we are able to show that β-reduction is confluent up to α-equivalence.

Theorem 1.31

Assume that t, u and u' are λ-terms.

If 

then there are r', s' and t' with 

Proof. *Students may want to see whether they can establish the confluence property now. The key ingredients are Theorem 1.30, Corollary 1.24, and Proposition 1.16.*

See page 275.

1.5 Non-termination

The previous section reassures us that our mental model of λ-terms as functions works in the sense that a calculation can not have two fundamentally different answers: if a single term β-reduces to two different terms, then there is always another term which both β-reduce to up to α-equivalence.

If we think about what else 'might go wrong' we have to worry about the perennial problem of non-termination. Does a calculation always have to yield some 'final answer', or are there terms which we can keep β-reducing forever?

To answer this question, let's first attend to another worry the reader might have about programming in the λ-calculus. Above, we encode various kinds of data as λ-terms, but this encoding is fragile in the sense that nothing stops us from taking a function written assuming it will be applied to correctly encoded data, and applying it to an arbitrary λ-term. We might wonder what other bad things we are not prevented from doing! Consider the λ-term

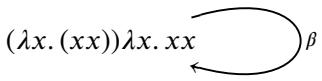
$$\lambda x.\, xx$$

This term clearly satisfies the definition of a λ-term, but what would it mean as a function? We have x applied to itself, and set theory tells us that this does not make sense. The reason is that a mathematical function always comes with a source and a target, and it can never be an element of its own source, so can never be applied to itself. But the λ-calculus is just a syntactic rewriting system: it doesn't care what we imagine!

So λ-terms come with a weird operation not available for functions: they can be applied to themselves. Having realized we have this operation of self-application we want to try it out, and what better experiment can we perform on the self-application operator than to *apply it to itself*? Doing that, we find

$$\begin{aligned}(\lambda x.\,(xx))\lambda x.\, xx &\\ &\xrightarrow{\beta} xx[\lambda x.\, xx/x]\\ &= (\lambda x.\,(xx))\lambda x.\, xx\end{aligned}$$

We may picture this situation as follows:



So we have an example of a term which β-reduces to itself. This means that someone who naïvely keeps β-reducing a λ-term until they run out of β-reductions to perform would keep going forever!

This might not seem very serious, because if this were the extent of the problem, we could just detect when a λ-term reduces to itself, and declare it to be the 'final answer' of the calculation. But life is not so simple. Consider the term

$$\lambda f.\,(\lambda x.\, f(xx))(\lambda x.\, f(xx))$$

If we apply this to a term g we find

$$\begin{aligned}(\lambda f.\,(\lambda x.\, f(xx))\lambda x.\, f(xx))g &\xrightarrow{\beta\alpha} (\lambda x.\, g(xx))\lambda x.\, g(xx)\\ &\xrightarrow{\beta\alpha} g(\lambda x.\, g(xx))\lambda x.\, g(xx)\\ &\xrightarrow{\beta\alpha} g(g(\lambda x.\, g(xx))\lambda x.\, g(xx))\\ &\vdots\end{aligned}$$

which means we don't just get a term which β-reduces to itself, we get one which β-reduces to something *more complicated than itself*! You can imagine how complicated the situation would become if instead of a free variable g, we had put a complicated λ-term, perhaps one which encodes a complicated calculation.

You can imagine how mysterious this was when it was first discovered. Indeed, it meant that Church's original idea for what the λ-calculus would be good for

didn't work. He had hoped to use it as a foundation for mathematics, but the self-referential nature of the term above gave rise to paradoxes.

As modern computer scientists, we are used to thinking of the meaning of a program as *partial* function. Indeed, we know that a language in which all programs halt can't encode every program: if it could, the Halting Problem would be trivial, instead of uncomputable.

But that doesn't completely solve this problem: it leads us to think of β-reduction of λ-terms as a low-level model of computation, like Turing machines or the operational semantics of the While language. We started studying the λ-calculus because of its origins in modelling functions—we were hoping to ignore low-level details and reason about computations as if they were ordinary mathematical functions, without having to actually work through long chains of β-reductions. But self-application just isn't something we can do with functions, so we have to improve our syntactic model. We said above that the reason applying a function to itself is meaningless in maths is that functions can only be applied to inputs from their source, so we need a way of specifying what kind of data a variable can be, or that a function accepts, and only allow applications where these match up—we need a type system! We develop such a system for the λ-calculus in the following chapter.

Chapter 2

The Simply Typed λ-Calculus

2.1 Introduction

Let's take stock of the lessons of the previous chapter. The λ-calculus is a formalism inspired by how we evaluate functions, and our perspective is to consider it a candidate for a programming language whose programs can be reasoned about as functions.

However it contains a paradoxical 'self-application' term which breaks this intuition. If we look at what we have modelled we can see that this is only part of what it is to be a function, namely the 'rule' for evaluating the function by substitution. What is missed by this approach is the fact that a function also comes with a source and target, and that a function application only makes sense if the input is suitable for that function—in other words, the input has to be an element of the source of the function.

The same problem presented itself to Church who devised the λ-calculus to investigate issues of mathematical logic. He too realized that the solution lay in modelling the sources and targets of functions. Following earlier work by Russel and Whitehead which resolved similar paradoxes in set theory, Church called this extra information the *type* of a function, and this is how typed programming languages came to be.

In this chapter we define a typed system known as the **simply typed λ-calculus**.

2.2 Types and Correctly Typed Terms

Adding types to the λ-calculus turns out to require a rather more complicated apparatus than one might think, in particular given that the idea of what we are trying to do seems quite simple. The definition proceeds in three stages:

- We have to say what the types of the system are. This part is straightforward and is covered in Section 2.2.1.
- We have to adjust the notion of term to allow for some typing decorations to appear. We then have to look at the machinery from the previous chapter to see what adjustments are required to deal with such decorated terms. We call these terms 'preterms', because ultimately we are only interested in terms that can be typed. This part is also quite straightforward and covered in Section 2.2.2.

- We have to build the machinery that ensures that the typing mechanism has the desired effect. It turns out that the machinery required is fairly elaborate. This material is covered in Sections 2.2.3 and 2.2.4, with first properties appearing in Section 2.3.1

2.2.1 Types

In order to add typing information to the λ-calculus to help with the problems identified above we need to settle on a collection of types. Below we use Greek letters, typically σ and τ, as variables ranging over types to ensure that types are easily distinguished from terms. We know that given two types σ and τ, we will want to form the type of 'functions' from σ to τ, which we will denote as

$$\sigma \to \tau.$$

But what types should we have to begin with, from which we may then construct function types? In a real programming language, there would be built-in types like booleans, various kinds of numbers, or strings. But for the time being we want to keep out model as simple as possible. For that reason, we will have just one basic type, which we will call ι, and we *will not specify* what the elements of this type are! That might seem strange, but it ensures that we do not build any assumptions into our system. For the time being we focus on the function types. One we've seen how these work, it will not be tricky to plug in more interesting notions of data.

Definition 13: types for the simply typed λ-calculus

The set of **types of the simply typed λ-calculus**, $\mathsf{Type}^{\to}$, is defined by recursion as follows

bc$\to$ $\iota \in \mathsf{Type}^{\to}$

sc$\to$ given $\sigma, \tau \in \mathsf{Type}^{\to}$ there is a type $\sigma \to \tau \in \mathsf{Type}^{\to}$.

Note that in order to refer to types unambiguously we have to put some brackets into the corresponding expression. Once again in order to make such expressions easier to read we adopt a convention, namely that by default, we carry out the $\to$ operation *from right to left*. As a consequence, when we write

$$\sigma \to \sigma \to \sigma$$

we are referring to the type whose fully bracketed name

$$\sigma \to (\sigma \to \sigma),$$

and if we want to refer to the other type that contains three copies of σ we have to put in the brackets explicitly and name it as

$$(\sigma \to \sigma) \to \sigma.$$

We might sometimes use brackets not strictly needed if we think it improves readability.

Types that are not base types are sometimes referred to as **higher types**.

It's important to realize that, as with all recursive definitions of syntactic entities, two types can only be equal if they are syntactically equal. While this is readily accepted for items we think of as syntactic terms, when we are concerned with entities that look as if they might be the interpretation of something it is not uncommon to get this wrong.

2.2.2 Typing decorations

Now we can move on thinking about when a term has a certain type.

Note that there are a couple of subtleties to consider. First, a given term such as $\lambda y.\, y$ can have many types. Intuitively, this could be the 'identity term' of type $\iota \to \iota$, but it would also make sense to apply it to other terms, for instance $(\lambda y.\, y)\lambda f.\, \lambda x.\, f\, x$ makes sense, and the simplest interpretation of $\lambda f.\, \lambda x.\, f\, x$ is that f should be a function of type $\iota \to \iota$, which would mean $\lambda f.\, \lambda x.\, f\, x$ would have type $(\iota \to \iota) \to \iota \to \iota$. So it should also be possible to type $\lambda y.\, y$ as having type $((\iota \to \iota) \to \iota \to \iota) \to ((\iota \to \iota) \to \iota \to \iota)$ too. Indeed, surely it should be possible to give $\lambda y.\, y$ the type $\sigma \to \sigma$ for *every* type σ.

Our motivation for introducing types is to incorporate the idea that terms should only be applied to suitable inputs, and in order to ensure that we cannot have one term inhabit several types. We want our system to only allow us to have such a term provided that all applications respect the idea of providing suitable inputs, and that means we have to incorporate some of this information into the term itself. To solve this problem, we add an annotation to the argument of a λ-abstraction, indicating the type we claim it should have, writing for instance $\lambda y : (\iota \to \iota).\, y$ for the 'identity term' which we expect to have type

$$(\iota \to \iota) \to (\iota \to \iota).$$

For these annotated terms we can transfer notions such as β-reduction by simply ignoring the annotations—we give more information about this transferral below.

Accordingly we change our notion of what terms we allow to include information about types when we perform a λ-abstraction. Note that many of the terms so defined cannot be assigned a type in the system we introduce below, and you may think of them as 'non-sensical' terms, which is why we call them **preterms**. We introduce these preterms by recursion:

bcPVar For $x \in \mathsf{Vars}$ we have $x \in \Lambda\mathsf{PTrm}$

scPAbs For $x \in \mathsf{Vars}$, $\sigma \in \mathsf{Type}^{\to}$ and $t \in \Lambda\mathsf{PTrm}$ we have $\lambda x : \sigma.\, t \in \Lambda\mathsf{PTrm}$.

scPApp For $t, t' \in \Lambda\mathsf{PTrm}$ we have $tt' \in \Lambda\mathsf{PTrm}$.

Before we worry about how we are going to assign types to (pre)terms we have to revisit some of the notions from the previous chapter. Most of the ideas presented there can be easily adjusted to type-annotated terms. For example, defining subterms, free and bound variables, or renaming variables, can be done in much the same way as before. However, when we define α-equivalence, we have to make a change when both terms are abstractions.

Definition 14: α-equivalence for preterms

Two preterms are **α-equivalent** if and only if one of the following cases applies. We use $\sim_\alpha$ for this relation.

bcPVarα Both terms are variables, and they are identical.

scPAbsα $\lambda x{:}\sigma.\, t \sim_\alpha \lambda x'{:}\sigma'.\, t'$ if and only if

$$\sigma = \sigma'$$

and for $w = \mathrm{freshv}(\mathrm{vars}(\lambda x.\, t) \cup \mathrm{vars}(\lambda x'.\, t'))$ we have

$$\mathrm{ren}^w_x\, t \sim_\alpha \mathrm{ren}^w_{x'}\, t'.$$

scPAppα $tt' \sim_\alpha uu'$ if and only if $t \sim_\alpha u$ and $t' \sim_\alpha u'$.

We can then easily adjust the notion of capture-avoiding substitution and β-reduction, noting that we need to adjust the base case of the latter to

bcPVarβ $(\lambda x{:}\sigma.\, t)a \xrightarrow{\beta} t[a/x]$,

while the other cases can stay the same.

The various results from Chapter 1 all hold for this adjusted system. You may want to leaf through that chapter to convince yourself that there is nothing to worry about in that respect.

2.2.3 Type environments

Example 2.1. To see the second subtlety mentioned in the discussion above, let's look more closely at our argument that it should be possible for

$$\lambda f.\, \lambda x.\, f\, x$$

to have the type

$$(\iota \to \iota) \to \iota \to \iota.$$

We can now record the assumptions we made about the input variables explicitly, writing the term as

$$\lambda f{:}(\iota \to \iota).\, \lambda x{:}\iota.\, f\, x.$$

Now we can see from these annotations that the type of this term must be of the form

$$(\iota \to \iota) \to \iota \to ?$$

where we have to work out the type of the output which should replace the '?'. We do this by looking at the inner subterm $f\, x$ in which f and x are free variables. We tell ourselves 'assuming that f has type $\iota \to \iota$ and x has type ι, the type of $f\, x$ should be ι'.

The preceding example shows that we need to be able to assign types to subterms with free variables to build up the type of a larger expression, and when we do this we need to remember the assumptions we should make about the

types of those free variables given how they are bound by λ-abstractions in the surrounding typing information. We call a set of assumptions[1] about the types of variables a *type environment*; read on for a formal definition.

However, this business of making assumptions can become complicated in the presence of shadowing which is discussed in Example 1.11.

Example 2.2. Consider the term $\lambda x\colon(\iota \to \iota).\ \lambda y\colon\iota.\ (\lambda x\colon\iota.\ x)(xy)$ in which the inner term $\lambda x\colon\iota.\ x$ shadows the variable x which is already used by the outer λ-abstraction. We might work our way through typing this term (with an extra pair of brackets for clarity) as follows:

$\lambda x\colon(\iota \to \iota).$	assume x has type $\iota \to \iota$
$(\lambda y\colon\iota.$	assume y has type ι
$(\lambda x\colon\iota.$	new assumption: x has type ι
x	free variable x has type ι so $\lambda x\colon\iota.\ x$ has type $\iota \to \iota$
$)$	exit local environment: forget new assumption about x
(xy)	by old assumption $x\colon\iota \to \iota$ and $y\colon\iota$ so xy has type ι
$)$	since $(\lambda x\colon\iota.\ x)\colon\iota \to \iota$ and $xy\colon\iota$ deduce $((\lambda x\colon\iota.\ x)(xy))\colon\iota$
	so $(\lambda y\colon\iota.\ (\lambda x\colon\iota.\ x)(xy))$ has type $\iota \to \iota$
	so $(\lambda x\colon(\iota \to \iota).\ \lambda y\colon\iota.\ (\lambda x\colon\iota.\ x)(xy))$ has type $(\iota \to \iota) \to (\iota \to \iota)$

From this example we conclude that to deal with assumptions properly we need some sort of stack to keep track of the most recent assumption about a particular variable, and so that we can go back to the previous assumption once we exit the local environment which caused the current assumption.

Now, most notes on the simply typed λ-calculus try to sweep this under the carpet. One common strategy is to only consider terms which do not involve shadowing. But then the type system would not be invariant under α-equivalence: there would be typeable terms which would be α-equivalent to untypeable terms. One of the most important correctness criteria of a type system is that this does *not* happen. One way to fix this would be to add a rule to the type system which simply asserts that α-equivalent terms have the same type. But this would significantly complicate the study of other the correctness properties proved below. An alternative is to assign each variable a type, and only consider terms in which variables are used with their assigned types, but this makes the system bizarre from a programming language point of view: no modern, real-world programming languages take this approach. In these notes, we present the full version of a system which allows shadowing while making sure that our examples can be typed without the extra complication.

We need notation for type environment and the accepted practice is to list the typing assumptions. However this is a notion where equality is *not* given by syntactic equality, see below.

Definition 15: type environment

A **type environment for the simply typed λ-calculus** is given by the following recursive definition.

[1] Some authors use 'context', but we reserve that terminology for something quite different.

bcTE There is an empty type environment.

scTE If Γ is a type environment, $x \in \mathsf{Vars}$ and $\tau \in \mathsf{Type}^{\rightarrow}$ then $\Gamma, x:\tau$ is a type environment.

This definition is quite short and not hard to understand, so the reader may wonder why we have taken up so much space in explaining it. The answer is that while type environments are easy to define, the question of when two type environments should be considered equal is more complicated due to the idea that we should be able to change the order of some of the assumptions.

In Example 2.2 it is clear that when we have several typing assumptions for the same variable we need them to be supplied in the correct order. So we want our equality of type environments to follow these principles:

- Tying assumptions for distinct variables may be swapped without changing the type environment.
- Swapping typing assumptions for the same variable changes the type environment.

We define equality of type environments making use of machinery we have used before. We first define an auxiliary relation.

For distinct variables x and y, and type environments Γ and Δ we define

$$\Gamma, x:\rho, y:\sigma, \Delta \approx \Gamma, y:\sigma, x:\rho, \Delta.$$

Note that this relation is symmetric, so the equality relation defined in the following is an equivalence relation.

Definition 16: equality of type environments

Equality of type environments is given by the reflexive transitive closure of the relation $\approx$.

We give an example to illustrate the idea of this equality.

Example 2.3. Assume we have

$$\Gamma = x:\alpha, y:\sigma, x:\sigma, y:\tau.$$

Then Γ is equal to the following type environments where arrows indicate the swapping of two assumptions.

$$\begin{array}{ccccc}
x:\alpha, x:\sigma, y:\beta, y:\tau & & & & \\
\updownarrow & & & & \\
x:\alpha, y:\sigma, x:\sigma, y:\tau & \longleftrightarrow & x:\alpha, y:\sigma, y:\tau, x:\sigma & & \\
\updownarrow & & \updownarrow & & \\
y:\sigma, x:\alpha, x:\sigma, y:\tau & \longleftrightarrow & y:\sigma, x:\alpha, y:\sigma, x:\sigma & & \\
 & & \updownarrow & & \\
 & & y:\sigma, y:\sigma, x:\alpha, x:\sigma & &
\end{array}$$

For readers who would like to have an alternative characterization of equality of type environments the following exercise provides one.

Exercise 12. Show that two type environments Γ and Δ are equal if and only if

- they contain the same typing assumptions and
- for every variable $x \in \mathsf{Vars}$, the typing assumptions about x appear in the same order in Γ and Δ.

A solution appears on page 276.

Note that there is an obvious operation for type environments: If Γ and Δ are type environments then Γ, Δ is a type environment.

Moreover, this operation is well behaved with respect to equality of type environments.

Proposition 2.1

If Γ, Γ', Δ and Δ' are type environments then

$$\Gamma = \Gamma' \text{ and } \Delta = \Delta' \qquad \text{implies} \qquad \Gamma, \Delta = \Gamma', \Delta'.$$

Proof. See page 239.

We further find the following useful when it comes to manipulating type environments.

Proposition 2.2

Let Γ, Γ' and be type environments, x a variable and σ a type.

(a) There is a type environment Δ such that $\Gamma, \Gamma' = \Delta, x : \sigma$ if and only if

- There is a type environment Γ'' with $\Gamma' = \Gamma'', x : \sigma$ or
- there is a type environment Γ'' with $\Gamma = \Gamma'', x : \sigma$ and $x : \sigma, \Gamma' = \Gamma', x : \sigma$.

(b) If y is a variable distinct from x and ρ is a type such that $\Delta, y : \rho = \Gamma, x : \sigma$ then there exists a type environment Δ' with $\Delta = \Delta', x : \sigma$.

Proof. See page 239.

2.2.4 Typing rules

Now we are finally in a position to give a formal definition of the type of a term. You may find this reminiscent of the natural deduction system for classical logic taught on COMP11120, and indeed there are quite deep reasons for this resemblance. We may think of the types as being analogous to logical formulae (and the terms give additional infrastructure that is new here).

We will use the notation $\Gamma \vdash t : \tau$ to mean that if we make all the assumptions in Γ then the term t is given the type τ.

We recall some terminology from COMP11120: A statement of the form

$$\Gamma \vdash t : \tau \qquad \text{is known as a } \textbf{judgement}.$$

We may think of the statements in Γ as assumptions.

We define this typing relation as an inference system of formal logic. Recall that in an inference rule is of the form

$$\frac{\Gamma_1 \vdash t_1 : \tau_1 \quad \Gamma_2 \vdash t_2 : \tau_2 \quad \dots \quad \Gamma_n \vdash t_n : \tau_n}{\Gamma \vdash t : \tau}$$

where the judgements above the line are the **premises** and the statement below the line is the **conclusion**. In our system we have none to two premises for each rule.

Definition 17: typing rules for the simply typed λ-calculus

The typing rules for the simply typed λ-calculus are as follows.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{ax}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma . t) : \sigma \to \tau}\ \text{abs} \qquad \frac{\Gamma \vdash f : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash f t : \tau}\ \text{app}$$

The axiom rule makes it look as if we are only able to extract a statement giving a type for the variable whose type declaration is the final one that appears in Γ. However, remember we are allowed to change the order of the typing assumptions in Γ based on the definition of equality of type environments, see Definition 16..

It is important to note that in contrast with the previous paragraph we are not allowed to change the order in which declarations for the same variable appear.

Alternatively we could have formulated the axiom rule as follows: We have

$$\frac{}{\Gamma \vdash x : \tau}\ \text{ax}$$

if and only if there is a type environment Δ with $\Gamma = \Delta, x : \tau$.

Example 2.4. The two warnings above are maybe best underlined by an example, and we revisit the type environment from Example 2.3.

Assume we have

$$\Gamma = x : \alpha, y : \sigma, x : \sigma, y : \tau.$$

Then we may move declarations for x past y, but it always has to be the case that $x : \alpha$ occurs before $x : \sigma$, and $y : \sigma$ must occur before $y : \tau$. In particular this has the consequence that the only valid usages of the axiom rule for Γ are

$$\frac{}{\Gamma \vdash x : \sigma}\ \text{ax} \quad \text{and} \quad \frac{}{\Gamma \vdash y : \tau}\ \text{ax}.$$

If we look at the rules one by one we may see that every rule is reminiscent of a rule for a natural deduction system.

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\,\text{ax}$$

This rule is reminiscent of the axiom rule for natural deduction, and it tells us that if x being of type τ is one of our assumptions, then that statement is entailed by the assumptions.

$$\frac{\Gamma, x:\sigma \vdash t:\tau}{\Gamma \vdash (\lambda x:\sigma.\, t):\sigma \to \tau}\,\text{abs}$$

The abstraction rule tells us that if the assumptions include x being a variable of type σ, and allow us to conclude that t is a term of type τ then under those assumptions, sans the one about x being of type σ, we may obtain a term of type $\sigma \to \tau$ by forming the λ-abstraction consisting of prefixing t by λ and the variable x. This fits well with our motivation of modelling something similar to substitutions for functions: When we carry out β reduction for such a term instances of the variable x will be replaced by capture avoiding substitution, and so we may think of such a term as modelling a function from type σ to type τ.

$$\frac{\Gamma \vdash f:\sigma \to \tau \quad \Gamma \vdash t:\sigma}{\Gamma \vdash ft:\tau}\,\text{app}$$

The application rule is possibly the most intuitive one: If our assumptions allow us to determine that f is of the type $\sigma \to \tau$, and also that t is a suitable input, that is, of type σ, then the application of f to t is of type τ.

So in an intuitive sense we may see that the typing rules we give incorporate the information about each function coming with a source and a target, and that we may only apply a function to an input of the appropriate kind.

In COMP11120 all the rules for connectives in the natural deduction system are split into *introduction* and *elimination* rules. We may think of the introduction rules as telling us how to build a formula that contains a particular connective, and the elimination rules tell us how we may 'get rid' of a connective. One might want to name the two non-axiom rules above similarly. We emphasize this idea by *removing the terms* from the rule:

$$\frac{}{\Gamma, \tau \vdash \tau}\,\text{ax}$$

This rule now looks like the axiom rule for natural deduction.

$$\frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau}\,\text{abs}$$

This rule now looks like the natural deduction introduction rule for implication.

$$\frac{\Gamma \vdash \sigma \to \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau}\,\text{app}$$

This rule now looks like the natural deduction elimination rule for implication.

We do not have space here to go into detail about the connection between this system and a natural deduction system, but there is much more that could be said.

Example 2.5. We check that the term $\lambda x:\iota.\, \lambda y:\iota \to \iota.\, yx$ has type

$$\iota \to ((\iota \to \iota) \to \iota)$$

by giving a derivation in our system.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\iota, y:\iota\to\iota \vdash y:\iota\to\iota}\,\text{ax} \quad \dfrac{}{x:\iota, y:\iota\to\iota \vdash x:\iota}\,\text{ax}}{x:\iota, y:\iota\to\iota \vdash yx:\iota}\,\text{app}}{x:\iota \vdash (\lambda y{:}\iota\to\iota.\, yx):((\iota\to\iota)\to\iota)}\,\text{abs}}{\vdash (\lambda x{:}\iota.\,\lambda y{:}\iota\to\iota.\, yx):\iota\to((\iota\to\iota)\to\iota)}\,\text{abs}$$

Example 2.6. We go through another example for those readers who are interested in shadowing. The motivating example above is

$$\lambda x{:}(\iota\to\iota).\,\lambda y{:}\iota.\,(\lambda x{:}\iota.\,x)(xy).$$

We give a derivation of a type for this term overleaf. Note how the derivation mirrors our thoughts from Example 2.2 left-most axiom rule which assigns two different types to the variable x.

Example 2.7. Let us consider the encoding of the boolean values 'true' and 'false' from 37 and RExercise 9. The untyped term which we used to represent true is (up to α-equivalence) $\lambda x.\,\lambda y.\,x$. The simplest typed term corresponding to this is the one where we assign the variables the type ι. Then we have $\lambda x{:}\iota.\,\lambda y{:}\iota.\,x$. The type of this term (without needing any typing assumptions, since there are no free variables) is $\iota\to\iota\to\iota$.

RExercise 13. Give a typing derivation to show that

$$\vdash (\lambda x{:}\iota.\,\lambda y{:}\iota.\,x){:}\iota\to\iota\to\iota.$$

See page 276.

Example 2.8. Consider the negation function on booleans (see Example 2.7), which we can encode as the term $\lambda b{:}\iota\to\iota\to\iota.\,\lambda x{:}\iota.\,\lambda y{:}\iota.\,byx$. Note that the type annotated term is easier to read than it would be without the type annotations, as they help us to see which argument is which. We show that this term has type $(\iota\to\iota\to\iota)\to(\iota\to\iota\to\iota)$ overleaf.

Derivation for Example 2.6

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x:(\iota\to\iota),y:\iota,x:\iota\vdash x:\iota}\,\text{ax}}{x:(\iota\to\iota),y:\iota\vdash(\lambda x:\iota.\,x):\iota\to\iota}\,\text{abs}\quad\dfrac{\dfrac{}{x:(\iota\to\iota),y:\iota\vdash(x):\iota\to\iota}\,\text{ax}\quad\dfrac{}{x:(\iota\to\iota),y:\iota\vdash(y):\iota}\,\text{ax}}{x:(\iota\to\iota),y:\iota\vdash(xy):\iota}\,\text{app}}{x:(\iota\to\iota),y:\iota\vdash((\lambda x:\iota.\,x)(xy)):\iota}\,\text{app}}{x:(\iota\to\iota)\vdash(\lambda y:\iota.\,(\lambda x:\iota.\,x)(xy)):\iota\to\iota}\,\text{abs}}{\vdash(\lambda x:(\iota\to\iota).\,\lambda y:\iota.\,(\lambda x:\iota.\,x)(xy)):(\iota\to\iota)\to(\iota\to\iota)}\,\text{abs}$$

Derivation for Example 2.8

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b:\iota\to\iota\to\iota,x:\iota,y:\iota\vdash b:\iota\to\iota\to\iota}\,\text{ax}\quad\dfrac{}{b:\iota\to\iota\to\iota,x:\iota,y:\iota\vdash y:\iota}\,\text{ax}}{b:\iota\to\iota\to\iota,x:\iota,y:\iota\vdash by:\iota\to\iota}\,\text{app}\quad\dfrac{}{b:\iota\to\iota\to\iota,x:\iota,y:\iota\vdash x:\iota}\,\text{ax}}{b:\iota\to\iota\to\iota,x:\iota,y:\iota\vdash byx:\iota}\,\text{app}}{b:\iota\to\iota\to\iota,x:\iota\vdash\lambda y:\iota.\,byx:\iota\to\iota}\,\text{abs}}{b:\iota\to\iota\to\iota\vdash\lambda x:\iota.\,\lambda y:\iota.\,byx:\iota\to\iota\to\iota}\,\text{abs}}{\vdash\lambda b:\iota\to\iota\to\iota.\,\lambda x:\iota.\,\lambda y:\iota.\,byx:(\iota\to\iota\to\iota)\to(\iota\to\iota\to\iota)}\,\text{abs}$$

RExercise 14. Consider the term

$$\lambda b : \iota \to \iota \to \iota.\ \lambda c : \iota \to \iota \to \iota.\ \lambda x : \iota.\ \lambda y : \iota.\ bx(cxy)$$

which encodes a binary operation on booleans. Show that it has the expected type, namely $(\iota \to \iota \to \iota) \to (\iota \to \iota \to \iota) \to (\iota \to \iota \to \iota)$.

See page 277.

We refer to the structures we build to show that a particular term has given type as formal **derivations**, here in the system defined by the typing rules for the simply typed λ-calculus. They are also known as *proof trees* since they can be thought of as putting rules together in a tree-like structure.

It is important to note that these formal derivations are themselves *recursively defined structures*. To emphasize this fact we list the base and step cases of these structures in the format familiar from the previous chapter.

bcTAx The base case is given by the axiom rule.

scTAbs The step case of abstraction requires us to have an existing derivation that establishes that a particular term is of type τ, and to know that x is a variable of type σ.

scTApp The step case of application requires us to have constructed two derivations, one to establish that a particular term is of type $\sigma \to \tau$ and one to establish that another term is of type σ. These have to match in terms of the assumptions we make.

As a consequence we are able to prove statements about derivations by induction. A number of examples appear in the proof of Proposition 2.4 and Lemma 2.5. The reader interested in seeing such a proof might want to skip ahead to the latter result, which allows us to obtain very strong properties for any derivation of a given typing judgement.

We might wonder at this point what we can already say about terms that can be typed. We might wonder in particular what terms of the base type ι we might expect. We can see that certainly we can have any variable as being of the base type, and $\Gamma \vdash x : \iota$ can be derived provided that the final typing information for x in Γ tells us that $x : \iota$. What about other terms? If we have a term of type $\iota \to \iota$ and one of type ι we may use application to create a term of type ι, but we are now running into the issue that we need a term of type ι to create another, and so far we have only created terms that are variables for that.

2.3 Basic Properties of Derivations

We build up some properties of derivations that allow us to derive important properties of our typed system in Section 2.4, such as the fact that our system is well-behaved with respect to α-equivalence and β-reduction. We also look at the fact that some terms that cause problems in the form of infinite reductions cannot be typed in our system.

2.3.1 Typing environments in derivations

Given our previous comparison between our system and a sequent calculus for a particular system of logic we might wonder whether we can add additional assumptions without affecting derivations. Thanks to us allowing shadowing the situation here is more subtle than getting a straight-forward weakening rule which you may remember from COMP11120. We collect some relevant results.

We first have to cover shadowing since we require this result for most other statements we might want to show. We need the second of these statements subsequently, but when trying to show that by induction we find that the induction hypothesis is not strong enough to give us what we require.

Proposition 2.3

Assume that Γ is a type environment, y a variable, t a term and τ a type.

(a) Given a type α and any two list of types $\sigma_1, \ldots, \sigma_n$ and $\rho_1, \ldots, \rho_m$ if the judgement

$$\Gamma, y{:}\sigma_1, \ldots, y{:}\sigma_n, y{:}\alpha \vdash t{:}\tau$$

then so is

$$\Gamma, y{:}\rho_1, \ldots, y{:}\rho_m, y{:}\alpha \vdash t{:}\tau.$$

(b) For types α and β the judgement $\Gamma, y{:}\alpha \vdash t{:}\tau$ is derivable if and only if $\Gamma, y{:}\beta, y{:}\alpha \vdash t{:}\tau$ is.

Proof. *This is a reasonably straightforward induction, but students may prefer instead to prove one of the statements in the following result..*

See page 278.

Note that this result *does not* tell us that we can ignore shadowing—it merely tells us that we can find a derivation of the type whose concluding judgement has a type environment that assigns at most one type to each variable. If you look back to Example 2.6 you can see that the final judgement of the derivation has an empty type environment, and so does not give more than one type for any variable. RExercise 16 shows that without shadowing we are not able to derive a type for that term.

Note that parts (b) and (c) of the following proposition may be understood as forms of weakening as in the sequent calculus: We may add to our assumptions and still obtain a given judgement, but we have to be careful how we weaken! The final statement tells us that we only need to make assumptions about free variables in a term in order to derive a typing judgement for it. We might think of this as a *strengthening*, as opposed to weakening, property.

Proposition 2.4

Assume that Γ and Γ' are type environments, that x is a variable, t a term and σ and τ are types.

(a) We have $\Gamma, \Gamma' \vdash x{:}\sigma$ if and only if we have

- $\Gamma' \vdash x : \sigma$ is derivable or
- $\Gamma \vdash x : \sigma$ is derivable and $x : \sigma, \Gamma' = \Gamma', x : \sigma$.

(b) If $\Gamma \vdash t : \tau$ is derivable then so is $\Gamma', \Gamma \vdash t : \tau$.

(c) If $\Gamma \vdash t : \tau$ is derivable and $y \notin \mathrm{fv}\, t$ then $\Gamma, y : \alpha \vdash t : \tau$ is derivable.

(d) If $\Gamma, y : \alpha \vdash t : \tau$ is derivable and $y \notin \mathrm{fv}\, t$ then $\Gamma \vdash t : \tau$ is derivable.

Proof. *The first of these results is concerned with fairly low level detail, but the other parts are suitable for students who would like to practice carrying out induction proofs for derivations.*

See page 279.

2.3.2 Terms provide a typing blueprint.

We look a bit further into our derivation system to better understand its workings.

How should we prove things about the type system? So far we have carried out proofs by induction over the structure of a derivation. However, derivation trees carry a lot of data which can make inductions a bit tricky if we are also trying to say something about the *types* involved. Consider a derivation which ends with the application rule:

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash f : \sigma \to \tau & \Gamma \vdash t : \sigma\end{array}}{\Gamma \vdash f t : \tau}\ \text{app}$$

We might have to reason about such a derivation in one of the step cases of an induction on derivation trees. But note that our smaller sub-proofs involve a type, σ, which is not mentioned in the conclusion. Though the sub-proofs are smaller, this type could be huge! So even though types and proof trees are recursively defined structures with a close relationship to each other, proof by induction on one of them complicates matters for the other. However, things are not as bad as they seem; the derivation system we have is 'self-documenting', in that the information which seems to have disappeared must actually be encoded in f, and hence in the term ft. In addition, each case of the definition of λ-terms has a unique rule which can type terms of that form. Therefore, when faced with a derivation of $\Gamma \vdash t : \tau$, it is not the type τ that one wants to explore recursively, but the term t.

We note first of all that if we have a derivation that a particular term has a particular type then we can say something about that derivation based on the term.

Lemma 2.5: inversion

(a) If $x \in \mathsf{Vars}$ and $\Gamma \vdash x : \tau$ is derivable, then there is a type environment Δ with $\Gamma = \Delta, x : \tau$ and any such derivation consists of the axiom rule.

(b) If $\Gamma \vdash (\lambda x : \sigma.\, t) : \tau$ is derivable then there is a type α such that $\tau = \sigma \to \alpha$ and $\Gamma, x : \sigma \vdash t : \alpha$ is derivable. Moreover, the concluding rule of any such derivation is the abstraction rule.

(c) If $\Gamma \vdash tu : \tau$ is derivable then there is a type α such that $\Gamma \vdash t : \alpha \to \tau$ and $\Gamma \vdash u : \alpha$ are both derivable. Moreover, the concluding rule of any such derivation is the application rule.

Proof. All these properties are proved by looking at the rules of the derivation system one by one, and realizing that only one of them can justify giving a type to a term of the required form.

(a) If $x \in \mathsf{Vars}$ and $\Gamma \vdash x : \tau$, then the only rule which is applicable is ax, and then the form of the rule implies that there is a type environment Δ such that $\Gamma = \Delta, x : \tau$ as required.

(b) If $\Gamma \vdash (\lambda x : \sigma . t) : \tau$ then the only rule which could be the last step of the derivation is the abstraction rule. So we must have a type α such that the derivation is of the form

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, x : \sigma \vdash t : \alpha\end{array}}{\Gamma \vdash (\lambda x : \sigma . t) : \sigma \to \alpha}\,\text{abs}$$

with $\tau = \sigma \to \alpha$ and so there must be a derivation of $\Gamma, x : \sigma \vdash t : \alpha$ as required.

(c) If $\Gamma \vdash tu : \tau$ then the only rule which could be the last step of the derivation is the application rule. So we must have a type σ such that the derivation is of the form

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash f : \sigma \to \tau\end{array} \quad \begin{array}{c}\vdots\\ \Gamma \vdash t : \sigma\end{array}}{\Gamma \vdash ft : \tau}\,\text{app}$$

and the derivations of the premises of this rule show that $\Gamma \vdash t : \sigma \to \tau$ and $\Gamma \vdash u : \sigma$ as required.

We note that inversion allows us to turn an induction over the shape of a term into one over the shape of the corresponding derivation since the shape of the former determines the shape of the latter.

To show this at work we show that if a type environment allows us to type a term t then it must give us a type for each free variable in t.

Proposition 2.6

If $\Gamma \vdash t : \tau$ is derivable and $x \in \mathrm{fv}\, t$ then there exists a type environment Δ and a type α with $\Gamma = \Delta, x : \alpha$.

Proof. *This is another typical induction proof.*

See page 282.

Note that this tells us that the only terms we can expect to type without using any typing assumptions are terms that don't contain any free variables. So when we want to use the empty type environment we will only be able to type very

special terms, and in the cause of any derivations that such a typing is correct non-empty tying environments are required:

- The only way of starting a derivation is to use the axiom rule, and for that we need a non-empty type environment.
- The only way of forming an abstraction is to have the variable we abstract to appear in the type environment of the premise of the abstraction rule.

This means that even in situations when we are only interested in terms that are typeable with an empty type environment we have to reason about other terms.

The previous result allows us to show that we may always weaken on the left hand side of a type environment.

One of our declared goals in introducing the type system is to rule out the terms from Section 1.5 which allow an infinite number of reduction steps.

RExercise 15. *This exercise invites the reader to convince themselves that certain weird terms from the previous chapter cannot be typed in our system.*

Argue that the following terms cannot be typed in our system.

(a) $\lambda x.\, xx.$

(b) $\lambda f.\, (\lambda x.\, f(xx))\lambda x.\, f(xx).$

Hint: You want to use the inversion result.

For a solution see page 283.

We look at the question of why shadowing is required in order to derive types for certain terms in the following exercise.

RExercise 16. *This exercise invites students to think about why the system introduced here allows more than one type to be assigned to a given variable.*

Convince yourself that it is not possible to give a derivation that assigns a type to the term

$$\lambda x : (\iota \to \iota).\, \lambda y : \iota.\, (\lambda x : \iota.\, x)(xy)$$

without using a type environment that assigns more than one type to x.

A solution may be found on page 283.

2.4 Correctness of the Type System

In the previous section, we define a type system which categorizes certain (annotated) λ-terms as having certain types. But *a priori* this type system might have nothing at all to do with β-reduction which gives the terms of our language their meaning, or with other notions such as α-equivalence. In particular, we want to think of β-reduction as 'simplifying the expression' or 'calculating an answer'. This means that if we β-reduce a term to another, the resulting term should have the same type as the one we started with. We tried to take this into account informally in the previous section: the time has come to prove that the job was done right.

2.4.1 Typing and α-equivalence

We know from Chapter 1 that in order to establish properties for α-equivalence we first need to look at renaming.

Proposition 2.7

Suppose Γ is a type environment, t is a term and y and z are variables, such that z is fresh for t. If there is a derivation of $\Gamma, y : \alpha \vdash t : \tau$ then $\Gamma, z : \alpha \vdash \mathrm{ren}_y^z\, t : \tau$ is also derivable.

Proof. See page 240.

We use inversion (Lemma 2.5) to establish that our typing rules interact well with α-equivalence. We expect α-equivalent terms to have the same type, and the following proposition ensures that this is indeed the case.

Proposition 2.8

If the judgement $\Gamma \vdash t : \tau$ is derivable and t is α-equivalent to t' then $\Gamma \vdash t' : \tau$ is derivable.

Proof. *The proof is another induction.*

See page 283.

2.4.2 Uniqueness of types

One might wonder whether the way our system has been defined we can assign more than one type to each (pre)term.

Proposition 2.9

If $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \tau'$ are both derivable then $\tau = \tau'$.

RExercise 17. *This proof gives a good case for practising induction proofs for our system.*

Show the preceding result.

See page 284 for a solution.

2.4.3 Types and reduction

We are now ready to prove that the types we assign are preserved by β-reduction. Our experience with the untyped λ-calculus in the previous chapter tells us that the core of this proof will deal with substitution, so we investigate this first.

Proposition 2.10

If $\Gamma \vdash a : \alpha$ and $\Gamma, x : \alpha \vdash t : \tau$ are derivable then $\Gamma \vdash t[a/x] : \tau$ is also derivable.

Proof. See page 242.

We can now prove that typing is preserved by β-reduction.

Theorem 2.11

If $\Gamma \vdash t : \tau$ is derivable and $t \xrightarrow{\beta} u$ then $\Gamma \vdash u : \tau$ is derivable.

RExercise 18. *This is another proof by induction that is good for practising.*

Show the preceding result.

A solution may be found on page 285.

We obtain the following result from the preceding one by a straightforward induction over the number of reduction steps involved.

Corollary 2.12: subject reduction

If $\Gamma \vdash t : \tau$ is derivable and $t \xrightarrow{\beta}\!\!\!\!\rightarrow u$ then $\Gamma \vdash u : \tau$ is derivable.

Why do we care about this property when looking at it from a programming perspective? What it tells us is that when we write a program that we think is going to give an output of a particular type then as the computation proceeds, the type cannot change and any result we obtain has to be of the expected type.

2.5 Logical Predicates

Now that we have restricted our language by adding a type system, and proved that the type system matches the dynamics of the language given by β-reduction, we expect to get some sort of pay off. We would hope that a restricted language should have stronger properties, and be easier to reason about, and this is indeed the case. But to exploit this, we need to develop a reasoning technique which lets us take advantage of the type system.

Below we use a theoretical device known as a *logical predicate* to look into the question whether insisting on terms being typeable solves our problems regarding non-termination of computations. Intuitively a logical predicate is a way of defining a property across all types in a uniform way. These properties which vary uniformly across types are precisely the ones which the type system gives us a reasoning principle for.

2.5.1 Uniformity across types

In this case we model the properties we are interested in by defining the set of things which have the property. So to a first approximation a logical predicate should be a set of terms, though as we see below we need to embellish this idea a bit to make it work.

Since we are attending to how the property varies with the type of the terms involved, it makes sense to define not just one set of 'terms which satisfy the property of interest', but instead to pick out these terms separately for each type τ. This makes it easier to state what we mean by 'varying uniformly' with the type. The general idea is that whenever our language gives a way for two terms to

interact, a well behaved property should be preserved by this interaction. In the simply typed lambda calculus, the only way for two different terms to interact with each other is for one of them to be applied to the other. So we need a way of saying that a property must still hold for the application of one term to another whenever it holds for both of the terms involved. In fact, we want an even more uniform notion: It is the term of function type which plays the more significant role in an application, and we will focus on properties which hold for a term of function type *if and only if* all applications involving that term preserve the property. The reader might start to see how a property which is uniform in that way might come with some sort of inductive reasoning principle.

Above we describe the idea of *sets of terms* as a way of modelling properties of terms. But as indicated, to make this work we need to add a bit of extra information to the terms involved. The reason is that terms may contain free variables, and we can only find their type if we have a type environment which gives types to the free variables involved. We could try restricting only to terms which do not contain free variables, but then we would have no examples at the base type ι. Even at other types, we would make our lives difficult because subterms of terms containing no free variables may have free variables. It is much easier to think about properties of terms which come with a suitable type environment.

A logical predicate $\mathcal{R}$, at a given type τ, is a selection of pairs

$$(\Gamma, t),$$

where

- Γ is a type environment and
- t is a term of type τ such that $\Gamma \vdash t : \tau$ is derivable.

Let us use $\mathcal{R}_\tau$ for the set all the pairs of this kind selected.

The notation can be confusing because a pair is written with a comma, but we also use commas to combine type environments. We adopt the convention of dropping the comma when it comes to separating type environments in what follows.

We now forge a connection between these selections at different types by demanding that for all types σ and τ we have that the selection of pairs at the type $\sigma \to \tau$ is completely determined by the selection at the type σ, and the selection at the type τ. We do this by requiring that

$$\mathcal{R}_{\sigma\to\tau}$$

consists of all those

$$(\Gamma, f) \qquad \text{with} \qquad \Gamma \vdash f : \sigma \to \tau$$

such that f, when applied to a suitable term from $\mathcal{R}_\sigma$, gives a term in $\mathcal{R}_\tau$—we just have to work out how to include type environments in an appropriate way.

We are aiming to build a condition which says that

$$(\Gamma, f) \in \mathcal{R}_{\sigma\to\tau}$$

if and only if

$$\text{for all } (\Delta, u) \in \mathcal{R}_\sigma \qquad (?, fu) \in \mathcal{R}_\tau.$$

We have to ensure that the type environment ? is sufficient to type the application of f to u, and we know from Lemma 2.5 that for that purpose we must be able to derive

$$? \vdash f : \sigma \to \tau \qquad \text{as well as} \qquad ? \vdash u : \sigma.$$

We know that Γ contains sufficient information to derive the type $\sigma \to \tau$ for f, so ? must contain the information given in Γ. However, we also must have the information from Δ in order to be sure that we are able to derive the type σ for u, but we can't just put $\Gamma\Delta$ together:

- We know that it is possible to derive $\Gamma\Delta \vdash u : \sigma$ (see Proposition 2.4), but
- in general it is not the case that $\Gamma\Delta \vdash f : \sigma \to \tau$ is derivable since Δ might 'overwrite' the typing information of some of the variables from Γ required to give the correct type, namely $\sigma \to \tau$, to f.

For that reason we can't quantify over all type environments Δ. We have to put a technical condition, described in the next section.

2.5.2 Compatibility of type environments

The required condition for type environments can be formulated as follows.

Definition 18: compatibility of type environments

We say that a type environment Δ is **compatible with a given type environment** Γ provided that for all variables x and types σ we have

$$\Gamma \vdash x : \sigma \text{ derivable} \qquad \text{implies} \qquad \Gamma\Delta \vdash x : \sigma \text{ derivable}.$$

This means that if we can find Γ' with $\Gamma = \Gamma' x : \sigma$ then $\Gamma\Delta = \Gamma'\Delta x : \sigma$.

Note in particular that if Γ is a type environment and z is a variable that does not occur in Γ, then for every type σ we have that $z : \sigma$ is compatible with Γ.

We need to check that this definition has the intended consequences, that is that if we may derive a type for a term t in the type environment Γ, then we may derive the same type when we move to the type environment $\Gamma\Delta$ whenever Δ is compatible with Γ. This is the final statement in the following result, but in order to show that we need to establish the stronger statement which immediately precedes it in order to have an induction hypothesis that is sufficiently strong for the proof to go through.

Proposition 2.13

Let Γ, Γ' and Δ be type environments such that Δ is compatible with Γ. For all variables x and types σ we have the following.

(a) If $\Gamma\Gamma' \vdash x : \sigma$ is derivable then the judgement $\Gamma\Delta\Gamma' \vdash x : \sigma$ is derivable.

(b) If t is a term and τ is a type such that the judgement $\Gamma\Gamma' \vdash t : \tau$ is derivable then $\Gamma\Delta\Gamma' \vdash t : \tau$ is derivable.

(c) If $\Gamma \vdash t : \tau$ is derivable and Δ is compatible with Γ then $\Gamma\Delta \vdash t : \tau$ is derivable.

Proof. See page 243.

2.5.3 Logical predicates

We are finally in a position to give the formal definition for this concept.

Definition 19: logical predicate

A **logical predicate** R **for the simply typed λ-calculus** is given by the following data: For each type τ we have $\mathcal{R}_\tau$ with

$$(\Gamma, t) \in \mathcal{R}_\tau \qquad \text{implies} \qquad \Gamma \vdash t : \tau \text{ is derivable}$$

and for all types σ and τ it is the case that

$$\mathcal{R}_{\sigma\to\tau} = \{(\Gamma, f) \mid \quad (\Gamma\Delta, u) \in \mathcal{R}_\sigma \text{ and } \Delta \text{ compatible with } \Gamma \text{ implies } (\Gamma\Delta, fu) \in \mathcal{R}_\tau\}.$$

Note that since we have one base type ι, and all the other types are function types, a logical predicate is uniquely determined by its choice at ι—what happens for all the other types is determined by the condition given above.

Before looking at a real logical relation, it is good to get some intuition for what the definition means by considering some assignments of sets to types and working out whether they satisfy the definition.

RExercise 19. Consider the following assignments of sets to types, and work out whether or not they are logical predicates.

(a) $\mathcal{P}_\tau = \{(\Gamma, t) \mid \Gamma \vdash t : \tau \text{ derivable and } t \text{ does not contain any abstractions}\}$,

(b) $\mathcal{Q}_\tau = \{(\Gamma, t) \mid \Gamma \vdash t : \tau \text{ derivable and } t \text{ does not contain any applications}\}$,

(c) $\mathcal{R}_\tau = \{(\Gamma, t) \mid \Gamma \vdash t : \tau \text{ derivable and } t \text{ contains at most one free occurrence of the variable } x\}$,

(d) $\mathcal{T}_\tau = \{(\Gamma, t) \mid \Gamma \vdash t : \tau \text{ derivable and } t \text{ contains at least one free occurrence of the variable } x\}$.

See page 286 for a solution.

One can also take a specification of a logical predicate at base type ι and work out what the assignment must be at another type.

RExercise 20. For the assignments of sets to types in RExercise 19 above which are *not* logical predicates, one can still always form a logical predicate with the same elements at type ι, as described above. Work out what the elements at type $\iota \to \iota$ are for each such logical predicate.

A solution may be found on page 286.

We see logical predicates in action in the following section, with an example provided in Section 2.6.2.

Proposition 2.14

Let $\mathcal{R}$ be a logical predicate such that at the base type we have that

$$(\Gamma, t) \in \mathcal{R}_i \qquad \text{implies} \qquad (\Gamma\Delta, t) \in \mathcal{R}_i$$

for all Δ which are compatible with Γ. If $(\Gamma, t) \in \mathcal{R}_\tau$ and Δ is compatible with Γ then $(\Gamma\Delta, t) \in \mathcal{R}_\tau$.

Proof. See page 244.

We need to worry about how logical relations interact with α-equivalence, and that is taken care of by the following proposition.

Proposition 2.15

Let $\mathcal{R}$ be a logical predicate such that at the base type we have that if $(\Gamma, t) \in \mathcal{R}_\iota$ and $t \sim_\alpha t'$ then $(\Gamma, t') \in \mathcal{R}_\iota$.

Then the corresponding property holds for all types.

Proof. *This is a short proof by induction which is a good exercise in understanding the property defining logical predicates.*

See page 287.

2.6 Strong Normalization

One of our concerns with the untyped λ-calculus as a programming language is that a computation might continue forever by becoming circular. This is one of the ways in which we are prevented from reasoning about programs as if they were functions: a function always has a well defined output for any input. While we know from the Halting Problem that a sufficiently powerful programming language must contain programs that might continue forever, we show in this section that our type system is so strong that it has eliminated non-termination completely.

This might strike the reader as either worse or better than the situation in the untyped λ-calculus. Worse because while reasoning about programs has become simpler, the language is no longer useful in practice; better because we have a solid foundation with good properties, on top of which useful features could be added carefully, rather than discovered as consequences of mind-bending hacks! Whichever of these alternatives suits the reader's sensibilities, it is clearly valuable to *prove* that the simply typed λ-calculus really does contain only terminating programs. Whether good or bad news, it is certainly a big claim. In this section, we prove this claim by constructing a suitable logical relation. But first we ought to clarify the idea of a program that always halts.

2.6.1 Terminating reductions

One way of addressing this idea is to look at those terms that cannot keep reducing forever, and we call such terms *strongly normalizing*. In other words, no matter

how we reduce such a term, a sequence of reduction steps starting from that term finishes after a finite number of reductions. The relation $\xrightarrow{\beta}\!\!\!\!\!\rightarrow$ is reflexive, and so every term can reduce to itself forever, and for that reason the mathematical definition of this property has to work around that: We consider a sequence of reduction as having the desired property if they only way of achieving infinite reduction sequences is to reduce to the same term.

We can picture this situation as follows:

$$t = t_0 \xrightarrow{\beta}\!\!\!\!\!\rightarrow t_1 \xrightarrow{\beta}\!\!\!\!\!\rightarrow t_2 \xrightarrow{\beta}\!\!\!\!\!\rightarrow \ldots \xrightarrow{\beta}\!\!\!\!\!\rightarrow t_n \circlearrowleft \beta$$

where there are no alternative reductions from t_n.

Definition 20: strongly normalizing

A λ-term t is **strongly normalizing** if and only if whenever we have terms t_i for $i \in \mathbb{N}$ such that $t_0 = t$ and for all $i \in \mathbb{N}$, $t_i \xrightarrow{\beta}\!\!\!\!\!\rightarrow t_{i+1}$, then there is a $n \in \mathbb{N}$ such that for all $m \geq n$, $t_n = t_m$.

What this definition tells us is that whenever we draw a graph indicating all the possible transitions for a strongly normalizing term, all the branches end after a finite number of steps with any additional $\xrightarrow{\beta}\!\!\!\!\!\rightarrow$ reductions being from terms to themselves.

From the point of view of a compiler writer this property is comforting because it means that to calculate the 'final answer' from a strongly normalizing term, one can implement any reduction strategy and it will eventually succeed–one does not need to cleverly choose which subterm to reduce to avoid getting stuck in an infinite loop. In this section we show that in the simply typed λ-calculus all terms are strongly normalizing.

This discussion of getting to a 'final answer' might make the reader wonder whether we can detect that the answer has been reached. The definition of strong normalization given tells us one way: after every β-reduction, one can check whether the term is equal to the one before the reduction happened. A nicer definition of a 'final answer' is one in which there are no redexes. But this is a problem because we know there are terms like $(\lambda x.\, xx)\lambda x.\, xx$ which contain redexes but still reduce to themselves. They do not make very nice 'final answers'. In particular, if we consider a slightly modified term, like $(\lambda x.\, xx)\lambda y.\, yy$, this reduces to a term α-equivalent to, rather than equal to, itself. We might worry about how long such pointless sequences of rewritings can go on for!

When we introduced the type system we observed that these paradoxical term can not be typed. We can turn this intuition into a more positive statement about typeable terms.

Proposition 2.16

Let Γ be a type environment, τ a type and t and t' terms such that $\Gamma \vdash t : \tau$ and $(\Gamma, \vdash t' : \tau$ are derivable. Show that the following properties hold.

(a) if $t \xrightarrow{\beta} t' \sim_\alpha t$ then $t = t'$, and

(b) if $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow t' \sim_\alpha t$ then $t = t'$.

Proof. See page 244.

RExercise 21. Find a way of formulating the property of being strongly normalizing using the relation $\xrightarrow{\beta}$ rather than $\xrightarrow{\beta}\!\!\!\!\rightarrow$. What reasons can you see for using the second of these relations in the definition given above?

A solution appears on page 287.

One of the recurring themes in these notes is that all our notions should work well with α-equivalence and we check that this is also the case for strong normalization.

RExercise 22. Show that for all λ-terms t and t', if t is strongly normalizing and $t \sim_\alpha t'$, then t' is strongly normalizing.

See page 287 for a solution.

What examples can we give of strongly normalizing terms? At the very least all variables of base type must be strongly normalizing since there are no reductions from a term that's a variable. But that's pretty much all that we can say at this stage. On the other hand, we can show various crucial properties of strongly normalizing terms. Since the definition of strong normalization above is not always easy to work with we collect useful properties here which we make use of.

Proposition 2.17

Strong normalization has the following properties:

(a) If t is strongly normalizing and $t \xrightarrow{\beta} t'$, then t' is strongly normalizing.

(b) For a a term t if for all t' such that $t \xrightarrow{\beta} t'$ we have that t' is strongly normalizing, then t is strongly normalizing.

(c) If t is strongly normalizing and t' is a subterm of t then t' is strongly normalizing.

(d) If x is variable and we have strongly normalizing terms $t_1, t_2, \ldots, t_k$ where $k \in \mathbb{N}$, then the application $xt_0t_1 \cdots t_k$ is strongly normalizing.

(e) If t_0 and $s[t_0/x]t_2 \ldots t_k$, where $k \in \mathbb{N}$, are strongly normalizing then so is $(\lambda x : \alpha.\, s)t_0t_2 \cdots t_k$.

Proof. *Going through at least some of the following statements provides good familiarization with reasoning about strong normalization.*

See page 287 for a solution.

2.6.2 A logical predicate for strong normalization

Our strategy to prove that typeable terms must be strongly normalizing is to build a logical predicate S which obviously implies strong normalization.

As indicated above, it is sufficient to define the logical predicate for the base type. We set

$$\mathcal{S}_\iota = \{(\Gamma, t) \mid \Gamma \vdash t : \iota \text{ is derivable and } t \text{ is strongly normalizing}\}.$$

What can we say about the elements of this logical predicate at a particular type? Based on what we know about strong normalization at this stage we know that at the very least that if we can derive $\Gamma \vdash x : i$ then $(\Gamma, x) \in \mathcal{S}_\iota$.

What about variables at higher types? Assume that $\Gamma \vdash x : \sigma \to \tau$ is derivable. We need to check the given condition, so let's assume we have $(\Delta, u) \in \mathcal{S}_\sigma$ such that Δ is compatible with Γ. We have to worry about $(\Gamma\Delta, xu)$ being in $\mathcal{S}_\tau$. We can see that we need to find out something about $\mathcal{S}_\sigma$ first! But we can also see that if we knew that terms in $\mathcal{S}_\sigma$ are strongly normalizing then we would know by Proposition 2.17 that xu is strongly normalizing, and so if we knew that $\mathcal{S}_\tau$ contains all the strongly normalizing terms we could be sure that our (Γ, x) is indeed an element of $\mathcal{S}_{\sigma\to\tau}$.

In the following section we check that all the terms in this logical predicate are strongly normalizing, and that all typeable terms are in the logical predicate —in other words, for all types τ we have

$$(\Gamma, t) \in \mathcal{S}_\tau \qquad \text{if and only if} \qquad \Gamma \vdash t : \tau \text{ is derivable.}$$

2.6.3 Proving strong normalization

The general strategy for using a logical predicate to prove something is first to show that all terms which are in the logical predicate have the property of interest. We do this in the following result: note that the first statement is exactly this result, since in this case the property of interest is strong normalization. The others are technical results which help us to get around the issues discussed above.

Lemma 2.18

For all types τ, we have the following. Let x be a variable and let $s, t, t_0, t_1, \ldots, t_k$ be terms.

(a) If $(\Gamma, t) \in \mathcal{S}_\tau$ then t is strongly normalizing.

(b) If $t_1, \ldots, t_k$ are strongly normalizing and $\Gamma \vdash xt_1t_2\cdots t_k : \tau$ is derivable then

$$(\Gamma, xt_1t_2\cdots t_k) \in \mathcal{S}_\tau.$$

(c) If t_0 is a strongly normalizing term such that $\Gamma \vdash t_0 : \alpha$ is derivable for some type α and $(\Gamma, s[t_0/x]t_1t_2\cdots t_k) \in \mathcal{S}_\tau$ then

$$(\Gamma, (\lambda x : \alpha.\, s)t_0t_1\cdots t_k) \in \mathcal{S}_\tau.$$

Proof. See page 247.

Next we prove the main lemma which helps us show that all typeable terms are strongly normalizing. As a first try we might simply aim to prove that for all type environments Γ, terms t and types τ such that $\Gamma \vdash t : \tau$ is derivable, we have $t \in \mathcal{S}_\tau$. The natural approach is to use induction on the term t. But we find that to prove

the abstraction step case the definition of logical relation asks us to consider what happens when we apply the given abstraction to an argument. By now we are used to the fact that this means considering a substitution, so we must strengthen the induction hypothesis to involve a substitution. What we actually want is to be able to do 'one more substitution' in the step case, and so the induction hypothesis has to talk about all sequences of substitutions of any length. This leads us to the following result.

Lemma 2.19

Suppose $\Gamma \vdash t : \tau$ is derivable. Let

$$\{(x_1, \alpha_1), \ldots, (x_k, \alpha_k)\} = \{(x, \alpha) \mid x \in \mathsf{Vars}, \Gamma \vdash x : \alpha \text{ is derivable}\}.$$

Assume that Δ is a type environment such that for $1 \leq i \leq k$ we have a term a_i such that $(\Delta, a_i) \in \mathcal{S}_{\alpha_i}$. Then $(\Delta, t[a_1/x_1] \cdots [a_k/x_k]) \in \mathcal{S}_\tau$.

Proof. See page 248.

To justify the form of Lemma 2.19 we started off by wanting to show that every typeable term is in the logical relation $\mathcal{S}$. But to prove this by induction we had to state a more general result since this gives a more general induction hypothesis. It is perhaps no longer clear, however, that this really is a generalization of the result we set out to prove, since to use Lemma 2.19 we have to modify the term t by substituting values for some of its variables. However, we can *substitute the variables for themselves* which will give us an α-equivalent term.

Theorem 2.20

If $\Gamma \vdash t : \tau$ is derivable then t is strongly normalizing.

Proof. Assume we have a derivable judgement as given in the statement. We collect the final type assignments from Γ by setting

$$\{(x_1, \alpha_1), \ldots, (x_k, \alpha_k)\} = \{(x, \alpha) \mid x \in \mathsf{Vars}, \Gamma \vdash x : \alpha \text{ is derivable}\}.$$

Let Δ be the type assignment given by $x_i : \alpha_i$ for $1 \leq i \leq k$. Then for all $1 \leq i \leq k$ we have by construction that $\Delta \vdash x_i : \alpha_i$ is derivable, and moreover by part (ii) of Lemma 2.18 (for $k = 0$) we have $(\Delta, x_i) \in \mathcal{S}_{\alpha_i}$ for $1 \leq i \leq k$.

We may apply Lemma 2.19 to obtain that

$$(\Delta, t[x_1/x_1] \cdots [x_k/x_k]) \in \mathcal{S}_\tau,$$

which by (i) of Lemma 2.18 means that it is strongly normalizing. But we know from Proposition 1.11 that

$$t[x_1/x_1] \cdots [x_k/x_k] \sim_\alpha t,$$

and so by RExercise 22 we find that t is strongly normalizing.

2.7 Contextual Equivalence

Strong normalization seems to be good evidence that terms in the simply typed λ-calculus are very well-behaved. We now turn to proving that they 'behave like functions', which will have to include a formal definition of what we mean by 'behave like'. Clearly, whatever we mean by the behaviour of a λ-term, it should be independent of the precise 'implementation details' of how exactly the term is written.

2.7.1 $\alpha\beta$-equivalence

First, if two terms are α-equivalent, then clearly we should think of them as having the same behaviour. Next, we should surely identify two terms if they β-reduce to a common term. We know from the confluence result that in that case any computation involving one of the terms will eventually be the same as a computation involving the other: they will forget the difference between them as computation proceeds.

Definition 21: $\alpha\beta$-equivalence

We say that two λ-terms t and u are **$\alpha\beta$-equivalent** if there exist terms t' and u' such that $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow t'$ and $u \xrightarrow{\beta}\!\!\!\!\!\rightarrow u'$ and $t' \sim_\alpha u'$. In that case we write $t \sim_{\alpha\beta} u$.

We may picture this condition as follows:

$$t \sim_{\alpha\beta} u \qquad \text{if and only if} \qquad \exists t', u' \text{ with } \begin{array}{ccc} t & & u \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ t' & \sim_\alpha & u' \end{array}$$

However, this picture is a little misleading: Note in particular that in the picture above we also have that

$$t \sim_{\alpha\beta} u' \qquad \text{and} \qquad u \sim_{\alpha\beta} t',$$

since, for example,

$$\begin{array}{ccc} t & & \\ {\scriptstyle\beta}\downarrow & & \\ t' & \sim_\alpha & u' \circlearrowright \beta \end{array} \qquad \text{and so} \qquad \begin{array}{ccc} t & & u' \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ t' & \sim_\alpha & u' \end{array}$$

and, maybe even more confusing, that

$$t' \sim_{\alpha\beta} t,$$

since

$$\begin{array}{ccc} & & t \\ & & {\scriptstyle\beta}\downarrow \\ \beta\circlearrowright & t' \sim_\alpha & t' \end{array} \qquad \text{and so} \qquad \begin{array}{ccc} t' & & t \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ t' & \sim_\alpha & t' \end{array}$$

Proposition 2.21

The relation $\alpha\beta$-equivalence is an equivalence relation, and it is the smallest relation containing both, the α-equivalence and β-reduction, relations.

Proof. *It is a relatively straightforward proof to establish the three properties required. The second part is fairly abstract and based on the idea that the smallest equivalence relation containing some given relation R can be characterized by being contained in every equivalence relation containing R.*

See page 288.

We show that when we build terms replacing a subterm by an $\alpha\beta$-equivalent one gives an $\alpha\beta$-equivalent one.

Proposition 2.22

Assume that t, t' and u are λ-terms.

(a) If $t \sim_{\alpha\beta} t'$ then $tu \sim_{\alpha\beta} t'u$ and $ut \sim_{\alpha\beta} ut'$.

(b) If x, x' and z are variables with $z \notin \operatorname{vars} t \cup \operatorname{vars} t'$ and $\operatorname{ren}_x^z t \sim_{\alpha\beta} \operatorname{ren}_{x'}^z t'$ then $\lambda x.\, t \sim_{\alpha\beta} \lambda x'.\, t'$.

Proof. *This proof requires pulling together a number of previous results.*

See page 289.

Is $\sim_{\alpha\beta}$ a good notion of 'same behaviour' for λ-terms? This works well enough for terms of type ι, which we think of as observable pieces of data which are equal exactly when they compute to the same value. But there is a problem at function types.

Example 2.9. Consider the λ-terms $\lambda f : \iota \to \iota.\, f$ and $\lambda f : \iota \to \iota.\, \lambda x : \iota.\, fx$. Intuitively, if we can reason about terms as if they were functions, these terms should have the same behaviour, because if we apply both of them to a arguments g and u of the appropriate types, then we have

$$(\lambda f : \iota \to \iota.\, f)gu \xrightarrow{\beta\alpha} gu$$

and

$$(\lambda f : \iota \to \iota.\, \lambda x : \iota.\, fx)gu \xrightarrow{\beta\alpha} (\lambda x : \iota.\, gx)u \xrightarrow{\beta\alpha} gu$$

The example above shows that two λ-terms which are not $\alpha\beta$-equivalent to each other can still have the same behaviour when we test them by applying them to arguments. This shows that if we can really reason about terms as if they were functions, we need to go beyond $\alpha\beta$-equivalence. Terms of higher type require a non-trivial notion of equivalence.

2.7.2 Contextual equivalence

To settle the question whether we can indeed think of the terms of the previous example as equivalent, we need to specify what we mean by equivalent terms. Intuitively, two programs should be considered equivalent if in any larger program we could replace one with the other without changing the result of computation. This means they would pass all the same tests, and that documentation for how to use one would apply to the other, and so on.

To formalize this for the simply typed λ-calculus, we first need to decide what we consider to be a 'result' of computation. Since terms of function type can be equivalent while having non-trivial internal differences, we don't want to consider those to be results of computation. Indeed, it is to investigate them that we are trying to come up with a notion of equivalence. Therefore, we consider terms of type ι to be acceptable 'results': to see if they are equivalent is is sufficient to compare them possible using $\alpha\beta$-equivalence.

Next we must define all the possible contexts in which a term can be put. A context is like a $\lambda-$term, but with exactly one 'hole' in it. Substituting into this 'hole' is simpler than substituting for a variable, because we do not allow abstraction over the 'hole'.

Definition 22: context

The set of **contexts** of the simply typed λ-calculus is defined as follows.

bcVar$\Box$ A special symbol, $\Box$, is a context.

scAbs$\Box$ If C is a context, x is a variable and σ is a type, then $\lambda x{:}\sigma.\, C$ is a context.

scApp$\Box$ If C is a context and t is a λ-term, then Ct and tC are contexts.

We may think of a context as an almost λ-term that has exactly one occurrence of the special symbol $\Box$ into which we may substitute.

Definition 23: substitution into a context

The operation of **substitution** of a term t into a context C, which we write as $C\{t\}$, is a preterm of the simply typed λ-calculus defined by recursion on C, as follows.

bcVar$\{\}$ $\Box\{t\} = t$.

scAbs$\{\}$ If C is a context, x is a variable and σ is a type, then

$$(\lambda x{:}\sigma.\, C)\{t\} = \lambda x{:}\sigma.\, C\{t\}.$$

scApp$\{\}$ If C is a context and u is a λ-term

$$(Cu)\{t\} = (C\{t\})u \qquad \text{and} \qquad (uC)\{t\} = u(C\{t\}).$$

Note that substitution into a context is *not* capture avoiding! It is really designed to model a simply copy-and-paste of a term into a larger program. For this reason, we ought to restrict which terms we expect to behave like functions to only consider terms with no free variables. We call a term with no free variables a **closed term**. Otherwise, the context definitely has a sneaky way of interacting

with a term other than applying it to an argument: it can abstract over one of the free variables and apply the resulting abstraction to a term. Our restriction is not very severe, since if we find a good notion of equivalence for terms with no free variables, we could extend it to terms with free variables by declaring them to be equivalent if the corresponding closed terms where we have abstracted over those free variables are equivalent.

In the λ-calculus, we can simplify the notion of contexts once we have decided only to concern ourselves with closed terms. This is because for closed terms, there is little difference between substitution into a context and applying a term to an argument. The reason for this is that for such terms, the difference between ordinary substitution and capture-avoiding substitution is not very significant.

Definition 24: variable not occurring in a context

A variable y **does not occur in a context** C if any only if for all terms t, if y is fresh for t then y is fresh for $C\{t\}$.

Proposition 2.23

For every context C and variable y not occurring in C there exists a term $\{C\}_y$ such that for every closed term f we have $C\{f\} \sim_\alpha C_y[f/y]$.

Proof. This is a proof by induction on the structure of C.

bcVar$\Box$ Let $\Box_y = y$. Then we have $\Box\{f\} = f \sim_\alpha f = y[f/y] = \Box_y[f/y]$ as required.

scAbs$\Box$ Let $(\lambda x{:}\sigma.\, C)_y = \lambda x{:}\sigma.\, C_y$. We assume that z is fresh for f, y, and $\lambda x{:}\sigma.\, C_y$.. That allows us to calculate as follows:

$$\begin{aligned}
(\lambda x{:}\sigma.\, C)_y\{f\} &= (\lambda x{:}\sigma.\, C_y)\{f\} && \text{def } (\lambda x{:}\sigma.\, C)_y \\
&= \lambda x{:}\sigma.\, C_y\{f\} && \text{scAbs}\{\} \\
&\sim_\alpha \lambda x{:}\sigma.\, (C_y[f/y]) && \text{ind hyp} \\
&\sim_\alpha \lambda z{:}\sigma.\, \mathrm{ren}^z_x\,(C_y[f/y]) && \text{Proposition 1.7} \\
&\sim_\alpha (\lambda x{:}\sigma.\, C_y)[f/y]. && \text{Proposition 1.10}
\end{aligned}$$

scApp$\Box$ Let $(Ct)_y = C_y t$, and note that by assumption we have that y is fresh for t. Now

$$\begin{aligned}
(Ct)_y\{f\} &= C_y\{f\}t && \text{scApp}\{\} \\
&\sim_\alpha (C_y[f/y])t && \text{ind hyp} \\
&\sim_\alpha (C_y t)[f/y] && \text{scApp}[\,] \text{ and Proposition 1.10}
\end{aligned}$$

The case for tC is analogous.

This is one way in which a language with more advanced features can be easier to study than an apparently simpler language. If we were studying a language with no notion of 'functions' or 'procedures' which allow us to represent a context as a term, we would have to treat contexts as a truly separate notion. This corresponds

to the fact that in a language with functions' or 'procedures' we can wrap up tests as pieces of code, whereas if the language lacks these features, tests which embed the code to be tested in a larger program would have to be constructed by external scripts, which would be more unwieldy.

Recalling that the results of computation are terms of type ι, we need to define the notion of a context which can be used to get a 'test result' from a term of of a given type τ. We want these contexts to have a 'hole' in them, such that if we replace the 'hole' with a term of type σ, then the resulting term has type ι. In fact, it is useful to consider the general idea of a context which transforms a term of type σ into one of some type τ. In the usual way we have to allow for free variables to appear in our terms and so we have to make use of a type environment once again to define this notion.

Definition 25: (Γ, σ, τ)-context

Given a type environment Γ and types σ and τ, a (Γ, σ, τ)-**context** is a context C such that for all terms t

$$\text{if } \Gamma \vdash t : \sigma \text{ is derivable} \qquad \text{then} \qquad \Gamma \vdash C\{t\} : \tau \text{ is derivable.}$$

Note that because we do not have any built-in data types, Γ can't be empty in a (Γ, τ)-context, because the only way we have to get a term of type ι is to assume we have a variable of that type in Γ.

This might sound a bit abstract, but in fact is it quite a down-to-earth observation. One can think of the environment Γ as a 'header file' describing the type signatures of some 'constants' or 'constructors' provided by a library. For example we may want a constant of type ι to represent zero and one of type $\iota \to \iota$ to represent successor. Since we have no built in data types, the only data we have to work with must be provided by such an 'external library'. We don't know how these are implemented, but we can see how the λ-terms we are interested in make use of them, as a way of performing experiments on those terms.

Example 2.10. Consider the term $\lambda x{:}\iota.\, \lambda y{:}\iota.\, x$. We have

$$\vdash \lambda x{:}\iota.\, \lambda y{:}\iota.\, x : \iota \to \iota \to \iota,$$

but if we work in the empty type environment, we don't have any inputs of type ι to test it with. Hence there are no $(\Gamma, \iota \to \iota \to \iota, \iota)$-contexts when Γ is empty. On the other hand if we make an assumption $k : \iota$ then we can form a $(k : \iota, \iota \to \iota \to \iota, \iota)$-context, namely $\Box k k$. However, in this typing environment, we can't find any contexts which would let us observe a difference in behaviour between the term above and $\lambda x{:}\iota.\, \lambda y{:}\iota.\, y$.

For that we need at least two different terms of type ι. However if we consider $(j : \iota, k : \iota, \iota \to \iota \to \iota, \iota)$-contexts, we can perform a test with $\Box j k$, and now if we substitute the two terms into this context they will give different results:

$$\begin{aligned}
\Box j k\{\lambda x{:}\iota.\, \lambda y{:}\iota.\, x\} &= \Box j\{\lambda x{:}\iota.\, \lambda y{:}\iota.\, x\} k && \text{scApp}\{\} \\
&= \Box\{\lambda x{:}\iota.\, \lambda y{:}\iota.\, x\} j k && \text{scApp}\{\} \\
&= (\lambda x{:}\iota.\, \lambda y{:}\iota.\, x) j k && \text{bcVar}\{\} \\
&\xrightarrow{\beta\alpha} (\lambda y{:}\iota.\, j) k && \text{bcVar}\beta
\end{aligned}$$

$$\xrightarrow{\beta} j \qquad \text{bcVar}\beta$$

while

$$\begin{aligned}
\Box jk\{\lambda x{:}\iota.\,\lambda y{:}\iota.\,y\} &= \Box j\{\lambda x{:}\iota.\,\lambda y{:}\iota.\,y\}k && \text{scApp}\{\}\\
&= \Box\{\lambda x{:}\iota.\,\lambda y{:}\iota.\,y\}jk && \text{scApp}\{\}\\
&= (\lambda x{:}\iota.\,\lambda y{:}\iota.\,y)jk && \text{bcVar}\{\}\\
&\xrightarrow{\beta\alpha} (\lambda y{:}\iota.\,y)k && \text{bcVar}\beta\\
&\xrightarrow{\beta} k && \text{bcVar}\beta.
\end{aligned}$$

Example 2.11. In the example above, we were able to distinguish two terms by using a context with two different variables of type ι. But what if we don't know in advance how many terms of type ι we might need in order to probe the behaviour of a term of interest?

In that case, we could use the environment $z{:}\iota, s{:}\iota \to \iota$. Now we have countably many terms of type ι to use, since we have:

$$\begin{aligned}
z{:}\iota, s{:}\iota \to \iota &\vdash z{:}\iota\\
z{:}\iota, s{:}\iota \to \iota &\vdash sz{:}\iota\\
z{:}\iota, s{:}\iota \to \iota &\vdash s(sz){:}\iota\\
&\vdots
\end{aligned}$$

Note that in the examples above, we only need 'input data' of type ι or $\iota \to \iota$, but never input data which includes a higher-order function type which takes a function as an argument, like $(\iota \to \iota) \to \iota$. This is a general phenomenon. Just as we only want to consider tests which output a concrete piece of data (i.e. a term of type ι), we only want to have to put in concrete data in order to test our terms.

Definition 26: first order type

We recursively define the notion of **first order type** for the simply-typed λ-calculus.

bc→ The base type ι is a first order type, and

sc→ if τ is a first order type, then so is $\iota \to \tau$

A type that is not first order is **higher order**.

A type environment is called a **first order** if it only assigns first order types to variables.

Exercise 23. *This exercise gives some idea that in the presence of restrictions to type environments that require them to be first order one may prove stronger properties for terms that are typeable in this way.*

Show that if Γ is a first order type environment, τ is a higher order type and

$$\Gamma \vdash t{:}\tau$$

is derivable, then there exists a variable x, a type σ and terms t' and u such

that

$$t \xrightarrow{\beta}\!\!\!\!\!\rightarrow t' \sim_\alpha \lambda x{:}\sigma.\, u.$$

See page 290 for a solution.

Definition 27: Γ-concrete terms

Let Γ be a first order type environment. A term t is called Γ-**concrete** if and only if

bcVar t is a variable and there is a type τ such that $\Gamma \vdash t{:}\tau$ is derivable;

scApp $t = xy$ where x and y are Γ-concrete

Note that our definition ensures that a Γ-concrete term t is created by using only the rules bcVar and scApp, without any abstractions.

Proposition 2.24

Let Γ be a first order type environment, τ a first order type, and t a term such that $\Gamma \vdash t{:}\tau$ is derivable. Then

- either t reduces to a term which is formed using only the rules bcVar and scApp, or
- there exists a variable x, a type σ and terms t' and u such that

$$t \xrightarrow{\beta}\!\!\!\!\!\rightarrow t' \sim_\alpha \lambda x{:}\sigma.\, u.$$

Proof. This is a proof by induction on t.

bcVar This case is immediate since the given term reduces to itself in 0 steps, and so matches the first case.

scAbs If t is an abstraction then it has to be of the form $\lambda x{:}\sigma.\, u$ and since this term reduces to itself in 0 steps it falls into the second case.

scApp If t is an application, say rs, then we know by inversion that we must be able to find a type ρ such that

$$\Gamma \vdash r{:}\rho \to \tau \qquad \text{and} \qquad \Gamma \vdash s{:}\rho$$

are both derivable.

If $\rho = \iota$ then both ρ and $\rho \to \tau$ are first order, and we may apply the induction hypothesis to both r and s. In particular we know that s is of base type and so must match the first case.

- If r matches the first case, then so does rs, since we may reduce the two subterms until we are in the first case.
- If r matches the second case then it must be the case that $\sigma = \iota$ in order for rs to typeable, and we an find a term r' with

$$r \xrightarrow{\beta}\!\!\!\!\!\rightarrow r' \sim_\alpha \lambda x{:}\iota.\, u.$$

. We pick up this case below.

Otherwise $\rho \to \tau$ is a higher order type, and by Exercise 23 there exists a variable x, a type which we know has to be ρ and terms r' and u such that

$$r \xrightarrow{\beta}\!\!\!\!\!\rightarrow r' \sim_\alpha \lambda x : \rho.\, u.$$

So either rs reduces to a term built using only bcVar and scApp, or we may find terms as described above with

$$rs \xrightarrow{\beta}\!\!\!\!\!\rightarrow r's \sim_\alpha (\lambda x : \rho.\, u)s.$$

By Proposition 1.16, the term $r's$ further reduces by bcVarβ.

By strong normalization, after finitely many steps we reach a term that cannot be reduced any further. We have just seen that applications are always reducible, and so we know that we end up with a variable or an abstraction, which match the two cases given.

Proposition 2.25

Let Γ be a first order type environment and let t be a term such that $\Gamma \vdash t : \iota$ is derivable and such that there is no term t' such that $t \xrightarrow{\beta} t'$. Then t is built using only the rules bcVar and scApp.

Proof. This is a proof by cases for t

bcVar This case is immediate.

scAbs This cannot occur since $\Gamma \vdash t : \iota$ and abstractions always have a function type.

scApp If t is an application, say rs, then we know by inversion that we must be able to find a type ρ such that

$$\Gamma \vdash r : \rho \to \iota \qquad \text{and} \qquad \Gamma \vdash s : \rho$$

are both derivable.

If $\rho \to \iota$ is a higher order type then by Exercise 23 r must be an abstraction, as by assumption (and the definition of β-reduction) r cannot be β-reduced. But then t would be an application with left-hand side an abstraction which would β-reduce. Therefore this case cannot occur.

Hence $\rho \to \iota$ must be a first order type, which means that $\rho = \iota$ must hold. In this case, by Proposition 2.24 and since the term cannot be further β-reduced, r either contains only variables and applications, or is an abstraction. But r cannot be an abstraction since then t would β-reduce. So r must be built using only bcVar and scApp, and the same

must be true for s by Proposition 2.24, and so $t = rs$ satisfies the same restriction.

We note that for terms of base type the two previous results taken together tell us that for a term t such that $\Gamma \vdash t : \iota$ is derivable for some first order type environment, then t β-reduces to a term t' that is α-equivalent to one built using only the rules bcVar and scApp. But if we have such a term t' then it consists entirely of free variables, and the only α-equivalent term to t' is t' itself. This gives us the following result.

Corollary 2.26

Let Γ be a first order type environment. For every term t such that $\Gamma \vdash t : \iota$ there is a unique Γ-concrete term $\mathrm{val}\, t$ such that $t \xrightarrow{\beta}\!\!\!\!\rightarrow \mathrm{val}\, t$.

Now we are ready to define the key notion of equivalence, which corresponds to the programmer's intuition that two programs are equivalent if one could be used in place of the other in any larger program without affecting the observable output.

Definition 28: contextual equivalence

If τ is a type and t and t' are closed terms such that $\vdash t : \tau$ and $\vdash t' : \tau$ are derivable then t is **contextually equivalent** to t', written

$$t \simeq t',$$

if for all first order type environments Γ and all (Γ, τ, ι)-contexts C we have

$$C\{t\} \sim_{\alpha\beta} C\{t'\}.$$

We can see that (Γ, τ, ι)-contexts are important and we look at what we can say about how (Γ, τ, σ)-contexts might be built.

Contextual equivalence is well motivated, but it is hard to work with because of the quantification over all contexts. Once one has defined contextual equivalence for a language, one is therefore interested in finding more concrete ways to describe the same relation.

We can, however, already simplify the definition of contextual equivalence somewhat, making use of Proposition 2.23 to replace the quantification over contexts by a quantification over all terms (based on the idea that in a language with 'functions', tests are not external to the language but can be implemented by terms).

Corollary 2.27

If τ is a type and t and t' are closed terms such that $\vdash t : \tau$ and $\vdash t' : \tau$ are derivable then t is contextually equivalent to t' if and only if for all first order type environments Γ and all terms f such that $\Gamma \vdash f : \tau \to \iota$ we have

$$ft \sim_{\alpha\beta} ft'.$$

This is an improvement, but there is still a quantification over a large number of terms, which could be complicated syntactic objects. We therefore look for

alternative definitions of the same relation which are more concrete in more significant ways.

We will look at two. One is given by a rewriting relation on terms inspired by Example 2.9 above, and we discuss this in the following subsections. The other is a way of mapping terms to mathematical functions, and we look at this in Section 2.8.

2.7.3 η-Conversion for the untyped λ-calculus

Suppose we want to turn the idea of Example 2.9 into a way of rewriting terms. A rough idea is that a term of the form $\lambda x.\, f\, x$ can always be simplified to just f. We have to take a bit of care, however, because if x occurs free in the subterm f, then removing the outer abstraction will in fact change the behaviour of the term $\lambda x.\, f\, x$ by making x free in the whole thing. We also want to be able to carry out the simplification anywhere inside a larger term. We define a relation for λ-terms capturing these ideas. We begin by doing so for untyped terms.

Definition 29: η-conversion

The relation of **η-conversion**, written $\xrightarrow{\eta}$, for λ-terms is defined recursively as follows:

bcVarη $\lambda x.\, t\, x \xrightarrow{\eta} t$ provided $x \notin \mathrm{fv}\, t$.

scAbsη $\lambda x.\, t \xrightarrow{\eta} \lambda x.\, t'$ if $t \xrightarrow{\eta} t'$.

scAppη $t\, u \xrightarrow{\eta} t'\, u'$ if $(t \xrightarrow{\eta} t'$ and $u = u')$ or $(t = t'$ and $u \xrightarrow{\eta} u')$.

As we have done previously we use $\xrightarrow{\eta}{\!\!\!\!\!\rightarrow}$ for the reflexive transitive closure of this relation.

RExercise 24. *This exercise aims students to gain a first understanding of η-conversion.*

Show that for the untyped versions of the terms from Example 2.9 we obtain a connection via η-conversion.

A solution may be found on page 290.

We show that η-conversion is well-behaved with respect to renaming.

Proposition 2.28

If t and t' are λ-terms and y and z are variables with $z \notin (\mathrm{vars}\, t \cup \mathrm{vars}\, t')$ then $t \xrightarrow{\eta} t'$ implies $\mathrm{ren}^z_y\, t \xrightarrow{\eta} \mathrm{ren}^z_y\, t'$.

Proof. See page 249.

Proposition 2.29

If t, t' and u are λ-terms with $t \xrightarrow{\eta} t'$ and $t \sim_\alpha u$ then there exists a term u' with $t' \sim_\alpha u'$ and $u \xrightarrow{\eta} u'$.

Further if we have terms t, t' and u' with $t \xrightarrow{\eta} t'$ and $t' \sim_\alpha u'$ then there exists a term u with $t \sim_\alpha u$ and $u \xrightarrow{\eta} u'$.

Proof. *This exercise is a typical proof by induction over the structure of the term t, paying attention of why there is an η-conversion step. The two proofs are similar so students probably only want to think about one of them.*

See page 291.

We can picture this as follows:

$$\begin{array}{ccc} t & \sim_\alpha & u \\ {\scriptstyle\eta}\downarrow & & \\ t' & & \end{array} \quad \text{then there exists } u' \text{ with} \quad \begin{array}{ccc} t & \sim_\alpha & u \\ {\scriptstyle\eta}\downarrow & & \downarrow{\scriptstyle\eta} \\ t' & \sim_\alpha & u' \end{array}$$

as well as

$$\begin{array}{ccc} t & & \\ {\scriptstyle\eta}\downarrow & & \\ t' & \sim_\alpha & u' \end{array} \quad \text{then there exists } u \text{ with} \quad \begin{array}{ccc} t & \sim_\alpha & u \\ {\scriptstyle\eta}\downarrow & & \downarrow{\scriptstyle\eta} \\ t' & \sim_\alpha & u' \end{array}$$

2.7.4 $\alpha\beta\eta$-equivalence for the untyped λ-calculus

We want to use η-conversion to study the notion of terms being contextually equivalent. To do that, we need to combine it with the other syntactic notions of equivalence. There are three ways in which distinct terms might show 'the same' behaviour according to our rewriting relations:

- The terms might be α-equivalent, and we don't want to worry about names of bound variables.
- One term might β-reduce to the other, in which case any term that has the first term as a subterm β-reduces (up to α-equivalence) to a term that has the second term as a subterm in the same position.
- One term might η-convert to the other.

But we have to allow for any combination of these to occur. This motivates the following definition. You may want to recall that given any relation we can always find the smallest equivalence relation containing it by forming its reflexive symmetric transitive closure.

Definition 30: $\alpha\beta\eta$-equivalence

For λ-terms **$\alpha\beta\eta$-equivalence** is the smallest equivalence relation $\sim_{\alpha\beta\eta}$ containing $\alpha\beta$-equivalence and η-conversion.

We know how to form this smallest equivalence relation, and so we know that for terms t and t' we have that

$$t \sim_{\alpha\beta\eta} t' \quad \text{if and only if} \quad \exists n \in \mathbb{N},\ t = t_0, t_1, \ldots t_n = t'.$$
$$\forall 0 \leq i < n.\, t_i \sim_{\alpha\beta} t_{i+1} \text{ or } t_i \xrightarrow{\eta} t_{i+1} \text{ or } t_{i+1} \xrightarrow{\eta} t_i.$$

Exercise 25. *This is proof only requires an abstract understanding of 'the smallest equivalence relation with some property'.*

Show that $\alpha\beta\eta$-equivalence is the smallest equivalence relation containing α-equivalence, β-reduction and η-conversion. *Hint: Most of the work has already been done by proving Proposition 2.21.*

See page 292 for a solution.

2.7.5 $\alpha\beta\eta$-equivalence for the simply typed λ-calculus

We note first of all that we may adjust the notion of $\alpha\beta$ equivalence to typed preterms by adopting Definition 21 by using our previous definition for α-equivalence and β-reduction, and that Proposition 2.22 can be transferred to this setting.

We adjust the material regarding η-conversion to the simply typed λ-calculus by accounting for the fact that we are only interested in typeable terms. This requires us to take typing environments into account.

Definition 31: η-conversion

We define the relation of **η-conversion relative to a type environment** Γ, written $\xrightarrow{\eta}_\Gamma$, recursively as follows

bcVarη $\lambda x{:}\sigma.tx \xrightarrow{\eta}_\Gamma t$ if $\Gamma \vdash t{:}\sigma \to \tau$ is derivable and $x \notin \mathrm{fv}\, t$.

scAbsη $\lambda x{:}\sigma.t \xrightarrow{\eta}_\Gamma \lambda x{:}\sigma.t'$ if $t \xrightarrow{\eta}_{\Gamma,x:\sigma} t'$.

scAppη $tu \xrightarrow{\eta}_\Gamma t'u'$ if

- $t \xrightarrow{\eta}_\Gamma t'$, $u = u'$ and there are types σ, τ with $\Gamma \vdash t' : \sigma \to \tau$ and $\Gamma \vdash u : \sigma$ being derivable or
- $t = t'$ and $u \xrightarrow{\eta}_\Gamma u'$ and there are types σ, τ with $\Gamma \vdash t : \sigma \to \tau$ and $\Gamma \vdash u' : \sigma$ being derivable.

We connect η-conversion relative to Γ with typing judgements for that type environment, and we start doing so with the base case.

RExercise 26. *The aim of this exercise is for students to better understand the connection between η-conversion and typing a term.*

If $\Gamma \vdash t : \sigma \to \tau$ is derivable and x is a variable that does not occur free in t then $\Gamma \vdash \lambda x{:}\sigma.tx : \sigma \to \tau$ is also derivable.

See page 292 for a solution.

Proposition 2.30

If $t \xrightarrow{\eta}_\Gamma t'$ then there is a type τ such that $\Gamma \vdash t : \tau$ and $\Gamma \vdash t' : \tau$ are derivable.

Proof. *This is an induction over the reason why $t \xrightarrow{\eta} t'$.*

See page 292.

This tells us that η-conversion relative to some Γ is type-preserving, and also that the way we have defined η-conversion it only applies to terms, not preterms.

Before we can look at how η-conversion interacts with α-equivalence we have to transfer the renaming operation to typed terms.

Proposition 2.31

If t and t' are λ-terms and y and z are variables with $z \notin (\operatorname{vars} t \cup \operatorname{vars} t')$ then $t \xrightarrow{\eta}_{\Gamma, y:\sigma} t'$ implies $\operatorname{ren}_y^z t \xrightarrow{\eta}_{\Gamma, z:\sigma} \operatorname{ren}_y^z t'$.

Proof. See page 250.

We now show that α-equivalent terms may carry out parallel η-conversion steps for the simply typed setting, compare Proposition 2.29.

Proposition 2.32

If t and t' are terms, Γ is a type environment such that $t \xrightarrow{\eta}_\Gamma t'$ then for all terms u such that $u \sim_\alpha t$ there exists a term u' such that $t' \sim_\alpha u'$ and $u \xrightarrow{\eta}_\Gamma u'$.

Proof. *This proof follows in the footsteps of Proposition 2.29 checking that the additional requirements for η-conversion relative to a type environment are satisfied.*

See page 293.

We adjust the notion of $\alpha\beta\eta$-equivalence for typed terms.

Definition 32: $\alpha\beta\eta$-equivalence

We define the relation of **$\alpha\beta\eta$-equivalence relative to a type environment** Γ, written $\sim^\Gamma_{\alpha\beta\eta}$, to be the smallest equivalence relation containing $\sim_{\alpha\beta}$ and $\xrightarrow{\eta}_\Gamma$.

We show that when we build terms then we may replace $\alpha\beta\eta$-equivalent terms by each other and obtain $\alpha\beta\eta$-equivalent terms.

Proposition 2.33

Assume that t, t', u and u' are preterms. The following statements hold:

(a) If for a type environment Γ we have $t \sim^\Gamma_{\alpha\beta\eta} t'$ and there are types σ and τ such that both

$$\Gamma \vdash t, t' : \sigma \to \tau \qquad \text{and} \qquad \Gamma \vdash u : \sigma$$

are derivable then

$$tu \sim^\Gamma_{\alpha\beta\eta} t'u.$$

(b) If for a type environment Γ we have $u \sim^\Gamma_{\alpha\beta\eta} u'$ and there are types σ and τ such that both

$$\Gamma \vdash t : \sigma \to \tau \qquad \text{and} \qquad \Gamma \vdash u, u' : \sigma$$

are derivable then

$$tu \sim^\Gamma_{\alpha\beta\eta} tu'.$$

(c) If for a type environment Γ and variables x, x' and z with

$$z \notin \operatorname{vars} t \cup \operatorname{vars} t'$$

we have $\mathrm{ren}^z_x\, t \sim^{\Gamma,z:\sigma}_{\alpha\beta\eta} \mathrm{ren}^z_{x'}\, t'$ then

$$\lambda x{:}\sigma.\, t \sim^{\Gamma}_{\alpha\beta\eta} \lambda x'{:}\sigma.\, t.$$

(d) If for a type environment $\Gamma, x:\sigma$ we have $t \sim^{\Gamma}_{\alpha\beta\eta} t'$ then

$$\lambda x{:}\sigma.\, t \sim^{\Gamma}_{\alpha\beta\eta} \lambda x{:}\sigma.\, t'.$$

Proof. *The proofs required are not very long, but only if one adopts a suitable perspective of the equivalence relation—students might find them frustrating otherwise.*

See page 293.

2.8 The Function Model

In this section we compare terms of the simply typed λ-calculus to ordinary mathematical functions. We check whether there is a way of interpreting λ-terms as actual functions.

By now you may have become accustomed to the fact that getting to the point where we can actually give a formal definition there is a lot that has to be put in place first.

- We have to say which functions we are going to use, and that means we have to interpret types as sets in such a way that where appropriate the elements of these sets are functions.
- We have to find a way of coping with free variables in terms, and that means we have to define valuations.
- We have to have a way of adjusting valuations to cope with λ-abstraction.
- We are finally able to describe how to find a function that interprets a typeable term—but that means we have to do this relative to a type environment.

We point out here that the interpretation of a term in some model is also known as its **denotation**.

2.8.1 Idea

We give the basic idea here. For the moment we assume that we are only interested in interpreting typeable terms that do not contain free variables. That means we don't have to worry about interpreting terms that consist of just a variable.

We will write

$$[\![\sigma]\!]$$

for the interpretation of a type[2] and we assume that the interpretation of a function type is

$$[\![\sigma \to \tau]\!] = \mathsf{Fun}([\![\sigma]\!], [\![\tau]\!]),$$

[2] But note that below there's an additional parameter to worry about.

where

$$\mathsf{Fun}(A, B)$$

is the set of all functions with source set A and target set B.[3]

It's quite easy to see how this would work for terms that are created using the application rule. If we assume that the term t can be shown, with no assumptions, to have type $\sigma \to \tau$, and similarly u can be shown to have type σ then we expect the following to hold for their interpretations, which we write in a similar way to the interpretation of a type:

$$[\![t]\!] \in [\![\sigma \to \tau]\!] = \mathsf{Fun}([\![\sigma]\!], [\![\tau]\!]) \qquad \text{and} \qquad [\![u]\!] \in [\![\sigma]\!].$$

It then makes sense based on our intuition for λ-terms to interpret

$$[\![tu]\!] = [\![t]\!]([\![u]\!]) \in [\![\tau]\!]$$

since this means that we interpret t by a function and we apply that to the interpretation of u which is an element of the source set of that function.

When we look at terms created using the abstraction rule we realize that we really have to think about terms that include a free variable.

As an example to illustrate our idea let's look at what we have called the identity term

$$\lambda x : \sigma.\, x.$$

By the philosophy laid out above we expect that

$$[\![\lambda x : \sigma.\, x]\!] \in [\![\sigma \to \sigma]\!] = \mathsf{Fun}([\![\sigma]\!], [\![\sigma]\!]).$$

But which function should we pick? It should be built on our interpretation of the body of the abstraction, that is the term x, so we have to worry about how to interpret the term x. Clearly there is no way of picking an element of $[\![\sigma]\!]$, and in order to solve this problem we use a solution from logic: We require that we have a *valuation* which ensures that we know how to interpret free variables for a given term. We delay our description of how we then turn that into a function from $[\![\sigma]\!]$ to $[\![\sigma]\!]$.

2.8.2 Interpreting types

To start, we need to interpret the types as sets, and above we have given the basic idea. However we have not thought about how to interpret the base type ι. This can be any set we like! For example we might use the booleans or natural numbers, or even sets of λ-terms. It is also interesting to think about what happens when we use the empty set or a one-element set. Finally, we sometimes want to think about the meaning of a term with respect to *any* set, which is the same as considering a set whose elements we leave unspecified.

Definition 33: denotation of a type

Given a set X, the denotation of a type τ **relative to** X, written $[\![\tau]\!]^X$, is defined by recursion on τ as follows.

bcι $[\![\iota]\!]^X = X$.

[3] In mathematical writing this set is sometimes written B^A, but we avoid that notation here.

sc→ $[\![\rho \to \sigma]\!]^X = \mathsf{Fun}([\![\rho]\!]^X, [\![\sigma]\!]^X)$.

In Section 4.1 we develop some notation for sets with a finite source set which the reader may want to refer to before looking at the following example and the associated exercises. We also look there a little bit into the hierarchy of functions.

We may calculate the denotations of some of the simpler types relative to the set $\{0, 1\}$.

Example 2.12. We calculate, for $'iota = \{0, 1\}$,

$$
\begin{aligned}
&[\![(\iota \to \iota) \to \iota]\!]^{\{0,1\}} = \\
&\{\{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 0, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 0, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 0\},\\
&\ \{\{0 \mapsto 0, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 0\} \mapsto 1, \{0 \mapsto 0, 1 \mapsto 1\} \mapsto 1, \{0 \mapsto 1, 1 \mapsto 1\} \mapsto 1\}\}
\end{aligned}
$$

You might find it a bit ridiculous to actually write this down, but it is worth emphasizing that the denotation of a type is a concrete, familiar object, namely a set of ordinary functions.

RExercise 27. *The aim of this exercise is for students to acquire some familiarity with the interpretations of types.*

Compute $[\![\iota \to (\iota \to \iota)]\!]^{\{0,1\}}$

A solution may be found on page 294.

RExercise 28. *Again the aim of this exercise is for students to acquire some familiarity with the interpretations of types.*

How many elements does $[\![\iota \to (\iota \to \iota)]\!]^{\{0,1,2\}}$ have? Write down an example of an element.

See page 295 for a solution.

2.8.3 Valuations

It is convenient to be able to refer to all those elements which appear in the denotation of some type. For convenience, we assume that the denotations of

different types are disjoint,[4] and then we can just take the union of the denotations of all the types to assemble all these elements. We write

$$\mathbb{V}^X = \bigcup_{\tau \in \mathsf{Type}^{\to}} [\![\tau]\!]^X.$$

Why might we need a set that contains all these elements? When we interpret terms with free variables we need to have a way of supplying potential interpretations for those free variables, and we do this by using the idea of a *valuation*.

Example 2.13. The idea of using a valuation also occurs when we look at the interpretation of a formula in some logic. If we have a formula A containing the propositional variables $Z_1, Z_2 \ldots, Z_n$ then in order to give an interpretation of A we need to know how to interpret these variables, for which purpose we assume there is a valuation which gives us this information.

If we want to provide the boolean interpretation of A then we need a valuation

$$v : \{Z_1, Z_2, \ldots, Z_N\} \longrightarrow \{0, 1\},$$

whereas if we are giving the powerset interpretation relative to some set X we need a valuation

$$f : \{Z_1, Z_2, \ldots, Z_N\} \longrightarrow \mathcal{P}\, X.$$

For the former case we are then able, for example, to define the interpretation relative to v of $(\neg Z_1 \vee Z_2) \wedge Z_3$, $I_v A$, as

$$I_v((\neg Z_1 \vee Z_2) \wedge Z_3) = (\neg v Z_1 \vee v Z_2) \wedge v Z_3,$$

and more generally we may give a recursive definition of I_v where the base case is provided by the valuation v.

We say that the **support of a type environment** Γ, $\lfloor \Gamma \rfloor$, is given as

$$\lfloor \Gamma \rfloor = \{x \in \mathsf{Vars} \mid \exists \tau \in \mathsf{Type}^{\to}. \Gamma \vdash x : \tau \text{ derivable}\}.$$

Definition 34: valuation

Given a set X and a type environment Γ, an X-**valuation for** Γ is a function

$$\phi : \lfloor \Gamma \rfloor \longrightarrow \mathbb{V}^X$$

such that if $\Gamma \vdash x : \tau$ is derivable then $\phi x \in [\![\tau]\!]^X$.

When the set X is clear from the context we simply call these valuations for Γ.

We note that we may weaken Γ and obtain a type environment for which valuations do not change.

If we have a type environment Γ then we have that

$$\lfloor \Gamma, x : \tau \rfloor = \lfloor \Gamma, x : \sigma, x : \tau \rfloor,$$

and so X-valuations for $\Gamma, x : \tau$ have exactly the same source and target sets as X-valuations for $\Gamma, x : \sigma, x : \tau$.

[4] This is true for all standard ways of modelling set theory.

Proposition 2.34

If Γ is a type environment, x a variable and σ and τ are types then ϕ is an X-valuation for $\Gamma, x:\tau$ if and only if it is an X-valuation for $\Gamma, x:\sigma, x:\tau$.

Proof. This statement follows immediately from the observation that in our system the only type we may derive for the variable x under these circumstances is τ

By now we are familiar with the idea that as we process a λ-term recursively, we need to be able to deal with subterms with more free variables which form the bodies of abstractions. In order to address this situation we need a way to move from a valuation for one type environment to a valuation for another. For a type environment Γ, a valuation ϕ for Γ and a variable y we would like to be able to perform two tasks:

- If the variable y is already in the support of Γ, change the value it is assigned.
- If the variable y is not in the support of Γ, add it to the source set for ϕ.

We use the same notation for both tasks:

$$\phi[y \mapsto a]$$

is the valuation where the value ϕ assigns stays the same for all variables with the exception of y, which is mapped to a. Formally we set

$$\phi[y \mapsto a](x) = \begin{cases} a & \text{if } x = y \\ \phi x & \text{otherwise.} \end{cases}$$

Note that if $a \in [\![\tau]\!]^X$ and $y \notin \lfloor\Gamma\rfloor$ then $\phi[y \mapsto a]$ is a valuation for $\Gamma, y:\tau$.

We also note that if Γ is the empty type environment then it has empty support, and so a valuation for that type environment is the empty function. We do use the notation $[x \mapsto a]$ for extending the empty function with the assignment that sends x to a.

This is a very simple operation compared to substitution, or even renaming, because it operates on functions rather than on recursively defined syntactic objects.

2.8.4 Interpreting terms

Now we can define the denotations of terms by mapping them to elements of the denotations of their types. As indicated above we need to do this relative to a suitable type environment.

Definition 35: denotation of a term

Let Γ be a type environment, τ a type and t a term such that $\Gamma \vdash t:\tau$ is derivable. Given a set X and an X-valuation ϕ for Γ the Γ-**denotation of** t **relative to** X **and** ϕ, written

$$[\![(\Gamma, t)]\!]^X_\phi$$

is defined by recursion on t as follows.

bcVar$[\![\,]\!]$ $[\![(\Gamma, x)]\!]^X_\phi = \phi x$ if $x \in \mathsf{Vars}$.

scAbs$[\![\ ]\!]$ $[\![(\Gamma, \lambda x : \sigma.\, u)]\!]^X_\phi$ is the function from $[\![\sigma]\!]^X$ to $[\![\tau]\!]^X$ which sends an element $a \in [\![\sigma]\!]^X$ to $[\![(\Gamma x : \sigma, u)]\!]^X_{\phi[x \mapsto a]}$.

scApp$[\![\ ]\!]$ $[\![(\Gamma, rs)]\!]^X_\phi = [\![(\Gamma, r)]\!]^X_\phi([\![(\Gamma, s)]\!]^X_\phi)$.

When the term t can be typed with the empty type environment we don't require a valuation and we write

$$[\![t]\!]^X$$

for its denotation relative to X. Proposition 2.36 below tells us that in this situation it does not matter what valuation we use in this situation, so the notation is safe.

Example 2.14. We are now able to revisit the question asked above regarding the denotation of

$$\lambda x : \sigma.\, x.$$

We may derive $\vdash \lambda x : \sigma.\, x : \sigma \to \sigma$, and so we can ask what $[\![\lambda x : \sigma.\, x]\!]^X$ is. It is supposed to be the function from $[\![\sigma]\!]^X$ to $[\![\sigma]\!]^X$ given by the assignment

$$a \mapsto [\![(x : \sigma, x)]\!]^X_{[x \mapsto a]}$$

and the base case tells us that

$$[\![(x : \sigma, x)]\!]^X_{[x \mapsto a]} = \{x \mapsto a\}(x) = a,$$

so the interpretation of the identity term $\lambda x : \sigma.\, x$ is the identity function on $[\![\sigma]\!]^X$, justifying our name.

We note that working with the definition of the abstraction rule is a little awkward, but we can make that easier by doing the following: We know that the denotation of the given term is a function that expects an element of $[\![\sigma]\!]^X$ as argument, so instead of calculating the function 'as a whole', we can ask what it does when given an argument $a \in [\![\sigma]\!]^X$:

$$\begin{aligned} [\![\lambda x : \sigma.\, x]\!]^X a &= [\![(x : \sigma, x)]\!]^X_{[x \mapsto a]} && \text{scAbs}[\![\ ]\!] \\ &= \{x \mapsto a\}(x) && \text{bcVar}[\![\ ]\!] \\ &= a \end{aligned}$$

Note how the argument a moves into the valuation, and this is a way in which one may employ equational reasoning for denotations.

We look at an example containing a free variable to see how a valuation is used.

Example 2.15. Consider the term

$$\lambda x : \iota.\, f(fx).$$

We look at its interpretation where we use the set of real numbers $\mathbb{R}$ to interpret the base type ι.

In order to derive a type for this term we need to have a type environment that gives us a type for f, but we can also see that the only way in which we obtain a well-typed term is if $f : \iota \to \iota$.

We may therefore ask what we get when calculating the interpretation of the given term relative to that type environment,

$$[\![(f : \iota \to \iota, \lambda x : \iota.\, f(f\,x))]\!]^{\mathbb{R}}_{?},$$

but we need a valuation in place of ? to tell us how to interpret the variable f. This has to be an element of

$$\mathsf{Fun}(\mathbb{R}, \mathbb{R}),$$

so a function from $\mathbb{R}$ to itself. We use the valuation

$$f \mapsto (x \mapsto x^2),$$

that is, the function that maps an element of $\mathbb{R}$ to its square. For a slightly shorter notation, we are going to use

$$f \mapsto (-)^2.$$

Now we are ready to calculate

$$[\![(f : \iota \to \iota, \lambda x : \iota.\, f(f\,x))]\!]^{\mathbb{R}}_{[f \mapsto (-)^2]}.$$

We have a λ-abstraction, so the rules tell us that this is the function in

$$\mathsf{Fun}(\iota, \iota) = \mathsf{Fun}(\mathbb{R}, \mathbb{R})$$

which maps an element r of $\mathbb{R}$ to

$$[\![(f : \iota \to \iota\, x : \iota, f(f\,x))]\!]^{\mathbb{R}}_{\substack{[f \mapsto (-)^2] \\ [x \mapsto r]}}.$$

This is simply a term consisting of variables applied to each other, and we know that this is modelled by function application, so this is

$$(r^2)^2 = r^4.$$

In other words, the interpretation of the original term with the given valuation is the function

$$x \mapsto x^4.$$

Again we see what happens if instead of calculating the function we check what it does when applied to a suitable argument $r \in [\![\iota]\!]^{\mathbb{R}} = \mathbb{R}$, using Γ for the type environment $f : \iota \to \iota, x : \iota$.

$$\begin{aligned}
&[\![(f : \iota \to \iota, \lambda x : \iota.\, f(f\,x))]\!]^{\mathbb{R}}_{[f \mapsto (-)^2]} r \\
&\quad = [\![(\Gamma, f(f\,x))]\!]^{\mathbb{R}}_{\substack{[f \mapsto (-)^2] \\ [x \mapsto r]}} && \text{scAbs}\,[\![\,]\!] \\
&\quad = [\![(\Gamma, f)]\!]^{\mathbb{R}}_{\substack{[f \mapsto (-)^2] \\ [x \mapsto r]}}([\![(\Gamma, f\,x)]\!]^{\mathbb{R}}_{\substack{[f \mapsto (-)^2] \\ [x \mapsto r]}}) && \text{scApp}\,[\![\,]\!] \\
&\quad = [\![(\Gamma, f)]\!]^{\mathbb{R}}_{\substack{[f \mapsto (-)^2] \\ [x \mapsto r]}}([\![(\Gamma, f)]\!]^{\mathbb{R}}_{\substack{[f \mapsto (-)^2] \\ [x \mapsto r]}}([\![(\Gamma, x)]\!]^{\mathbb{R}}_{\substack{[f \mapsto (-)^2] \\ [x \mapsto r]}})) && \text{scApp}\,[\![\,]\!]
\end{aligned}$$

$$
\begin{aligned}
&= [\![(\Gamma, f)]\!]^{\mathbb{R}}_{\substack{[f\mapsto(-)^2]\\[x\mapsto r]}}([\![(\Gamma, f)]\!]^{\mathbb{R}}_{\substack{[f\mapsto(-)^2]\\[x\mapsto r]}} r) && \text{bcVar}[\![\,]\!]\\
&= [\![(\Gamma, f)]\!]^{\mathbb{R}}_{\substack{[f\mapsto(-)^2]\\[x\mapsto r]}}(r^2) && \text{bcVar}[\![\,]\!]\\
&= (r^2)^2 && \text{bcVar}[\![\,]\!]
\end{aligned}
$$

One can see that one might easily apply some shortcuts to this application—we only carried out each step separately for clarity on this occasion. We can see that by ensuring that we provide the denotation of a term with its expected arguments we may use equational reasoning.

Example 2.16. We consider a second example building on the previous one, for the term

$$(\lambda x : \iota.\, f(f\,x))y,$$

where again we use $\mathbb{R}$ to interpret the base type ι.

This term contains two free variables, f and y, so we need a type environment and a valuation for that type environment. For the former we use

$$f : \iota \to \iota, y : \iota,$$

and for the latter

$$
\begin{aligned}
f &\mapsto (-)^2\\
y &\mapsto \pi.
\end{aligned}
$$

By scApp$[\![\,]\!]$ we know that

$$
\begin{aligned}
&[\![(f : \iota \to \iota\, y : \iota, (\lambda x : \iota.\, f(f\,x))y)]\!]^{R}_{\substack{[f\mapsto(-)^2]\\[y\mapsto\pi]}}\\
&\quad = [\![(f : \iota \to \iota\, y : \iota, \lambda x : \iota.\, f(f\,x))]\!]^{R}_{\substack{[f\mapsto(-)^2]\\[y\mapsto\pi]}}([\![(f : \iota \to \iota\, y : \iota, y)]\!]^{R}_{\substack{[f\mapsto(-)^2]\\[y\mapsto\pi]}}).
\end{aligned}
$$

We know the former from the previous example, and by bc$[\![\,]\!]$ the second is equal to π, so the given term is interpreted by

$$\pi^4.$$

We note that the term as given β-reduces to $f(f\,y)$, and that

$$[\![(f : \iota \to \iota\, y : \iota, f(f\,y))]\!]^{R}_{\substack{[f\mapsto(-)^2]\\[y\mapsto\pi]}} = (\pi^2)^2 = \pi^4$$

as well. We see below that this is no accident—if t β-reduces to t' then, for the same set, type environment and valuation, the two terms have the same denotation.

RExercise 29. *The aim of this exercise is for students to familiarize themselves with how the denotation of terms works.*

Consider the term

$$\lambda x : \iota.\, \lambda y : \iota.\, y.$$

Where should its denotation live if once again we use the set of real numbers to interpret the base type? What is its denotation?

A solution may be found on page 295.

RExercise 30. *Again we would like students to carry out this exercise to become more familiar with how the denotations of terms are formed.*

Let $\sigma = \iota \to \iota$. Consider the term

$$\lambda f : \sigma.\, \lambda x : \iota.\, f(f\,x).$$

Where should the denotation of this term live if we again interpret use the set $\mathbb{R}$ to interpret the base type? Calculate its denotation.

What is the denotation of

$$(\lambda f : \sigma.\, \lambda x : \iota.\, f(f\,x))\lambda x : \iota.\, f\,x,$$

where we use the same set, type environment and valuation as in Example 2.15?

See page 296 for a solution.

2.8.5 Properties of denotations

Having defined the denotation of a term, we need to show that our definition is relevant to the notion of computation given by β-reduction and α-equivalence. We think of terms as syntactic expressions and their denotations as the underlying mathematical values which they mean. We have stipulated that β-reduction and α-equivalence give correct calculation steps, and this ought to mean that they do not change the value of the terms involved.

Usually in mathematics the values come first and a syntactic system for expressing and manipulating them comes second: consider how the rules for manipulating fractions are justified by reference to a pre-existing idea of what they should mean. But from our point of view the case of terms of the simply typed λ-calculus is more reminiscent of the history of the complex numbers, where the rules for manipulating them came first, and a geometric model in terms of the complex plane was discovered later. In such a case, we don't justify calculation rules by showing that they preserve values, but we justify our assignment of values by checking that they are preserved by the rules. We address this task in this section, building up to the main results about β-reduction and α-equivalence via a series of smaller propositions as usual.

We note that the interpretation of a type does not change if we apply suitable weakening to the underlying type-environment.

Proposition 2.35

Let $\Gamma, x : \sigma$ be a type environment, t a term and τ a type such that $\Gamma, x : \sigma \vdash t : \tau$

is derivable. If ϕ is an X-valuation for Γ then for every type ρ we have

$$[\![(\Gamma x : \sigma, t)]\!]^X_\phi = [\![(\Gamma x : \rho x : \sigma, t)]\!]^X_\phi.$$

Proof. This follows immediately by inspection of the definition of a denotation when we use the fact that $\Gamma, x : \rho, x : \sigma \vdash t : \tau$ is derivable by Proposition 2.4 when we invoke Proposition 2.34.

We observe that the denotation of a term does not depend on what the valuation does for variables which don't occur free in it.

Proposition 2.36

Let $\Gamma, x : \sigma$ be a type environment, t a term and τ a type such that $\Gamma, x : \sigma \vdash t : \tau$ is derivable. If ϕ is an X-valuation for Γ and $x \notin \mathrm{fv}\, t$ then

$$[\![(\Gamma x : \sigma, t)]\!]^X_\phi = [\![(\Gamma, t)]\!]^X_\phi$$

and for every $b \in [\![\sigma]\!]^X$

$$[\![(\Gamma x : \sigma, t)]\!]^X_{\phi[x \mapsto b]} = [\![(\Gamma, t)]\!]^X_\phi.$$

Proof. For the first statement note that we know from Proposition 2.4 that under the assumptions given $\Gamma \vdash t : \tau$ is derivable and therefore the Γ-denotation of t relative to X and ϕ is defined. Again we may establish this by inspection of the definition of a denotation.

For the second statement we carry out a proof by induction over the structure of t.

bcVar If the term is a variable then by assumption that variable can't be x and so the value of the valuation at x ha no impact on the interpretation of the term.

scAbs If the term is an abstraction there are two cases. Either x does not occur free in the body of the abstraction and the claim follows from the induction hypothesis or the variable being abstracted is x, in which case whatever value the original valuation gave to x is overwritten in the cause of assigning an interpretation to the abstraction.

scApp If the term is an application then x does not occur freely in either of the immediate subterms and the claim follows immediately from the induction hypothesis.

We now turn towards α-equivalence, beginning as usual with a statement about renaming.

Proposition 2.37

Suppose we have a type environment Γ, a term t and a type τ such that $\Gamma \vdash t : \tau$ is derivable. If z is a variable with $z \notin (\mathrm{fv}\, t \cup y)$, X is a set and ϕ an X-valuation

for Γ then

$$[\![(\Gamma y : \sigma, t)]\!]^X_\phi = [\![(\Gamma z : \sigma, \mathrm{ren}^z_y\, t)]\!]^X_{\phi[z \mapsto \phi y]}.$$

Proof. See page 250.

With a renaming result, we can show that the denotation of a term is invariant under α-equivalence.

Proposition 2.38

If Γ is a type environment, t a term and τ a type such that $\Gamma \vdash t : \tau$ is derivable and $t \sim_\alpha t'$ then for every X-valuation ϕ for Γ we have that the Γ-denotations relative to X and ϕ of t and t' agree, that is

$$[\![(\Gamma, t)]\!]^X_\phi = [\![(\Gamma, t')]\!]^X_\phi.$$

Proof. Proving this result is an assessed coursework exercise.

We turn to looking at how β-reduction relates to our interpretation. As usual we address substitution first.

Proposition 2.39

Let Γ be a type environment, t a term, x a variable and σ, τ types such that $\Gamma \vdash a : \sigma$ and $\Gamma, x : \sigma \vdash t : \tau$ are derivable. Then for every set X and every X-valuation ϕ for Γ

$$[\![(\Gamma x : \sigma, t)]\!]^X_{\phi[x \mapsto [\![(\Gamma, a)]\!]^X_\phi]} = [\![(\Gamma, t[a/x])]\!]^X_\phi.$$

Proof. See page 252.

Proposition 2.40

Let Γ be a type environment, t a term and τ a type such that $\Gamma, t : \tau$ is derivable. If $t \xrightarrow{\beta} u$ then for every set X and every X-valuation ϕ we have that the Γ-denotations relative to X and ϕ of t and u agree, that is

$$[\![(\Gamma, t)]\!]^X_\phi = [\![(\Gamma, u)]\!]^X_\phi.$$

Proof. *This is a proof by induction over the structure of t, and is good practice in using the formal definition of the denotation of a term.*
See page 296.

As usual we may now move from taking one β-reduction step to taking any finite number of these.

Corollary 2.41

Let Γ be a type environment, t a term and τ a type such that $\Gamma, t : \tau$ is derivable. If $t \twoheadrightarrow_{\beta} u$ then for every set X and every X-valuation ϕ we have

$$[\![(\Gamma, t)]\!]^X_\phi = [\![(\Gamma, u)]\!]^X_\phi.$$

Proof. This is immediate from the previous result—formally it is an induction over the number of reduction steps, and we establish that at each step the interpretation stays the same.

We have now shown that, since the denotational semantics is preserved by α-equivalence and β-reduction, it is a worthwhile technical tool for studying the λ-calculus. Seeing how λ-terms, as syntactic objects, are radically transformed by β-reduction, one might have wondered if there was any property of the term preserved by these transformations, and here we have one answer: the denotation of the term relative to any set is one such invariant property, expressing what all the terms connected by β-reduction (and α-equivalence) have in common.

2.8.6 Adequacy of the semantics

Now that we have established the basic correctness criterion for our denotational semantics, namely that it is preserved by computations, we turn to the question of how it interacts with contextual equivalence.

We hope that denotational semantics will let us reason about terms as if they were functions, by mapping them to ordinary functions. An important job we might want such a translation to do for us is to allow us to check the correctness of an optimized program. One way to do this would be to compute the denotation of the original program and the optimized version, and see if they come out to be the same. Since our semantics depends on a choice of a set X and a valuation, we would have to check that they are the same for *every* set and relevant valuation. If the two programs have the same denotation for all these translations, can we conclude that we may safely use the optimized version in place of the original?

Clearly, this is really asking whether, if two terms denote the same function for all valuations, they must be contextually equivalent. This property is called **computational adequacy**, since it is needed in order for the denotational semantics to have any practical use at all. In this section, we give the main idea of the proof that the function model of the simply typed λ-calculus is adequate.

Since the equality of denotations is something we need to assume for all sets and valuations, we can exploit that by defining, for each type environment, a special set and which makes our task as easy as possible. The idea is that while in general a denotational semantics is defined relative to an arbitrary set which might have nothing to do with the original terms, we are free to pick a set which happens to be a set of terms. Doing so allows us to form a bridge between the syntactic and semantic worlds.

Although contextual equivalence deals only with closed terms, we need a first order type environment Γ to be non-empty in order to provide suitable input and output values for testing terms. This means working relative to some type environment Γ.

Now the idea is to make use of Corollary 2.26 which tells us that in a first order context Γ, terms of type ι always β-reduce to Γ-concrete terms which cannot be

further reduced. This means that at type ι we have a collection of special terms which can serve as the denotations of terms of type ι, so we choose to work relative to this set.

The following definition is based on Definition 27.

Given a first order type environment Γ and a type σ we define $\widehat{\Gamma}_\sigma$ to be the set of Γ-concrete terms of type σ..

We use

$$\widehat{\Gamma}$$

as an abbreviation for the set $\widehat{\Gamma}_\iota$ of Γ-concrete terms of type ι which is the set relative to which our semantics is defined.

We want to show that for the function semantics built using $\widehat{\Gamma}$ has the adequacy property discussed above, that is, two terms that have the same interpretation for all valuations must be contextually equivalent.

We now need to define a special valuation to go with the set $\widehat{\Gamma}$ to prove this result, and here we make use of Corollary 2.26 again. We use the fact that Γ must be a first order type environment, so a variable given a type by Γ must be mapped to a function which takes some inputs from the set $\widehat{\Gamma}$ (which is the denotation of the type ι) and outputs an element of $\widehat{\Gamma}$.

Intuitively, we send a variable of type ι to itself, and a variable f of a more complicated first order type to the function which, when supplied with sufficiently many arguments from $\widehat{\Gamma}$ (which by definition are Γ-concrete terms) returns the Γ-concrete term which is the application of f to those terms. Although the valuation itself need only assign values to variables, it is convenient to define it in terms of an auxiliary function which is defined on all Γ-concrete terms of a given type.

Given a first order type environment Γ for each first order type σ, we define a function

$$\bar{\Gamma}_\sigma$$

from the set $\widehat{\Gamma}_\sigma$ of Γ-concrete terms of type σ to the interpretation $[\![\sigma]\!]^{\widehat{\Gamma}}$ of the type σ relative to the set $\widehat{\Gamma}$ by recursion on σ as follows.

bc$\bar{\Gamma}$ $\bar{\Gamma}_\iota x = x$.

sc$\bar{\Gamma}$ For a Γ-concrete terms t of type ι we define

$$\begin{aligned} \bar{\Gamma}_{\iota\to\sigma} t : [\![\iota]\!]^{\widehat{\Gamma}} &\longrightarrow [\![\sigma]\!]^{\widehat{\Gamma}} \\ u &\longmapsto \bar{\Gamma}_\sigma(tu) \end{aligned}$$

We may now use this function to define a particular valuation that we make use of below. For a variable x with the property that $\Gamma \vdash x : \sigma$ is derivable we may now think of having a default value in our semantics, and using that idea we set

$$\phi^{\widehat{\Gamma}} x = \bar{\Gamma}_\sigma x.$$

The family of auxiliary functions $\bar{\Gamma}_\sigma$ is useful because it lets us reason about the denotations of Γ-concrete terms.

Proposition 2.42

Let Γ be a first order type environment and t be a Γ-concrete term such that

$\Gamma \vdash t : \sigma$ is derivable. Then

$$[\![(\Gamma, t)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} = \bar{\Gamma}_\sigma t.$$

Proof. This is a proof by cases on the structure of t.

bcVar If the term is a variable, say x, then by assumption $\Gamma \vdash x : \sigma$ is derivable. Then we have

$$\begin{aligned} [\![(\Gamma, x)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} &= \phi^{\widehat{\Gamma}} x && \text{bcVar}[\![\,]\!] \\ &= \bar{\Gamma}_\sigma x && \text{def } \phi^{\widehat{\Gamma}} \end{aligned}$$

as required. Note that if the type of x happens to be ι then we know that the value in question is x.

scAbs This case cannot occur since t is Γ-concrete.

scApp If the term is an application then it must be of the form $f\,x$ where f is a variable of type $\iota \to \sigma$ and x is a variable of type ι. Then

$$\begin{aligned} [\![(\Gamma, f\,x)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} &= [\![(\Gamma, f)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} [\![(\Gamma, x)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} && \text{scApp}[\![\,]\!] \\ &= (\phi^{\widehat{\Gamma}} f)x && \text{bcVar}[\![\,]\!] \text{ and bcVar from above} \\ &= (\bar{\Gamma}_{\iota\to\sigma} f)x && \text{def } \phi^{\widehat{\Gamma}} \\ &= \bar{\Gamma}_\sigma(f\,x) && \text{def } \bar{\Gamma} \end{aligned}$$

as required.

The crucial property of this valuation is that when we find the denotation of a term t of type ι, we get a term which t β-reduces to. This would be a nonsensical thing to ask for for a general semantics, since the set X which interprets the type ι may have nothing at all to do with terms! However, in this special case, we have carefully used the set of Γ-concrete terms of type ι. This *is* a set of terms, so it makes sense to ask whether a term t β-reduces to its denotation, giving us the bridge between syntax and semantics we were hoping for.

Proposition 2.43

Let Γ be a first order type environment and t be a term such that $\Gamma \vdash t : \iota$ is derivable. Then

$$[\![(\Gamma, t)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} = \text{val}\, t.$$

Proof. We know from Corollary 2.26 that $t \xrightarrow{\beta\,*} \text{val}(t)$ and so we may invoke Proposition 2.40 to help us calculate as follows, taking note of the fact that $\text{val}\, t$ is Γ-concrete.

$$[\![(\Gamma, t)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} = [\![(\Gamma, \text{val}\, t)]\!]^{\widehat{\Gamma}}_{\phi^{\widehat{\Gamma}}} \qquad \text{Proposition 2.40}$$

$$
\begin{aligned}
&= \bar{\Gamma}_\iota(\mathrm{val}\, t) && \text{Proposition 2.42}\\
&= \mathrm{val}(t) && \text{def } \bar{\Gamma}_\iota.
\end{aligned}
$$

So not only do we have a default valuation for this particular semantics we also have a default value that interprets every term that can be given the base type in a first order type environment.

Putting the previous result together with Corollary 2.26 we obtain the following.

Corollary 2.44

Let Γ be a first order type environment and let t be a term such that $\Gamma \vdash t : \iota$ is derivable. Then

$$t \twoheadrightarrow_\beta [\![(\Gamma, t)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}.$$

Now that we have this connection between syntax and semantics, we can prove that the semantics has the crucial property of adequacy.

Corollary 2.45: computational adequacy of the function model

Let t and t' be closed terms such that $\vdash t : \tau$ and $\vdash t' : \tau$ are derivable. Suppose that for all sets X we have

$$[\![t]\!]^X = [\![t']\!]^X.$$

Then

$$t \simeq t'.$$

Proof. Assume that we have two terms t and t' with the given property. To show that these terms are contextually equivalent we would like to invoke Corollary 2.27, so let us assume we have a first order type environment Γ and a term f such that $\Gamma \vdash f : \tau \to \iota$. We begin by showing that in the context Γ, ft and ft' have the same interpretation in our model.

$$
\begin{aligned}
[\![(\Gamma, ft)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}} &= [\![(\Gamma, f)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}([\![(\Gamma, t)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}) && \text{scApp}\,[\![\,]\!]\\
&= [\![(\Gamma, f)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}([\![t]\!]^{\hat{\Gamma}}) && \text{Proposition 2.36}\\
&= [\![(\Gamma, f)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}([\![t']\!]^{\hat{\Gamma}}) && \text{assumption}\\
&= [\![(\Gamma, f)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}([\![(\Gamma, t')]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}) && \text{Proposition 2.36}\\
&= [\![(\Gamma, ft')]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}} && \text{scApp}\,[\![\,]\!]
\end{aligned}
$$

By Corollary 2.44 we have

$$ft \twoheadrightarrow_\beta [\![(\Gamma, ft)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}} \quad \text{as well as} \quad ft' \twoheadrightarrow_\beta [\![(\Gamma, ft')]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}} = [\![(\Gamma, ft)]\!]^{\hat{\Gamma}}_{\phi^{\hat{\Gamma}}}$$

and so ft and ft' both β-reduce to a common term which means that

$$ft \sim_{\alpha\beta} ft'.$$

We have now established that the denotational semantics does not lie when it tells us that two terms are indistinguishable: as programmers we will never be able to tell the terms apart.

Example 2.17. It can often by more enlightening, and indeed less work, to use adequacy to show that two terms are contextually equivalent, rather than using $\alpha\beta\eta$-equivalence, if we strictly apply the definitions step by step. Consider the term

$$\lambda a{:}\iota \to \iota \to \iota.\, (\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx)((\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx)a).$$

Intuitively, this term takes a two argument function, and applies a 'swap arguments' function twice. This should result in the same thing as not swapping the arguments at all, that is we expect this to be contextually equivalent to the term

$$\lambda a{:}\iota \to \iota \to \iota.\, a.$$

We show this using computational adequacy. To that end let X be an arbitrary set. We have to show that the denotations of the two terms relative to X are equal as functions. We know that, as the terms in question are of type

$$(\iota \to \iota \to \iota) \to (\iota \to \iota \to \iota)$$

we need to apply them to an element of

$$\mathsf{Fun}(X, \mathsf{Fun}(X, X))$$

in order to find out what they do. The output will again be an element of $\mathsf{Fun}(X, \mathsf{Fun}(X, X))$, so to find out which function we have got as an output, we need to apply it to two elements of X. Thus, let

$$f \in \mathsf{Fun}(X, \mathsf{Fun}(X, X)) \qquad \text{and} \qquad e_0, e_1 \in X.$$

We calculate

$$\begin{aligned}
&[\![\lambda a{:}\iota \to \iota \to \iota.\, (\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx)((\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx)a)]\!]^X f e_0 e_1 \\
&\quad = [\![(\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx)((\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx)a)]\!]^X_{[a \mapsto f]} e_0 e_1 \\
&\quad = [\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^X_{[a \mapsto f]} ([\![(\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx)a]\!]^X_{[a \mapsto f]}) e_0 e_1 \\
&\quad = [\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^X_{[a \mapsto f]} ([\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^X_{[a \mapsto f]} ([\![a]\!]^X_{[a \mapsto f]})) e_0 e_1 \\
&\quad = [\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^X_{[a \mapsto f]} ([\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^X_{[a \mapsto f]} (f)) e_0 e_1.
\end{aligned}$$

At this point, we could keep going methodically through the term by following the definition of $[\![\,]\!]^X$. However, we spot that we are dealing with the denotation of a repeated, self-contained term which looks like it should denote a sensible function in itself. Let's calculate what that function is by seeing what happens when we apply it to suitable arguments. It is of type $\iota \to \iota \to \iota$ so let $f \in \mathsf{Fun}(X, \mathsf{Fun}(X, X))$ and $d_0, d_1 \in X$ then we find[5]

$$\begin{aligned}
&[\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^X_{[a \mapsto f]} g d_0 d_1 \\
&= [\![\lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^X_{[a \mapsto f],[b \mapsto g]} d_0 d_1 \\
&= [\![\lambda y{:}\iota.\, byx]\!]^X_{\substack{[a \mapsto f],[b \mapsto g] \\ [x \mapsto d_0]}} d_1
\end{aligned}$$

$$
\begin{aligned}
&= [\![byx]\!]^{X}_{\substack{[a\mapsto f],[b\mapsto g]\\ [x\mapsto d_0],[y\mapsto d_1]}} \\
&= [\![by]\!]^{X}_{\substack{[a\mapsto f],[b\mapsto g]\\ [x\mapsto d_0],[y\mapsto d_1]}} \left([\![x]\!]^{X}_{\substack{[a\mapsto f],[b\mapsto g]\\ [x\mapsto d_0],[y\mapsto d_1]}}\right) \\
&= \left([\![b]\!]^{X}_{\substack{[a\mapsto f],[b\mapsto g]\\ [x\mapsto d_0],[y\mapsto d_1]}} \left([\![y]\!]^{X}_{\substack{[a\mapsto f],[b\mapsto g]\\ [x\mapsto d_0],[y\mapsto d_1]}}\right)\right)\left([\![x]\!]^{X}_{\substack{[a\mapsto f],[b\mapsto gg]\\ [x\mapsto d_0],[y\mapsto d_1]}}\right) \\
&= g d_1 d_0.
\end{aligned}
$$

Now we know what the behaviour of the function

$$[\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^{X}_{[a\mapsto f]}$$

is, we have escaped the world of λ-terms and are just dealing with ordinary functions. We calculate

$$
\begin{aligned}
&[\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^{X}_{[a\mapsto f]}([\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^{X}_{[a\mapsto f]}(f))e_0e_1 \\
&\quad = [\![\lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, byx]\!]^{X}_{[a\mapsto f]} f e_1 e_0 \\
&\quad = f e_0 e_1.
\end{aligned}
$$

Now

$$[\![\lambda a{:}\iota \to \iota \to \iota.\, a]\!]^{X} f e_0 e_1 = [\![a]\!]^{X}_{[a\mapsto f]} e_0 e_1 = f e_0 e_1$$

So we have calculated that for any set X, the denotations of the two terms are the same, and hence, the terms are contextually equivalent. The above calculation looks laborious because we strictly applied the definition of the semantics step-by-step. If we had tried to use $\alpha\beta\eta$-equivalence with the same level of detail, there would be a great deal of intricate syntactic work.

However, we do not expect students to go through a calculation like this in such detail, unless we explicitly ask for it. Instead, when calculating the denotation of a term, students should feel free to make as many steps at once as they feel confident they have got correct. A good rule of thumb is to treat bracketed sub-terms which are abstractions separately, to make sure that they are evaluated using the correct valuation for the given situation.

RExercise 31. *This exercise asks students to think about how to establish that two terms are contextually equivalent.*

Let

$$
\begin{aligned}
N &= \lambda p{:}\iota \to \iota \to \iota.\, [\lambda x{:}\iota.\, \lambda y{:}\iota.\, pyx] \\
C &= \lambda c{:}\iota \to \iota \to \iota.\, \lambda d{:}\iota \to \iota \to \iota.\, [\lambda x{:}\iota.\, \lambda y{:}\iota.\, c(dxy)y] \text{ and} \\
D &= \lambda c{:}\iota \to \iota \to \iota.\, \lambda d{:}\iota \to \iota \to \iota.\, [\lambda x{:}\iota.\, \lambda y{:}\iota.\, cx(dxy)]
\end{aligned}
$$

Prove that the terms

$$T = \lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, C(Na)(Nb)$$

[5] We have a valuation here that needs us to explicitly set the values of a number of variables and we omit the square brackets to improve readability.

and

$$U = \lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, N(Dab)$$

are contextually equivalent.

A solution appears on page 298.

2.8.7 Full abstraction

Having proved that denotationally equal terms are contextually equivalent: the semantics is adequate. We might wonder about the converse: if two terms are contextually equivalent, do they have the same denotation? If a semantics has both these properties it is called 'fully abstract'.

This is a much harder property to show than adequacy. After all, we could cook up a very syntactic semantics where we interpret terms as themselves, or perhaps themselves after we have done all possible beta reductions. Done right, this will give an adequate, if unenlightening, semantics. But clearly it remembers lots about the implementation details of terms, not just their observable behaviour. On the other hand, a fully abstract semantics means you have a mathematical theory of behaviour which completely hides all implementation details. Hence the name 'full abstraction'.

If you go to a theoretical conference and present a semantics for a language which fails to be adequate, you are likely to be laughed out of town. On the other hand, if you present a fully abstract semantics for an important language, you will be a hero!

Proving full abstraction for even a simple language takes a formidable technical apparatus, and generally involves peculiarities of the language which give rise to some mathematical miracles. For that reason, there is little point in studying a proof of full abstraction in detail in a course: there just isn't much of a general lesson to take away.

Nevertheless, if we take the two facts below for granted, we can link full abstraction for the simply typed λ-calculus to the more concrete equivalence relation of $\alpha\beta\eta$-equivalence.

2.8.8 Connecting $\alpha\beta\eta$-equivalence and contextual equivalence

Having defined $\alpha\beta\eta$-equivalence as a syntactic notion for terms, we should check whether it has any relevance to the notion of contextual equivalence which is the equivalence of terms which we really care about. The most important property is that two terms are related by $\alpha\beta\eta$-equivalence, then they are contextually equivalent.

Fact 1

Assume that t and t' are terms such that $\vdash t, t' : \sigma \to \tau$ is derivable and $t \xrightarrow{\eta} t'$. If Γ is a first order type environment and C is a $(\Gamma, \sigma \to \tau, \iota)$-context then

$$C\{t\} \xrightarrow{\beta\alpha} C\{t'\}.$$

Proposition 2.46

Let t and t' be closed terms, τ a type and Γ a first order context such that $\Gamma \vdash t.t' : \tau$. If $t \sim^{\Gamma}_{\alpha\beta\eta} t'$ then $t \simeq t'$.

Proof. See page 253.

This tells us that we can safely replace one term by another, $\alpha\beta\eta$-equivalent one, without having to worry about them behaving differently according to observations we could make about them at base type.

The second fact we require is that the converse implication of the previous result also holds.

Fact 2

Suppose t and t' are closed terms such that $\vdash t : \tau$ and $\vdash t' : \tau$. Then

$$t \simeq t' \qquad \text{implies} \qquad t \sim_{\alpha\beta\eta} t'.$$

The interested reader is directed to [DP01] for a proof of this fact, or to consult [BDS13]. One can also use the (extensive) theory developed in the latter reference to establish Fact 1.

2.8.9 Obtaining full abstraction

The results from the previous section tell us that we can show that the function model is fully abstract by proving that the denotation is preserved by $\alpha\beta\eta$-equivalence. Since we proved above that α-equivalence and β-reduction preserve the semantics as a basic correctness check, the only work remaining is to show that η-conversion does too. We look at how η-conversion connects with our interpretation.

Proposition 2.47

If Γ is a type environment, t a term and σ, τ types such that $\Gamma \vdash t : \sigma \to \tau$ is derivable and $x \notin \mathrm{fv}\, t$ then if X is a set and ϕ an X-valuation we have

$$[\![(\Gamma, \lambda x : \sigma.\, tx)]\!]^X_\phi = [\![(\Gamma, t)]\!]^X_\phi.$$

Proof. *Carrying out this proof means delving a little deeper into the denotation of terms.*
See page 299.

Proposition 2.48

If Γ is a type environment and t and t' are terms with $t \xrightarrow{\eta}_\Gamma t'$ then for every set X and X-valuation ϕ

$$[\![(\Gamma, t)]\!]^X_\phi = [\![(\Gamma, t')]\!]^X_\phi.$$

Proof. *This is a proof by induction over the structure of t, and is good practice in using the formal definition of the denotation of a term.*
See page 299.

Theorem 2.49

If Γ is a type environment and t and t' are terms with $t \sim^{\Gamma}_{\alpha\beta\eta} t'$ then for every set X and every X-valuation ϕ

$$[\![(\Gamma, t)]\!]^X_\phi = [\![(\Gamma, t')]\!]^X_\phi.$$

Proof. This follows from the fact that $\sim^{\Gamma}_{\alpha\beta\eta}$ is the smallest equivalence relation containing α-equivalence, β-reduction and η-conversion, which means that two terms are equivalent if and only if we can connect them by a finite sequence of terms which is such that any two neighbouring terms are in the symmetric closure of one of the given relations. Because of Propositions 2.38 and 2.40 and Proposition 2.48 we know that any two neighbouring terms must have the same interpretation, and so the interpretation stays the same along the given sequence.

This is an important result in its own right, since it tells us that $\alpha\beta\eta$-rewriting gives a complete equational reasoning system for the function model. To put it another way, if you ever wonder whether two functions are equal, and you spot that they are definable in the simply typed λ-calculus, we can prove they are equal by using just these three rewriting rules. But together with Fact 2 it takes on a more grandiose meaning: full abstraction for the function model.

Corollary 2.50: full abstraction for the function model

Let t and t' be closed terms such that $\vdash t : \tau$ and $\vdash t' : \tau$ are derivable. Suppose that

$$t \simeq t'.$$

Then for all sets X we have

$$[\![t]\!]^X = [\![t']\!]^X.$$

Proof. By Fact 2 the assumption $t \simeq t'$ gives us $t \sim_{\alpha\beta\eta} t'$, and then by Theorem 2.49 for all sets X and valuations ϕ, we have $[\![t]\!]^X_\phi = [\![t']\!]^X_\phi$.

In a way, these results tell us that the simply typed λ-calculus is a very simple language. Not only does strong normalization mean that every program terminates, which rules out many algorithms, but Theorem 2.49 tells us that we can even prove equality of programs *on all inputs* by a finite number of applications of three rewriting rules. These properties make the simply typed λ-calculus unsuitable as a general purpose programming languages, although they can be an advantage in a domain-specific language, or a well behaved fragment of a more powerful language.

Example 2.18. We demonstrate how to use full abstraction to argue about contextual equivalence. Consider the terms

$$\lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, a((bxy)y)$$

and

$$\lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, b((axy)y).$$

When we use the encoding for the booleans from RExercise 9 these terms encode conjunction,[6] with the first term encoding $A \wedge B$ and the second $B \wedge A$. We know that as formulae, these are semantically equivalent, and so it is tempting to assume that the terms will be contextually equivalent as well.

To work this out based on the definition of contextual equivalence is hard since that quantifies over all contexts of a certain kind. It usually is easier instead to think about the denotations of such terms.

It's a priori not clear what set X we should use to interpret the base type, so for the moment we assume we have an arbitrary such set. Both these are terms have a lot of abstractions, so their interpretations will take a lot of inputs: two functions from $\mathsf{Fun}(X, \mathsf{Fun}(X, X))$ and two inputs from X, producing an output from X.

We can see that for the term on the left, given inputs two functions f and g and two elements s and t of X, will output

$$f(g(s)(t))(t)$$

while the term on the right will output

$$g(f(s)(t))(t),$$

and writing this down using actual functions gives the idea that maybe we can find such functions and suitable inputs that produce different outputs. We also get the impression that maybe we don't need a very large set X for this.

We are more used to thinking of functions of this type as being akin to binary operations, and if we allow ourselves to write the expressions from above in this manner, using infix notation, we get

$$(s \oplus t) \otimes t \qquad \text{and} \qquad (s \otimes t) \oplus t.$$

Let $X = \{0, 1, 2\}$, and let $\oplus$ be addition modulo 3 and $\otimes$ be multiplication modulo 3. Now if $s = 0$ and $t = 2$ we have

$$(0 \oplus 2) \otimes 2 = 2 \otimes 2 = 1,$$

while

$$(0 \otimes 2) \oplus 2 = 0 \oplus 2 = 2.$$

Hence we have found a set $X = \{0, 1, 2\}$ and inputs f, g, 0 and 2 so that

$$[\![\lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, a((bxy)y)]\!]^X(f)(g)(0)(2) = 1,$$

while

$$[\![[\lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, b((axy)y)]\!]^X(f)(g)(0)(2) = 2,$$

so that the terms have different denotations, and therefore are not contextually equivalent by Theorem 2.49.

It is worth mentioning here that the two given terms are contextually equivalent if they are only applied to closed terms.

RExercise 32. *This is another exercise asking students to use available tools to establish that two terms are, or are not, contextually equivalent.*

Are the following terms contextually equivalent? Justify your answer.

$$\lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, \lambda c{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, a(bx(cxy))y$$

and

$$\lambda a{:}\iota \to \iota \to \iota.\, \lambda b{:}\iota \to \iota \to \iota.\, \lambda c{:}\iota \to \iota \to \iota.\, \lambda x{:}\iota.\, \lambda y{:}\iota.\, a(bx(a(cxy)y))(a(cxy)y).$$

A solution appears on page 300.

We have seen that our type system rules out the ingenious hacks which let us get recursion into the untyped λ-calculus. But they also give us a platform for building a more powerful language on top of, if we can work out how to add recursion back in in such a way that we get a reasoning principle. That is the programme we pursue in the next chapter, for the language PCF.

[6] Note that in RExercise 9 we gave untyped terms; the ones given there are not typeable, which explains the discrepancy with the ones given here.

Chapter 3

PCF

3.1 A Realistic Functional Programming Language

The simply typed λ-calculus is not a realistic programming language in many ways. The most salient is the lack of recursion, which is part of the reason we get properties like strong normalization, but which also prevents many programs from being written. It is perhaps best thought of as a template which can be modified in various ways to produce programming languages. It can be viewed as embodying the fundamental operation of substitution, which is at the core of any functional language, and explaining how this interacts with typing. In this chapter, we describe the language PCF by adding—but also removing—aspects of the simply typed λ-calculus. Before we get to recursion, there are a few other changes to make.

3.1.1 Built in data and an evaluation strategy

One of the ways in which the simply typed λ-calculus is unrealistic is its lack of any built-in datatypes. This is an advantage for a general study, since it doesn't commit us to particular details, but it is not suitable when we want to write programs. In PCF, instead of an abstract base type ι, we have a base type nat intended to model the natural numbers. This means we need a way of writing numbers. We add constants $\bar{0}$ and $\bar{\mathsf{s}}$ to model zero and successor so that the natural number n can be written

$$\bar{\mathsf{s}}^n\bar{0}.$$

To work with numbers, we need a way to inspect them. Since we are aiming to use recursion to compute, we need to be able to tell if an elements of nat is $\bar{0}$ or a successor of something, and if it is a successor, we would like to be able to find the number it is the successor of. To that end, we add two built-in operations ifz and pred to the language. The operation ifz takes three arguments of type nat, returning the second if the first is $\bar{0}$ and the third otherwise, while pred is just the predecessor function (which given $\bar{0}$ outputs $\bar{0}$). The typing rules and β-reduction rules for these new constants are given formally in Definition 37 and Definition 38, but we preview them here in order to make some remarks. The typing rules are as follows:

$$\frac{}{\Gamma \vdash \bar{0} : \mathsf{nat}}\ \text{zero} \qquad \frac{\Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \bar{\mathsf{s}}n : \mathsf{nat}}\ \text{succ}$$

$$\frac{\Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \mathsf{pred}\, n : \mathsf{nat}}\ \mathrm{pred} \qquad \frac{\Gamma \vdash n : \mathsf{nat} \quad \Gamma \vdash p : \mathsf{nat} \quad \Gamma \vdash q : \mathsf{nat}}{\Gamma \vdash \mathsf{ifz}\, npq : \mathsf{nat}}\ \mathrm{ifz}$$

We adopt bracketing rules for $\bar{\mathsf{s}}n$ and $\mathsf{pred}\, n$ by treating them analogously to application, except that instead of a left-hand term, they have a special constant. We write $\bar{\mathsf{s}}^n\bar{0}$ to mean the term in which $\bar{\mathsf{s}}$ is applied to $\bar{0}$ n times, which is how we represent the number n.

Similarly ifz is bracketed as if it was a term of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$, even though it is a special constant (in particular, we never define terms in which it is not supplied with all three arguments). However, we usually opt to add brackets for clarity when using these terms. For ifz in particular, we generally use one style of bracket for the first argument, and another style to enclose the second and third arguments which represent the 'if' and 'else' cases. So we would usually write $\mathsf{ifz}\, npq$ as $\mathsf{ifz}\, n[p][q]$ or $\mathsf{ifz}\,(n)[p][q]$ wherever this aids clarity.

We also preview the β-reduction rules for the new constants; they come in two groups. First, the built-in operations are defined by cases as follows.

bcPredβ $\mathsf{pred}\,\bar{0} \xrightarrow{\beta} \bar{0}$, $\mathsf{pred}\,\bar{\mathsf{s}}(\bar{\mathsf{s}}^n\bar{0}) \xrightarrow{\beta} \bar{\mathsf{s}}^n\bar{0}$.

bcIfzβ $\mathsf{ifz}\,\bar{0}tu \xrightarrow{\beta} t$, $\mathsf{ifz}\,\bar{\mathsf{s}}(\bar{\mathsf{s}}^n\bar{0})tu \xrightarrow{\beta} u$.

Note that $\bar{0}$ and $\bar{\mathsf{s}}$ do not have corresponding rules to the above, because they just represent data; they are not supposed to 'do' anything. On the other hand, if we end up with a term which is the successor of something, that 'something' might still need to finish computing before we can see which number it is, so we want to allow computation to happen inside a term of the form $\bar{\mathsf{s}}t$. Similarly, pred and ifz don't know what to do with arguments which are not of the form $\bar{\mathsf{s}}^n\bar{0}$, so in those cases we need to let the argument continue computing, in the hope that it eventually becomes a term of a suitable form.

scSuccβ If $t \xrightarrow{\beta} t'$ then $\bar{\mathsf{s}}t \xrightarrow{\beta} \bar{\mathsf{s}}t'$.

scPredβ If $t \xrightarrow{\beta} t'$ then $\mathsf{pred}\, t \xrightarrow{\beta} \mathsf{pred}\, t'$.

scIfzβ If $t \xrightarrow{\beta} t'$ then $\mathsf{ifz}\, trs \xrightarrow{\beta} \mathsf{ifz}\, t'rs$.

On the surface, adding all this new syntax makes proofs more laborious as there are now many more step cases to deal with. However, the added concreteness actually *simplifies* many matters. In the simply typed λ-calculus, we faced a serious problem when trying to specify the notion of contextual equivalence, because we have no built-in elements of type ι to serve as inputs and outputs of the experiments we perform to test the behaviour of terms. This lead to a subtle theory which makes careful use of type environments to provide input and output data. None of this is necessary when the language has a base type with built-in elements.

The reader might feel that adding only one type (of natural numbers) does not really give us a 'realistic' language since many other important datatypes are lacking. This restriction is, of course, to keep the language simple so that our study of it does not get swamped in details. But it is an honest restriction in the sense that adding more datatypes would not anything *intellectually* new once we have seen how to add one concrete datatype.

The second change we make to go from the simply typed λ-calculus is that we *restrict* β-reduction. This is one way in which we definitely *remove* something

when passing to a more realistic language. The full definition of β-reduction is an account of when a calculation with λ-terms is correct, and it is as liberal as possible about which calculation steps to perform when. But in a programming language, we usually expect program execution to be deterministic. That is, the language must make a choice about which β-reduction step to take in a given situation. Such a choice is called an 'evaluation strategy'. In PCF, we choose the 'leftmost, outermost' strategy which says that a subterm of a large term can only reduce if it is allowed to by one of the new cases above, or the whole term is an application and the subterm is the left-hand side. This means that the arguments of applications are banned from β-reducing, and so are the bodies of abstractions. In other words, for abstraction and application, Definition 38 in the next section lacks the rules (from the original Definition 9) in red below

bcVarβ $(\lambda x.\, t)a \xrightarrow{\beta} t[a/x]$.

scAbsβ If $t \xrightarrow{\beta} t'$ then $\lambda x.\, t \xrightarrow{\beta} \lambda x.\, t'$.

scAppβ if $t \xrightarrow{\beta} t'$ then $tu \xrightarrow{\beta} t'u$ and if $u \xrightarrow{\beta} u'$ then $tu \xrightarrow{\beta} tu'$.

Again, this change simplifies many things: confluence becomes a triviality since there is never a choice about how to reduce, and the irreducible terms (i.e. those which can't β-reduce) become easy to classify. However, it ought to make us pause and wonder if it means there are strange elements of our types which can only be shown to be equal to ordinary elements by the forbidden β-reduction rules! Our interest in seeing a denotational semantics which tells us how to think of terms increases.

3.1.2 Reintroducing recursion

Of course, the main change we need to make to go from the simply typed λ-calculus to a realistic programming language is to add *recursion* which allows us to write complicated programs, including search procedures which might run forever if there is no solution to the problem we have set them. We saw a form of recursion in the untyped λ-calculus, but that relies on self-referential terms which are not typeable and are difficult to reason about conceptually. So we seek a way to add recursion back in in a principled way which has a sensible type.

Let us consider an example. In ordinary maths we could define a (very boring!) recursive function from $\mathbb{N}$ to $\mathbb{N}$ as follows

$$f\,n = \begin{cases} 0 \text{ if } n = 0 \\ f(n-1) \text{ otherwise.} \end{cases}$$

Much of the body of the definition can already be translated into a PCF term on the basis of the features of the language we have already described, but the recursion itself seems to rely on the fact that we are able to name the function f and refer to this name in the definition. It complicates a programming language if we have to deal with things like named declarations. We want to be able to write down a definition of the function above without relying on creating a 'global' name f. Temporarily, while we investigate how such a thing should behave, let's use the notation

$$\text{"the recursive function } f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]\text{"}$$

for an imaginary PCF term which represents this function. Once we have‘ finished investigating the behaviour of such a term, we can replace this awkward notation with some real PCF syntax.

At this point we pause and remark that f is a bound, local name in the expression above: whereas the traditional definition introduces a name f we could refer back to outside of the definition, we have rephrased it so that this is no longer grammatically possible, which matches our λ-calculus based programming language better. Since the λ-calculus is supposed to be a universal theory of bound names, we expect that once we formalize the above expression, the local name f will turn into a $\lambda f. \ldots$ somehow. But we leave this thought to one side for now.

We would like to be able to ‘calculate which function this really is’ in the sense that we would like to express the above function in the form $\lambda n : \mathsf{nat}. \ldots$ so that it can be applied to an argument by β-reduction. To see how to fill in the blank, we should look ahead to what we expect this β-reduction to produce, by plugging n into the body of the definition ourselves. Naïvely, it seems like we should have

$$\begin{aligned} &\text{“the recursive function } f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]\text{”}n \\ &\quad \xrightarrow{\beta} \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n] \end{aligned}$$

but this is problematic because it contains a free f. This is why we usually make use of global names in order to define recursive functions! But there is a way of fixing the problem. We can just replace this free f with the definition of f, so we would have

$$\begin{aligned} &\text{“the recursive function } f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]\text{”}n \\ &\quad \xrightarrow{\beta} \mathsf{ifz}\,(n)[0][(\text{“the recursive function } f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]\text{”})\mathsf{pred}\,n]. \end{aligned}$$

This lets us solve the problem of how we expect the recursive definition to look after it has been rewritten into the form of an abstraction; we want

$$\begin{aligned} &\text{“the recursive function } f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]\text{”} \\ &\quad \xrightarrow{\beta} \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0] \\ &\qquad\qquad [(\text{“the recursive function } f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]\text{”})\mathsf{pred}\,n]. \end{aligned}$$

This is reminiscent of ‘unrolling a loop’ in an imperative language. Students might want to compare this with the transition rule for loops in the **While** programming language from COMP112, which is

$$\langle \text{while b do S}, \sigma \rangle \quad \Rightarrow \quad \langle \text{if b then (S;while b do S) else skip}, \sigma \rangle.$$

But what is the final expression above? It is nothing but the body

$$\lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]$$

of the recursive definition, with

$$\text{“the recursive function } f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]\text{”}$$

substituted in place of f. In other words, it is what we would get if we were to β-reduce

$$(\lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n])$$

("the recursive function $f = \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]$").

Aha! Here is the $\lambda f \ldots$ we were expecting to show up. It looks like the body of a recursive expression is best thought of as a *higher order function*, one which takes a function as an input which it uses to interpret the recursive call. This lets us at least come up with a sensible notation for the idea we are formalizing. We write the recursive function above as

$$\mathsf{rec}\,\lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n]$$

Then we want the β-reduction rule of rec to express the fact that the function to use in the recursive call is *the recursively defined function itself*. In other words, in our running example we see that the β-reduction rule for rec must be such that

$$\begin{aligned}&\mathsf{rec}\,(\lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n])\\ &\quad \xrightarrow{\beta} (\lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n])\\ &\qquad (\mathsf{rec}\,\lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda n : \mathsf{nat}.\, \mathsf{ifz}\,(n)[0][f\,\mathsf{pred}\,n])\end{aligned}$$

What have we learned about recursion in general from this discussion? The body of a recursive definition should be an abstraction, where we interpret the variable abstracted over as the term to be used to interpret the recursive call. What makes a recursive definition special is that it supplies *itself* to its body as the term to use in a recursive call. From this we can deduce something about the types involved. The body, being an abstraction, must have a function type, and the recursive function itself must be a suitable input for an abstraction of this type. That is, to use recursion to specify an element of type τ, we need to supply a term t of type $\tau \to \tau$, and then the desired term of type τ is $\mathsf{rec}\,t$. Then the β-reduction rule for $\mathsf{rec}\,t$ can safely state that $\mathsf{rec}\,t \xrightarrow{\beta} t\,\mathsf{rec}\,t$.

So the typing rule for rec is

$$\frac{\Gamma \vdash g : \tau \to \tau}{\Gamma \vdash \mathsf{rec}\,g : \tau}\,\mathrm{rec}$$

and it's reduction rule is

bcRecβ $\mathsf{rec}\,t \xrightarrow{\beta} t\,\mathsf{rec}\,t$.

As for the other new constants, we bracket expressions involving rec as if they were applications of something called rec to a term, even though rec by itself will never be allowed as a term. We often add extra brackets where this helps with readability. Now we can put all these changes together into a formal definition of PCF.

3.2 The Language PCF

In this section we give a formal definition of the language PCF, formalizing the discussion of the previous section. After looking at some example programs, we investigate some basic but important properties of the language, which have implications for the denotational semantics.

3.2.1 Terms, types and β-reduction

To formalize PCF, we proceed along familiar lines. Just like the simply typed λ-calculus, we begin with a collection of preterms which might not typecheck, define typing rules which determine the set of terms, and then give a definition of β-reduction which explains how these terms compute.

Definition 36: PCF preterms

The set of PCF preterms PCFTrm is defined recursively as follows.

bcVar If $x \in \mathsf{Vars}$ then $x \in \mathsf{PCFTrm}$.

scAbs If $x \in \mathsf{Vars}$ and $u \in \mathsf{PCFTrm}$ then $\lambda x{:}\sigma.\, u \in \mathsf{PCFTrm}$.

scApp If $r, s \in \mathsf{PCFTrm}$ then $rs \in \mathsf{PCFTrm}$.

bcZero $\bar{0} \in \mathsf{PCFTrm}$.

scSucc If $t \in \mathsf{PCFTrm}$ then $\bar{\mathsf{s}}t \in \mathsf{PCFTrm}$.

scPred If $t \in \mathsf{PCFTrm}$ then $\mathsf{pred}\, t \in \mathsf{PCFTrm}$.

scIfz If $r, s, t \in \mathsf{PCFTrm}$ then $\mathsf{ifz}\, trs \in \mathsf{PCFTrm}$.

scRec If $t \in \mathsf{PCFTrm}$ then $\mathsf{rec}\, t \in \mathsf{PCFTrm}$.

We note that the notion of α-equivalence can be extended to PCF terms, but since in this chapter we concentrate on the new aspects of the language we do not give details.

We reserve the name *PCF term* for those that are typeable via the following rules. In general, when we refer to terms or types in this chapter we are referring to those for PCF.

Definition 37: typing rules for PCF

The typing rules for PCF are as follows.

$$\frac{}{\Gamma \vdash \bar{0} : \mathsf{nat}}\ \text{zero} \qquad \frac{\Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \bar{\mathsf{s}}n : \mathsf{nat}}\ \text{succ}$$

$$\frac{\Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \mathsf{pred}\, n : \mathsf{nat}}\ \text{pred} \qquad \frac{\Gamma \vdash n : \mathsf{nat} \quad \Gamma \vdash p : \mathsf{nat} \quad \Gamma \vdash q : \mathsf{nat}}{\Gamma \vdash \mathsf{ifz}\, npq : \mathsf{nat}}\ \text{if}$$

$$\frac{\Gamma \vdash g : \tau \to \tau}{\Gamma \vdash \mathsf{rec}\, g : \tau}\ \text{rec}$$

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}\ \text{ax}$$

$$\frac{\Gamma, x{:}\sigma \vdash t : \tau}{\Gamma \vdash (\lambda x{:}\sigma.\, t) : \sigma \to \tau}\ \text{abs} \qquad \frac{\Gamma \vdash f : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash ft : \tau}\ \text{app}$$

We may extend the notion of capture-avoiding substitution to PCF recursively and we do not give that definition here. The only interesting case is the abstraction

step case, which is dealt with already in the λ-calculus. In proofs we assume it behaves as expected on the new terms, and indeed one could write down a formal definition by studying how it is assumed to behave in proofs.

Definition 38: β-reduction for PCF

We define the relation of β-reduction, denoted by $\xrightarrow{\beta}$, on PCF preterms as follow.

bcVarβ $(\lambda x{:}\sigma.\, t)a \xrightarrow{\beta} t[a/x]$.

bcPredβ $\mathsf{pred}\,\bar{0} \xrightarrow{\beta} \bar{0}$, $\mathsf{pred}\,\bar{s}(\bar{s}^n\bar{0}) \xrightarrow{\beta} \bar{s}^n\bar{0}$.

bcIfzβ $\mathsf{ifz}\,\bar{0}tu \xrightarrow{\beta} t$, $\mathsf{ifz}\,\bar{s}(\bar{s}^n\bar{0})tu \xrightarrow{\beta} u$.

bcRecβ $\mathsf{rec}\,t \xrightarrow{\beta} t\mathsf{rec}\,t$.

scAppβ If $t \xrightarrow{\beta} t'$ then $tu \xrightarrow{\beta} t'u$.

scSuccβ If $t \xrightarrow{\beta} t'$ then $\bar{s}t \xrightarrow{\beta} \bar{s}t'$.

scPredβ If $t \xrightarrow{\beta} t'$ then $\mathsf{pred}\,t \xrightarrow{\beta} \mathsf{pred}\,t'$.

scIfzβ If $t \xrightarrow{\beta} t'$ then $\mathsf{ifz}\,trs \xrightarrow{\beta} \mathsf{ifz}\,t'rs$.

As before we use $\xrightarrow{\beta}{\!\!\!\!\!\!\rightarrow}$ for the reflexive transitive closure of $\xrightarrow{\beta}$. Note that although we use the same name as before, this is a different relation. Not only is it defined on a different set, but it behaves quite differently as discussed in the previous section. Nevertheless while the *details* of the PCF β-reduction relation are quite different from those of the λ-calculus, the *ideas* needed to reason about this relation are essentially the same as the ones involved in studying the original.

Note that in general it is *not* the case that if we have terms t, t' of the simply typed λ-calculus with $t \xrightarrow{\beta} t'$ that t reduces to t' when we view them as being PCF terms.

RExercise 33. *This exercise points to differences between β-reduction in the λ-calculus and PCF.*

Give a term in the simply typed λ-calculus that contains a redex, but that cannot reduce if viewed as a term of PCF.

See page 301 for a solution.

It is, however, the case that if we have two terms t and t' from the simply typed λ-calculus, and $t \xrightarrow{\beta} t'$ if we view them as PCF terms, then they we also have $t \xrightarrow{\beta} t'$ in the simply typed λ-calculus. We now look at some example PCF programs to get a feel for what the definition of the language means.

Example 3.1 (Addition in PCF). Let us consider how we might implement addition in PCF. A mathematical definition of addition in terms of successor

(represented by adding 1) is given by the recursive definition

$$\operatorname{add} mn = \begin{cases} n & m = 0 \\ 1 + (\operatorname{add}(m-1)n) & \text{otherwise.} \end{cases}$$

To convert this to a PCF term, we have to think of add as an input to a higher order function. Clearly, the type of the term corresponding to add must be $\mathsf{nat} \to \mathsf{nat}$. The higher order function then expends the definition of the recursive function, using the supplied input where the recursive call occurs. So the higher order term in question is

$$\lambda a : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.\ \lambda x : \mathsf{nat}.\ \lambda y : \mathsf{nat}.\ \mathsf{ifz}\, x [y][\bar{\mathsf{s}}(a(\mathsf{pred}\, x)y)]$$

We check the type of this term, and find that it is

$$(\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}),$$

which is of the form $\tau \to \tau$ for $\tau = (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat})$. That means we can apply rec to this term to get a term of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ as follows

$$\mathsf{rec}\,(\lambda a : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.\ \lambda x : \mathsf{nat}.\ \lambda y : \mathsf{nat}.\ \mathsf{ifz}\, x [y][\bar{\mathsf{s}}(a(\mathsf{pred}\, x)y)])$$

RExercise 34. *This exercise asks students to practice reducing a PCF term to gain intuition for how reduction behaves.*

Carry out four reduction steps for the term in Example 3.1 applied to $\bar{\mathsf{s}}\bar{0}$ and $\bar{0}$.

See page 301 for a solution.

RExercise 35. *This exercise asks students to create a PCF term to help them understand the recursion operator.*

Let A be the addition term given in Example 3.1. In terms of A, write a PCF term which implements multiplication.

A solution may be found on page 301.

Example 3.2 (Nested recursion). Sometimes we may need to use one recursive term inside another. For instance, suppose we want to define the function

$$f\,k = \sum_{i=0}^{k} \sum_{j=0}^{i} j$$

in terms of a term A which does addition. We tackle this by first working out how to write the function which takes i to

$$\sum_{j=0}^{i} j.$$

This is the term

$$\mathsf{rec}\, \lambda s : \mathsf{nat} \to \mathsf{nat}.\ \lambda i : \mathsf{nat}.\ \mathsf{ifz}\, i [\bar{0}][A(s(\mathsf{pred}\, i))i].$$

Now we can apply the same idea for k to obtain

$$\mathsf{rec}\,\lambda z\!:\!\mathsf{nat}\to\mathsf{nat}.\,\lambda k\!:\!\mathsf{nat}.\,\mathsf{ifz}\,k\langle\bar{0}\rangle \\ \langle A(\mathsf{rec}\,\lambda s\!:\!\mathsf{nat}\to\mathsf{nat}.\,\lambda i\!:\!\mathsf{nat}.\,\mathsf{ifz}\,i[\bar{0}][A(s(\mathsf{pred}\,i))i]z)(z\mathsf{pred}\,k)\rangle.$$

Example 3.3 (Unbounded search). The examples above are carefully constructed so that they terminate. But we can also ask PCF to search for a number with a certain property, even if that property doesn't hold for any natural number. For instance, suppose we are given terms A and M which carry out addition and multiplication. We can instruct PCF to find a natural number x such that $x^2+1=0$ with the term

$$(\mathsf{rec}\,\lambda f\!:\!\mathsf{nat}\to\mathsf{nat}.\,\lambda x\!:\!\mathsf{nat}.\,\mathsf{ifz}\,(A(\bar{s}\bar{0})Mxx)[x][f\bar{s}x])\bar{0}.$$

3.2.2 Irreducible terms

The reduction relation we give for PCF is easier to reason about than that for either version of the λ-calculus. One place where this is particularly apparent is the ease with which we can classify the terms which cannot perform β-reduction.

Definition 39: irreducible term

A PCF term t is **irreducible** provided that there is no term t' with $t \xrightarrow{\beta} t'$.

RExercise 36. *We give the answer to this exercise in the text, but we encourage the reader to pause here and think about this exercise before reading on.*

Identify the irreducible terms for PCF which are of type nat.

A solution is given on page 301.

These terms are of interest because they are the kind of thing which can be the 'final result' of a computation. For instance, if we have a term of function type in the empty type environment, there are three possibilities for the 'answer' we get by β-reducing as much as possible. In the best case, it reduces to an abstraction, which is good because that explicitly tells us how to apply it to an input. Alternatively it could go on β-reducing forever: we have to accept that possibility as the price we pay for a useful programming language. But in principle there could have been a third option: computation would get 'stuck' at a term of function type which can never be reduced to an abstraction. This would be mysterious because we wouldn't be able to work out how to apply the resulting term to an argument. However, we can reassure ourselves that such things never happen: if computation does come to an end, the resulting term is in a 'useful' form for the particular type at hand.

This is complicated slightly in the presence of a type environment. In that case, we think of the variables with declared types as externally provided functions, following the analogy of a type environment as being like a header file. That means it is also ok for a term to reduce to a variable.

Proposition 3.1

If $\vdash t:\tau$ is derivable then t is irreducible if and only if one of the following cases applies.

- The irreducible terms of type $\tau = \mathsf{nat}$ are
  - $\bar{0}$ and
  - if u is an irreducible term of type nat then so is $\bar{\mathsf{s}}u$.
- The irreducible terms of type $\sigma \to \tau$ are abstractions $\lambda x : \sigma . u$.

If $\Gamma \vdash t : \tau$ is derivable then t is irreducible if and only if one of the previous cases applies or we have

- $t = x$ is a variable or
- $t = rs$ and the left-most symbol of t is a variable.

Proof. *This exercise encourages students to think about the nature of the β-reduction relation for PCF.*

See page 302.

This helps us to make sense of the β-reduction rules a little better. If we are trying to evaluate an expression involving, say, pred, we want to be able to apply the base case which is the case which explains the computational meaning of pred. For this to work, we might need to reduce the argument of pred, and if this does not run forever, the above reassures us that it reaches a form which the base case of pred can use. The same goes for all the other base cases of β-reduction: if their arguments finish computing, then they must be in the appropriate form for the base case to apply.

3.2.3 Properties of reduction

We revisit some of the properties that discussed in Chapter 2 for the simply typed λ-calculus. As promised, the definition of β-reduction for PCF makes some properties much simpler.

We had to work hard to show that β-reduction for the λ-calculus satisfies confluence, but the reduction relation we have defined for PCF is completely deterministic: there is never any choice about what to do next.

Proposition 3.2

If t is a PCF term then there is at most one term t' with $t \xrightarrow{\beta} t'$.

RExercise 37. *We encourage students to think about why reduction is deterministic.*

Prove the preceding proposition.

A solution may be found on page 302.

In our extended setting we still have the important property that β-reduction does not affect the type of a term.

Proposition 3.3: subject reduction

If Γ is a type environment, t a PCF term and τ a PCF type such that $\Gamma \vdash t : \tau$ is derivable. If $t \xrightarrow{\beta}\!\!\!\!\rightarrow t'$ then $\Gamma \vdash t' : \tau$ is derivable.

Proof. *Students may want to think about how this is proved; however we see no point in once again proving those cases which are treated in the previous chapter.*

See page 253.

Although we do not give a formal definition of α-equivalence for PCF terms, it is worth remarking that the properties one expects to hold for the way in which α-equivalence and β-reduction do hold. For example if for terms t, t' and u if $t \sim_\alpha t'$, t β-reduces to u and u is irreducible, then there is an irreducible term u' such that t' reduces u' and $u \sim_\alpha u'$. We simply assume any such property to be true if needed: we have seen enough of the detail of how such properties are proved for the simply typed λ-calculus, and the techniques used to establish them for PCF are not interestingly different.

3.2.4 Non-termination

In Example 3.3 a term is given which represents an unbounded search over the natural numbers. The ability to write such terms is the main motivation for moving to this more expressive programming language. However, the Halting Problem tells us that with this power comes the possibility of non-termination.

Example 3.4. The simplest term we can build using recursion is

$$\mathsf{rec}\, \lambda x : \sigma.\, x.$$

We can see that our β-reduction rules allow us to keep reducing this term forever:

$$\mathsf{rec}\, \lambda x : \sigma.\, x \xrightarrow{\beta} (\lambda x : \sigma.\, x)\mathsf{rec}\, \lambda x : \sigma.\, x \xrightarrow{\beta} \mathsf{rec}\, \lambda x : \sigma.\, x \xrightarrow{\beta} \ldots$$

Hence PCF does not have the property that every typeable term is strongly normalizing.

This example tells us something interesting about what a denotational semantics can and can't be like. For instance, we might have hoped that, however we come to define the denotational semantics of types in PCF, we could have $[\![\mathsf{nat}]\!] = \mathbb{N}$. But if we instantiate the term from the example at $\sigma = \mathsf{nat}$, then we have a term $\mathsf{rec}\, \lambda x : \mathsf{nat}.\, x$ which is of type nat but which is clearly observable different from outputting any particular natural number.

Therefore, we need the denotation of every type to contain a special value $\bot$ which represents 'never observing an output'. Note that this is conceptually very different from observing some kind of error value. In languages with error values, when we observe one as an output, we *know* that something has gone wrong. But one never actually observes $\bot$: if you try you will sit there watching your computer forever!

It is useful to think of $\bot$ as meaning 'no information'. One reason that this intuition is useful is that programs in PCF do *not* necessarily run forever if one of their inputs runs forever, but there are restrictions on what is possible.

Example 3.5. Consider the term $\lambda m : \mathsf{nat}.\, \bar{0}$. If we apply that to the non-terminating program of type nat from Example 3.4, we find

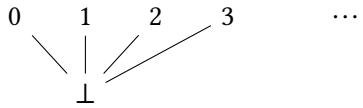
$$(\lambda m : \mathsf{nat}.\, \bar{0})(\mathsf{rec}\, \lambda x : \sigma.\, x) \xrightarrow{\beta} \bar{0}.$$

So this program halts even if its argument is a non-terminating computation. This is one of the advantages of the 'leftmost, outermost' evaluation strategy, but clearly it means any denotational semantics will have to be quite subtle!

Could we have a program that outputs one answer if its input runs forever and another if its input halts? The Halting Problem tells us that this is impossible in general, but in PCF this is easy to see by examining what all the constants of the language do when given an input that runs forever. For example, ifz tries to reduce its first argument to something of the form $\bar{\mathsf{s}}^n\bar{0}$ to see which case it should apply. But if that argument never terminates, neither will ifz as we will always have to apply the step case of its β-reduction rules.

This agrees with the intuition of 'no information' because it says, intuitively 'if you claim you can output an answer when given no information about your input, that means you can't then look at the input to decide what to output!' Another way to see this is that if the program needs to check anything about its input, for instance whether the input is $\bar{0}$, it does not know how long it needs to wait before giving up and assuming that the input will never compute to $\bar{0}$.

For these reasons, we interpret the type nat not just by a set, but by an *ordered* set, with $\bot$ on the bottom, the 'least informative' element, and all the natural numbers maximal elements since they tell us precisely which natural number they are. We picture this situation as follows.



The situation at function types is more complicated. We can, of course, use Example 3.4 to show that there must be a $\bot$ value in every type, including function types. But recursively defined functions can terminate on some inputs and run forever on others.

Example 3.6. Consider the PCF term

$$\mathsf{rec}\, \lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][f\, x].$$

If we apply this to $\bar{0}$ then, using $\sigma = \mathsf{nat} \to \mathsf{nat}$,

$$\begin{aligned}
&(\mathsf{rec}\, \lambda f : \sigma.\, \lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][f\, x])\bar{0} \\
\xrightarrow{\beta}\ &(\lambda f : \sigma.\, \lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][f\, x])(\mathsf{rec}\, \lambda f : \sigma.\, \lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][f\, x])\bar{0} \\
\xrightarrow{\beta\alpha}\ &\lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][\mathsf{rec}\, \lambda f : \sigma.\, \lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][f\, x]].\bar{0} \\
\xrightarrow{\beta}\ &\mathsf{ifz}\, \bar{0}[\bar{0}][\mathsf{rec}\, \lambda f : \sigma.\, \lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][f\, x]] \\
\xrightarrow{\beta}\ &\bar{0}.
\end{aligned}$$

But if we apply it to $\overline{\mathsf{s0}}$, or the representation of any non-zero number, it β-reduces to itself.

This function seems to be more informative than the function which doesn't halt for any input, but less informative than, say, the identity function on the type nat. On the other hand, it is observably different from the constant function which always returns $\overline{\mathsf{s0}}$ because these functions can be evaluated at $\overline{\mathsf{0}}$ where they output observably different values. Clearly, we can't expect to be able to work out the information ordering on the values of complicated types by intuition. To give a denotational semantics to PCF we need a mathematical theory which describes what sort of ordered structures these are, and comes with a 'function space' construction which builds the correct ordered structure for us at higher type, and this mathematical theory is described in Section 4.4.

The β-reduction rule for rec puts another very strong constraint on the mathematical objects we need for denotational semantics. Consider a term t of type $\tau \to \tau$ for any type τ. Suppose we have a denotational semantics which maps t to some sort of mathematical function $[\![t]\!]$. It must send $\mathsf{rec}\, t$ to a suitable element $[\![\mathsf{rec}\, t]\!]$ of the set $[\![\tau]\!]$.

Now we expect these denotations to be invariant under β-reduction: part of the point of denotational semantics is that it lets us reason equationally rather than running programs in our heads. Similarly we want application to be interpreted by ordinary function application. But since

$$\mathsf{rec}\, t \xrightarrow{\beta} t\,\mathsf{rec}\, t$$

we must have

$$[\![\mathsf{rec}\, t]\!] \;=\; [\![t]\!]\,([\![\mathsf{rec}\, t]\!]).$$

So $[\![\mathsf{rec}\, t]\!]$ is a 'fixed point' of $[\![t]\!]$: an element which is sent to itself by the function. But since we made no special assumptions about t, the semantics must have the property that every function which can be the denotation of a term must have a fixed point.

It is not immediately clear that we can construct a mathematical theory where this is possible. Our considerations about non-termination in this section have given us a clue: on the set $\mathbb{N}$, for instance, the successor function has no fixed point. But now we have added $\bot$ to represent 'no information returned', we can see that this has had the side-effect of adding a fixed point when applying the successor operation to programs. But clearly a sophisticated semantic theory is needed to explain how productive uses of recursion work. In order to study denotational semantics properly, we first have to work out what observations can be performed on terms—a task we turn to in the next section.

3.3 Comparing Terms by Carrying Out Experiments

In this section we describe the notion of contextual equivalence for PCF. In light of the discussion above, we actually define something more refined than an equivalence. We are not just interested in when two terms behave in the same way, but more generally when one term is more informative than another in the sense that it passes the same tests but may also pass tests which cause the other term to run forever. The intuition for this is that we never actually observe a term running forever—we are left wondering whether it *will* pass the test, but in a

million years! Therefore, terms which behave the same as other terms except that they run forever in more situations will never actually be observed to behave differently. We define order-like relations to take account of this.

We first define a simplified version of this notion, where we only care about tests which *apply* the given terms to inputs, not ones where they may be the inputs to larger programs.

Above we argue that if we have a term t of base type (that is nat) then one of the following happens:

- either we can find a unique $n \in \mathbb{N}$ with $t \xrightarrow{\beta}\!\!\!\!\rightarrow \bar{\mathsf{s}}^n\bar{0}$ or
- t reduces forever.

When we have a term t of type $\mathsf{nat} \to \mathsf{nat}$ we can think about what happens if we apply t to a term u of type nat, and since tu is of type nat one of the above possibilities applies.

We make formal the idea of a term being more informative in the following definition, after first generalizing it to all types. A definition of the notion of *preorder* is given in Definition 50, and there we also make the connection with the notion of a *partial order*.

Definition 40: applicative preorder

The **applicative preorder** for PCF terms is defined recursively as follows: Assume that t and t' are PCF terms such that $\Gamma \vdash t, t' : \tau$ is derivable for some type environment Γ and some type τ. We have

$$t \preceq t'$$

if and only if

$\mathbf{bc}\mathsf{nat} \preceq \ \tau = \mathsf{nat}$ and, for $n \in \mathbb{N}$,

$$t \xrightarrow{\beta}\!\!\!\!\rightarrow \bar{\mathsf{s}}^n\bar{0} \qquad \text{implies} \qquad t' \xrightarrow{\beta}\!\!\!\!\rightarrow \bar{\mathsf{s}}^n\bar{0}.$$

$\mathbf{sc}{\to}\preceq \ \tau = \rho \to \sigma$ and for all u with $\vdash u : \rho$ we have

$$tu \preceq t'u.$$

It is worth noting that at type nat, the terms $\bar{0}$ and $\bar{\mathsf{s}}\bar{0}$, for example, are *not* related—indeed, none of the terms that reduce to irreducible terms we might think of as numbers are related. This is very different from how we usually compare natural numbers.

Example 3.7. Consider the PCF term given in Example 3.6.

$$\mathsf{rec}\, \lambda f : \mathsf{nat} \to \mathsf{nat}.\ \lambda x : \mathsf{nat}.\ \mathsf{ifz}\, x[\bar{0}][f\, x].$$

There we showed that this term returns $\bar{0}$ when given $\bar{0}$ as an input, and fails to terminate for all other inputs. Therefore, we have

$$\lambda x : \mathsf{nat}.\ (\mathsf{rec}\, \lambda y : \mathsf{nat}.\ y)$$

$$\preceq \mathsf{rec}\, \lambda f : \mathsf{nat} \to \mathsf{nat}.\ \lambda x : \mathsf{nat}.\ \mathsf{ifz}\, x[\bar{0}][f\, x]$$
$$\preceq \lambda x : \mathsf{nat}.\ x.$$

This is because the first runs forever on all inputs, while the third is the identity function. So when given input $\bar{0}$, it returns $\bar{0}$ as required. For all other inputs, the term from Example 3.6 never finishes reducing, so we know it never reaches a term of the form $\bar{s}^n\bar{0}$. The identity function term is more informative than the given term in the sense that while no-one will ever be able to observe them giving different outputs, it successfully outputs an answer for more inputs.

Note that we *also* have

$$\mathsf{rec}\, \lambda f : \mathsf{nat} \to \mathsf{nat}.\ \lambda x : \mathsf{nat}.\ \mathsf{ifz}\, x[\bar{0}][f\, x]$$
$$\preceq \lambda x : \mathsf{nat}.\ \bar{0}$$

for the same reason, even though the second term in this case is not related either way around to the identity function term. On the other hand, the term

$$\lambda x : \mathsf{nat}.\ \bar{s}\bar{0}$$

is not related either to $\lambda x : \mathsf{nat}.\ \bar{0}$ or to the term from Example 3.6, because it reduces to $\bar{s}\bar{0}$ when applied to $\bar{0}$, whereas the given term reduces to $\bar{0}$ when applied to $\bar{0}$.

It is worth noting that we may think of the applicative preorder as comparing terms of the same type by checking what they reduce to when we supply them with arguments until we reach a term of type nat.

We ought to prove that this is a preorder, and also that it does not distinguish between two terms if one reduces to the other. This is how we know that it is a useful order to use to study properties of reduction. The final condition we are interested about deserves a bit of thinking about types.

We may see that every type is of the form

$$\tau_1 \to \tau_2 \to \ldots \to \tau_n \to \mathsf{nat},$$

where when $n = 0$ the type is just nat. If we have a term t of this type we may then supply it with arguments $u_i : \tau_i$ to obtain a term of type nat by forming

$$t u_1 u_2 \ldots u_n.$$

Proposition 3.4

Let t and t' be PCF terms such that $\Gamma \vdash t, t' : \tau$ is derivable for some type environment Γ and some type τ. The following statements hold.

(a) The applicative preorder relation is indeed a preorder.

(b) If t and t' are terms with $t \xrightarrow{\beta} t'$ then $t' \preceq t$ and $t \preceq t'$.

(c) If $\tau = \tau_1 \to \tau_2 \to \ldots \to \tau_n \to \mathsf{nat}$ then $t \preceq t'$ if and only if for all terms

$u_1, \ldots, u_n$ with $\vdash u_i : \tau_i$ derivable for $1 \leq i \leq n$ we have

$$tu_1 \ldots u_n \preceq t' u_1 \ldots u_n.$$

Proof. We show each part.

(a) The given relation is clearly reflexive at type nat, and a simple proof by induction over the type of the given terms shows that reflexivity holds for all types.

Similarly the given relation is clearly transitive at type nat and once again a simple proof by induction over the type of the given terms shows that transitivity holds for all types.

(b) This is an induction over the type of t (which is the type of t').

If t and t' are terms of base type then we may argue as follows. If $t' \xrightarrow{\beta}\!\!\!\!\!\rightarrow \mathsf{s}^n\mathsf{0}$ for some $n \in \mathbb{N}$ then we can see that

$$t \xrightarrow{\beta} t' \xrightarrow{\beta}\!\!\!\!\!\rightarrow \mathsf{s}^n\mathsf{0}$$

and so the first claim holds.

But we also have that $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow \mathsf{s}^n\mathsf{0}$ for some $n \in \mathbb{N}$ then since reduction is deterministic the first step of that chain of reductions must be $t \xrightarrow{\beta} t'$, and so $t' \xrightarrow{\beta}\!\!\!\!\!\rightarrow \mathsf{s}^n\mathsf{0}$ must also hold.

If t and t' are terms of type $\rho \to \sigma$ and we have a term u of type ρ then we know that $t \xrightarrow{\beta} t'$ implies $tu \xrightarrow{\beta} t'u$, and so by the induction hypothesis we know that $tu \preceq t'u$ as well as $t'u \preceq tu$ and since this holds for all such u we get $t \preceq t'$ as well as $t' \preceq t$.

(c) We note that in the case where $\tau = \mathsf{nat}$ there is nothing to show. We prove the statement by induction over the type, where we have effectively already covered the base case.

If $\tau = \rho \to \sigma$ then this means that $\rho = \tau_1$ and $\sigma = \tau_2 \to \ldots \to \tau_n$.

Assume we have $t \preceq t'$ and that we have terms $u_1, \ldots, u_n$ satisfying the conditions from the statement.

We may see that tu_1 and $t'u_1$ are terms which, given the type environment Γ, we may give the type σ, and also that by sc$\to\preceq$ we know that $t \preceq t'$ implies $tu_1 \preceq t'u_1$. By the induction hypothesis this implies

$$tu_1 \ldots u_n \preceq t' u_1 \ldots u_n$$

as required.

Now assume that for all terms $u_1, \ldots, u_n$ satisfying the conditions from the statement and we have that

$$tu_1 \ldots u_n \preceq t' u_1 \ldots u_n.$$

In order to show that $t \preceq t'$ by sc$\to\preceq$ it is sufficient to show that for all terms u with $\vdash u : \rho$ we have $tu \preceq t'u$. We note that the requirements for

the term u exactly match the requirements for the term u_1, and by the induction hypothesis we know that the assumption

$$tu_1 \dots u_n \preceq t'u_1 \dots u_n$$

implies

$$tu_1 \preceq t'u_1$$

which, as we have just argued, implies $t \preceq t'$ as required.

Part (iii) of this result tells us that, as far as the applicative preorder is concerned, we may find out everything we need to know about a term by supplying it with arguments until we have a term of type nat. We may therefore justify the name 'applicative preorder' by noting that it can be thought of as being the result of seeing what happens when we *apply* the given term to suitable arguments, until it can take no further arguments (because we have obtained a term of base type).

We now turn to the notion of contextual equivalence proper. While applicative equivalence only tests terms by applying them to various inputs, we want to know what happens when the terms in question are used in more general contexts. In the simply typed λ-calculus we had to define a separate notion of 'context' in order to describe what it means for two terms to behave in the same way in all contexts. However, we proved in Proposition 2.23 that it is sufficient to consider contexts in which the given term is the right hand side of an application. The reasoning in PCF is completely analogous so we do not repeat it here, instead, we use the idea as the definition contextual equivalence in PCF.

As for the applicative preorder, we refine the notion of contextual equivalence to a contextual preorder to capture the subtle way in which one term can be less informative than another, but never actually be observed giving a different result to a test.

Definition 41: contextual preorder

The **contextual preorder** for PCF terms is defined as follows. If t and t' are terms and Γ is a type environment such that $\Gamma \vdash t, t' : \tau$ is derivable then

$$t \trianglelefteq t'$$

if and only if for all terms f such that $\vdash f : \tau \to \mathsf{nat}$ is derivable we have that

$$ft \preceq ft'.$$

Note that ft and ft' are both terms of type nat, and so we only invoke the preorder $\preceq$ at base type in this definition.

We may now use this idea to define contextual equivalence—below we see why it is useful to have the preorder as a concept rather than only the equivalence.

Definition 42: contextual equivalence

Let t and t' be PCF terms such that there is a type environment Γ and a type τ such that $\Gamma \vdash t, t' : \tau$ is derivable. Then t and t' are **contextually equivalent** if and only if we have

$$t \trianglelefteq t' \qquad \text{and} \qquad t' \trianglelefteq t.$$

We again use the notation

$$t \simeq t'$$

to indicate that t and t' are contextually equivalent.

Exercise 38. *This exercise is suitable for students who want to refresh their memory regarding how relations work.*

Show that if $\leq$ is a preorder on a set S then the relation $\sim$ defined by

$$s \sim s' \qquad \text{if and only if} \qquad s \leq s' \text{ and } s' \leq s$$

is an equivalence relation.

A solution appears on page 303.

Proposition 3.5

The contextual preorder relation satisfies the following properties.

(a) It is indeed a preorder.

(b) If t and t' are PCF terms with $t \trianglelefteq t'$ then $t \preceq t'$.

Proof. See page 254.

Now we have defined what it means for PCF terms to be equivalent, and, indeed, refined this notion to a preorder to capture the fact that non-termination complicates the notion of observable behaviour. With this in place, it makes sense to try to give a denotational semantics of the language, and ask how well it captures the observable behaviour of programs.

3.4 The Dcpo Model

In order to define the interpretation of PCF we need to make significant use of material from Chapter 4. In particular:

- The syntactic construction that is hardest to interpret is rec, for which we make use of *fixed points* (see Section 4.4.3), and in order to ensure that these always exist, we have to introduce *dcpos* (see Section 4.4).
- Again we use a function space construction to model function types, but for dcpos this is a more sophisticated concept than for the sets we used for the simply typed λ-calculus (see 175).

To make sure that these constructions work as expected, and to establish their properties, significant machinery is required.

Once we have chosen a suitable dcpo D as the base type we may define

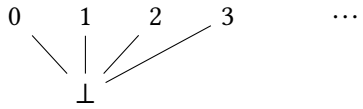
$$[\![\sigma \to \tau]\!]^D = [\![\sigma]\!]^D \Rightarrow [\![\tau]\!]^D$$

to interpret the remaining types.

In other words, subject to a different notation, this is the same approach as modelling the simply typed λ-calculus in that a function type is again interpreted by a set of functions between the interpretations of the immediate subtypes. The three main differences are the following.

- We ensure that there is a value we may think of meaning 'undefined', or 'no information', called $\bot$.
- We only allow functions that are well-behaved with respect to the partial order with which we encode this idea, in the language of Chapter 4, we only allow Scott-continuous functions.
- When we considered the simply typed λ-calculus we allowed the base type to be interpreted by an arbitrary set. To model PCF we fix the base type to be interpreted by the set $\mathbb{N}_\bot$.

The first requirement is necessary due to the fact that we know from the Halting Problem that in a sufficiently powerful language, some recursive programs will not terminate. The second requirements ensures that we have a fixed point operator that allows us to interpret recursion. Regarding the third requirement, we have to ensure that we are able to model constructions such as $\bar{\mathsf{s}}$, pred and ifz, and that motivates the choice of model for the base type, which we may picture like this.



It is worth emphasizing that we do not consider the number 1 as less than or equal to the number 2, for example—from the point of view of carrying out computations, one carries as much information as the other. It's just the value of $\bot$ that is assumed to convey no information at all. The (partially ordered) set $\mathbb{N}_\bot$ and related structures appear as examples throughout Section 4.3 and Section 4.4, and the reader is directed to those for more mathematical content.

We use brackets $[\![\,]\!]$ without a superscript to refer to this interpretation. Our results are restricted to this model.

3.4.1 Denoting types and terms

As for the simply typed λ-calculus we first say how we interpret the types for our language. This definition looks similar to the interpretation of function types for the simply typed λ-calculus, and $\Rightarrow$ does indeed denote a set of functions of some sort. However, this is not just a set, it is itself a partially ordered set, and we do not allow all the functions between the two underlying set. The construction is described in detail in Section 4.4.2, but understanding that section requires reading Section 4.3 and all of Section 4.4.

Definition 43: denotation of a PCF type

The **denotation of a type** τ, written $[\![\tau]\!]^X$, is defined by recursion on τ as follows.

bcnat $[\![\mathsf{nat}]\!] = \mathbb{N}_\bot$.

sc$\to$ $[\![\rho \to \sigma]\!] = [\![\rho]\!] \Rightarrow [\![\sigma]\!]$.

We discuss the idea behind our interpretation of PCF terms.

- For terms and term constructors to do with the type nat, we use corresponding infrastructure for $\mathbb{N}_\perp$. In particular, $\bar{0}$ is interpreted by the element $0 \in \mathbb{N}_\perp$, and for $\bar{s}$, pred and ifz we define appropriate functions in Definition 44.
- For variables, λ-abstractions and applications we use much the same interpretation as for terms of the simply typed λ-calculus, see Definition 35.
- For rec we use a fixed point construction as described in Section 4.4.3. This idea is foreshadowed towards the end of Section 3.2.4. The reader is encouraged to look at the examples following Definition 45 to see that idea in action, in particular Example 3.10.

Since we now have some infrastructure in our language to support the natural numbers as our base type, and since we also have conditionals we have to ensure that suitable infrastructure to interpret these exists in our model. The function succ appears in Example 4.8, and the reader is invited to consider the function pred in Exercise 43. The typing of ifz is discussed in Example 4.4.

Definition 44: useful functions

We have the following functions:

$$\begin{aligned}
&\mathrm{succ} : \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \\
&\qquad x \longmapsto \begin{cases} \perp & x = \perp \\ x+1 & \text{else} \end{cases} \\
&\mathrm{pred} : \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \\
&\qquad x \longmapsto \begin{cases} \perp & x = \perp \\ 0 & x = 0 \\ x-1 & \text{else} \end{cases} \\
&\mathrm{ifz} : \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \Rightarrow (\mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp) \\
&\qquad xyz \longmapsto \begin{cases} \perp & x = \perp \\ y & x = 0 \\ z & \text{else} \end{cases}
\end{aligned}$$

We have to ensure that these functions do live in our intended model.

Exercise 39. *For students who want to understand the model better this is a useful exercise. There are results in the previous section that help with this.*

Argue that the functions succ, pred and ifz are Scott-continuous, see Definition 57.

A solution appears on page 303.

We summarize the various cases of denotations of PCF terms.

Definition 45: denotation of a PCF term

Let Γ be a type environment, τ a type and t a PCF term such that $\Gamma \vdash t : \tau$ is derivable. Given a valuation ϕ for Γ the Γ-**denotation of t relative to ϕ**, written

$$[\![(\Gamma, t)]\!]_\phi$$

is defined by recursion on t as follows.

bcVar$[\![\,]\!]$ $[\![(\Gamma, x)]\!]_\phi = \phi x$ if $x \in \mathsf{Vars}$.

scAbs$[\![\,]\!]$ $[\![(\Gamma, \lambda x : \sigma.\, u)]\!]_\phi$ is the function from $[\![\sigma]\!]$ to $[\![\tau]\!]$ which sends an element $a \in [\![\sigma]\!]$ to $[\![(\Gamma x : \sigma, u)]\!]_{\phi[x \mapsto a]}$.

scApp$[\![\,]\!]$ $[\![(\Gamma, rs)]\!]_\phi = [\![(\Gamma, r)]\!]_\phi([\![(\Gamma, s)]\!]_\phi)$.

bcZero$[\![\,]\!]$ $[\![(\Gamma, \bar{0})]\!]_\phi = 0 \in \mathbb{N}_\perp$.

scSucc$[\![\,]\!]$ $[\![(\Gamma, \bar{\mathsf{s}} n)]\!]_\phi = \mathrm{succ}\,[\![(\Gamma, n)]\!]_\phi$.

scPred$[\![\,]\!]$ $[\![(\Gamma, \mathsf{pred}\, n)]\!]_\phi = \mathrm{pred}\,[\![(\Gamma, n)]\!]_\phi$.

scIfz$[\![\,]\!]$ $[\![(\Gamma, \mathsf{ifz}\, nrs)]\!]_\phi = \mathrm{ifz}\,[\![(\Gamma, n)]\!]_\phi [\![(\Gamma, r)]\!]_\phi [\![(\Gamma, s)]\!]_\phi$.

scRec$[\![\,]\!]$ $[\![(\Gamma, \mathsf{rec}\, u)]\!]_\phi = \mathsf{fix}\,[\![(\Gamma, u)]\!]_\phi$

We still owe further justifications that all the functions used in this definition are actually Scott-continuous, and so are elements of the relevant function type, and the reader may find these in Section 3.4.2. We turn to some examples, paying particular attention to the interpretation of recursion, which is the most sophisticated new feature of PCF when compared to the simply typed λ-calculus.

Example 3.8. As an exercise one might wonder what the interpretation of rec looks like for the simplest term we can type that uses recursion,

$$\mathsf{rec}\,(\lambda x : \sigma.\, x).$$

We may calculate this as

$$\begin{aligned}
[\![\mathsf{rec}\,(\lambda x : \sigma.\, x)]\!] &= \mathsf{fix}\,[\![\lambda x : \sigma.\, x]\!] && \text{scRec}[\![\,]\!] \\
&= \mathsf{fix}(d \mapsto [\![(x : \sigma, x)]\!]_{x \mapsto d}) && \text{scAbs}[\![\,]\!] \\
&= \mathsf{fix}(d \mapsto d) && \text{bcVar}[\![\,]\!] \\
&= \bigvee_{n \in \mathbb{N}} \mathrm{id}^n_{[\![\sigma]\!]}\, \perp && \text{Exercise 49} \\
&= \perp.
\end{aligned}$$

We provide a more interesting example of the denotation of a term.

Example 3.9. We return once again to the term

$$\mathsf{rec}\, \lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda x : \mathsf{nat}.\, \mathsf{ifz}\, x[\bar{0}][f\, x]$$

from Example 3.6.

We calculate its denotation in some detail to show how the fixed point

construction works. We typically would *not* work in this manner with expressions as given below, and the reader who finds this intimidating is invited to move on to the example that follows, which illustrates a more pragmatic way of proceeding, and which uses a more elegant way of reasoning.

$$
\begin{aligned}
&[\![\mathsf{rec}\,\lambda f:\sigma.\,\lambda x:\mathsf{nat}.\,\mathsf{ifz}\,x[\bar{0}][f\,x]]\!] \\
&= \mathsf{fix}\,[\![\lambda f:\sigma.\,\lambda x:\mathsf{nat}.\,\mathsf{ifz}\,x[\bar{0}][f\,x]]\!] \\
&= \mathsf{fix}(h \mapsto [\![(f:\sigma, \lambda x:\mathsf{nat}.\,\mathsf{ifz}\,x[\bar{0}][f\,x])]\!]_{[f\mapsto f]}) \\
&= \mathsf{fix}(h \mapsto (z \mapsto [\![(f:\sigma x:\mathsf{nat}, \mathsf{ifz}\,x[\bar{0}][f\,x])]\!]_{\substack{\phi[f\mapsto h] \\ [x\mapsto z]}})) \\
&= \mathsf{fix}\left(h \mapsto \left(z \mapsto \begin{cases} \bot & [\![(\Gamma,x)]\!]_{\substack{\phi[f\mapsto h] \\ [x\mapsto z]}} = \bot \\ 0 & [\![(\Gamma,x)]\!]_{\substack{\phi[f\mapsto h] \\ [x\mapsto z]}} = 0 \\ [\![(\Gamma, f\,x)]\!]_{\substack{\phi[f\mapsto h] \\ [x\mapsto z]}} & \text{else} \end{cases}\right)\right) \\
&= \mathsf{fix}\left(h \mapsto \left(z \mapsto \begin{cases} \bot & z = \bot \\ 0 & z = 0 \\ hz & \text{else.} \end{cases}\right)\right)
\end{aligned}
$$

We call the assignment that maps h to the given function F. Then we have that this fixed point is given by the assignment

$$x \longmapsto (\bigvee\nolimits_{n\in\mathbb{N}} F^n k_\bot)x\ ,$$

where $k_\bot$ is the function from $\mathbb{N}_\bot$ to itself that returns $\bot$ for all inputs. and we may now calculate that by Proposition 4.11 we have

$$(\bigvee_{n\in\mathbb{N}} F^n k_\bot)x = \bigvee_{n\in\mathbb{N}} (F^n k_\bot)x$$

and we can show by induction for $n \geq 1$ that

$$(F^n k_\bot)x = \begin{cases} \bot & x = \bot \\ 0 & x = 0 \\ \bot & \text{else} \end{cases}$$

as expected, and so we obtain a fixed point after applying F only once.

Example 3.10. Consider the term

$$\mathsf{rec}\,\lambda f:\mathsf{nat}\to\mathsf{nat}.\,\lambda x:n.\,\mathsf{ifz}\,x[\bar{0}][\bar{\mathsf{s}}\,f\,\mathsf{pred}\,x]$$

which appears in the discussion following Example 4.5. We think about its denotation by looking at the body of the abstraction, that is

$$\mathsf{ifz}\,x[\bar{0}][\bar{\mathsf{s}}\,f\,\mathsf{pred}\,x].$$

Its interpretation assumes that there is a function $g : \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp$ as well as $y \in \mathbb{N}_\perp$ and it gives

$$\text{ifz}\, y\, 0(\text{succ}\, g(\text{pred}\, y)).$$

We then want to take the least fixed point of this, taken as a function whose argument is g and so we may think of this as an operator, say G, where

$$G : \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp,$$

where

$$(Gg)y = \text{ifz}\, y\, 0(\text{succ}\, g(\text{pred}\, y)),$$

so $\mathsf{fix}\, G$ is an element of $\mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp$ and we may think of it as a function

$$\mathsf{fix}\, G : \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp.$$

From the previous discussion we believe that this fixed point is the identity function on $\mathbb{N}_\perp$, and we prove this by induction over the natural numbers, where we have an additional case of checking what happens if the input is $\perp$.

We make crucial use here of the fixed point property which tells us that

$$G(\mathsf{fix}\, G) = \mathsf{fix}\, G.$$

If we have the input $\perp$ then

$$\begin{aligned}
(\mathsf{fix}\, G)\perp &= G(\mathsf{fix}\, G)\perp && \text{fixed point prop}\\
&= \text{ifz}\, \perp\, 0(\text{succ}(\mathsf{fix}\, G)(\text{pred}\, \perp)) && \text{def } G\\
&= \perp && \text{def ifz}.
\end{aligned}$$

For the input 0 we may calculate similarly

$$\begin{aligned}
(\mathsf{fix}\, G)0 &= G(\mathsf{fix}\, G)0 && \text{fixed point prop}\\
&= \text{ifz}\, 0\, 0(\text{succ}(\mathsf{fix}\, G)(\text{pred}\, 0)) && \text{def } G\\
&= 0 && \text{def ifz}.
\end{aligned}$$

For both these cases $\mathsf{fix}\, G$ does indeed give the same output as the identity function on $\mathbb{N}_\perp$. We now assume that the claim holds for the number n and we show the step case, that is it also holds for $n + 1$.

$$\begin{aligned}
(\mathsf{fix}\, G)(n+1) &= G(\mathsf{fix}\, G)(n+1) && \text{fixed point prop}\\
&= \text{ifz}(n+1)\, 0(\text{succ}(\mathsf{fix}\, G)(\text{pred}(n+1))) && \text{def } G\\
&= \text{succ}((\mathsf{fix}\, G)(\text{pred}(n+1))) && \text{def ifz}\\
&= \text{succ}((\mathsf{fix}\, G)n) && \text{def pred}\\
&= \text{succ}\, n && \text{ind hyp}\\
&= n+1
\end{aligned}$$

and so it is indeed the case that

$$\mathsf{fix}\, G = \text{id}_{\mathbb{N}_\perp}.$$

RExercise 40. *This exercise is asking students to work out the denotation of a PCF term.*

Consider the term

$$\mathsf{rec}\,\lambda f : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.\ \lambda x : \mathsf{nat}.\ \lambda y : \mathsf{nat}.\ \mathsf{ifz}\, x [y][\overline{\mathsf{s}}(f\,(\mathsf{pred}\, x)\, y)].$$

Work out the denotation of this term, using similar techniques to Example 3.10.

A solution may be found on page 303.

3.4.2 Justification of the given model

It is not immediately obvious that our denotations behave as expected—for the simply typed λ-calculus we just had to check that we had obtained functions of the appropriate type, but now we need to ensure that interpretations of terms of function type are Scott-continuous: In order to have the fixed points we require in order to model recursion we form our function spaces by only allowing Scott-continuous functions (see Sections 4.4.2 and 4.4.3). We now have to check that when providing denotations for PCF terms we construct such functions.

We need a preparatory result to help us establish this.

Proposition 3.6

Assume that Γ is a type environment, τ a type and t a PCF term such that $\Gamma \vdash t : \tau$ is derivable. If $\Gamma \vdash x : \sigma$ is derivable and ϕ is a valuation for Γ and $d \in [\![\sigma]\!]$ then

$$d \longmapsto [\![(\Gamma, t)]\!]_{\phi[x \mapsto d]}$$

is a Scott-continuous assignment from $[\![\sigma]\!]$ to $[\![\tau]\!]$.

Proof. See page 255.

Proposition 3.7

Let Γ be a type environment, τ a type and t a PCF term such that $\Gamma \vdash t : \tau$ is derivable. If ϕ is a valuation for Γ then

$$[\![(\Gamma, t)]\!]_{\phi} \in [\![\tau]\!].$$

Proof. *This exercise requires some knowledge from Section 4.4, but is of interest to students who have followed that narrative in detail.*

See page 304.

3.5 Properties of Denotations

In Section 2.8.5 we show that the way we have defined denotations works well with the computational infrastructure, and we carry out a similar exercise here. We have not formally defined α-equivalence for PCF terms and so we do not give a formal proof that denotations are invariant under α-equivalence, but that is indeed the case (and we also have the usual renaming result). We turn to β-reduction.

Proposition 3.8

Assume that Γ is a type environment such that

$$\Gamma \vdash a : \sigma \qquad \text{and} \qquad \Gamma, x : \sigma \vdash \lambda x : \sigma . t : \tau$$

are derivable, and that ϕ is a valuation for Γ. Then

$$[\![(\Gamma, t[a/x])]\!]_\phi = [\![(\Gamma, t)]\!]_{\phi[x \mapsto [\![(\Gamma, a)]\!]_\phi]}.$$

Proof. See page 256.

Proposition 3.9

Let t and t' be PCF terms and Γ a type environment such that $\Gamma \vdash t, t' : \tau$ is derivable. If ϕ is a valuation for Γ we have that

$$t \xrightarrow{\beta} t' \qquad \text{implies} \qquad [\![(\Gamma, t)]\!]_\phi = [\![(\Gamma, t')]\!]_\phi.$$

Proof. *This is a good exercise to learn to reason about the denotational semantics for PCF.*

See page 305.

The following is a straightforward induction proof based on the previous result.

Corollary 3.10

If t is a PCF term such that $\Gamma \vdash t : \tau$ is derivable and

$$t \xrightarrow{\beta} u,$$

where u is irreducible, then

$$[\![(\Gamma, t)]\!]_\phi = [\![(\Gamma, u)]\!]_\phi.$$

Note that this tells us in particular that if an irreducible term t does not have free variables and so is typeable with the empty type environment then it is either an abstraction (if it is of a function type) or it is a term that represents a particular number, namely $\bar{0}$ or $\bar{s}$ applied a finite number of times to $\bar{0}$ (if it is of type nat).

Moreover, we can relate the order we obtain for the semantics to the preorder we have for terms of base type.

Proposition 3.11

If $\Gamma \vdash t, t' : \mathsf{nat}$ is derivable and

$$t \preceq t'$$

then for every valuation ϕ for Γ we have

$$[\![(\Gamma, t)]\!]_\phi \leq [\![(\Gamma, t')]\!]_\phi.$$

Proof. If we can find $n \in \mathbb{N}$ with $t \xrightarrow{\beta} \bar{\mathsf{s}}^n\bar{0}$ then by Corollary 3.10 we know that

$$[\![(\Gamma, t)]\!]_\phi = [\![(\Gamma, \bar{\mathsf{s}}^n\bar{0})]\!]_\phi = n \in \mathbb{N}_\perp.$$

In this situation we also know that since $t \preceq t'$ we have $t' \xrightarrow{\beta} \bar{\mathsf{s}}^n\bar{0}$ and we may employ the same reasoning to conclude that $[\![(\Gamma, t')]\!]_\phi = n$.

Hence $[\![(\Gamma, t)]\!]_\phi$ is either equal to $\perp \in \mathbb{N}_\perp$, in which case it is certainly less than or equal to $[\![(\Gamma, t')]\!]_\phi$, or it is equal to some $n \in \mathbb{N}$, in which case it is equal to $[\![(\Gamma, t')]\!]_\phi$, and so in every case we have

$$[\![(\Gamma, t)]\!]_\phi \leq [\![(\Gamma, t')]\!]_\phi$$

which establishes the claim.

We summarize our knowledge about the various (pre)orders:

Corollary 3.12

If $\Gamma \vdash t, t' : \mathsf{nat}$ is derivable then for every valuation ϕ for Γ we have

$$t \trianglelefteq t' \quad \text{implies} \quad t \preceq t' \quad \text{implies} \quad [\![(\Gamma, t)]\!]_\phi \leq [\![(\Gamma, t)]\!]_\phi.$$

Proof. The first implication comes from Proposition 3.5 and the second from the previous result.

We further check that the semantics is invariant for the base case for η-conversion.

Proposition 3.13

Assume we have a type environment Γ, a variable x, a term t and types σ as well as τ a type such that x dos not occur free in t and such that $\Gamma \vdash t : \tau$ is derivable. Then for every valuation ϕ for Γ we have

$$[\![(\Gamma, \lambda x : \sigma.\, tx)]\!]_\phi = [\![(\Gamma, t)]\!]_\phi.$$

Proof. *This is an exercise to see the similarities between the denotational semantics for PCF and that of the simply typed λ-calculus, but note that we have not restated ingredients required here for PCF (they are present in Chapter 2).*

See page 306.

3.6 Adequacy

Having put considerable effort into constructing a semantics for PCF, we must now ask the question whether it is a good model. Just as in the simply typed λ-calculus, the formalization of this question is to ask whether the semantics is *adequate*, that is, if the model tells us that two terms have equal denotations, then they must be contextually equivalent. This means that the model fulfils its fundamental purpose of letting us prove instances of contextual equivalence without explicitly

reasoning about all contexts. For PCF, we have refined the notion of contextual equivalence to the contextual preorder, and so we in fact prove something stronger than adequacy which says that the order in the model tells us something about the contextual preorder. We build up to that by studying in detail what happens for the base type nat, since this type contains our input and output data for testing terms.

The argument has a similar structure to the one for the simply-typed λ-calculus. Namely, we show that every term of type nat reduces to something which can be straightforwardly understood as something in $\mathbb{N}_\perp$ which gives the semantics of this type. In the simply-typed λ-calculus, we could freely choose the semantics of ι to be a set of syntactic terms. We can't do the same here as we want to model term of type nat by ordinary natural numbers if possible. However, given a natural number n, we have a term which represents it, namely $\bar{\mathsf{s}}^{[\![t]\!]}\,\bar{0}$, so below we prove Theorem 3.19 which connects syntax and semantics analogously to Corollary 2.44 from our study of the simply-typed λ-calculus. Once this connection is established, the proof of adequacy has a similar logical structure, albeit with some complications in PCF due to non-termination.

However, the possibility of non-termination makes the connection between syntax and semantics discussed above more complicated than in the simply-typed case. The reason is that while we only need a result at type nat, such a term can be constructed out of subterms of higher type. We need a way of generalising the property we need at type nat to higher types, in a way which uniformly tracks the issues caused by the possibility of non-termination. In the simply-typed λ-calculus, we saw a way in which a property at base type can be extended uniformly to all types: logical predicates. In this case, we are looking for a *relation* between syntax and semantics which works uniformly at all types, so we need the same idea but for relations instead of predicates.

3.6.1 Logical relations

Since contextual equivalence is all about the observable behaviour of terms when they are used in larger programs, and an observation in this case means a returned value of type nat, we need to study the relationship between denotations and computations at this type. We want to find out whether we may deduce something about a term whose interpretation is a natural number. In particular we want to show that if $\vdash t : \mathsf{nat}$ is derivable then

$$[\![t]\!] = n \in \mathbb{N}_\perp \qquad \text{implies} \qquad t \twoheadrightarrow_\beta \bar{\mathsf{s}}^n\bar{0},$$

and more generally we would like once again that if two typeable terms have the same denotations then they are contextually equivalent.

We define

$$\mathsf{PCFTrm}_\sigma = \{t \in \mathsf{PCFTrm} \mid \vdash t : \sigma \text{ is derivable}\}.$$

To prove the desired statement we generalize our notion of logical predicate to *logical relations*. In Section 2.5 we picked out sets of terms at each type. In this generalization we still do something at each type.

What we do at type σ is to define a relation between

$$[\![\sigma]\!] \qquad \text{and} \qquad \mathsf{PCFTrm}_\sigma.$$

This is done in a similar manner to the idea of defining logical predicates in that we define the relation at base type, and it is then completely determined at function types by how it is defined at lower types, and this occurs by checking that application to suitable inputs in the relation produces outputs in the relation.

We note that the idea of a *logical relation* does not have a fixed definition in the literature, and that it might be more productive to think of it as a technique that is used to show properties of programming languages.

In these notes we have two instances of a logical relation and we define formally those two instances. The first logical relation we look at connects denotations with terms, while the one we introduce in Section 3.7 connects denotations.

Definition 46: dt-logical relation

A (binary) **dt-logical relation** is a family of binary relations, one at each type, so that

$$\mathcal{R}_\tau$$

relates elements of‘

$$[\![\tau]\!] \qquad \text{and} \qquad \mathsf{PCFTrm}_\tau$$

with the property that at type $\rho \to \sigma$ it is the case that

$$f\ \mathcal{R}_{\rho\to\sigma}\ t \qquad \text{if and only if} \qquad (d,u) \in \mathcal{R}_\rho \text{ implies } f\,d\ \mathcal{R}_\sigma\ tu.$$

3.6.2 A logical relation for adequacy

In this section we only consider dt-logical relations, and we use just logical relation to talk about these. We define a logical relation which allows us to show the property given in the previous section, connecting denotational semantics to β-reduction as a stepping stone to proving adequacy.

In order to define the logical relation $\mathcal{A}$ we give the relation at base type as encoding what we would like to establish: For $x \in \mathbb{N}_\perp$ and $t \in \mathsf{PCFTrm}_{\mathsf{nat}}$

$$x\ \mathcal{A}_{\mathsf{nat}}\ t \qquad \text{if and only if} \qquad x \in \mathbb{N} \text{ implies } t \xrightarrow{\beta\!\!\!\!\rightarrow} \bar{\mathsf{s}}^x\bar{\mathsf{0}}.$$

We note that we may immediately list some instances of the relation: For $n \in \mathbb{N}$ we have

$$n\ \mathcal{A}_{\mathsf{nat}}\ \bar{\mathsf{s}}^n\bar{\mathsf{0}}.$$

We want to show that in general, for terms t such that $\vdash t : \tau$ is derivable we have, for example,

$$[\![t]\!]\ \mathcal{A}_\tau\ t,$$

which then allows us to show a number of useful properties for denotations.

We note that while the relation only talks about terms that can be typed based on the empty type environment some of the statements we need to show to get to our desired result are concerned with terms that have free variables.

We establish some properties of the logical relation $\mathcal{A}$ at type nat which serve as base cases for our desired statement.

Lemma 3.14

Let r, s, and t be PCF terms such that $\vdash r, s, t : \mathsf{nat}$ is derivable and further let $d, d', d'' \in \mathbb{N}_\perp$. If

$$d\ \mathcal{A}_{\mathsf{nat}}\ t$$

then the following hold:

(a) $\mathsf{succ}\, d\ \mathcal{A}_{\mathsf{nat}}\ \bar{\mathsf{s}}t$.

(b) $\mathsf{pred}\, d\ \mathcal{A}_{\mathsf{nat}}\ \mathsf{pred}\, t$.

(c) If further $d'\ \mathcal{A}_{\mathsf{nat}}\ r$ and $d''\ \mathcal{A}_{\mathsf{nat}}\ s$ then $\mathsf{ifz}\, d d' d''\ \mathcal{A}_{\mathsf{nat}}\ \mathsf{ifz}\, trs$.

Proof. *This is an exercise about understanding the notation and how to reason with the logical relation at base type.*

See page 307.

In order to obtain corresponding results for the remaining term constructors for PCF we need to collect more information about the logical relation $\mathcal{A}$. We note that the partial order for the semantics and the applicative preorder for terms are closely related.

Lemma 3.15

Let τ be type and assume that u and t are PCF terms such that $\vdash u, t : \tau$ is derivable. Then the following hold.

(a) $\bot\ \mathcal{A}_\tau\ t$.

(b) If $d, d' \in [\![\tau]\!]$ with $d' \leq d$ and $d\ \mathcal{A}_\tau\ t$ then $d'\ \mathcal{A}_\tau\ t$.

(c) If S is a directed subset of $[\![\sigma]\!]$ such that $s\ \mathcal{A}_\tau\ t$ holds for all $s \in S$ then $\bigvee^{\uparrow} S\ \mathcal{A}_\tau\ t$.

(d) If $d \in [\![\tau]\!]$, $d\ \mathcal{A}_\tau\ t$ and $t \preceq t'$ then $d\ \mathcal{A}_\tau\ t'$.

Proof. See page 257.

We address the term constructors recursion and abstraction.

Lemma 3.16

Let t be a PCF term such that $\vdash t : \tau$ is derivable.

(a) If $\tau = \sigma \to \sigma$ for some type σ and $f \in [\![\sigma]\!] \Rightarrow [\![\sigma]\!]$ with

$$f\ \mathcal{A}_{\sigma\to\sigma}\ t$$

then

$$\mathsf{fix}\, f\, \mathcal{A}_\sigma\, \mathsf{rec}\, t$$

(b) If ρ is a type then if $f \in [\![\rho]\!] \Rightarrow [\![\tau]\!]$ such that for all $d\ \mathcal{A}_\rho\ a$ we have

$$f d\ \mathcal{A}_\sigma\ t[a/x]$$

then

$$f\ \mathcal{A}_{\rho\to\tau}\ \lambda x{:}\rho.\, t.$$

Proof. *This is an exercise about understanding the notation and how to reason with the logical relation.*

See page 307.

3.6.3 The proof of adequacy

We show that our logical relation is constructed correctly, in that it is possible to prove that it relates the denotation of any term to the term itself. In order to do this, we must show that applies to terms with free variables when we substitute suitable terms for these free variables.

Lemma 3.17

Let t be a PCF term such that $\Gamma \vdash t : \tau$ is derivable for some type environment Γ, and let

$$\{(x_1, \alpha_1), \ldots, (x_k, \alpha_k)\} = \{(x, \alpha) \mid x \in \mathsf{Vars},\ \Gamma \vdash x : \alpha \text{ is derivable}\}.$$

Further assume that for $1 \leq i \leq k$ we have a term a_i and $d_i \in [\![\alpha_i]\!]$ such that

$$d\ \mathcal{A}_{\alpha_i}\ a_i.$$

Then for the valuation ϕ that maps each x_i to d_i we have

$$[\![(\Gamma, t)]\!]_\phi\ \mathcal{A}_\tau\ t[a_1/x_1] \cdots [a_k/x_k].$$

Proof. See page 258.

For terms that don't contain free variables there is no need to carry out any substitutions, so in particular we obtain from the previous result the intended statement which tells us how tightly the logical relation connects such terms and their denotations:

Corollary 3.18

If $\vdash t : \tau$ is derivable for a PCF term t then $[\![t]\!]\ \mathcal{A}_\tau\ t$.

Theorem 3.19

If $\vdash t : \mathsf{nat}$ is derivable for a PCF term t then if $[\![t]\!] \in \mathbb{N}$ we have

$$t \xrightarrow{\beta}\!\!\!\!\rightarrow\ \bar{\mathsf{s}}^{[\![t]\!]}\,\bar{0}.$$

Proof. This statement follows from the definition of $\mathcal{A}$ via the preceding result.

This is a powerful statement connecting our semantics to computations in the language: it tells us that if we can calculate the interpretation of a term of type nat that contains no free variables then the computation we obtain from that term matches its interpretation.

We may combine it with Proposition 3.11 to conclude that if $\vdash t : \mathsf{nat}$ is derivable then we can find $n \in \mathbb{N}$ with

$$t \xrightarrow{\beta}\!\!\!\!\rightarrow \bar{\mathsf{s}}^n\bar{0} \qquad \text{if and only if} \qquad [\![t]\!] = n.$$

For terms of base type that don't contain free variables we thus get a very strong connection between the semantics of a term and its behaviour under reduction. This raises two obvious questions: Can we generalize this to other types, and can we generalize this to terms with free variables?

We don't have the space to address the latter, but as far as the former goes we may show the following order-theoretic version of adequacy.

Proposition 3.20

Let t and t' be terms and τ be a type such that $\vdash t, t : \tau$ is derivable. We have that

$$[\![t]\!] \leq [\![t']\!] \qquad \text{implies} \qquad t \trianglelefteq t'.$$

Proof. To show that under the given conditions we have $t \trianglelefteq t'$, assume that f is a term such that $\vdash f : \tau \to \mathsf{nat}$ is derivable.

We want to show that

$$f\,t \preceq f\,t'.$$

But if we can find $n \in \mathbb{N}$ with $f\,t \xrightarrow{\beta}\!\!\!\!\rightarrow \bar{\mathsf{s}}^n\bar{0}$ we know from Proposition 3.9 that

$$n = [\![f\,t]\!] = [\![f]\!]([\![t]\!]) \leq [\![f]\!]([\![t']\!])$$

since $[\![f]\!]$ is an order-preserving function, and since n is a maximal element of $\mathbb{N}_\perp$ it follows that

$$n = [\![f]\!]([\![t']\!]),$$

which, using Theorem 3.19, tells us that

$$f\,t' \xrightarrow{\beta}\!\!\!\!\rightarrow \bar{\mathsf{s}}^n\bar{0},$$

which ensures that $f\,t \preceq f\,t'$ which in turn gives $t \trianglelefteq t'$.

Corollary 3.21: adequacy

Let t and t' be PCF terms and τ be a type such that $\vdash t, t' : \tau$ is derivable. If

$$[\![t]\!] = [\![t']\!] \qquad \text{then} \qquad t \simeq t'.$$

Proof. This is immediate from the theorem since $[\![t]\!] = [\![t']\!]$ implies both, $[\![t]\!] \leq [\![t']\!]$ as well as $[\![t']\!] \leq [\![t]\!]$, and the first of these implies $t \trianglelefteq t'$ while the second gives us $t' \trianglelefteq t$, and together these mean that t and t' are contextually equivalent by definition.

Example 3.11. Consider the term from Section 4.3.1 and Example 3.10,

$$\mathsf{rec}\,\lambda f : \mathsf{nat} \to \mathsf{nat}.\ \lambda x : n.\ \mathsf{ifz}\,x[\bar{0}][\bar{\mathsf{s}}\,f\,\mathsf{pred}\,x].$$

We may argue that this term is contextually equivalent to the term

$$\lambda x : \mathsf{nat}.\, x.$$

In Example 3.10 we show that the denotation of the first term is the identity function on $\mathbb{N}_\perp$, and it is not hard to show that this is also the case for the second term since, for $y \in \mathbb{N}_\perp$,

$$[\![\lambda x : \mathsf{nat}.\, x]\!]\, y = [\![(x : \mathsf{nat}, x)]\!]_{[x \mapsto y]} = y.$$

Hence the two terms have the same denotation and so must be contextually equivalent.

RExercise 41. *This exercise aims to show how one may reason about PCF terms making use of the adequacy result.*

Assume we have PCF terms

$$M : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \qquad \text{and} \qquad A : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$$

such that the denotations return $\perp$ if one of the inputs is $\perp$ and otherwise

$$([\![M]\!]\, n)m = nm$$
$$([\![A]\!]\, n)m = n + m$$

Prove that the following PCF terms of type $\mathsf{nat} \to \mathsf{nat}$ are contextually equivalent:

$$\mathsf{rec}\,(\lambda f : \mathsf{nat} \to \mathsf{nat}.\, \lambda n : \mathsf{nat}.\, \mathsf{ifz}\, n \bar{0} [A(M(\bar{\mathsf{s}}(\bar{\mathsf{s}}\bar{0}))n)[f(Pn)]])$$

and

$$\lambda x : \mathsf{nat}.\, M x [Ax(\bar{\mathsf{s}}\bar{0})]$$

A solution is given on page 307.

3.6.4 Further results for denotations

Note that by moving to the idea of having a partial order for both terms and denotations, we are able to provide quite nuanced statements connecting a term and its denotations. The following result is an immediate consequence from the previous one given that contextual equivalence is defined via the contextual preorder.

Corollary 3.22: full abstraction for nat

If t and t' are PCF terms such that $\vdash t, t' : \mathsf{nat}$ is derivable then

$$t \trianglelefteq t' \qquad \text{if and only if} \qquad t \preceq t' \qquad \text{if and only if} \qquad [\![t]\!] \leq [\![t']\!].$$

Proof. In Proposition 3.5 we already seen that we have implications from left to right for both parts for all typeable terms, so it is sufficient to establish that $[\![t]\!] \leq [\![t']\!]$ implies $t \trianglelefteq t'$, which we now have for terms of type nat that do not

contain free variables.

One might decide that we have full abstraction now, since the above implies that

$$t \simeq t' \qquad \text{if and only if} \qquad [\![t]\!] = [\![t']\!],$$

but we only have this for terms of type nat! In the following section we show that this result does not hold for all types.

We are able to show one more property that establishes that we may replace syntactic experiments by semantic ones.

Proposition 3.23

Let t and t' be PCF terms and τ be a type such that $\vdash t, t' : \tau$ is derivable. Then the following statements are equivalent:

(a) $t \trianglelefteq t'$;

(b) $t \preceq t'$;

(c) $[\![t]\!]\ \mathcal{A}_\tau\ t'$.

Proof. We know from Proposition 3.5 that (i) implies (ii).

We further know from Corollary 3.18 that $[\![t]\!]\ \mathcal{A}_\tau\ t'$, and if we know $t \preceq t$ then by Lemma 3.15

$$[\![t]\!]\ \mathcal{A}_\tau\ t',$$

so (ii) implies (iii).

For (iii) implies (i), assume that $[\![t]\!]\ \mathcal{A}_\tau t'$, and that $\vdash f : \tau \to \mathsf{nat}$ is derivable for some PCF term f. Again we may invoke Corollary 3.18 to find that

$$[\![f]\!]\ \mathcal{A}_\tau\ f,$$

which by the definition of a logical relation applied to the assumption $[\![t]\!]\ \mathcal{A}_\tau\ t'$ gives us

$$[\![f\,t]\!] = [\![f]\!]\,([\![t]\!])\ \mathcal{A}_{\mathsf{nat}}\ f\,t'.$$

If we now have that $f\,t \overset{\beta}{\twoheadrightarrow} \bar{\mathsf{s}}^n\bar{\mathsf{0}}$ then $[\![f\,t]\!] = [\![\bar{\mathsf{s}}^n\bar{\mathsf{0}}]\!] = n$ and so

$$n\ \mathcal{A}_{\mathsf{nat}}\ f\,t',$$

which by the definition of the logical relation $\mathcal{A}$ gives us $f\,t' \overset{\beta}{\twoheadrightarrow} \bar{\mathsf{s}}^n\bar{\mathsf{0}}$ which allows us to conclude that $t \preceq t'$ as required.

3.7 Failure of Full Abstraction

Based on the development carried out to this point it is a natural question whether we obtain another full abstraction result (compare Section 2.8.7), this time for the language PCF.

Since we are now in a more sophisticated setting there are two possibilities for interpreting this: We could ask for

$$[\![t]\!] \leq [\![t']\!] \qquad \text{if and only if} \qquad t \trianglelefteq t',$$

or

$$[\![t]\!] = [\![t']\!] \qquad \text{if and only if} \qquad t \simeq t',$$

where t and t' are typeable terms of the same type which contain no free variables.

The first statement implies the second (see the proof of Corollary 3.21), so it is stronger. However, neither of them holds.

3.7.1 Functions that do not interpret PCF terms

Example 3.12 (The parallel or function). An important example of a function that exists in our model but is not the interpretation of a PCF term is the following:

$$\mathrm{por} : \mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp \Rightarrow \mathbb{N}_\perp$$

given by

$$\mathrm{por}\ x\ y = \begin{cases} 0 & x = 0 \text{ or } y = 0 \\ 1 & x, y \in \mathbb{N} \setminus \{0\} \\ \perp & \text{else} \end{cases}$$

We may think of this function as implementing disjunction where we think of the inputs as encoding booleans as we do in the conditional, that is 0 encodes true and any other number encodes false. If at least one of our inputs is an encoding for true then we return 0, which encodes true, if both are encodings for false we return false, and otherwise we haven't got enough information to return anything other than $\perp$.

This seems harmless enough, but we need to bear in mind that the function por might be used as an input to the denotation of PCF terms of type

$$(\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat},$$

and it turns out that this idea allows us to show that our model is not fully abstract.

We give an informal argument that there can't be a PCF term whose interpretation is por. If such a term first triggered the computation that gives the first argument and if that computation did not terminate then the whole computation would not terminate, and so be interpreted by $\perp$. The second argument might give the number 0 and so the computation could in principle have returned 0. If the computation that gives the second argument is triggered first exactly the same issue arises.

Indeed this is a limitation of the way computations are carried out in PCF: there is no term whose interpretation is por.

Knowing that there is no term that implements por would not be particularly worrisome by itself, and indeed none of the properties of models that we have considered demands that every element in the model must be the interpretation of a term. However, it turns out that there are terms whose denotation can tell whether they have been given por as an argument, and these denotations can give different outputs for that argument.

3.7.2 Terms that are sensitive to por

We can build closed terms

$$\mathsf{portest}_0 \qquad \text{and} \qquad \mathsf{portest}_1$$

such that

$$\mathsf{portest}_0 \simeq \mathsf{portest}_1$$

but

$$[\![\mathsf{portest}_0]\!]\,(\mathrm{por}) = 0 \qquad \text{and} \qquad [\![\mathsf{portest}_1]\!]\,(\mathrm{por}) = 1,$$

and so terms that are contextually equivalent may have different denotations.

We define the two terms, making use of the term P defined as

$$P = \lambda x\!:\!\mathsf{nat}.\ \mathsf{ifz}\,x[\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x][\mathsf{ifz}\,(\mathsf{pred}\,x)\bar{0}(\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x)].$$

We now set

$$\begin{aligned}
&\mathsf{portest}_0\\
&\quad = \lambda f\!:\!\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}.\ \mathsf{ifz}\,(f\,(\bar{0})(\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x))\\
&\qquad\qquad [\mathsf{ifz}\,(f\,(\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x)(\bar{0}))\\
&\qquad\qquad\quad (\mathsf{ifz}\,(P\,f\,(\bar{\mathsf{s}}\bar{0})(\bar{\mathsf{s}}\bar{0}))\langle\bar{0}\rangle\langle\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x\rangle)\\
&\qquad\qquad\quad (\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x)]\\
&\qquad\qquad [\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x]
\end{aligned}$$

while

$$\begin{aligned}
&\mathsf{portest}_1\\
&\quad = \lambda f\!:\!\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}.\ \mathsf{ifz}\,(f\,(\bar{0})(\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x))\\
&\qquad\qquad [\mathsf{ifz}\,(f\,(\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x)(\bar{0}))\\
&\qquad\qquad\quad (\mathsf{ifz}\,(P\,f\,(\bar{\mathsf{s}}\bar{0})(\bar{\mathsf{s}}\bar{0}))\langle\bar{\mathsf{s}}\bar{0}\rangle\langle\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x\rangle)\\
&\qquad\qquad\quad (\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x)]\\
&\qquad\qquad [\mathsf{rec}\,\lambda x\!:\!\mathsf{nat}.\ x]
\end{aligned}$$

To work out what $[\![\mathsf{portest}_0]\!]$ and $[\![\mathsf{portest}_1]\!]$ do when we apply them to por we first have to work out the denotation of P, which gives us a function from $\mathbb{N}_\perp$ to $\mathbb{N}_\perp$.

$$\begin{aligned}
[\![P]\!]\,x &= \mathrm{ifz}\,x\perp(\mathrm{ifz}(\mathrm{pred}\,x)0\perp)\\
&= \begin{cases} \perp & x = \perp \text{ or } x = 0\\ 0 & x = 1\\ \perp & \text{else.}\end{cases}
\end{aligned}$$

We calculate

$$\begin{aligned}
[\![\mathsf{portest}_0]\!]\ \mathrm{por} &= \mathrm{ifz}\,\big(\mathrm{por}\,0\perp\big)\,\big(\,\mathrm{ifz}[\mathrm{por}\,\perp 0][\mathrm{ifz}([\![P]\!]\ \mathrm{por}\,1)0\perp][\perp]\big)\,\perp\\
&= \mathrm{ifz}\,0(\mathrm{ifz}\,0(\mathrm{ifz}([\![P]\!]\,1)0\perp)\perp)\perp\\
&= \mathrm{ifz}\,0(\mathrm{ifz}\,0(\mathrm{ifz}\,00\perp)\perp)\perp\\
&= \mathrm{ifz}\,0(\mathrm{ifz}\,00\perp)\perp\\
&= \mathrm{ifz}\,00\perp
\end{aligned}$$

$$= 0$$

while

$$\begin{aligned}
[\![\mathsf{portest}_1]\!]\ \mathrm{por} &= \mathrm{ifz}\ (\mathrm{por}\ 0\bot)\ (\ \mathrm{ifz}[\mathrm{por}\ \bot 0][\mathrm{ifz}([\![P]\!]\ \mathrm{por}\ 11)1\bot][\bot])\ \bot \\
&= \mathrm{ifz}\,0(\mathrm{ifz}\,0(\mathrm{ifz}([\![P]\!]\,1)1\bot)\bot)\bot \\
&= \mathrm{ifz}\,0(\mathrm{ifz}\,0(\mathrm{ifz}\,01\bot)\bot)\bot \\
&= \mathrm{ifz}\,0(\mathrm{ifz}\,01\bot)\bot \\
&= \mathrm{ifz}\,01\bot \\
&= 1.
\end{aligned}$$

3.7.3 Contextual equivalence of the terms testing for por

In order to show that $\mathsf{portest}_0$ and $\mathsf{portest}_1$ are contextually equivalent by Proposition 3.23 it is sufficient to show that

$$\mathsf{portest}_0 \preceq \mathsf{portest}_1 \qquad \text{as well as} \qquad \mathsf{portest}_1 \preceq \mathsf{portest}_0.$$

To show that it is sufficient to know that for all terms f

$$\vdash f : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$$

we have

$$\mathsf{portest}_0 f \preceq \mathsf{portest}_1 f \qquad \text{and} \qquad \mathsf{portest}_1 f \preceq \mathsf{portest}_0 f$$

and these are now terms of type nat, where we have a full abstraction result in the form of Theorem 3.19. Hence if we can show that for all such f we have

$$[\![\mathsf{portest}_0 f]\!] = [\![\mathsf{portest}_1 f]\!]$$

then we may concludes that

$$\mathsf{portest}_0 \simeq \mathsf{portest}_1.$$

We make a start by calculating the semantics of these terms.

Lemma 3.24

For a function $g : \mathbb{N}_\bot \longrightarrow \mathbb{N}_\bot \Rightarrow \mathbb{N}_\bot$ we say we say that g satisfies property (*) if and only if it satisfies

$$g0\bot = 0 \qquad g\bot 0 = 0 \qquad g11 = 1.$$

Then for a term f such that $\vdash f : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ is derivable we have

$$[\![\mathsf{portest}_0 f]\!] = \begin{cases} 0 & [\![f]\!] \text{ satisfies property (*)} \\ \bot & \text{else} \end{cases}$$

while

$$[\![\mathsf{portest}_1 f]\!] \begin{cases} 1 & [\![f]\!] \text{ satisfies property (*)} \\ \bot & \text{else} \end{cases}$$

Proof. *This exercise is directed at students who want to understand how the two test terms operate.*

See page 309.

If we can now show that there is no term f such that $\vdash f : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ is derivable and such that $[\![f]\!]$ has property (*) then we may conclude that $\mathsf{portest}_0$ and $\mathsf{portest}_1$ are contextually equivalent since the denotations given in the previous result agree.

3.7.4 Denotational logical relations

In order to show that functions that have property (*) are not definable we introduce a second notion of logical relation, this time one that only connects elements on the denotation side.

Definition 47: d-logical relation

A k-ary **d-logical relation** is a family of k-ary relations, one at each type, so that

$$\mathcal{R}_\tau$$

consists of k-tuples of elements of

$$[\![\tau]\!]$$

with the property that at type $\rho \to \sigma$ it is the case that

$(f_1, f_2, \ldots, f_k) \in \mathcal{R}_{\rho\to\sigma}$ if and only if
$(d_1, d_2, \ldots, d_k) \in \mathcal{R}_\rho$ implies $(f_1 d_1, f_2 d_2, \ldots, f_k d_k) \in \mathcal{R}_\sigma$.

We may establish results that mirror ideas for dt-logical relations discussed in Section 3.6. In the proof we only consider the binary case to keep the notation manageable since the general case works in exactly the same way, just with more indices. When we talk of logical relations in this section we always mean d-logical relations.

Lemma 3.25

For a d-logical relation $\mathcal{R}$ the following statements hold.

(a) If the k-tuple consisting of $\bot$ is an element of $\mathcal{R}_{\mathsf{nat}}$ then the same is true for all types.

(b) If for a type τ we have, for $n \in \mathbb{N}$,

$$(d_1^n, d_2^n, \ldots d_k^n) \in \mathcal{R}_\tau$$

and for all $1 \leq i \leq k$ and all $n \in \mathbb{N}$ we have

$$d_i^n \leq d_i^{n+1}$$

then

$$(\bigvee_{n\in\mathbb{N}} d_1^n, \bigvee_{n\in\mathbb{N}} d_2^n, \ldots, \bigvee_{n\in\mathbb{N}} d_k^n) \in \mathcal{R}_\tau.$$

(c) If the tuple consisting of $\bot$ is in $\mathcal{R}_{\mathsf{nat}}$ and $f_i \in [\![\sigma \to \sigma]\!]$ for $1 \leq i \leq k$ for some type σ and

$$(f_1, f_2, \ldots, f_n) \in \mathcal{R}_{\sigma\to\sigma}$$

then

$$(\mathsf{fix}\, f_1, \mathsf{fix}\, f_2, \ldots, \mathsf{fix}\, f_k) \in \mathcal{R}_\sigma.$$

Proof. We show each statement for the binary case.

(a) We show this statement by induction. The base case is given by assumption.

For the step case we know that the least element of a function type $\rho \to \sigma$ is the constant function $k_\bot$ that assigns $\bot$ in $[\![\sigma]\!]$ to all inputs. If we have $(d, e) \in \mathcal{R}_\rho$ then

$$(k_\bot d, k_\bot e) = (\bot, \bot) \in \mathcal{R}_\sigma$$

by the induction hypothesis, so $(k_\bot, k_\bot) \in \mathcal{R}_{\rho\to\sigma}$ by the definition of logical relation.

(b) At the base type this claim is trivial since every directed set has a maximal element, which means that the chains are constant at some point, and beyond that point the assumption gives the required statement.

In the step case we have functions, and to remind ourselves of this we assume these are (moving to subscripts rather than superscripts) f_n and g_n. We further know that $\bigvee_{n\in\mathbb{N}} f_n$ is computed pointwise, and the same is true for the g_n. Assume that $d\mathcal{R}_\rho e$. Then by the definition of logical relation $(f_n, g_n) \in \mathcal{R}_{\rho\to\sigma}$ gives us $(f_n d, g_n e) \in \mathcal{R}_\rho$, and so by the induction hypothesis

$$((\bigvee_{n\in\mathbb{N}} f_n)d, (\bigvee_{n\in\mathbb{N}} f_n)e_n) = (\bigvee_{n\in\mathbb{N}} f_n d, \bigvee_{n\in\mathbb{N}} g_n e) \in \mathcal{R}_\sigma$$

which implies, again by the definition of logical relation, that

$$(\bigvee_{n\in\mathbb{N}} f_n, \bigvee_{n\in\mathbb{N}} g_n) \in \mathcal{R}_{\rho\to\sigma}$$

as required.

(c) Assume we know that $(f, g) \in \mathcal{R}_{\sigma\to\sigma}$. We have that

$$\mathsf{fix}\, f = \bigvee_{n\in\mathbb{N}} f^n \bot,$$

and similarly for $\mathsf{fix}\, g$. By the previous part it is therefore sufficient to show that for all $n \in \mathbb{N}$ we have that

$$(f^n \bot, g^n \bot) \in \mathcal{R}_\sigma.$$

We prove this by induction. If $n = 0$ we have that

$$(f^0\bot, g^0\bot) = (\bot, \bot) \in \mathcal{R}_\sigma$$

by assumption and the first part. For the step case we know from $(f, g) \in \mathcal{R}_{\sigma\to\sigma}$ by the definition of d-logical relation and the induction hypothesis that

$$(f^{n+1}\bot, g^{n+1}\bot) = (f(f^n\bot), g(g^n\bot)) \in \mathcal{R}_\sigma,$$

which establishes the claim.

Definition 48: $\mathcal{R}$-invariance

Let $\mathcal{R}$ be a k-ary logical relation and let t be a term such that $\vdash t : \tau$ is derivable for some type τ. We say that t is **$\mathcal{R}$-invariant** if and only if the k-tuple that consists of k copies of $[\![t]\!]$ is an element of $\mathcal{R}_\tau$.

We may now state the theorem that tells us how logical relations give us information about terms, which in particular allows us to show that some elements of our denotational dcpos are not definable.

Theorem 3.26

Let $\mathcal{R}$ be a k-ary logical relation such that

- the k-tuple consisting of k copies of $\bot$ is in $\mathcal{R}_{\mathsf{nat}}$.
- the k-tuple consisting of k copies of 0 is in $\mathcal{R}_{\mathsf{nat}}$;
- the k-tuples consisting of k copies of succ, and k copies of pred, respectively, are in $\mathcal{R}_{\mathsf{nat}\to\mathsf{nat}}$;
- the k-tuple consisting of k copies of ifz is in $\mathcal{R}_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}$.

If t is a term such that $\vdash t : \tau$ is derivable for some type τ then $[\![t]\!]$ is $\mathcal{R}$-invariant.

Proof. We prove the statement for $k = 2$, the proof for the other cases is exactly the same. We have to show that for all terms t with $\vdash t : \sigma$ derivable we have

$$([\![t]\!], [\![t]\!]) \in \mathcal{R}_\sigma.$$

We extend the notion for terms that contain free variables as follows: If $\Gamma \vdash t : \tau$ is derivable, then we show that if

$$\lfloor\Gamma\rfloor = \{x_1, x_2, \ldots, x_n\}$$

and

$$\Gamma \vdash x_i : \alpha_i \text{ derivable for } 1 \leq i \leq n$$

and we choose

$$(d_i, e_i) \in \mathcal{R}_{\alpha_i}$$

for all $1 \leq i \leq n$ then for the valuations ϕ and ψ given by

$$x_i \mapsto d_i \qquad \text{and} \qquad x_i \mapsto e_i$$

respectively we have

$$(\llbracket t \rrbracket_\phi, \llbracket t \rrbracket_\psi) \in \mathcal{R}_\tau.$$

We prove the result by induction over the typing rules.

bcZero We have that $(\llbracket \bar{0} \rrbracket, \llbracket \bar{0} \rrbracket) = (0, 0) \in \mathcal{R}_{\mathsf{nat}}$ by assumption.

bcVar If we have a term that is a variable, say x, then we know that the corresponding type environment is of the form $\Gamma, x : \tau$ and then for the valuations ϕ as given above we have

$$\llbracket (\Gamma, x) \rrbracket_\phi = \phi x,$$

and correspondingly for the valuation ψ. Hence we obtain

$$(\llbracket (\Gamma, x) \rrbracket_\phi, \llbracket (\Gamma, x) \rrbracket_\psi) = (\phi x, \psi x) \in \mathcal{R}_\tau$$

by construction of ϕ and ψ.

scSucc If know that $\Gamma \vdash \bar{\mathsf{s}} t : \mathsf{nat}$ is derivable then by the induction hypothesis we know that $(\llbracket (\Gamma, t) \rrbracket_\phi, \llbracket (\Gamma, t) \rrbracket_\psi) \in \mathcal{R}_{\mathsf{nat}}$, and then by the assumption for succ and using the definition of logical relation we get that

$$(\llbracket (\Gamma, \bar{\mathsf{s}} t) \rrbracket_\phi, \llbracket (\Gamma, \bar{\mathsf{s}} t) \rrbracket_\psi) = (\mathrm{succ}\ \llbracket (\Gamma, t) \rrbracket_\phi, \mathrm{succ}\ \llbracket (\Gamma, t) \rrbracket_\psi) \in \mathcal{R}_{\mathsf{nat}}$$

as required.

scPred If we know that $\Gamma \vdash \mathsf{pred}\, t : \mathsf{nat}$ is derivable we may proceed as in the previous case.

scIfz If $\vdash \mathsf{ifz}\, trs : \mathsf{nat}$ is derivable then we may proceed much as in the previous two cases, but we spell out the argument because the type is more complicated. We have by the induction hypothesis that

$$\begin{aligned}&(\llbracket (\Gamma, t) \rrbracket_\phi, \llbracket (\Gamma, t) \rrbracket_\psi), (\llbracket (\Gamma, r) \rrbracket_\phi, \llbracket (\Gamma, r) \rrbracket_\psi),\\ &\quad (\llbracket (\Gamma, s) \rrbracket_\phi, \llbracket (\Gamma, s) \rrbracket_\psi) \in \mathcal{R}_{\mathsf{nat}}.\end{aligned}$$

Hence

$$\llbracket (\Gamma, \mathsf{ifz}\, trs) \rrbracket_\phi = \mathrm{ifz}\ \llbracket (\Gamma, t) \rrbracket_\phi \llbracket (\Gamma, r) \rrbracket_\phi \llbracket (\Gamma, s) \rrbracket_\phi$$

arises from applying ifz successively to three suitable arguments, and by the definition of logical relation we obtain that

$$\begin{aligned}&(\llbracket (\Gamma, \mathsf{ifz}\, trs) \rrbracket_\phi, \llbracket (\Gamma, \mathsf{ifz}\, trs) \rrbracket_\psi)\\ &\quad = (\mathrm{ifz}\ \llbracket (\Gamma, t) \rrbracket_\phi \llbracket (\Gamma, r) \rrbracket_\phi \llbracket (\Gamma, s) \rrbracket_\phi, \mathrm{ifz}\ \llbracket (\Gamma, t) \rrbracket_\psi \llbracket (\Gamma, r) \rrbracket_\psi \llbracket (\Gamma, s) \rrbracket_\psi)\end{aligned}$$

is an element of $\mathcal{R}_{\mathsf{nat}}$ as required.

scAbs If $\Gamma \vdash \lambda x : \rho.\, t : \rho \to \sigma$ is derivable we know by the induction hypothesis that for every $(d, e) \in \mathcal{R}_\rho$ we have that

$$(\llbracket (\Gamma x : \rho, t) \rrbracket_{\phi[x \mapsto d]}, \llbracket (\Gamma x : \rho, t) \rrbracket_{\psi[x \mapsto e]}) \in \mathcal{R}_\sigma$$

and so for $(d, e) \in \mathcal{R}_\rho$

$$\begin{aligned}&(\llbracket(\Gamma, \lambda x : \rho.\, t)\rrbracket_\phi d,\ \llbracket(\Gamma, \lambda x : \rho.\, t)\rrbracket_\psi e)\\ &\quad = (\llbracket(\Gamma x : \rho, t)\rrbracket_{\phi[x \mapsto d]},\ \llbracket(\Gamma x : \rho, t)\rrbracket_{\psi[x \mapsto e]}) \in \mathcal{R}_\sigma\end{aligned}$$

follows immediately which, by the definition of logical relation, gives us the desired

$$(\llbracket(\Gamma, \lambda x : \rho.\, t)\rrbracket_\phi,\ \llbracket(\Gamma, \lambda x : \rho.\, t)\rrbracket_\psi) \in \mathcal{R}_{\rho \to \sigma}.$$

scApp If we know that $\Gamma \vdash rs : \tau$ is derivable then we know that there is a type σ such that both,

$$\Gamma \vdash r : \sigma \to \tau \qquad\qquad \Gamma \vdash s : \sigma$$

are derivable, and by the induction hypothesis we obtain that

$$(\llbracket(\Gamma, r)\rrbracket_\phi,\ \llbracket(\Gamma, r)\rrbracket_\psi) \in \mathcal{R}_{\sigma \to \tau} \text{ and } (\llbracket(\Gamma, s)\rrbracket_\phi,\ \llbracket(\Gamma, s)\rrbracket_\psi) \in \mathcal{R}_\sigma$$

which by the definition of logical relation immediately give us that

$$\begin{aligned}&(\llbracket(\Gamma, rs)\rrbracket_\phi,\ \llbracket(\Gamma, rs)\rrbracket_\psi)\\ &\quad = (\llbracket(\Gamma, r)\rrbracket_\phi(\llbracket(\Gamma, s)\rrbracket_\phi),\ \llbracket(\Gamma, r)\rrbracket_\psi(\llbracket(\Gamma, s)\rrbracket_\psi)) \in \mathcal{R}_\tau\end{aligned}$$

as required.

scRec If know that $\Gamma \vdash \mathsf{rec}\, t : \tau$ is derivable then we know by the induction hypothesis that

$$(\llbracket(\Gamma, t)\rrbracket_\phi,\ \llbracket(\Gamma, t)\rrbracket_\psi) \in \mathcal{R}_{\tau \to \tau},$$

and since $\llbracket(\Gamma, \mathsf{rec}\, t)\rrbracket_\phi = \mathsf{fix}\ \llbracket(\Gamma, t)\rrbracket_\phi$ it is sufficient to show that

$$(\mathsf{fix}\ \llbracket(\Gamma, t)\rrbracket_\phi, \mathsf{fix}\ \llbracket(\Gamma, t)\rrbracket_\psi) \in \mathcal{R}_\tau,$$

which we obtain immediately from Lemma 3.25.

3.7.5 A definability logical relation for por

Property (*) defined above consists of three equalities and we show here how we may build a ternary logical relation to show that no PCF term has a denotation satisfying all three of these equalities.

We are interested in logical relations of the form

$$\mathcal{D}_\tau \subseteq \llbracket\tau\rrbracket^3.$$

At base type we define this to be

$$\mathcal{D}_{\mathsf{nat}} = \{(x, y, z) \in \mathbb{N}_\perp^3 \mid \ x = \perp \text{ or } y = \perp \text{ or } x = y = z\}.$$

At function types we know that we require

$$(f, g, h) \in \mathcal{D}_{\sigma \to \tau} \qquad \text{iff} \qquad (x, y, z) \in \mathcal{D}_\sigma \text{ implies } (f\,x, g\,y, h\,z) \in \mathcal{D}_\tau.$$

We show that $\mathcal{D}$ satisfies the assumptions from Theorem 3.26.

- We have that $(\bot, \bot, \bot)$ and $(0, 0, 0)$ are elements of $\mathcal{D}_{\mathsf{nat}}$.
- We check that
$$(\mathrm{succ}, \mathrm{succ}, \mathrm{succ}) \in \mathcal{D}_{\mathsf{nat}\to\mathsf{nat}} :$$
The various triples that can arise when applying this triple componentwise to elements of $\mathcal{D}_{\mathsf{nat}}$:
$$(\mathrm{succ}\,\bot, \mathrm{succ}\,y, \mathrm{succ}\,z) = (\bot, \mathrm{succ}\,y, \mathrm{succ}\,z) \in \mathcal{D}_{\mathsf{nat}},$$
$$(\mathrm{succ}\,x, \mathrm{succ}\,\bot, \mathrm{succ}\,z) = (\mathrm{succ}\,x, \bot, \mathrm{succ}\,z) \in \mathcal{D}_{\mathsf{nat}},$$
and
$$(\mathrm{succ}\,x, \mathrm{succ}\,x, \mathrm{succ}\,x) \in \mathcal{D}_{\mathsf{nat}},$$
so indeed $(\mathrm{succ}, \mathrm{succ}, \mathrm{succ}) \in \mathcal{D}_{\mathsf{nat}\to\mathsf{nat}}$. We may similarly show that triples consisting of pred are elements of $\mathcal{D}_{\mathsf{nat}\to\mathsf{nat}}$.
- To show that triples consisting of ifz are elements of $\mathcal{D}_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}$ we have to show that for all
$$(x, y, z), (x', y', z'), (x'', y'', z'') \in \mathcal{D}_{\mathsf{nat}}$$
we have
$$(\mathrm{ifz}\,xx'x'', \mathrm{ifz}\,yy'y'', \mathrm{ifz}\,zz'z'') \in \mathcal{D}_{\mathsf{nat}}.$$
For that we need to worry about the first two components not being $\bot$. For the first component this happens when
  - $x = 0$ and $x' \in \mathbb{N}$ or
  - $x \in \mathbb{N} \setminus \{0\}$ and $x'' \in \mathbb{N}$.

  and similar considerations apply to the second component. We note therefore that this can only happen if $x, x', y, y' \in \mathbb{N}$, so by the definition of $\mathcal{D}_{\mathsf{nat}}$ we must have $x = x' = x''$, $y = y' = y''$ and $z = z' = z''$, and so
$$(\mathrm{ifz}\,xx'x'', \mathrm{ifz}\,yy'y'', \mathrm{ifz}\,zz'z'') = (\mathrm{ifz}\,xx'x'', \mathrm{ifz}\,xx'x'', \mathrm{ifz}\,xx'x'') \in \mathcal{D}_{\mathsf{nat}}.$$

Hence if t is a term such that $\vdash t : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ is derivable then $[\![t]\!]$ is $\mathcal{D}$-invariant, and so
$$([\![t]\!], [\![t]\!], [\![t]\!]) \in \mathcal{D}_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}.$$

We now look at what happens when for an element f of $\mathbb{N}_\bot \Rightarrow (\mathbb{N}_\bot \Rightarrow \mathbb{N}_\bot)$ we form
$$(f, f, f).$$
For that triple to be an element of $\mathcal{D}_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}$ we would need that for all
$$(x, y, z), (x', y', z') \in \mathcal{D}_{\mathsf{nat}}$$
we have
$$(f\,xx', f\,yy', f\,zz') \in \mathcal{D}_{\mathsf{nat}}.$$
But we know that
$$(0, \bot, 1), (\bot, 0, 1) \in \mathcal{D}_{\mathsf{nat}},$$

while for a function f that satisfies property (*) we have

$$(f\,0\bot, f\,\bot 0, f\,1\,1) = (0, 0, 1) \notin \mathcal{D}_{\mathsf{nat}}$$

and so

$$(f, f, f) \notin \mathcal{D}_{\mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}}.$$

If there was a term t with the property that $[\![t]\!] = f$, where f satisfies property (*), then (f, f, f) would have to be an element of the logical relation, and we have just shown that this is impossible, and so such a function is not PCF-definable.

Further with $\mathsf{portest}_0$ and $\mathsf{portest}_1$ we have two terms which we now know are contextually equivalent, but whose denotations are different since they give different outputs for the input por. Hence the given model for PCF is not fully abstract.

Chapter 4

Mathematical Notions and Results

In this chapter we collect mathematical definitions and results that are required mostly for the two models we define, for the simply typed λ-calculus and for PCF.

Some of the concepts from the first year unit *Mathematical Techniques for Computer Science*, COMP11120, are relevant to our discussions. We repeat definitions here for convenience, and also for those students who did not take that unit.

4.1 Sets and Functions

The basic idea of our model for the simply typed λ-calculus is that we would like to interpret λ-abstractions as functions. For us, every function comes with a specified **source** and **target set**, and we write this as

$$f : S \longrightarrow T$$

to indicate that f takes inputs from the set S and produces outputs from the set T. We may then use a description of the underlying assignment to define a function, for example

$$\begin{aligned} \mathbb{R} &\longrightarrow \mathbb{R} \\ x &\longmapsto x^2, \end{aligned}$$

and if we need to give a name to a function we might use

$$\begin{aligned} g : \mathbb{R} &\longrightarrow \mathbb{R} \\ x &\longmapsto e^{\pi x}. \end{aligned}$$

One result we make repeated use of below is the idea that a function

$$f : S \longrightarrow T$$

is a **bijection** if and only if there is a function $g : T \longrightarrow S$ that is the **inverse of** f, that is it satisfies both

$$g \circ f = \mathrm{id}_S \qquad \text{and} \qquad f \circ g = \mathrm{id}_T .$$

In general, when we look at structures that are more sophisticated than sets, and we want to say that two such structures are very similar indeed (*isomorphic* is

the correct mathematical term) then asking for a (suitable) function that has an inverse is the correct way of defining this notion.[1]

We may then form new sets based on functions with a given source and target set by defining

$$\mathsf{Fun}(S, T)$$

to be the set of all functions with source set S and target set T.

For the simply typed λ-set, we model the type $\sigma \to \tau$ with the set $\mathsf{Fun}(\llbracket\sigma\rrbracket, \llbracket\tau\rrbracket)$, where $\llbracket\sigma\rrbracket$ is the interpretation of the type σ. The reader who has read about the simply typed λ-calculus may appreciate that as we go up in the hierarchy of types, we assemble nested occurrences of Fun. In AExercise 2 we ask you to think about some examples of this situation.

4.1.1 Functions with a finite source set

When we encounter the notion of a **valuation**, see Section 2.8.3 we find ourselves having to talk about functions with a finite source set, and we spend some time here to develop some notation for this situation. Also, whenever we use a finite set to interpret the base type, a term in the simply typed λ-calculus is denoted by a function of this kind.

Example 4.1. To write concrete examples of denotations of terms, it is useful to have a compact notation for functions with a finite source set. A finite function can be written as list of all of its outputs. for example, consider the function on the set $\{0, 1, 2, 3, 4\}$ which adds 1 to its argument (mod 5). This can be defined as

$$\begin{aligned} a : \{0,1,2,3,4\} &\to \{0,1,2,3,4\} \\ 0 &\mapsto 1 \\ 1 &\mapsto 2 \\ 2 &\mapsto 3 \\ 3 &\mapsto 4 \\ 4 &\mapsto 0 \end{aligned}$$

If we have to write lots of such functions, we need an inline notation. We introduce a shorthand notation whereby a function is written as a set of assignments, with the target set indicated as a superscript. For example, the function above is written as

$$\{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 0\}^{\{0,1,2,3,4\}}.$$

In practice, we will always ensure the target is clear from the context, and simply write $\{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 0\}$.

A further example of using this notation in a slightly more sophisticated setting appears in the following section.

[1] We may have conditions for our functions, for example see the idea of an order-preserving function (Definition 53), and demanding that both, a function and its inverse satisfy such conditions is stronger than asking for a bijection.

4.1.2 Higher order functions

When thinking about how to denote terms of a higher order type we have to think about functions that live in sets of the form

$$\mathsf{Fun}(\mathsf{Fun}(S,T),U) \qquad \text{and} \qquad \mathsf{Fun}(S,\mathsf{Fun}(T,U)),$$

and we look here how we might describe these.

An element F of $\mathsf{Fun}(\mathsf{Fun}(S,T),U)$ takes as its input a function, say f, from S to T and produces an output in U, so we would write Ff for the output of F for the input f. But often what F does is to use its argument f, applied to some input.

For example, if $T = U$ and $s \in S$ is fixed, we might define a function F by the assignment

$$Ff = fs.$$

Or, more generally, if $g : T \longrightarrow U$, and s is fixed, we might use the assignment

$$Ff = g(fs).$$

Indeed, our interpretation of application of λ terms is via application of functions, so these ideas appear whenever we look at interpreting particular terms.

We may now combine our notation for functions with a finite source set from above and see what that looks like when describing higher order functions.

Example 4.2. Assume for the set $B = \{0,1\}$ we want to describe a particular function F in

$$\mathsf{Fun}(\mathsf{Fun}(B,B),B)$$

which takes the input f (known to be an element of $\mathsf{Fun}(B,B)$ and so a function $B \longrightarrow B$), and puts out whatever f puts out for the input 0, in other words

$$Ff = f0.$$

We may describe the function F, which we might call the 'evaluate at 0 function', as

$$\{\{0\mapsto 0, 1\mapsto 0\}\mapsto 0, \{0\mapsto 1, 1\mapsto 0\}\mapsto 1, \{0\mapsto 0, 1\mapsto 1\}\mapsto 0, \{0\mapsto 1, 1\mapsto 1\}\mapsto 1\}.$$

An element f of $\mathsf{Fun}(S,\mathsf{Fun}(T,U))$, on the other hand, takes as its input an element from the set S and outputs a function $T \longrightarrow U$. For some $s \in S$, In order to understand what fs does, we have to supply it with an argument from T in order to get an output in U. In this situation one might almost want to think of f as a function that takes *two arguments* one from S and one from T, and then produces an element of U.[2]

Example 4.3. Assume for the set $B = \{0,1\}$ we want to describe a particular element F of

$$\mathsf{Fun}(B,\mathsf{Fun}(B,B))$$

that maps the element 1 to the identity on B and the element 0 to the function that maps 0 to 1 and 1 to 0. We may use our notation to describe this function

[2] Indeed there is a mathematical argument that establishes that $\mathsf{Fun}(S,\mathsf{Fun}(T,U))$ has a bijection to $\mathsf{Fun}(S\times T,U)$ justifying this notion, but we don't have the time to pursue this idea.

as

$$\{0 \mapsto \{0 \mapsto 1, 1 \mapsto 0\}, 1 \mapsto \{0 \mapsto 0, 1 \mapsto 1\}\}.$$

Example 4.4. In PCF we have a term constructor ifz, which takes three terms of the base type nat. Based on β-reduction in PCF (the formal definition is given in Definition 38, but there is more of a discussion in Section 3.1.1), it is clear that the behaviour of this term constructor is to

- evaluate its first argument by repeated β-reduction and
- if that argument evaluates to $\overline{0}$, return the second argument and
- if that argument evaluates to $\overline{\mathsf{s}}\overline{0}$, return the third argument.

Lets pretend for the moment that we are using the set $B = \{0, 1\}$ to interpret PCF. How would we expect to interpret ifz? One possibility is to treat it as an element of

$$\mathsf{Fun}(B, \mathsf{Fun}(B, \mathsf{Fun}(B, B))).$$

That allow us to supply the corresponding function with the interpretations of the first, second and third terms (which have to be of base type), and supplied with three suitable arguments, this function returns an element of the interpretation of the base type.

For our (restricted) interpretation (combining our notation for functions with finite source sets with more general notation using assignments) we would expect this function to be described by

$$\{0 \mapsto (y \mapsto (z \mapsto y)), 1 \mapsto (y \mapsto (z \mapsto z))\}.$$

This function takes three arguments, one of after the other, and if the first argument is 0 then it returns its second argument, and otherwise its third. This matches our expectation of the behaviour of the ifz term constructor but, of course, for our actual interpretation of PCF, the interpretation of the base type has more elements, and we also have to worry about the first term giving a non-terminating computation. The actual interpretation of ifz is given in Definition 44, and the reader is invited to compare the function from above with the one given there, but to fully understand the official definition, the remainder of the material from this chapter is required.

4.2 Binary Relations

Binary relations allow us to form connections between elements of the same set.

4.2.1 Equivalence relations

Definition 49: equivalence relation

An **equivalence relation on a set** S is a binary relation that is reflexive, symmetric and transitive.

We briefly look at the three required properties:

- A binary relation on a set S is **reflexive** if and only if every element of S is related to itself.
- A binary relation on a set S is **symmetric** if and only if for all $s, s' \in S$, the fact that s is related to s' implies that s' is related to s.
- A binary relation on a set S is **transitive** if and only if for all $s, s', s'' \in S$ if we have that s is related to s', and s' is related to s'' then s is related to s''.

A topical example of an equivalence relation is α-equivalence, see Proposition 1.8. When we have an equivalence relation $\sim$ on a set S we may form the **quotient of S with respect to $\sim$**, and use equivalence classes of elements as the elements of a new set. We may think of this as considering the elements of S 'up to equivalence', that is, we don't make a distinction between elements that are related by the given equivalence relation.

Many authors treat the λ calculus as a quotient for α-equivalence, but we prefer to work with terms rather than equivalence classes of terms. This is certainly closer to working with programs.

Proposition 4.1

Let R be a binary operation on a set S. The following statements hold

(a) The intersection of all equivalence relations containing R is an equivalence relation, $\widehat{R}$.

(b) If R' is an equivalence relation containing R then $\widehat{R} \subseteq R'$.

Proof. *This is a suitable exercise for students wanting to understand the notion of the smallest equivalence relation generated by a given binary relation.*
See page 310.

Hence we may think of $\widehat{R}$ as the **smallest equivalence relation containing R**, because every equivalence relation containing R must also contain $\widehat{R}$. There is a different way of obtaining $\widehat{R}$, namely as the reflexive symmetric transitive closure of R.

The reflexive closure of R is given by

$$R \cup \{(s, s) \mid s \in S\},$$

while the symmetric closure is

$$R \cup R^{\mathrm{op}},$$

where

$$R^{\mathrm{op}} = \{(s, s') \in S \times S \mid (s', s) \in R.$$

The transitive closure of a relation is the union

$$\bigcup_{n \in \mathbb{N} \setminus \{0\}} R^n,$$

where R^n is the n-fold relational composite of R, given by the recursive definition

- $R^0 = \{(s, s) \mid s \in S\} = \mathrm{id}_S$ and

- $R^{n+1} = R^n \,;\, R$,

where for binary relations R and R' we have

$$R \,;\, R' = \{(s, s'') \mid \exists s' \in S (s, s') \in R \text{ and } (s', s'') \in R''\}.$$

When we talk about the reflexive symmetric transitive closure of a relation R, we first apply the reflexive closure of R, then the symmetric closure of the result, and then the transitive closure of that result. In the notes for COMP11120 we show that the reflexive symmetric transitive closure of a relation R is an equivalence relation that contains R.

Proposition 4.2

The reflexive symmetric transitive closure of R is the smallest equivalence relation containing R.

Proof. *This exercise forms a bridge between the idea of the smallest equivalence relation containing a given binary relation and the notion of the rellexive symmetric transitive closure of R.*
See page 310.

4.2.2 Pre- and partial orders

Another important concept for binary relations is that of a preorder.

Definition 50: preorder

A **preorder** on a set S is a binary relation that is reflexive and transitive.

We may think of a preorder as allowing us to compare elements to each other; the first condition makes sure that every element is less than or equal to itself, and the second that if s is less than or equal to s' and s' is less than or equal to s'', then s is less than or equal to s''. Both these ensure that our ideas of comparing elements work for these relations.

What might be unexpected is that a preorder allows the following to occur: We may find *distinct* elements $s, s' \in S$ with

$$s \leq s' \qquad \text{and} \qquad s' \leq s.$$

In Section 3.3 we define preorders for terms of the language PCF based on their interactions with other terms, and in that setting it is certainly possible that two terms are related in both ways—this is certainly the case for α-equivalent terms.

If we don't want to allow situations like the above we may demand that our relation also by **anti-symmetric**, which means that $s \leq s'$ and $s' \leq s$ imply that $s = s'$.

Definition 51: partial order

A **partial order** on a set S is a preorder that is anti-symmetric.

We may combine all three of these ideas as follows: Assume we have a preorder on a set S. Then an equivalence relation on S is given by

$$s \sim s' \qquad \text{if and only if} \qquad s \leq s' \text{ and } s' \leq s.$$

We may now form the quotient of S with respect to that equivalence relation, and on the resulting set, whose elements are equivalence classes, such as $[s]$ with respect to that equivalence relation we may define a partial order as follows:

$$[s] \preceq [s'] \qquad \text{if and only if} \qquad s \leq s'.$$

The fact that this is a well-defined notion is a bit more subtle than the reader may think, and the interested members of our audience may want to think about why.

Partial orders are fundamental to our model of PCF, and we give details of the required mathematics in Section 4.4.

Definition 52: poset

A **poset** is given by a pair $(P, \leq)$ where P is a set and $\leq$ is a partial order for that set.

4.3 Posets

We first establish some results about partially ordered sets.

4.3.1 Ordering functions

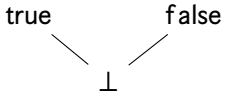
In the simply-typed λ-calculus, one can use ordinary sets and arbitrary functions between them as a model. Arguably, this is because it is not a very powerful language. We know that every program terminates with an essentially unique answer, so we don't have to wonder, for example, how we interpret a program that does not terminate. However, in the above sections we saw that non-termination in PCF is a subtle issue, requiring us to add extra structure to our model.

The basic idea we use for this purpose is quite simple: We assume that when we interpret the base type we have a separate element $\bot$ (pronounced 'bottom') which stands for non-termination. Although our base type in PCF corresponds to the natural numbers, let us first think about the idea for a simpler data type. One might think that the way we want to, for example, model the booleans would be to have three elements

$$\mathsf{true} \qquad \mathsf{false} \qquad \bot$$

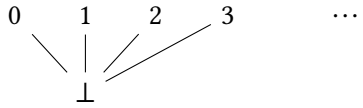
but it turns out that this is not quite the right thing to do. The reason for this is that $\bot$ cannot be thought of as an ordinary piece of data. If we start a program running, we can be sure that its value in the model should *not* be $\bot$ if we see it output a value. But we can never be sure that its value in the model *actually is* $\bot$, because we would have to watch it run forever to observe that. That means we think of the elements in the model of the booleans as being related to each other in the sense that two of them provide a complete answer, namely false and true, while one of them conveys no information, namely $\bot$.

We picture this idea as follows



We want to think of this set as being *partially ordered*, and what we have drawn above is the Hasse diagram of a poset. This might be a good time to remind the reader that a poset $(P, \leq)$ is a set P together with a relation $\leq$ which is reflexive, anti-symmetric and transitive.

Our potential base types are going to be quite simple: They will consist of a set, say S, together with a new element $\bot$, and the partial order consists of exactly all instances of $\bot \leq s$ where s is an element of S. We use $S_\bot$ for these kinds of (partially ordered) sets. We may picture the set $\mathbb{N}_\bot$ as follows:



We use the following convention: When we refer to an element x in this set and write $x \in \mathbb{N}$ we mean that x is one of the elements different from $\bot$. In the literature this partially ordered set is often known as *the flat natural numbers*.

Having interpreted potential base types we need to think about what to do for the remaining types. We expect to once again give a recursive definition of this interpretation, and we expect the interpretation of a function type to be a (partially ordered) set of functions. There are several questions to answer:

- Which functions should we pick?
- How should we put a partial order on these?
- How do we ensure that we can model the fixed point operator?

When we discussed ways of comparing PCF terms in Section 3.3, we noted that the situation at a function type is complicated. With a function, we can ask about its observable behaviour for each input. For some inputs, it might fail to terminate and we get no information, but on others it might return an answer, and different functions may, of course, output different values for the same input. The intuition we use is that one function ought to be considered more informative than another if it extends the other function in the sense that at every input, the function outputs something more informative than the other function, without observably contradicting the information which the other function already outputs. Intuitively, it is like considering someone who says "the next bus is at eleven, and the one after is at twelve", to be more informative than someone who says "the next bus is at eleven, and the one after is at er..." before trailing off. But we don't consider someone who says "the next bus is at ten thirty, and the one after is at twelve" to be more informative: they have supplied incomparable information to our first respondent.

Because we have an infinite base type, there can be infinite sequences of functions, each more informative than the last.

Example 4.5. We describe a family of functions from $\mathbb{N}_\bot$ to itself, one for each $n \in \mathbb{N}$.

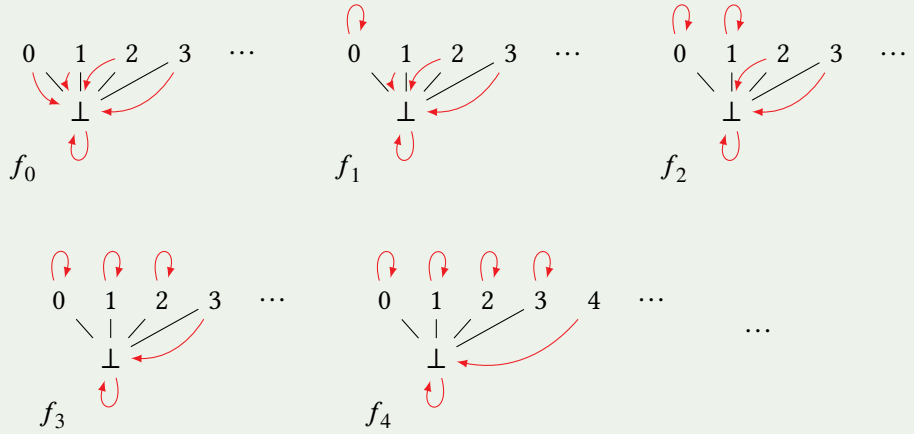
$$f_n x = \begin{cases} x & x \in \mathbb{N},\ x < n \\ \bot & \text{else} \end{cases}$$

So

- f_0 sends every element to $\bot$,

- f_1 sends 0 to 0 and all the other elements to $\bot$,
- f_2 sends 0 to 0 and 1 to 1 and all the other elements to $\bot$…

You might find it useful to picture the functions via the following diagrams:

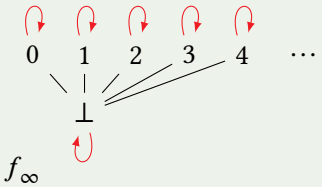


We observe the following:

- If two of these functions map an element to something other than $\bot$ they map it to the same element, that is, for $i, j \in \mathbb{N}$ we have $f_i x, f_j x \in \mathbb{N}$ implies $f_i x = f_j x$.
- If we think of a function as being 'more defined' than another if it sends more elements to something other than $\bot$ then our functions are increasingly more defined as the index increases.
- We can think of these functions as coming closer and closer to the following function:

$$f_\infty x = \begin{cases} x & x \in \mathbb{N} \\ \bot & \text{else.} \end{cases}$$

We may picture this function as being the one where all natural numbers are mapped to themselves.



There is something intellectually pleasing about the example above, because each of the functions f_n seems 'finite' in the sense that it returns an informative output for only finitely many inputs. Yet, taken together, they seem to approximate f_∞ in some sense. This looks like it might shed some light on one of the mysteries of programming—how can a finite program, which we run each time with finite resources, implement an infinite mathematical function? Looking at the example, we might realize that if we have a program implementing f_∞, when we run this program on a particular input, say m, if the program were only to calculate f_m and use this to answer, we would have no way to distinguish this from the 'real

thing' f_∞. So a program does not have to represent the infinite object f_∞ at any time, as long as it has a scheme for computing f_n for a big enough value of n for a given input. This intuition is related to the 'loop unrolling' behaviour we get when we reduce a recursive term implementing f_∞, such as

$$\mathsf{rec}\, \lambda f : \mathsf{nat} \to \mathsf{nat}.\ \lambda x : n.\ \mathsf{ifz}\, x[\bar{0}][\bar{\mathsf{s}}\, f\, \mathsf{pred}\, x]$$

which for any particular finite input will only have to expand the recursive definition finitely many times before reaching the base case.

We need quite a bit of mathematical machinery to make these ideas precise, and that is the aim of the remainder of this section.

Exercise 42. *This exercise encourages students to think about how to switch between partial and total functions.*

Another way of thinking of the element $\bot$ as introduced above is to think of it as denoting 'undefined'. Indeed, we may make a connection between partial functions, and adding this kind of element to the given set.

Let $\mathsf{PFun}(S, T)$ be the set of partial functions from S to T. Show that there is a bijection from $\mathsf{PFun}(S, T)$ to $\mathsf{Fun}(S, T_\bot)$.

A solution appears on page 311.

4.3.2 Basic facts about partial orders

Since our model is based on partial orders, we need to establish a few results about these. In particular it turns out that the order theoretic idea of a least upper bound of a set of elements plays a crucial role in making the notion of approximation from the previous section precise. Before we look into how exactly this works we need some properties of least upper bounds.

Proposition 4.3

If S is a subset of a poset P and S has a greatest element s then s is the least upper bound of S.

Proof. *This is a suitable exercise for students wanting to refamiliarize themselves with partial orders, but it is recommended that students carry out exercises in the following subsections first.*
See page 311.

Proposition 4.4

Assume that P is a poset and that S and S' are subsets of P such that their least upper bounds exist. Then the following statements hold.

(a) If $S \subseteq S'$ then the least upper bound of S is less than or equal to that of S'.

(b) If for all $s \in S$ we have that there is $s' \in S'$ with $s \leq s'$ then the least upper bound of S is less than or equal to that of S'.

Proof. See page 259.

Sometimes it is useful to have notation for the least upper bound for the subset S of some poset P. We use

$$\bigvee S$$

for this purpose.

Proposition 4.5

Assume that P is a poset and that $\mathcal{S}$ and $\mathcal{T}$ are families of subsets of P such that the following sets have least upper bounds:

- all S in $\mathcal{S}$ and
- the set of all the least upper bounds from the previous item and
- all T in $\mathcal{T}$ and
- the set of all the least upper bounds from the previous part.

(a) We have $\bigvee \bigcup_{S \in \mathcal{S}} S = \bigvee_{S \in \mathcal{S}} \bigvee S$.

(b) If $\bigcup_{S \in \mathcal{S}} S = \bigcup_{T \in \mathcal{T}} T$ then $\bigvee_{S \in \mathcal{S}} \bigvee S = \bigvee_{T \in \mathcal{T}} \bigvee T$.

Proof. See page 260.

4.3.3 Posets of functions

Above we argue that we would like to use posets to interpret the base type. In order to interpret function types we have to think about how we may put a partial order on sets of functions in the first instance, and which functions we should consider.

In this model we don't want to allow all functions, but ones that are well-behaved with respect to the partial order. There are two conditions to impose, and the first of these is that we only consider order-preserving functions. This condition models the crucial idea that we can't know for sure that we aren't going to get more information from a running computation. We are allowed to wait for information to come in, but if that information never comes then we will never be able to supply the information we would have worked out using the information we were waiting for. Intuitively, we expect terms of PCF to have the same limitations as us if they only have access to other PCF terms by applying them to inputs. This tells us that a more informative input can lead to a more informative output, but a *less* informative input can't lead to a *more* informative output! We can't both give a definitive answer on the basis of no information *and* reserve the right to change our answer if more information arrives later. Another way to say this is that if one input extends the information given by another, then a computation can extend the output, but it can't contradict the information already given.

Definition 53: order-preserving

Let $(P, \leq)$ and $(Q, \leq)$ be posets. We say that a function

$$f : P \longrightarrow Q$$

is order-preserving if and only if for all $p, p' \in P$ it is the case that

$$f\,p \leq f\,p'.$$

If our input and output sets are flat sets such as $\mathbb{N}_\perp$ the only non-trivial instances of the partial order are the ones that tell us that $\perp \leq n$ for $n \in \mathbb{N}$, and so for a function to be order-preserving all that is required is that

$$f\perp \leq f\,n \qquad \text{for all } n \in \mathbb{N}.$$

Example 4.6. In Example 4.5 all the functions f_n for $n \in \mathbb{N}$, as well as f_∞ are order-preserving. This is easy checked since all these functions map $\perp$ to $\perp$, and so the condition given above is satisfied.

We can think of this requirement as demanding that if we think of $p \leq p'$ as telling us that p' contains at least as much information as p then from that we know that $f\,p'$ contains at least as much information as $f\,p$.

Example 4.7. Simple examples of order-preserving functions are given by the fact that every constant function is order-preserving, that is if P and Q are posets and $q \in Q$ then

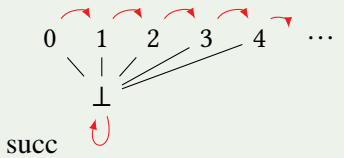
$$\begin{aligned} k_q : P &\longrightarrow Q \\ x &\longmapsto q \end{aligned}$$

is order-preserving. Further examples appear in Proposition 4.6.

Example 4.8. A more interesting example of an order-preserving function is given by

$$\begin{aligned} \mathrm{succ} : \mathbb{N}_\perp &\longrightarrow \mathbb{N}_\perp \\ x &\longmapsto \begin{cases} \perp & x = \perp \\ x+1 & \text{else,} \end{cases} \end{aligned}$$

which we may picture as follows:



We use this function to help us model PCF terms that contain the $\bar{\mathsf{s}}$ term constructor.

Exercise 43. *This exercise encourages students to think about how we might interpret a related PCF term constructor.*

Define a function from $\mathbb{N}_\perp$ to $\mathbb{N}_\perp$ which we might use to interpret the pred term constructor. Draw a picture of the function similar to the one for succ in the previous example.

A solution appears on page 311.

Exercise 44. *This exercise is suitable for students who want to get more practice in reasoning about partial orders for concrete examples and a simple general case.*

Show that the functions given in the three preceding examples and Exercise 43 are indeed order-preserving.

A solution appears on page 311.

Proposition 4.6

The Identity on a poset is order preserving, and the composite of two order-preserving functions is also order-preserving.

Proof. *This exercise is suitable for students who want to get more practice in reasoning about partial orders.*

See page 312.

For posets $(P, \leq)$ and $(Q, \leq)$ we write

$$P \Rightarrow_{\leq} Q$$

for the set of all order-preserving functions from P to Q. We may put a partial order on this set as follows.

Definition 54: pointwise order

Let $(P, \leq)$ and $(Q, \leq)$ be posets. The **pointwise order** on $P \Rightarrow_{\leq} Q$ is defined by setting

$$f \leq g \qquad \text{if and only if} \qquad \text{for all } p \in P,\ fp \leq gp.$$

Exercise 45. *This is a good exercise for students wishing to refamiliarize themselves with the notion of a partial order.*

Show that the relation defined in Definition 54 is indeed a partial order.

A solution may be found on page 312.

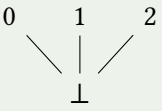
Example 4.9. Returning to Example 4.5 we may see that for $m \leq n$ in $\mathbb{N}$ we have that

$$f_m \leq f_n$$

in the pointwise order. We also observe that for all $n \in \mathbb{N}$ we have

$$f_n \leq f_\infty.$$

Example 4.10. We may now picture a function space as a poset. We consider the poset $\mathbb{3}_\perp = \{0, 1, 2\}_\perp$, which gives us an initial segment of $\mathbb{N}_\perp$. We can picture it as



When we look at order-preserving functions from $\mathbb{3}_\perp$ to itself we know we get all the constant functions, and the identity function. Are there any others?

Above we argued that such a function f is order preserving provided that for all elements $x \in \mathbb{3}_\perp$ we have that $f \perp \leq f x$. This tells us that if we map $\perp$ to an element other than $\perp$ the function has to be constant, and we can also see that there cannot be any elements above the constant functions, so these form some of the maximal elements of our poset.

But if we map $\perp$ to $\perp$ we can map the other three elements to anything we like: For example,

$$\begin{aligned} 0 &\mapsto 1 \\ 1 &\mapsto 2 \\ 2 &\mapsto 2 \end{aligned}$$

together with $\perp \mapsto \perp$ gives an order-preserving function from $\mathbb{3}_\perp$ to $\mathbb{3}_\perp$, and this is a maximal function in the poset.

We can see that there are $3^3 = 27$ elements of this kind. Three of those, which map the three non-bottom inputs to the same output, have one of the constant functions above them, and the others are also maximal elements of our poset.

We need to come up with some space-saving notation for these functions. We are going to encode such a function as a triple, where

$$0 - 1$$

is the function that maps 0 to 0, 1 to $\perp$ and 2 to 1. (We find it easier read these triples when using hyphen rather than $\perp$.)

The resulting poset is a bit too large to draw. There is a least element, $k_\perp$, and immediate above that element we have nine function that map exactly one of $0, 1, 2$ to an element in $\{0, 1, 2\}$, and all the other elements to $\perp$.

Using the notation from above we can see that the least upper bound of the functions given in this way by

$$0 - - \qquad \text{and} \qquad - 1 -$$

is given by

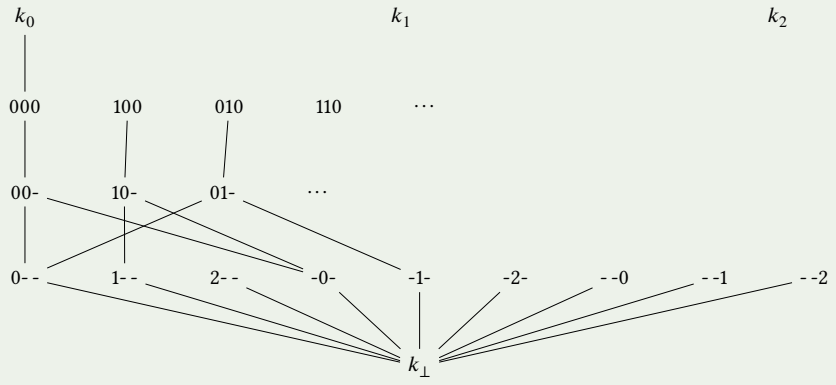
$$01 - .$$

There are 3 ways in which we can pick two out of three elements, and there are 6 ways of assigning those two elements to an element in $\{0, 1, 2\}$, so there are 18 function in that 'layer' of the poset.

The layer above this consists of functions where all three elements are mapped to non-bottom outputs, but $\bot$ is mapped to itself.

The final layer then consists of the maximal elements discussed already.

We don't draw the whole poset because it's a bit too large, but we hope that this picture shows enough of it.



RExercise 46. *This exercise is about getting a hands-on intuition for what the order on functions is like.*

Consider the poset of order preserving functions from $\mathbb{3}_\bot$ to $\mathbb{3}_\bot$ from Example 4.10. Draw a Hasse diagram for the elements which are at least as informative as the one called $-2-$ in the notation of that example.

A solution appears on page 312.

We note that our ideas for interpreting the base type assumes that we have a least element, $\bot$. This represents a computation which contains no information – concretely, it runs forever before giving any useful results. We need something like this at all types, because we know that we can apply rec to the PCF term which represents the identity function at any type, and the resulting term will reduce forever. If we build this into the base type we obtain this infrastructure at all types. We adopt the convention that we use $\bot$ for the least element of any poset that has such an element.

Proposition 4.7

If P and Q are posets with a least element then $P \Rightarrow_{\leq} Q$ also has a least element.

Proof. *This is a good exercise for students who would like to better understand the pointwise order.*

See page 313.

4.4 Directed-complete Partial Orders

Given that our ideas for interpreting the base type suggest posets that only have a finite height one might be forgiven for assuming that those are the only partially

ordered sets we have to worry about. However, the situation becomes more complicated:

We intend to interpret the type nat using the flat natural numbers $\mathbb{N}_\perp$, and again we intend to use a function space construction to interpret higher types. This forces us to accept that the posets we obtain in this manner contain infinite chains, such as

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_\infty$$

from Example 4.5, and we want to be able to consider the f_I as approximating f_∞, which is the identity on $\mathbb{N}_\perp$. All these functions appear in our interpretation of $\mathsf{nat} \to \mathsf{nat}$, and this means that when we consider the interpretation of even higher types, such as $N > N \to \mathsf{nat} \to \mathsf{nat}$, we have to make a decision regarding what elements of the corresponding function space in our model should do with such infinite approximations.

Above we talk about the idea that the element $\perp$ in the base type says there is no information regarding the result of a computation, and we can extend this idea to functions whose target set is the interpretation of the base type as indicated by Example 4.5 (see Definition 54 for a precise definition).

Above we argue that we want to think of the partial order on the base type as being about information, and we use the notion of a least upper bound with respect to such an order to formalize our notion of approximation. We should distinguish two very different ways in which one might think of using upper bounds to model the notion of approximation. The one which might come to mind first, which we do *not* use here, is to think of the upper bound as an approximation of the elements in a set. This is the way the expression "upper bound" is often used in everyday life. Here we mean the opposite: we think of the elements of a set as approximations to the upper bound.

This is not completely unfamiliar mathematically. An ancient method to approximate π is to inscribe a regular polygon inside a circle, and use the ratio of the diameter and circumference of the polygon as an approximation to π from below. Then π itself is an upper bound of the set of approximations one gets this way. But 4 and lots of other numbers are also upper bounds. We could define π as the *least* upper bound of the approximations we get from inscribed polygons. It is in this sense that we consider the least upper bound of a set to be approximated by the elements of the set. But we need to apply this idea to the functions we use to model computations, to make the ideas of Example 4.5 precise.

For that to work we have to ensure that least upper bounds of certain sets exist, and these sets need to be defined. The intuition is that a finite set of pieces of information has an upper bound exactly when they do not contradict each other: there is a way of extending all of them to get one combined piece of information. To generalize this consistency condition to infinite sets we think in terms of making observations: if every pair of elements are consistent with each other, then there is no finitely-observable reason why the pieces of information are not all consistent.

4.4.1 Directed sets and dcpos

Definition 55: directed set

A subset non-empty S of a poset $(P, \leq)$ is **directed** if and only if for all $s, s' \in S$ there exists $s'' \in S$ with $s, s' \leq s''$.

This means that if we pick finitely many elements $s_1, s_2, \ldots s_n$ in S then there is an element in S that is greater than or equal to all of these.

Now in order to have a mathematical theory which allows us to talk about the infinite objects approximated by finite computations we have to restrict attention to special posets, namely those where a set of consistent pieces of information can always be combined to form a least upper bound, modelling the infinite object which they describe.

Definition 56: dcpo

A poset $(P, \leq)$ is a **directed complete partial order** (dcpo) provided that every directed subset of D has a least upper bound.

If S is a directed subset of D we write

$$\bigvee^{\uparrow} S$$

for its least upper bound. When we are forming a more complicated expressions we may sometimes find it useful to write

$$\bigvee_{s \in S}^{\uparrow} s.$$

We may readily establish that the partial orders suggested for the base type are dcpos: The only directed sets in these 'flat' partially ordered sets have at most two elements, for example $\{\bot, n\}$ for $n \in \mathbb{N}$ in $\mathbb{N}_\bot$, and a two-element directed set has a greatest element and so also a least upper bound.

Proposition 4.8

Assume that P is a poset with the property that it has a countable number of elements, and so that we may find a number $n \in \mathbb{N}$ such that for all $p, p' \in P$ we have that if we can find

$$p = p_0 < p_1 < \cdots < p_k = p'$$

then

$$k \leq n.$$

Then P is a dcpo.

Proof. See page 260.

Hence in particular we know that the set $\mathbb{N}_\bot$ is a dcpo.

4.4.2 Suitable functions, and function spaces

The situation becomes more interesting when we move to posets which interpret function types. To ensure that we re able to interpret recursion we need to demand an additional property for such functions: they have to be well-behaved with respect to least upper bounds of directed sets.

Proposition 4.9

If P and Q are posets and $f : P \longrightarrow Q$ is an order-preserving function then the image of a directed subset of P is a directed subset of Q.

Proof. See page 260.

Definition 57: Scott-continuous

We say that an order-preserving function $f : D \longrightarrow E$ between dcpos is **Scott-continuous** if and only if for every directed subsets S of D it is the case that

$$f \bigvee^{\uparrow} S = \bigvee^{\uparrow} \{ f s \mid s \in S \}.$$

Note that we may also write this equality as

$$f \bigvee^{\uparrow} S = \bigvee_{s \in S}^{\uparrow} f s.$$

We note that Proposition 4.9 tells us that the set given in the definition is directed, since it is the image of a directed set under an order-preserving function, and so our definition makes sense.

Alternatively people say that such an f *preserves directed suprema*, where 'suprema' means least upper bounds and 'directed suprema' is a shortcut for 'suprema of directed sets'.

The computational intuition for this is related to the reason why finite programs an encode infinite objects. We mention above that we have no way as observers to distinguish between a program which somehow computes an infinite function and one which just computes as much of a finite approximation as needed for the input it is working on. If terms of PCF really behave like functions, then other terms are in the same situation as observers: they can't look inside the given term to see if it really builds an infinite object, they can only evaluate it on inputs, and then the given term only needs to compute a finite approximation in order to answer.

Because of this, the behaviour of a PCF term on an infinite function is determined by what it does to all the finite approximations of that function: it has no way of finding out the global behaviour by applying it to infinitely many arguments, because then it would have to run forever! This means that we can approximate the output of a PCF computable function by feeding it approximations of the input. This rules out behaviour like running forever for (inputs encoding) all the functions f_n from Example 4.5, but suddenly returning $\bar{0}$ when supplied with f_∞. The function corresponding to that behaviour is order-preserving, but fails to preserve directed suprema. A program with that behaviour can't be written because it would have to look at its argument on all inputs before outputting the answer of 1, but trying to do that will, quite literally, take forever, so the real observable output is $\bot$, the supremum of the outputs on the functions f_n.

For our flat partially ordered sets there are no non-trivial directed sets and so there are no 'interesting' suprema, and so every order-preserving function is Scott-continuous. We need to move to function spaces to obtain more interesting examples, but for those we have to establish first that we do obtain dcpos.

Before we look at function spaces we consider examples of Scott-continuous functions.

Example 4.11. Simple examples of Scott-continuous functions are the same that we used to provide examples of order-preserving functions;

- The identity function $P \longrightarrow P$ is Scott-continuous for every poset P.
- Every constant function is Scott-continuous, that is if P and Q are posets and $q \in Q$ then
$$\begin{aligned} k_q : P &\longrightarrow Q \\ x &\longmapsto q \end{aligned}$$
is Scott-continuous.
- The composite of two Scott-continuous functions is Scott-continuous.

Exercise 47. *This exercise is suitable for students who want to get more practice in reasoning about partial orders.*

Show that the functions given in the preceding example are indeed Scott-continuous.

A solution may be found on page 313.

Proposition 4.10

Let $f : P \longrightarrow Q$ be an order-preserving function between posets, and further assume that P satisfies the condition from Proposition 4.8. Then f is Scott-continuous.

Proof. See page 260.

The second condition we impose on functions we use in our interpretation is that they have to be Scott-continuous. If D and E are dcpos we write

$$D \Rightarrow E$$

for the set of all Scott-continuous functions from D to E, and we put the pointwise order on this set to view it as a poset.

Proposition 4.11

We have the following for dcpos D and E:

- If $f : D \longrightarrow E$ an order-preserving function. Show that if S is a directed subset of D then
$$\bigvee^{\uparrow} \{ f\, s \mid s \in S \} \leq f \bigvee^{\uparrow} S.$$
- In this situation $D \Rightarrow E$ is a dcpo.

If D and E are dcpos then so is $D \Rightarrow E$.

Proof. *Hint: To prove the second part you may find it helpful to know that if $\mathcal{F}$ is a directed subset of $D \Rightarrow E$ then its greatest lower bound is given by*

$$\begin{aligned} \bigvee^{\uparrow} \mathcal{F} : D &\longrightarrow E \\ d &\longmapsto \bigvee^{\uparrow}_{f \in \mathcal{F}} f\, d. \end{aligned}$$

See page 261.

Example 4.12. Looking back at Example 4.5 we may see that the f_n form a directed set (since we know that $m \leq n$ implies $f_m \leq f_n$) and that we have

$$\bigvee_{n \in \mathbb{N}}^{\uparrow} f_n = f_\infty.$$

Proposition 4.12

Assume that D, E and F are dcpos and that

$$f : D \longrightarrow E \Rightarrow F \qquad \text{and} \qquad g : D \longrightarrow E$$

are Scott-continuous functions. Then the assignment

$$d \longmapsto f\, d\, (g\, d)$$

is a Scott-continuous function from D to F.

In particular for $e \in E$ the assignment

$$d \longmapsto f\, d\, e$$

is Scott-continuous.

Proof. See page 262.

4.4.3 Fixed points

In order to interpret the rec operator from PCF we have to ensure that there is suitable infrastructure in our model. In Section 3.2.4 we argue that the meaning of rec must be a 'fixed point' of its argument. In an ordinary set, there is no guarantee that every function has a fixed point (think about the 'not' operation on booleans!). But now we have added the infrastructure of a dcpo and restricted attention to order preserving Scott-continuous functions, it turns out that we can solve this problem. It is the fact that this can be made to work using dcpos and Scott-continuous functions that first attracted researchers to this model.

Definition 58: fixed point

If S is a set and $f : S \longrightarrow S$ is a function we say that $s \in S$ is a **fixed point of** f provided that $f\, s = s$.

RExercise 48. *The purpose of this exercise is to think about the notion of fixed points in a more mathematically familiar setting, before we move on to dcpos.*

Consider the set $\mathbb{R}^+$ of non-negative real numbers, and the function

$$f : \mathbb{R}^+ \to \mathbb{R}^+$$

given by $f(x) = \lfloor x/2 \rfloor + 1$. What is the set of fixed points of f?

A solution appears on page 313.

Now we turn to showing the key result, that we always have fixed points of Scott-continuous functions on dcpos.

Proposition 4.13

If D is a dcpo with a least element and $f : D \longrightarrow D$ is Scott-continuous then it has a least fixed point.

We take a little time here to explain the proof of this result since it brings together much of the infrastructure we have built up to this point. The dcpo D has a least element $\bot$ to which we may apply the function f, and then we know that the result must be greater than or equal to the least element, so we get

$$\bot \leq f\bot.$$

Since f is order-preserving we know that we may apply f on both sides to obtain

$$f\bot \leq f^2\bot,$$

and if we apply f again to this instance of the partial order we obtain

$$f^2\bot \leq f^3\bot.$$

We see that what we are building here can be summarized n just one diagram

$$\bot \leq f\bot \leq f^2\bot \leq f^3\bot \leq \cdots.$$

The key idea now is to realize that this is a directed set: for m and n in $\mathbb{N}$ we have that

$$f^m\bot,\ f^n\bot \leq f^{\max\{m,n\}}\bot,$$

and so we know that

$$\bigvee_{n\in\mathbb{N}} f^n\bot$$

exists. Moreover since f is Scott-continuous we know that

$$f\left(\bigvee_{n\in\mathbb{N}} f^n\bot\right) = \bigvee_{n\in\mathbb{N}} f^{n+1}\bot.$$

It turns out that the expression on the right is equal to

$$\bigvee_{n\in\mathbb{N}} f^n\bot,$$

and so this is a fixed point of f. For this final step we may use Proposition 4.4 to obtain

$$\bigvee_{n\in\mathbb{N}} f^n\bot = \bigvee_{n\in\mathbb{N}} f^{n+1}\bot$$

since for every element of f^{k+1} of

$$\{ f^{n+1}\bot \mid n \in \mathbb{N}\}$$

there is an element f^{k+1} of

$$\{ f^{n}\bot \mid n \in \mathbb{N}\}$$

greater than or equal to it.

The reader might like to contemplate the similarity between this proof and the procedure of 'loop unrolling' or expanding the definition of a recursive function. At first we have no idea about what the recursive function does, but when we expend the definition of rec by applying the body we hope to get a term which does not need the recursive call in order to output an answer for some of the cases.

This is more information than we started with, but it might not cover the case we are interested in. When expanding the definition again, we are now expanding the body of the term rec is applied to for a more informative input, so there may be more inputs for which we can calculate the answer without needing to do another recursive expansion.

If repeating this process a finite number of times allows us to calculate an answer for the input of interest, then that is the output of the term defined using rec. If not we reduce forever and then no observable information is output. This also helps to explain why we take the *least* fixed point': For instance, every element is a fixed point of the identity function, but if we apply rec to a term representing the identity function, the resulting term reduces forever. The reason is that applying the identity function never adds any specific information about what the output should be.

Exercise 49. *This statement has a proof that is not very long but is quite tricky for those new to reasoning about partial orders.*

Show that $\bigvee^{\uparrow}_{n\in\mathbb{N}} f^n\bot$ is the least fixed point of f. *Hint: The proof reprises some of the ideas from above.*

A solution appears on page 314.

In the process of interpreting PCF terms we apply the fixed point operator to interpretations of subterms, and we should think about its type, which is

$$D \Rightarrow D \longrightarrow D.$$

We use fix to name this operator.[3] The property of being a fixed point then becomes, for

$$f : D \longrightarrow D,$$

the equality

$$f(\mathsf{fix}\, f) = \mathsf{fix}\, f. \qquad \text{fixed point property}$$

We prefer this way of naming the fixed point to the explicit calculation given above. We note that the given formulation provides a very useful reasoning principle. We also note that this being the least fixed point means that, for $d \in D$, we have

$$f\, d = d \qquad \text{implies} \qquad \mathsf{fix}\, f \leq d.$$

[3] It is important to realize that a function from some dcpo D to itself may have more than one fixed point, and we expect fix to give us the least one of these.

Definition 59: fixed point operator

Given a dcpo D a **fixed point operator** is given by

$$\Phi : D \Rightarrow D \longrightarrow D$$

such that for all $f : D \longrightarrow D$ we have

$$f(\Phi f) = \Phi f.$$

We can see that fix for a particular dcpo is a fixed point operator, and that it has a special property that means we can refer to it as the *least fixed point operator*.

We may now ask the question whether fixed point itself is an operator in our model, that is, whether it is Scott-continuous.

Proposition 4.14

If D is a dcpo then

$$\mathsf{fix} : D \Rightarrow D \longrightarrow D$$

is Scott continuous.

Proof. See page 262.

Exercise 50. *The following statement has a fairly short proof, but assembling the available information correctly is non-trivial, so students may find the working on it frustrating.*

Let D be a dcpo. Define

$$\Phi : (D \Rightarrow D) \Rightarrow D \longrightarrow (D \Rightarrow D) \Rightarrow D$$

as

$$\Phi(F)(f) = f(Ff).$$

Show that F is a fixed point of Φ if and only if F is a fixed point operator for D.

We require a number of results to establish that our denotations for PCF terms really do live in our model—that is, we have to show that a number of functions are Scott-continuous. Because we can write a term of the form $\lambda f : \tau.\, \mathsf{rec}\, f$, we have to show that the fix operator itself is a Scott-continuous transformation.

Proposition 4.15

If D is a dcpo and

$$F : D \Rightarrow D \longrightarrow D \Rightarrow D$$

is Scott-continuous then so is

$$\mathsf{fix}\, F : D \longrightarrow D\ .$$

Proof. *This proof requires arguing with functions and many students may find this quite a challenge.*

See page 315.

Glossary

$\alpha\beta$-equivalence, $\sim_{\alpha\beta}$ 85

The smallest equivalence relation containing both α-equivalence and β-reduction.

$\alpha\beta\eta$-equivalence, $\sim_{\alpha\beta\eta}$ 92, 94

The smallest equivalence relation containing α-equivalence, β-reduction and η-conversion.

α-equivalence, $\sim_{\alpha}$ 20, 60

Two λ-terms are α-equivalent provided they only differ in the choice of name for some of their bound variables; in a typed setting abstracted variables must have the same type.

application 12

A λ-term that has been formed using the scApp rule.

applicative preorder 130

A preorder on PCF terms that says a term is greater than or equal to another if it terminates with an output for more inputs.

β-reduction, $\xrightarrow{\beta}$ 28, 122

β-reduction is the mechanism that provides a dynamics for λ-terms, and that allows us to program with these.

bound variables 14

The bound variables in a λ-term are those that appear under a λ-abstraction for that variable.

closed term 88

A term that contains no free variables.

compatible 78

A typing environment Δ is compatible with a given one Γ provided that $\Gamma\Delta$ allows us to derive all the typing assignments for variables that Γ does.

context 87

A λ-term with a hole in it.

context substitution 87

We may substitute a term into the hole in a context.

contextual equivalence 90, 134

Two terms which are typeable with the same type are contextually equivalent at a type environment Γ if and only if when substituted into a suitable context they give $\alpha\beta\eta$-equivalent results.

contextual preorder 133

A preorder on PCF terms that says a term is less than or equal to another if when we apply a term that results in a term of type nat we obtain terms that are related by the observational preorder.

d-logical relation 153

For a given arity a relation between elements of the denotation of the same type that is defined across all types in such a way that application preserves the relation; a generalization of a logical predicate.

denotation of a term 101, 137

We may define the denotation of a typeable term relative to a type environment and a valuation that provides interpretation for the variables that occur in said type environment.

denotation of a type 97, 136

Given a set to interpret the base type we may recursively define a set of functions to interpret all the types in the hierarchy.

diamond property 47

A binary relation has the diamond property if and only if one term is related to two terms, then there is a fourth term such that those two terms are related to that.

directed 175

A subset of a partially ordered set is directed if every two elements of that set have an upper bound in the set.

directed complete partial order (dcpo) 175

A dcpo is a partially ordered set in which every directed subset has a least upper bound.

dt-logical relation 144

A binary relation between elements of the denotation of types and terms of that type that is defined across all types in such a way that application preserves the relation; a generalization of a logical predicate.

equivalence relation 164

An equivalence relation on a given set is a binary relation for that set which is reflexive, symmetric and transitive.

first order type 89

A type that takes inputs of base type only and produces an output of base type.

first order type environment 90

A type environment that exclusively assigns first order types to variables.

fixed point 179

For a function whose source and target sets are the same a fixed point is an element of that set which is mapped to itself.

fixed point operator 181

An operator that takes as input a function from a dcpo to itself and returns a fixed point of that function.

free variables 16

The free variables in a λ-term are those that have occurrences that are not bound by a λ-abstraction for that variable.

η-conversion, $\xrightarrow{\eta}$ 91, 93

This is a relation that allows us to simplify a term to one that has the same behaviour when substituted into a larger term.

higher order type 89

A type that is not first order.

invariant 155

A typeable term t is invariant for a given logical relation if the tupel that consists of the denotation of t is an element of the logical relation at the appropriate type.

irreducible 125

A term is irreducible if there are no β-reductions possible from that term.

λ-abstraction 12

A λ-term that has been formed using the scAbs rule.

λ-term 12

A term defined recursively from variables using lambda abstraction and application. The set of all λ terms is $\Lambda\mathsf{Trm}$.

logical predicate 79

A logical predicate is, at each type, a selection of pairs consisting of a typing environment and a term such that there is a derivation that given that typing environment the given term has that type; and this is done in such a way that the selection at a function type is determined by the selection at the immediate subtypes.

order-preserving 170

A function between partially ordered sets is order-preserving provided that if two inputs are related by the partial order then so are the corresponding outputs.

parallel reduct, $\Diamond$ 49

An operation on λ-terms that is helpful in proving confluence.

partial order 165

A partial order on a given set is a preorder which is anti-symmetric..

PCF preterm 121

A term in a language where apart from abstraction and application, we have some infrastructure for natural numbers, a conditional, and recursion.

pointwise order 172

The partial order on a set of functions between partially ordered sets where one function is less than or equal to another if for all possible inputs the first function gives a value less than or equal to that given by the second function.

poset 165

A set equipped with a partial order.

preorder 164

A preorder on a given set is a binary relation for that set which is reflexive and transitive.

preterm 59

A preterm is a λ-term where for every abstraction the variable which is abstracted is annotated with a type.

redex 29

A redex is λ-term consisting of an application of a λ-abstraction to another λ-term. This is a situation where we may apply β-reduction in the base case.

renaming, ren 17

We have a renaming operation that allows us to change the name of a particular variable in a λ-term.

Scott-continuous 176

An order-preserving function is Scott-continuous provided that it maps the greatest lower bound of directed sets to the greatest lower bound of their images.

strongly normalizing 81

A term is strongly normalizing if any infinite reduction sequence starting at that term, after a finite number of steps, consists of reduction to the same term.

substitution, $-[-/-]$ 24

We have a particular notion of *capture-avoiding* substitution that substitutes a term for a variable while preserving α-equivalence of terms.

subterm 12

We can think of the subterms of a λ-terms as the valid λ-terms we need to assemble when we are using the recursive instructions to construct the given term.

$\mathsf{Type}^{\rightarrow}$ 58

The type system for the simply typed λ-calculus is defined recursively from a base type by adding function types.

type environment 62

A list of assumptions, where some variable is given none to several types. Equality of type environments is not syntactic equality.

valuation 99

A valuation provides an interpretation of variables, which allows us to interpret λ-terms that contain free variables.

variables 16

The variables in a λ-term is the union of its sets of free and bound variables.

COMP31311

Coursework 1

Deadline: 7th November, 18.00

AExercise 1. Carry out the following tasks:

(a) Select a λ-term that does not appear in the lecture notes and that contains at least two redexes and that has at least one reduction sequence that consists of four steps. Draw a graph which shows all the possible β-reductions (or $\beta\alpha$-reductions) starting from your term; see the graph on Page 42 as an example. (4 marks)

(b) Give a stepwise calculation (with justifications) of the parallel reduct of your term. It's okay to calculate several steps at once for ren and substitution, or where you operate on different parts of the term. (3 marks)

(c) In your graph from part (a) identify a term that is α-equivalent to the term you obtained in part (b). (1 mark)

(d) Give a proof of Proposition 1.23. (12 marks)

We appreciate that this a longish technical result, and that proving a statement like this might be quite daunting. Students should write down their best attempt, making it clear which results from the notes they are making use of. Marks will be awarded for partial attempts. It is okay to indicate gaps in the process where student are unsure how to obtain the intended statement.

Remember the following:

- you need to submit your coursework on Blackboard;
- you may only use definitions and results from the course notes;
- you should justify each step in your proofs;
- discussing assessed work with others before submission amounts to academic malpractice;
- if you have question about the work set you may ask them in the workshops or on the Blackboard discussion forum.

COMP31311

Coursework 2

Deadline: 19th December, 18.00

AExercise 2. Carry out the following tasks:

(a) For the type environment $c : \iota \to \iota \to \iota$, what is the type of the term

$$\lambda x : \iota.\, (\lambda y : \iota.\, cxy)(cxx)?$$

Give a formal derivation. (3 marks)

(b) Recall that there is a unique (empty) function $! : \emptyset \longrightarrow \emptyset$. Compute the following denotions of types:

(i) $[\![\iota]\!]^{\emptyset}$

(ii) $[\![\iota \to \iota]\!]^{\emptyset}$

(iii) $[\![(\iota \to \iota) \to (\iota \to \iota)]\!]^{\emptyset}$

(iv) $[\![(\iota \to \iota) \to \iota]\!]^{\emptyset}$

For each of these denotations, determine its number of elements.

Can you conclude anything from your calculation about the set of terms t such that $\vdash t : (\iota \to \iota) \to \iota$ is derivable? (3 marks)

(c) Prove Proposition 2.38. (8 marks)

(d) Use the adequacy of the function model of the simply typed λ-calculus to show that the terms

$$\lambda x : \iota.\, \lambda s : \iota \to \iota.\, (\lambda g : \iota \to \iota.\, \langle \lambda f : \iota \to \iota.\, \lambda y : \iota.\, f(f(f\, y)))\rangle \langle \lambda z : \iota.\, g(g(g(g\, z)))\rangle x)s$$

and

$$\lambda x : \iota.\, \lambda s : \iota \to \iota.\, (\lambda f : \iota \to \iota.\, \langle \lambda g : \iota \to \iota.\, \lambda z : \iota.\, g(g(g(g\, z)))\rangle \langle \lambda y : \iota.\, f(f(f\, y)))\rangle x)s$$

are contextually equivalent. (6 marks)

We are happy for students to supply denotations for the terms in angle brackets without explanation, but the remaining steps should be justified.

Remember the following:

- you need to submit your coursework on Blackboard;
- you may only use definitions and results from the course notes;
- you should justify each step in your proofs;
- discussing assessed work with others before submission amounts to academic malpractice;
- if you have question about the work set you may ask them in the workshops or on the Blackboard discussion forum.

Technical Proofs

Technical Proofs from Chapter 1

Proposition 1.2

Proof. We show each statement.

(a) This wants to be a proof by induction over the structure of the argument of ren.

bcVar In the base case we have t is a variable, say y, and

$$\operatorname{ren}^x_x y = \begin{cases} x & x = y \\ y & \text{else} \end{cases}$$

and we get y back in either case.

scAbs

$$\begin{aligned} \operatorname{ren}^x_x(\lambda y.\, t) &= \lambda(\operatorname{ren}^x_x y).\, (\operatorname{ren}^x_x t) && \text{scAbsren} \\ &= \lambda y.\, t && \text{base case above and ind hyp} \end{aligned}$$

scApp

$$\begin{aligned} \operatorname{ren}^x_x(tt') &= (\operatorname{ren}^x_x t)(\operatorname{ren}^x_x t') && \text{ScAppren} \\ &= tt' && \text{induction hypothesis} \end{aligned}$$

(b) This wants to be another proof by induction.

bcVarren In the base case we have

$$\operatorname{ren}^y_x x' = \begin{cases} y & x = x' \\ x' & \text{else} \end{cases}$$

We now apply ren^z_y to the result.

$$\operatorname{ren}^z_y \operatorname{ren}^y_x x' = \begin{cases} z & x = x' \\ z & x \neq x' \text{ and } y = x' \\ x' & \text{else} \end{cases}$$

We know from the assumption that $y \notin \operatorname{vars} t = \operatorname{vars} v = \{x\}$, so the case that appears in the middle cannot arise. Bearing that in mind, if we look at

$$\operatorname{ren}^z_x x' = \begin{cases} z & x' = x \\ x' & \text{else} \end{cases}$$

we can see that the two functions agree.

scAbs

$$\begin{aligned}
\mathrm{ren}^z_y\,\mathrm{ren}^y_x(\lambda x'.\,t) &= \mathrm{ren}^z_y(\lambda(\mathrm{ren}^y_x\,x').\,(\mathrm{ren}^y_x\,t)) && \text{scAbsren}\\
&= \lambda(\mathrm{ren}^z_y\,\mathrm{ren}^y_x\,x').\,(\mathrm{ren}^z_y\,\mathrm{ren}^y_x\,t) && \text{scAbsren}\\
&= \lambda(\mathrm{ren}^z_x\,x').\,(\mathrm{ren}^z_x\,t) && \text{bc from above and ind hyp}\\
&= \mathrm{ren}^z_x(\lambda x'.\,t) && \text{scAbsren}
\end{aligned}$$

scApp

$$\begin{aligned}
\mathrm{ren}^z_y\,\mathrm{ren}^y_x(tt') &= \mathrm{ren}^z_y((\mathrm{ren}^y_x\,t)(\mathrm{ren}^y_x\,t')) && \text{scApp}\\
&= (\mathrm{ren}^z_y\,\mathrm{ren}^y_x\,t)(\mathrm{ren}^z_y\,\mathrm{ren}^y_x\,t) && \text{scAppren}\\
&= (\mathrm{ren}^z_x\,t)(\mathrm{ren}^z_x\,t') && \text{induction hypothesis}\\
&= \mathrm{ren}^z_x(tt') && \text{scAppren}
\end{aligned}$$

(c) We don't need a proof by induction this time—this follows from the previous two parts since

$$\begin{aligned}
\mathrm{ren}^x_y\,\mathrm{ren}^y_x\,t &= \mathrm{ren}^x_x\,t && \text{previous part}\\
&= t && \text{first part}
\end{aligned}$$

(d) This is another proof by induction.

bcVar There are a few case distinctions to be made if the term is a variable, say z.

- If z is distinct from x and y then

$$\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y\,z = \mathrm{ren}^{x'}_x\,z = z = \mathrm{ren}^{y'}_y\,z = \mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x\,z.$$

- If $z = x$ (in which case it can't also be equal to any of the other variables) we have

$$\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y\,z = \mathrm{ren}^{x'}_x\,z = x' = \mathrm{ren}^{y'}_y\,x' = \mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x\,z.$$

- If $z = y$ (in which case it can't also be equal to any of the other variables) the argument is similar.

scAbs If the term is a λ-abstraction, say $\lambda z.\,u$ we have that

$$\begin{aligned}
\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y(\lambda z.\,u) &= \mathrm{ren}^{x'}_x(\lambda(\mathrm{ren}^y_y\,z).\,\mathrm{ren}^{y'}_y\,u) && \text{scAbsren}\\
&= \lambda(\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y\,z).\,\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y\,u && \text{scAbsren}\\
&= \lambda(\mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x\,z).\,\mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x\,u && \text{ind hyp and base case above}\\
&= \mathrm{ren}^{y'}_y(\lambda(\mathrm{ren}^{x'}_x\,z).\,\mathrm{ren}^{x'}_x\,u) && \text{scAbsren}\\
&= \mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x(\lambda z.\,u) && \text{scAbsren}
\end{aligned}$$

scApp If the term is an application, say rs, then

$$\begin{aligned}
\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y\,rs &= \mathrm{ren}^{x'}_x(\mathrm{ren}^{y'}_y\,r)(\mathrm{ren}^{y'}_y\,s) && \text{scAppren}\\
&= (\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y\,r)(\mathrm{ren}^{x'}_x\,\mathrm{ren}^{y'}_y\,s) && \text{scAppren}\\
&= (\mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x\,r)(\mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x\,s) && \text{ind hyp}\\
&= \mathrm{ren}^{y'}_y(\mathrm{ren}^{x'}_x\,r)(\mathrm{ren}^{x'}_x\,s) && \text{scAppren}\\
&= \mathrm{ren}^{y'}_y\,\mathrm{ren}^{x'}_x\,rs
\end{aligned}$$

(e) This is a proof by induction.

bcVar If t is a variable, say z, then we know by assumption that $z \neq y$ and so by bcVarren we have $\mathrm{ren}_y^x\, z = z$.

scAbs If t is a λ-abstraction, say $\lambda z.\, u$, then using the assumption that y does not occur in in the term at all, we know in particular $y \neq z$.

$$\begin{aligned} \mathrm{ren}_y^x(\lambda z.\, u) &= \lambda(\mathrm{ren}_y^x\, z).\, (\mathrm{ren}_y^x\, u) && \text{scAbs,} \\ &= \lambda z.\, u && y \neq z \text{ and ind hyp.} \end{aligned}$$

scApp If t is an application, say rs then

$$\begin{aligned} \mathrm{ren}_y^x(rs) &= (\mathrm{ren}_y^x\, r)(\mathrm{ren}_y^x\, s) && \text{scAppren} \\ &= rs && \text{induction hypothesis} \end{aligned}$$

as required.

Proposition 1.3

Proof. We show each statement.

(a) We carry out the proof by induction following the definition of ren. Throughout we use the the alternative definition of vars from Exercise 2.

bcVar If the term is a variable, say z, then there are two cases. If $z = x$ then we get

$$\begin{aligned} \mathrm{vars}\,\mathrm{ren}_x^y\, z &= \mathrm{vars}\, y && \text{bcVarren, } z = x \\ &= \{y\} && \text{bcVarv}' \\ &= (\{x\} \cup \{y\}) \setminus \{x\} && \text{calcs with sets} \\ &= (\mathrm{vars}\, z \cup \{y\}) \setminus \{x\} && z = x \end{aligned}$$

In the other case we have

$$\begin{aligned} \mathrm{fv}\,\mathrm{ren}_x^y\, z &= \mathrm{fv}\, z && \text{bcVarren, } x \neq v \\ &= \{z\} && \text{bcVarv}' \\ &\subseteq (\{z\} \cup \{y\}) \setminus \{x\} && z \neq x \end{aligned}$$

scAbs If the term is a λ-abstraction, say $t = \lambda x.\, u$ then

$$\begin{aligned} &\mathrm{vars}(\mathrm{ren}_x^y(\lambda z.\, u)) \\ &\quad = \mathrm{vars}(\lambda(\mathrm{ren}_x^y\, z).\, (\mathrm{ren}_x^y\, u)) && \text{scAbsren} \\ &\quad = \mathrm{vars}(\mathrm{ren}_x^y\, z) \cup \mathrm{vars}\,\mathrm{ren}_x^y\, u && \text{scAbsv}' \\ &\quad \subseteq ((\{z\} \cup \{y\} \setminus \{x\}) \cup (\mathrm{vars}\, u \cup \{y\}) \setminus \{x\} && \text{base case above and ind hyp} \\ &\quad = (\{z\} \cup \mathrm{vars}\, u \cup \{y\}) \setminus \{x\} && \text{calcs with sets} \\ &\quad = (\mathrm{vars}\, t \cup \{y\}) \setminus \{x\} && \text{scAbs"} \end{aligned}$$

scApp If the term is an application, say $t = rs$, then we have

$$\begin{aligned} \mathrm{vars}(\mathrm{ren}_x^y\, rs) &= \mathrm{vars}((\mathrm{ren}_x^y\, r)(\mathrm{ren}_x^y\, s)) && \text{scAppren} \\ &= \mathrm{vars}(\mathrm{ren}_x^y\, r) \cup \mathrm{fv}(\mathrm{ren}_x^y\, s) && \text{scAppv'} \end{aligned}$$

$$
\begin{aligned}
&\subseteq ((\operatorname{vars} r \cup \{y\}) \setminus \{x\}) \cup ((\operatorname{vars} s \cup \{y\}) \setminus \{x\} && \text{ind hyp}\\
&= (\operatorname{vars} r \cup \operatorname{vars} s \cup \{y\}) \setminus \{x\} && \text{calcs with sets}\\
&= (\operatorname{vars}(rs) \cup \{y\}) \setminus \{x\} && \text{scAppfv}\\
&= (\operatorname{vars} t \cup \{y\}) \setminus \{x\} && t = rs
\end{aligned}
$$

(b) We have that

$$
\begin{aligned}
\operatorname{fv}(\lambda y.\ \operatorname{ren}^y_x t) &= \operatorname{fv}(\operatorname{ren}^y_x t) \setminus \{y\} && \text{scAbsfv}\\
&\subseteq \operatorname{vars}(\operatorname{ren}^y_x t) \setminus \{y\} && \operatorname{fv} t \subseteq \operatorname{vars} t\\
&\subseteq ((\operatorname{vars} t \cup \{y\}) \setminus \{x\}) \setminus \{y\} && \text{previous part}\\
&= \operatorname{vars} t \setminus \{x, y\} && \text{calcs with sets}
\end{aligned}
$$

and so x is not an element of the given set.

(c) This is another proof by induction.

bcVar If the term is a variable, say z, then there are two cases.

- If $z = x$ then

$$\operatorname{fv}(\operatorname{ren}^y_x z) = \operatorname{fv}(\operatorname{ren}^y_x z) = \operatorname{fv} z = \{z\} = (\{x\} \setminus \{x\}) \cup \{z\} = (\operatorname{fv} x \setminus \{x\}) \cup \{z\},$$

  and we are in the second case.
- If $z \neq x$ then

$$\operatorname{fv}(\operatorname{ren}^y_x z) = \operatorname{fv} z$$

  and we are in the second case.

scAbs If the term is an abstraction, say $t = \lambda z.\, u$ then we have to make a case distinction.

- If $z = x$ then we know that $x = z \notin \operatorname{fv}(\lambda z.\, u)$, and we are in the first case of the claimed property, so we have to show that

$$\operatorname{fv}(\operatorname{ren}^y_x(\lambda z.\, u)) = \operatorname{fv}(\lambda z.\, u) = \operatorname{fv} u \setminus \{z\}.$$

  We calculate

$$
\begin{aligned}
\operatorname{fv}(\operatorname{ren}^y_x(\lambda z.\, u)) &= \operatorname{fv}(\operatorname{ren}^y_x(\lambda x.\, u)) && && z = x\\
&= \operatorname{fv}(\lambda y.\ \operatorname{ren}^y_x u) && && \text{scAbsren}\\
&= (\operatorname{fv}(\operatorname{ren}^y_x u)) \setminus \{y\} && && \text{scAbsfv}\\
&= \begin{cases} \operatorname{fv} u \setminus \{y\} & x \notin \operatorname{fv} u\\ ((\operatorname{fv} u \setminus \{x\}) \cup \{y\}) \setminus \{y\} = \operatorname{fv} u \setminus \{x\} & \text{else,} \end{cases}
\end{aligned}
$$

  where the final step is justified by the induction hypothesis. If $x = z \notin \operatorname{fv} u$ and we are in the first case then

$$
\begin{aligned}
\operatorname{fv} u \setminus \{y\} &= \operatorname{fv} u && \text{conditions for } y\\
&= \operatorname{fv} u \setminus \{z\} && z = x \notin \operatorname{fv} u.
\end{aligned}
$$

  If we are in the second case then

$$\operatorname{fv} u \setminus \{x\} = \operatorname{fv} u \setminus \{z\} \qquad x = z$$

  and so in both cases we match the required set.

- If $z \neq x$ then

$$\begin{aligned} \mathrm{fv}(\mathrm{ren}^y_x(\lambda z.\, u)) &= \mathrm{fv}\ \lambda z.\ \mathrm{ren}^y_x\, u && \text{scAbsren} \\ &= (\mathrm{fv}(\mathrm{ren}^y_x\, u)) \setminus \{z\} && \text{scAbsfv} \\ &= \begin{cases} \mathrm{fv}\, u \setminus \{z\} & x \notin \mathrm{fv}\, u \\ ((\mathrm{fv}\, u \setminus \{x\}) \cup \{y\}) \setminus \{z\} & \text{else} \end{cases} \end{aligned}$$

  where again the last step is justified by the induction hypothesis.
  We have two further cases to think about and match up with the above. If $x \notin \mathrm{fv}(\lambda z.\, u)$ then we have to match the above with $\mathrm{fv}(\lambda z.\, u)$, but since the assumption also means $x \notin \mathrm{fv}\, u$ we are in the first of the above cases and we can see that

$$\mathrm{fv}\, u \setminus \{z\} = \mathrm{fv}(\lambda z.\, u)$$

  as required
  If $x \in \mathrm{fv}(\lambda z.\, u)$ then we have to match the above with

$$(\mathrm{fv}(\lambda z.\, u) \setminus \{x\}) \cup \{y\},$$

  but in this situation we also have $x \in \mathrm{fv}\, u$ and so we are in the second of the above cases, and again we can see that

$$\begin{aligned} ((\mathrm{fv}\, u \setminus \{x\}) \cup \{y\}) \setminus \{z\} &= ((\mathrm{fv}\, u \setminus \{z\}) \setminus \{x\} \cup \{y\} && z \neq y,\ z \neq x \\ &= (\mathrm{fv}(\lambda z.\, u) \setminus \{x\}) \cup \{y\} && \text{scAbsfv}. \end{aligned}$$

  Hence in all cases we have matched up the sets as required.

scApp If the term is an application, say $t = rs$ then

$$\begin{aligned} \mathrm{fv}(\mathrm{ren}^y_x(rs)) &= \mathrm{fv}(\mathrm{ren}^y_x\, r\, \mathrm{ren}^y_x\, s) && \text{scAppren} \\ &= \mathrm{fv}\, \mathrm{ren}^y_x\, r \cup \mathrm{fv}\, \mathrm{ren}^y_x\, s && \text{scAppfv} \end{aligned}$$

We want to apply the induction hypothesis and that requires some case distinctions.

- If $x \notin \mathrm{fv}\, rs$ then we may continue

$$\begin{aligned} \ldots &= \mathrm{fv}\, r \cup \mathrm{fv}\, s && \text{ind hyp} \\ &= \mathrm{fv}(rs) && \text{scAppfv}. \end{aligned}$$

- If $x \in \mathrm{fv}\, rs$ we have to make further case distinctions:
  - If $x \in \mathrm{fv}\, r$ and $x \in \mathrm{fv}\, s$ then we may continue

$$\begin{aligned} \ldots &= ((\mathrm{fv}\, r \setminus \{x\}) \cup \{w\}) \cup ((\mathrm{fv}\, s \setminus \{x\}) \cup \{y\}) && \text{ind hyp} \\ &= ((\mathrm{fv}\, r \setminus \{x\}) \cup (\mathrm{fv}\, s \setminus \{x\}) \cup \{y\} && (A \cup B) \cup C = (A \cup C) \cup (B \cup C) \\ &= ((\mathrm{fv}\, r \cup \mathrm{fv}\, s) \setminus \{x\}) \cup \{y\} && (A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C) \\ &= (\mathrm{fv}(rs') \setminus \{x\}) \cup \{y\} && \text{scAppfv} \end{aligned}$$

  - If $x \in \mathrm{fv}\, r$ and $x \notin \mathrm{fv}\, s$ then we may continue

$$\begin{aligned} \ldots &= ((\mathrm{fv}\, r \setminus \{x\}) \cup \{y\}) \cup \mathrm{fv}\, s && \text{ind hyp} \\ &= ((\mathrm{fv}\, r \setminus \{x\}) \cup \mathrm{fv}\, s) \cup \{y\} && (A \cup B) \cup C = (A \cup C) \cup B \\ &= ((\mathrm{fv}\, r \cup \mathrm{fv}\, s) \setminus \{x\}) \cup \{y\} && (A \setminus \{s\}) \cup B = (A \cup B) \setminus \{s\} \text{ if } s \notin B \\ &= ((\mathrm{fv}\, rs) \setminus \{x\}) \cup \{y\} && \text{scAppfv}. \end{aligned}$$

  - The case where $x \notin \mathrm{fv}\, r$ and $x \in \mathrm{fv}\, s$ is symmetric to the previous one.

Proposition 1.4

Proof. We prove this by induction for the term t. We begin by showing that

$$t \sim_\forall t' \qquad \text{implies} \qquad t \sim_\exists t'$$

for all terms t and t'. Assume that we have terms t and t' such that $t \sim_\forall t'$.

bcVar If t is a variable, then so must t' be, and then they must be equal, so we immediately have that $t \sim_\exists t'$ by bcVar$\sim_\exists$.

scAbs If t is an abstraction, say $\lambda x.\, u$ then so must t' be, say $\lambda x'.\, u'$. Let z be a variable that is fresh for the two terms. By scAbs$\sim_\forall$ we know that $\mathrm{ren}^z_x\, u \sim_\forall \mathrm{ren}^z_{x'}\, u'$; the induction hypothesis gives $\mathrm{ren}^z_x\, u \sim_\exists \mathrm{ren}^z_{x'}\, u'$. We have now exhibited a fresh variable, namely z, with the property required to conclude that

$$\lambda x.\, u \sim_\exists \lambda x'.\, u'$$

as required.

scApp If t is an application, say rs, then it must be the case that t' is also an application, say $r's'$ and that

$$r \sim_\forall r' \qquad \text{and} \qquad s \sim_\forall s'.$$

The induction hypothesis immediately gives us that

$$r \sim_\exists r' \qquad \text{and} \qquad s \sim_\exists s'$$

as required.

We now show the opposite implication, so assume that

$$t \sim_\exists t'.$$

This is again a proof by induction, and the base case and the application step case are much the same as those we have just given (just swap the two relations), so we do not repeat them.

The remaining case is the abstraction case, which means we assume that

$$\lambda x.\, u \sim_\exists \lambda x'.\, u'.$$

By scAbs$\sim_\exists$ we know that that there exists a variable z fresh for the two given terms such that $\mathrm{ren}^z_x\, u \sim_\exists \mathrm{ren}^z_{x'}\, u'$ By the induction hypothesis, this means that $\mathrm{ren}^z_x\, t \sim_\forall \mathrm{ren}^z_{x'}\, t'$.

We want to show that $\lambda x.\, u \sim_\forall \lambda x'.\, u'$, that is for all w fresh for the two terms we have

$$\mathrm{ren}^w_x\, u \sim_\forall \mathrm{ren}^w_{x'}\, u'.$$

It is sufficient to show that for every such w we have $\mathrm{ren}^w_x\, u \sim_\exists \mathrm{ren}^w_{x'}\, u'$, since the induction hypothesis tells us that this implies what we want to show.

To prove this, we need to examine the terms u and u', so we will do another induction proof here, for the term u. Luckily this is not really a nested induction; we don't need to use the old induction hypothesis any more, so we can think of what follows as a separate induction proof, so we give the new statement for an arbitrary term t.

We will show the following by induction on t.

For all terms t,
for all terms t', variables x and x'
if there exists z fresh for x, x', t, t' such that $\mathrm{ren}^z_x\, t \sim_\forall \mathrm{ren}^z_{x'}\, t'$
then for all w fresh for x, x', t, t' we have $\mathrm{ren}^w_x\, t \sim_\exists \mathrm{ren}^w_{x'}\, t'$.

bcVar If t is a variable, then so must t' be. If $t = y$ and $t' = y'$ then for $\mathrm{ren}^z_x\, y \sim_\forall \mathrm{ren}^z_{x'}\, y'$ to hold by bcVar$\sim_\forall$ we must have $\mathrm{ren}^z_x\, y = \mathrm{ren}^z_{x'}\, y'$.
But if these two terms are equal then they are also related, by bcVar$\sim_\exists$, by that relation.

scAbs If t is an abstraction, say $t = \lambda y.\, u$ then so is t', say $t' = \lambda y'.\, u'$. By assumption there exists z fresh for x, x', y, y', u, u' such that $\mathrm{ren}^z_x(\lambda y.\, s) \sim_\forall \mathrm{ren}^z_{x'}(\lambda y'.\, s')$.
To unpack this definition we will have to work with $\mathrm{ren}^z_x\, y$ and $\mathrm{ren}^z_{x'}\, y'$ frequently, so we define $y_z = \mathrm{ren}^z_x\, y$ and $y'_z = \mathrm{ren}^z_{x'}\, y'$. The given statement tells us by scAbs$\sim_\forall$ that for all z' fresh for $y_z, y'_z, \mathrm{ren}^z_x\, u, \mathrm{ren}^z_{x'}\, u'$ we have $\mathrm{ren}^{z'}_{y_z}\, \mathrm{ren}^z_x\, u \sim_\forall \mathrm{ren}^{z'}_{y'_z}\, \mathrm{ren}^z_{x'}\, u'$.

Using similar shortcuts $y_w = \mathrm{ren}^w_x\, y$ and$y'_w = \mathrm{ren}^w_{x'}\, y'$, we want to show that for all w fresh for x, x', y, y', u, u' there exists w' fresh for y_w, y'_w,$\mathrm{ren}^w_x\, u, \mathrm{ren}^w_{x'}\, u''$ such that

$$\mathrm{ren}^{w'}_{y_w}\, \mathrm{ren}^w_x\, u \sim_\exists \mathrm{ren}^{z'}_{y'_w}\, \mathrm{ren}^w_{x'}\, u'.$$

To that end let w be any variable fresh for x, x', y, y', u, u'. Note that if $w = z$ then what we have immediately implies what we want to show. So we proceed to prove that case where w and z are different. Note that this implies that w is fresh for $\mathrm{ren}^{z'}_x\, u$ and $\mathrm{ren}^z_{x'}\, u'$.

The trick with fresh variables is usually to make them as fresh as possible! So we now choose w' to be fresh for $x, x', y, y', u, u', w, z$. Note that this implies that w' is fresh for y_z, y'_z,$\mathrm{ren}^z_x\, u, \mathrm{ren}^z_{x'}\, u'$,$y_w, y'_w, \mathrm{ren}^w_x\, u, \mathrm{ren}^w_{x'}\, u'$.
Now since w' is fresh for $y_z, y'_z, \mathrm{ren}^z_x\, u, \mathrm{ren}^z_{x'}\, u'$ we have $\mathrm{ren}^{w'}_{y_z}\, \mathrm{ren}^z_x\, u \sim_\forall \mathrm{ren}^{w'}_{y'_z}\, \mathrm{ren}^z_{x'}\, u'$ by our assumption.
At this point, we would like to use the induction hypothesis, but it is not immediately clear how! First, we show that the terms involved in both the information we have and the statement we want to show can be written in a particular form; once this is done it the induction hypothesis applies.
Our plan is to find a term l such that we can write

$$\mathrm{ren}^{w'}_{y_z}\, \mathrm{ren}^z_x\, u \text{ as } \mathrm{ren}^z_x\, l \qquad \text{and} \qquad \mathrm{ren}^{w'}_{y_w}\, \mathrm{ren}^w_x\, u \text{ as } \mathrm{ren}^w_x\, l,$$

and a term r such that we can write

$$\mathrm{ren}^{w'}_{y'_z}\, \mathrm{ren}^z_{x'}\, u' \text{ as } \mathrm{ren}^z_{x'}\, r \qquad \text{and} \qquad \mathrm{ren}^{w'}_{y'_w}\, \mathrm{ren}^w_{x'}\, u' \text{ as } \mathrm{ren}^w_{x'}\, r.$$

Since the situation is symmetric, we only prove the version involving l and leave the reader to check that exactly the same argument works on the right hand side of the equivalence.
Because there are two variables x and y involved, we have to work in cases. Consider first the case where $y = x$. Then we have

$$y_z = \mathrm{ren}^z_x\, y = z \qquad \text{as well as} \qquad y_w = \mathrm{ren}^w_x\, y = w$$

and so

$$\begin{aligned} \mathrm{ren}^{w'}_{y_z}\, \mathrm{ren}^z_x\, u &= \mathrm{ren}^{w'}_z\, \mathrm{ren}^z_x\, u \\ &= \mathrm{ren}^{w'}_x\, u \\ &= \mathrm{ren}^z_x\, \mathrm{ren}^{w'}_x\, u \end{aligned}$$

and

$$\mathrm{ren}^{w'}_{y_w}\, \mathrm{ren}^w_x\, u = \mathrm{ren}^{w'}_w\, \mathrm{ren}^w_x\, u$$

$$= \mathrm{ren}_x^{w'}\, u$$
$$= \mathrm{ren}_x^{w}\, \mathrm{ren}_x^{w'}\, u$$

so we can pick $l = \mathrm{ren}_x^{w'}\, u$.
On the other hand, suppose $x \neq y$. In that case we have

$$y_z = \mathrm{ren}_x^z\, y = y \qquad \text{as well as} \qquad y_w = \mathrm{ren}_x^w\, y = y$$

and so

$$\begin{aligned}\mathrm{ren}_{y_z}^{w'}\, \mathrm{ren}_x^z\, u &= \mathrm{ren}_y^{w'}\, \mathrm{ren}_x^z\, u \\ &= \mathrm{ren}_x^z\, \mathrm{ren}_y^{w'}\, u \qquad \text{by Proposition 1.2 since } y \neq x \neq w', y \neq z\end{aligned}$$

and

$$\begin{aligned}\mathrm{ren}_{y_w}^{w'}\, \mathrm{ren}_x^z\, u &= \mathrm{ren}_y^{w'}\, \mathrm{ren}_x^w\, u \\ &= \mathrm{ren}_x^w\, \mathrm{ren}_y^{w'}\, u \qquad \text{by Proposition 1.2 since } y \neq x \neq w', y \neq w\end{aligned}$$

so we can pick $l = \mathrm{ren}_y^{w'}\, u$. Hence in both cases we have found a suitable l.

We may now reformulate our assumption as

$$\mathrm{ren}_x^z\, l \sim_\forall \mathrm{ren}_{x'}^z\, r$$

so the induction hypothesis tells us that

$$\mathrm{ren}_x^w\, l \sim_\exists \mathrm{ren}_{x'}^w\, r$$

and reformulating the terms again we obtain the required

$$\mathrm{ren}_{y_w}^{w'}\, \mathrm{ren}_x^w\, u \sim_\exists \mathrm{ren}_{y'_w}^{w'}\, \mathrm{ren}_{x'}^w\, u'.$$

scApp Suppose $t = rs$ which means we must have and $t' = r's'$. Then the assumption gives us, by scAppren

$$\mathrm{ren}_x^z\, r\, \mathrm{ren}_{x'}^z\, s \sim_\exists \mathrm{ren}_{x'}^z\, r'\, \mathrm{ren}_{x'}^z\, s',$$

which by scApp$\sim_\exists$ says that

$$\mathrm{ren}_x^z\, r \sim_\exists \mathrm{ren}_{x'}^z\, r' \qquad \text{and} \qquad \mathrm{ren}_x^z\, s \sim_\exists \mathrm{ren}_{x'}^z\, s'$$

and by the induction hypothesis we have that for all fresh variables w we have

$$\mathrm{ren}_x^w\, r \sim_\exists \mathrm{ren}_{x'}^w\, r' \qquad \text{and} \qquad \mathrm{ren}_x^w\, s \sim_\exists \mathrm{ren}_{x'}^w\, s',$$

which by scApp$\sim_\exists$ and scAppren gives the required

$$\mathrm{ren}_x^w(rs) \sim_\exists \mathrm{ren}_{x'}^w(r's').$$

Proposition 1.7

Solution. We show each part.

(a) While this is a special case of the following part we find it convenient to establish this separately and then use it in the proof of that part. This is a proof by induction over the structure of t.

bcVar If t is a variable, say $t = y$, then we know based on the assumptions that $x \notin \operatorname{fv} y = \{y\}$ that $\operatorname{ren}_x^y y = y$ which is α-equivalent to itself by bcVarα.

scAbs If t is an abstraction, say $\lambda y.\, u$, then we have that

$$x \notin \operatorname{fv}(\lambda y.\, u) = \operatorname{fv} u \setminus \{y\}).$$

We have to make a case distinction.

- If $y = x$ then $\operatorname{ren}_x^z t = \operatorname{ren}_x^z(\lambda x.\, u) = \lambda z.\ \operatorname{ren}_x^z u$. In order to show that

$$\lambda x.\, u \sim_\alpha \lambda z.\ \operatorname{ren}_x^z u$$

it is sufficient to show that for a variable z' that is fresh for these two terms we have

$$\operatorname{ren}_x^{z'} u \sim_\alpha \operatorname{ren}_z^{z'} \operatorname{ren}_x^z u$$

but we know that the expression on the right is equal to $\operatorname{ren}_x^{z'} u$ by Proposition 1.2, which we may employ since we know that $z \notin \operatorname{vars} u$. Hence the two expressions are equal and so α-equivalent by Proposition 1.6.

- If $y \neq x$ then $\operatorname{ren}_x^z t = \operatorname{ren}_x^z(\lambda y.\, u) = \lambda y.\ \operatorname{ren}_x^z u$. In this situation we know that $x \notin \operatorname{fv} u$ and $z \notin \operatorname{vars} u$ and we may apply the induction hypothesis to deduce that $u \sim_\alpha \operatorname{ren}_x^z u$.

scApp If t is an application, say rs, then we know that $x \notin \operatorname{fv}(rs) = \operatorname{fv} r \cup \operatorname{fv} s$, as well as $z \notin \operatorname{vars}(rs) = \operatorname{vars} r \cup \operatorname{vars} s$, so in particular

$$x \notin \operatorname{fv} r \qquad \text{and} \qquad z \notin \operatorname{fv} s$$

and similarly

$$x \notin \operatorname{vars} r \qquad \text{and} \qquad z \notin \operatorname{vars} s.$$

This allows us to apply the induction hypothesis to obtain that $r \sim_\alpha \operatorname{ren}_x^z r$ and $s \sim_\alpha \operatorname{ren}_x^z s$, and so by scApp$\alpha$ we get $rs \sim_\alpha (\operatorname{ren}_x^z r)(\operatorname{ren}_x^z s) = \operatorname{ren}_x^z(rs)$ as required.

(b) This is a proof by induction over the structure of t. The induction hypothesis is that the claim holds for proper subterms where one variable may have been renamed.

bcVar If t is a variable, say y then for $t \sim_\alpha t'$ to hold we must have $t' = y = t$. In this situation $\operatorname{ren}_x^z t = \operatorname{ren}_x^z t'$ and so the two terms are α-equivalent by reflexivity of that relation.

scAbs If t is a λ-abstraction, say $t = \lambda y.\, u$, then for $t \sim_\alpha t'$ to hold t' must also be a λ-abstraction, say $\lambda y x'.\, u'$, and we must have for every variable z' fresh for t and t' that

$$\operatorname{ren}_y^{z'} u \sim_\alpha \operatorname{ren}_{y'}^{z'} u'.$$

When we calculate

$$\operatorname{ren}_x^z(\lambda y.\, u) \qquad \text{and} \qquad \operatorname{ren}_x^z \lambda y'.\, u'$$

we find it useful to make a case distinction.

- If $y = x$ or $y' = x$ then x does not occur free in at least one of $t\ t'$, which means it also does not occur free in the other by Proposition 1.6 and then we have by the previous part that

$$\operatorname{ren}_x^z t \sim_\alpha t \sim_\alpha t' \sim_\alpha \operatorname{ren}_x^{z'} t.$$

- If $y \neq x$ and $y' \neq x$ then

$$\mathrm{ren}^z_x(\lambda y.\, u) = \lambda y.\, \mathrm{ren}^z_x\, u \qquad \text{and} \qquad \mathrm{ren}^z_x(\lambda y'.\, u') = \lambda y'.\, \mathrm{ren}^z_x\, u'.$$

  It is sufficient to show that there is a variable z'' which is fresh for the two terms with the property that

$$\mathrm{ren}^{z''}_y\, \mathrm{ren}^z_x\, u \sim_\alpha \mathrm{ren}^{z''}_{y'}\, \mathrm{ren}^z_x\, u.$$

  We may choose our variable z'' to be different from all variables named so far, and we already know that $z \neq y$ and $x \neq y$, and so we may apply Proposition 1.2 to conclude that we have $\mathrm{ren}^{z'}_x\, \mathrm{ren}^z_v\, u = \mathrm{ren}^z_v\, \mathrm{ren}^{z'}_x\, u$ and similarly for the other term.
  Our claim now follows from the induction hypothesis applied to $\mathrm{ren}^{z''}_x\, u \sim_\alpha \mathrm{ren}^{z''}_{x'}\, u'$ which we get from the assumption by setting $z' = z''$.

scApp If t is an application, say $t = rs$, then for $t \sim_\alpha t'$ to hold t' must also be an application, say $r's'$, and we must have $r \sim_\alpha r'$ and $s \sim_\alpha s'$. By the induction hypothesis this implies $\mathrm{ren}^z_x\, r \sim_\alpha \mathrm{ren}^z_x\, r'$ and $\mathrm{ren}^z_x\, s \sim_\alpha \mathrm{ren}^z_x\, s'$ and by scAppα we have

$$\mathrm{ren}^z_x\, t = \mathrm{ren}^z_x(rs) = (\mathrm{ren}^z_x\, r)(\mathrm{ren}^z_x\, s) \sim_\alpha (\mathrm{ren}^z_x\, r')(\mathrm{ren}^z_x\, s') = \mathrm{ren}^z_x(r's') = \mathrm{ren}^z_x\, t'.$$

(c) This follows from part (b) since x does not occur freely in $\lambda x.\, t$.

(d) These are two λ-abstractions and so we have to show that for there is a variable z which is fresh for both terms such that $\mathrm{ren}^z_x\, t \sim_\alpha \mathrm{ren}^z_x\, t'$, and so this immediately follows from part (c).

Proposition 1.10

Proof.

(a) We proceed by induction following the definition of substitution. The induction hypothesis is that the claim applies to subterms, as well as subterms where one variable has been renamed.

bcVar If t is a variable, say y, then if $y = x$ the result of the substitution is a, and so

$$\mathrm{fv}(t[a/x]) = \mathrm{fv}(x[a/x]) = \mathrm{fv}\, a = (\mathrm{fv}\, x \setminus \{x\}) \cup \mathrm{fv}\, a.$$

If $y \neq x$ then

$$\mathrm{fv}(t[a/x]) = \mathrm{fv}(y[a/x]) = \mathrm{fv}\, y = \{y\} \subseteq \{y\} \cup \mathrm{fv}\, a = (\{y\} \setminus \{x\}) \cup \mathrm{fv}\, a.$$

scAbs If t is a λ-abstraction, say $t = \lambda y.\, u$ then for a va$w = \mathrm{freshv}(\mathrm{vars}\, a \cup \mathrm{vars}\, t \cup \{x\})$ we have

$$t[a/x] = \lambda w.\, (\mathrm{ren}^w_y\, u)[a/x].$$

Hence

$$\begin{aligned}
\mathrm{fv}(t[a/x]) &= \mathrm{fv}(\lambda w.\, (\mathrm{ren}^w_y\, u)[a/x]) && \text{above}\\
&= \mathrm{fv}((\mathrm{ren}^w_y\, u)[a/x]) \setminus \{w\} && \text{scAbsfv}\\
&\subseteq ((\mathrm{fv}(\mathrm{ren}^w_y\, u) \setminus \{x\}) \cup \mathrm{fv}\, a) \setminus \{w\} && \text{ind hyp}\\
&\subseteq (((\mathrm{fv}\, u \cup \{w\}) \setminus \{x\}) \cup \mathrm{fv}\, a) \setminus \{w\} && \text{Proposition 1.3}\\
&\subseteq (\mathrm{fv}\, u \setminus \{x, w\}) \cup \mathrm{fv}\, a && \text{calcs with sets}\\
&= (\mathrm{fv}(\lambda y.\, u) \setminus \{x\}) \cup \mathrm{fv}\, a && \text{scAbsfv}\\
&= (\mathrm{fv}\, t \setminus \{x\}) \cup \mathrm{fv}\, a && t = \lambda y.\, u.
\end{aligned}$$

scApp If we have an application, say $t = rs$ then $(rs)[a/x] = (r[a/x])(s[a/x])$ and

$$\begin{aligned}
\operatorname{fv}((rs)[a/x]) &= \operatorname{fv}((r[a/x])(s[a/x])) && \text{above}\\
&= \operatorname{fv}(r[a/x]) \cup \operatorname{fv}(s[a/x]) && \text{scApp[ ]}\\
&\subseteq (\operatorname{fv}(r \setminus \{x\}) \cup \operatorname{fv} a) \cup (\operatorname{fv}(s \setminus \{x\}) \cup \operatorname{fv} a) && \text{ind hyp}\\
&= ((\operatorname{fv} r \cup \operatorname{fv} s) \setminus \{x\}) \cup \operatorname{fv} a && \text{calcs with sets}\\
&= (\operatorname{fv}(rs) \setminus \{x\}) \cup \operatorname{fv} a && \text{scAppfv}\\
&= (\operatorname{fv} t \setminus \{x\}) \cup \operatorname{fv} a && t = rs.
\end{aligned}$$

This completes the proof.

(b) This is another proof by induction.

bcVar If our term is a variable, say, y then the condition means that this variable cannot be x, and $y[a/x] = y$, which is α-equivalent to itself.

scAbs If our term is a λ-abstraction, say $\lambda y.\, u$, then we get, for $w = \operatorname{freshv}(\operatorname{vars} a \cup \operatorname{vars} t \cup \{x\})$,

$$(\lambda y.\, u)[a/x] = \lambda w.\, (\operatorname{ren}_y^w t)[a/x].$$

In order to show our claim we have to show that this term is α-equivalent to $\lambda y.\, u$, and that means showing that for a variable z which is fresh for $\lambda y.\, u)$ and $\lambda w.\ \operatorname{ren}_y^w u[a/x]))$ we have

$$\operatorname{ren}_y^z u \sim_\alpha \operatorname{ren}_w^z((\operatorname{ren}_y^w u)[a/x]).$$

Knowing that $x \notin \operatorname{fv}(\lambda y.\, u) = \operatorname{fv} u \setminus \{y\}$, combined with $x \neq w$, tells us that

$$x \notin (\operatorname{fv} u \setminus \{y\}) \cup \{w\} \supseteq \operatorname{fv}(\operatorname{ren}_y^w u),$$

so we may apply the induction hypothesis to give us that

$$\operatorname{ren}_y^w u \sim_\alpha (\operatorname{ren}_y^w u)[a/x].$$

We get the desired result from Proposition 1.7 by applying ren_w^z on both sides noting that by Proposition 1.2 we have $\operatorname{ren}_w^z \operatorname{ren}_y^w u = \operatorname{ren}_y^z u$.

scApp If our term is an application, say rs then

$$\begin{aligned}
(rs)[x/v] &= (r[x/v])(s[x/v]) && \text{scApp[ ]}\\
&\sim_\alpha rs && \text{ind hyp, scApp}\alpha
\end{aligned}$$

(c) We show the statement by induction over the structure of the term t.

bcVar If t is a variable, say y', then

$$(\operatorname{ren}_y^z y')[a/x] = \begin{cases} a & \operatorname{ren}_y^z y' = x \\ \operatorname{ren}_y^z y' & \text{else.} \end{cases}$$

We observe that we are in the first case if and only if the renaming has no effect (since $z \neq x$) and also $y' = x$, and so this case arises if and only $y' = x$ and $y' \neq y$. We further observe that z cannot occur in a and so $\operatorname{ren}_z^{z'} a = a$. Hence

$$\operatorname{ren}_z^{z'}((\operatorname{ren}_y^z y')[a/x]) = \begin{cases} \operatorname{ren}_z^{z'} a = a & y' = x \text{ and } y' \neq y \\ \operatorname{ren}_z^{z'} \operatorname{ren}_y^z y' = \operatorname{ren}_y^{z'} y' & \text{else} \end{cases}$$

For the second term we similarly obtain

$$(\operatorname{ren}_y^{z'} y')[a/x] = \begin{cases} a & y' = x \text{ and } y' \neq y \\ \operatorname{ren}_y^z y' & \text{else.} \end{cases}$$

Hence we obtain the same result for both terms.

scAbs If t is an abstraction, say $\lambda y'.\, u$ then for our first term, for

$$w = \mathrm{freshv}(\mathrm{vars}(\mathrm{ren}^{z}_{y}\, t) \cup \mathrm{vars}\, a \cup \{x\}),$$

we have

$$\begin{aligned}
\mathrm{ren}^{z'}_{z}((\mathrm{ren}^{z}_{y}(\lambda y'.\, u))[a/x]) &= \mathrm{ren}^{z'}_{z}((\lambda(\mathrm{ren}^{z}_{y}\, y').\ \mathrm{ren}^{z}_{y}\, u)[a/x]) && \text{scAbsren}\\
&= \mathrm{ren}^{z'}_{z}(\lambda w.\, (\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && \text{scAbs[ ]}\\
&= \lambda(\mathrm{ren}^{z'}_{z}\, w).\ \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && \text{scAbsren.}
\end{aligned}$$

For our second term we may calculate, for

$$w' = \mathrm{freshv}(\mathrm{vars}(\mathrm{ren}^{z'}_{y}\, t) \cup \mathrm{vars}\, a \cup \{x\}),$$

that

$$\begin{aligned}
(\mathrm{ren}^{z'}_{y}(\lambda y'.\, u))[a/x] &= (\lambda(\mathrm{ren}^{z'}_{y}\, y').\ \mathrm{ren}^{z'}_{y}\, u)[a/x] && \text{scAbsren}\\
&= \lambda w'.\, (\mathrm{ren}^{w'}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z'}_{y}\, u)[a/x] && \text{scAbs[,],}
\end{aligned}$$

and we have to establish that these two terms are α-equivalent, that is we may find a fresh variable z'' such that

$$\mathrm{ren}^{z''}_{\mathrm{ren}^{z'}_{z}\, w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) \sim_{\alpha} \mathrm{ren}^{z''}_{w'}((\mathrm{ren}^{w'}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z'}_{y}\, u)[a/x]).$$

We note that the instances of renaming

$$\mathrm{ren}^{z''}_{\mathrm{ren}^{z'}_{z}\, w}, \qquad \mathrm{ren}^{z}_{z}, \qquad \text{and } \mathrm{ren}^{z''}_{w'}$$

that appear on the outside of these terms potentially satisfy the requirements of the statement under consideration regarding the freshness of variables and so potentially allow us to apply the induction hypothesis. By said induction hypothesis we have

$$\mathrm{ren}^{z''}_{w'}((\mathrm{ren}^{w'}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]) \sim_{\alpha} (\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]$$

for the term on the right hand side. The situation for the term on the left is slightly more complicated, and we have to make a case distinction.

- We have to consider that we might have $w = z$. Given the definition of w this can only happen if z does not occur in $\mathrm{ren}^{z}_{y}(\lambda y'.\, u)$, which by freshness of z can only occur inf y does not occur in $\lambda y'.\, u$. In that situation our terms simplify a bit. We have

  $$\begin{aligned}
  &\mathrm{ren}^{z''}_{\mathrm{ren}^{z'}_{z}\, w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{z}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x])\\
  &\quad = \mathrm{ren}^{z''}_{z'}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{z}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]) && w = z,\ y \notin \mathrm{vars}\, u\\
  &\quad = \mathrm{ren}^{z''}_{z}(\mathrm{ren}^{z}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x] && \text{Proposition 1.2}\\
  &\quad \sim_{\alpha} (\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x] && \text{ind hyp}
  \end{aligned}$$

  In this situation, where we know that y does not occur in u, the term which we need to be α-equivalent to the above is

  $$(\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x] = (\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]$$

  and we are done.

- If $w \neq z$ and $w = y$ then by definition of w the variable y does not occur in $\lambda y'.\, u$ and we may calculate for the term on the left hand side:

$$\begin{aligned}
&\mathrm{ren}^{z''}_{\mathrm{ren}^{z'}_{z}\, w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && \\
&\quad = \mathrm{ren}^{z''}_{w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]) && w \neq z,\ y \notin \mathrm{vars}(\lambda y'.\, u) \\
&\quad = \mathrm{ren}^{z''}_{w}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]) && z \text{ fresh for } t, x, a, w \\
&\quad \sim_{\alpha} (\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x] && \text{ind hyp.}
\end{aligned}$$

  Again we know that y does not occur in u and so the term which we require to be α-equivalent to the above is

$$(\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x] = (\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]$$

  and again we are done.
- If $w \neq z$ and $w \neq y$ and $y = y'$ we obtain the following for the term on the right hand side.

$$\begin{aligned}
&\mathrm{ren}^{z''}_{\mathrm{ren}^{z'}_{z}\, w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && \\
&\quad = \mathrm{ren}^{z''}_{w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{z}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && w \neq z, y = y' \\
&\quad = \mathrm{ren}^{z''}_{w}((\mathrm{ren}^{w}_{z}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && z \notin \mathrm{vars}(\mathrm{ren}^{w}_{z}\, r) \text{ for all terms } r \\
&\quad = \mathrm{ren}^{z''}_{w}((\mathrm{ren}^{w}_{y}\, u)[a/x]) && \text{Proposition 1.2} \\
&\quad \sim_{\alpha} (\mathrm{ren}^{z''}_{y}\, u)[a/x] && \text{ind hyp}
\end{aligned}$$

  and since we know that $\mathrm{ren}^{z}_{y}\, y' = z$ our target term is

$$(\mathrm{ren}^{z''}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x] = (\mathrm{ren}^{z''}_{z}\, \mathrm{ren}^{z}_{y}\, u)[a/x] = (\mathrm{ren}^{z''}_{y}\, u)[a/x]$$

  and we are done once again.
- If $w \neq z$ and $w \neq y$ and $y \neq y'$ then we have that our variables are sufficiently distinct to allow us to swap the two renaming operations for the term we substitute into when computing the term on the left hand side, using Proposition 1.2. We have $y \neq y'$ and $y \neq w$ by assumption, and we know that z is fresh for y'.
  Hence we obtain

$$\begin{aligned}
&\mathrm{ren}^{z''}_{\mathrm{ren}^{z'}_{z}\, w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, \mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && \\
&\quad = \mathrm{ren}^{z''}_{w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, \mathrm{ren}^{z}_{y}\, u)[a/x]) && w \neq z, y \neq y' \\
&\quad = \mathrm{ren}^{z''}_{w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{z}_{y}\, \mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]) && \text{Proposition 1.2.}
\end{aligned}$$

  The induction hypothesis tells us that

$$\mathrm{ren}^{z'}_{z}((\mathrm{ren}^{z}_{y}\, \mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]) \sim_{\alpha} (\mathrm{ren}^{z}_{y}\, \mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x],$$

  and we may apply the renaming operator on both sides and continue our calculation as follows.

$$\mathrm{ren}^{z''}_{w}\, \mathrm{ren}^{z'}_{z}((\mathrm{ren}^{z}_{y}\, \mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x]) \sim_{\alpha} \mathrm{ren}^{z''}_{w}(\mathrm{ren}^{z}_{y}\, \mathrm{ren}^{w}_{\mathrm{ren}^{z}_{y}\, y'}\, u)[a/x] \quad \text{Proposition 1.7}$$

$$= \mathrm{ren}_w^{z''}((\mathrm{ren}^w_{\mathrm{ren}^z_y\, y'}\, \mathrm{ren}^z_y\, u)[a/x]) \quad \text{Proposition 1.2}$$
$$\sim_\alpha (\mathrm{ren}^{z''}_{\mathrm{ren}^z_y\, y'}\, \mathrm{ren}^z_y)[a/x] \quad \text{ind hyp}$$

gives the required instance of the α-equivalence relation.

scApp If t is an application, say rs, we may calculate as follows:

$$\begin{aligned}
\mathrm{ren}^{z'}_z((\mathrm{ren}^z_{y'}(rs))[a/x]) &= \mathrm{ren}^{z'}_z(((\mathrm{ren}^z_y r)(\mathrm{ren}^z_y s))[a/x]) && \text{scAppren}\\
&= \mathrm{ren}^{z'}_z(((\mathrm{ren}^z_y r)[a/x])((\mathrm{ren}^z_y s)[a/x])) && \text{scApp[ ]}\\
&= (\mathrm{ren}^{z'}_z((\mathrm{ren}^z_y r)[a/x]))(\mathrm{ren}^{z'}_z((\mathrm{ren}^z_y s)[a/x])) && \text{scAppren}\\
&\sim_\alpha ((\mathrm{ren}^{z'}_y r)[a/x])((\mathrm{ren}^{z'}_y s)[a/x]) && \text{ind hyp}\\
&= ((\mathrm{ren}^{z'}_y r)(\mathrm{ren}^{z'}_y s))[a/x] && \text{scApp[ ]}\\
&= (\mathrm{ren}^{z'}_y (rs))[a/x] && \text{scAppren.}
\end{aligned}$$

which establishes the required instance of the equivalence relation.

(d) We may use the previous part to establish this result. We may calculate that for

$$w = \mathrm{freshv}(\mathrm{vars}(\lambda y.\, t) \cup \mathrm{vars}\, a \cup \{x\})$$

we have

$$(\lambda y.\, t)[a/x] = \lambda w.\, (\mathrm{ren}^w_y\, t)[a/x],$$

and if z' is a variable which is fresh for $\lambda y.\, t$, a, x, w and z, then

$$\begin{aligned}
\mathrm{ren}^{z'}_w((\mathrm{ren}^w_y u)[a/x]) &\sim_\alpha (\mathrm{ren}^{z'}_y u)[a/x] && \text{previous part}\\
&\sim_\alpha \mathrm{ren}^{z'}_z((\mathrm{ren}^z_y u[a/x]) && \text{previous part}
\end{aligned}$$

which is sufficient to show the required

$$\lambda w.\, (\mathrm{ren}^w_y\, t)[a/x] \sim_\alpha \lambda z.\, (\mathrm{ren}^z_y\, u)[a/x].$$

(e) There are quite a few variables in this statement, and that gives rise to a number of case distinctions being required. We find it convenient to make one of these case distinctions at the start, and so we provide two separate proofs by induction. This case distinction is based on whether or not we have

$$x = y,$$

that is, whether the variable we are renaming is also the variable we are substituting.
We first assume that $x = y$, and we only use the name x in the proof,[1] so the statement at issue is

$$\mathrm{ren}^z_x(t[a/x]) \sim_\alpha (\mathrm{ren}^z_x\, t)[\mathrm{ren}^z_x\, a/\, \mathrm{ren}^z_x\, x].$$

bcVar If t is a variable we have to make a case distinction. If that variable is x, then

$$\begin{aligned}
\mathrm{ren}^z_x(x[a/x]) &= \mathrm{ren}^z_x\, a && \text{bcVar[ ]}\\
&= z[\mathrm{ren}^z_x\, a/z] && \text{bcVar[ ]}\\
&= \mathrm{ren}^z_x\, x[\mathrm{ren}^z_x\, a/\, \mathrm{ren}^z_x\, x] && \text{bcVarren}
\end{aligned}$$

If that variable is y distinct from x then

$$\begin{aligned}
\mathrm{ren}^z_x(y[a/x]) &= \mathrm{ren}^z_x\, y && \text{bcVar[ ]}\\
&= y && \text{bcVarren}\\
&= y[\mathrm{ren}^z_x\, a/z] && \text{bcVar[ ], } z \neq y\\
&= (\mathrm{ren}^z_x\, y)[\mathrm{ren}^z_x\, a/\, \mathrm{ren}^z_x\, x] && \text{bcVarren}
\end{aligned}$$

[1] But be warned that a different variable called y appears below!

scAbs If t is a λ-abstraction, say $\lambda y.\, u$ we make another case distinction. For

$$w = \mathrm{freshv}(\mathrm{vars}\, t \cup \mathrm{vars}\, a \cup \{x\})$$

we have

$$\begin{aligned}
\mathrm{ren}^z_x((\lambda y.\, u)[a/x]) &= \mathrm{ren}^z_x(\lambda w.\, (\mathrm{ren}^w_y\, u)[a/x]) && \text{scAbs[ ]}\\
&= \lambda\, \mathrm{ren}^z_x\, w.\ \mathrm{ren}^z_x((\mathrm{ren}^w_y\, u)[a/x]) && \text{scAbsren}\\
&= \lambda w.\ \mathrm{ren}^z_x((\mathrm{ren}^w_y\, u)[a/x]) && \text{bcVarren, } w \neq x.\\
&\sim_\alpha \lambda w.\, (\mathrm{ren}^z_x\, \mathrm{ren}^w_y\, u)[\mathrm{ren}^z_x\, a/\, \mathrm{ren}^z_x\, x] && \text{ind hyp and Proposition 1.7,}\\
&= \lambda w.\, (\mathrm{ren}^z_x\, \mathrm{ren}^w_y\, u)[\mathrm{ren}^z_x\, a/z] && \text{bcVarren,}
\end{aligned}$$

where in the penultimate step we use the induction hypothesis to obtain an α-equivalent term under the abstraction, and Proposition 1.7 to tell us that when we abstract the same variable for two α-equivalent terms, we obtain α-equivalent results.
For

$$w' = \mathrm{freshv}(\mathrm{vars}(\lambda(\mathrm{ren}^z_x\, y).\ \mathrm{ren}^z_x\, u) \cup \mathrm{vars}(\mathrm{ren}^z_x\, a) \cup \{z\})$$

we have that

$$\begin{aligned}
(\mathrm{ren}^z_x(\lambda y.\, u))[\mathrm{ren}^z_x\, a/\, \mathrm{ren}^z_x\, x] &= (\lambda(\mathrm{ren}^z_x\, y).\ \mathrm{ren}^z_x\, u)[\mathrm{ren}^z_x\, a/z] && \text{scAbsren, bcVarren}\\
&= (\lambda(\mathrm{ren}^z_x\, y).\ \mathrm{ren}^z_x\, u)[\mathrm{ren}^z_x\, a/z] && \text{bcVarren}\\
&= \lambda w'.\, (\mathrm{ren}^{w'}_{\mathrm{ren}^z_x\, y}\, \mathrm{ren}^z_x\, u)[\mathrm{ren}^z_x\, a/z] && \text{scAbs[ ]}
\end{aligned}$$

To show that the two resulting terms are α-equivalent we make a case distinction.

- If $x = y$ then after renaming $y = x$ to w there are no occurrences of x left in u and so renaming x to z has no effect, meaning that the first term is

  $$\lambda w.\, (\mathrm{ren}^w_y\, u)[\mathrm{ren}^z_x\, a/\, \mathrm{ren}^z_x\, x],$$

  while for the second term we observe that in this situation we have $\mathrm{ren}^z_x\, y = z$, so that term is

  $$\lambda w'.\, (\mathrm{ren}^{w'}_z\ \mathrm{ren}^z_x\, u)[\mathrm{ren}^z_x\, a/z] = \lambda w'.\, (\mathrm{ren}^{w'}_x\, u)[\mathrm{ren}^z_x\, a/z].$$

  These two terms are α-equivalent by the previous part.
- If $x \neq y$ we have that $\mathrm{ren}^z_x\, y = y$, and so the second term is

  $$\lambda w'.\, (\mathrm{ren}^{w'}_y\ \mathrm{ren}^z_x\, u)[\mathrm{ren}^z_x\, a/z]$$

  and we may use Proposition 1.2 to swap the two instances of the renaming operator since we know that

  $$y \neq x, \qquad w' \neq x, \qquad \text{and } z \neq y,$$

  so this term is equal to

  $$\lambda w'.\, (\mathrm{ren}^z_x\, \mathrm{ren}^{w'}_y\, u)[\mathrm{ren}^z_x\, a/z]$$

  Again we may invoke the previous part to invoke that this term is α-equivalent to

  $$\lambda w.\, (\mathrm{ren}^z_x\, \mathrm{ren}^{[}_y\, w[u/)]\mathrm{ren}^z_x\, az$$

  as required.

scApp If t is an application, say rs, then

$$\begin{aligned}
\operatorname{ren}^z_x(rs[a/x]) &= \operatorname{ren}^z_x((r[a/x])(s[a/x])) && \text{scApp[ ]}\\
&= (\operatorname{ren}^z_x r[a/x])(\operatorname{ren}^z_x s[a/x]) && \text{scAppren}\\
&\sim_\alpha (\operatorname{ren}^z_x r[\operatorname{ren}^z_x a/\operatorname{ren}^z_x x])(\operatorname{ren}^z_x s[\operatorname{ren}^z_x a/\operatorname{ren}^z_x x]) && \text{scApp}\alpha\text{, ind hyp}\\
&= (\operatorname{ren}^z_x r)(\operatorname{ren}^z_x s)[\operatorname{ren}^z_x a/\operatorname{ren}^z_x x] && \text{scApp[ ]}\\
&= \operatorname{ren}^z_x(rs)[\operatorname{ren}^z_x a/\operatorname{ren}^z_x x] && \text{scAppren}
\end{aligned}$$

We look at the case $x \neq y$. We perform another proof by induction where the induction hypothesis is that the statement holds for proper subterms where a variable may have been renamed. We note that in this case we know that in this case $\operatorname{ren}^z_y x = x$ and we use this fact freely below.

bcVar If t is a variable it might be equal to other variables, namely x or y, so we have several cases to consider.
If $t = x$ then

$$\begin{aligned}
\operatorname{ren}^z_y(x[a/x]) &= \operatorname{ren}^z_y a && \text{bcVar[ ]}\\
&= x[\operatorname{ren}^z_y a/x] && \text{bcVar[ ]}\\
&= (\operatorname{ren}^z_y x)[\operatorname{ren}^z_y a/\operatorname{ren}^z_y x] && \text{bcVarren, } x \neq y
\end{aligned}$$

If $t = y$ then

$$\begin{aligned}
\operatorname{ren}^z_y(y[a/x]) &= \operatorname{ren}^z_y y && \text{bcVar[ ]}\\
&= z && \text{bcVarren}\\
&= z[\operatorname{ren}^z_y a/x] && \text{bcVar[ ]}\\
&= \operatorname{ren}^z_y y[\operatorname{ren}^z_y a/x] && \text{bcVarren}
\end{aligned}$$

If $t = y'$ is a variable different from both x and y then

$$\begin{aligned}
\operatorname{ren}^z_y(y'[a/x]) &= \operatorname{ren}^z_y y' && \text{bcVar[ ], } y' \neq x\\
&= y' && \text{bcVarren, } y' \neq y\\
&= y'[\operatorname{ren}^z_y a/x] && \text{bcVar[ ], } x \neq y'\\
&= (\operatorname{ren}^z_y y')[\operatorname{ren}^z_x a/x] && \text{bcVar[ ], } y \neq y'
\end{aligned}$$

scAbs If t is a λ-abstraction, say $\lambda y.\, u$, we may calculate the two terms as follows. For

$$w = \operatorname{freshv}(\operatorname{vars}(\lambda y'.\, u) \cup \operatorname{vars} a \cup \{x\})$$

we have

$$\begin{aligned}
\operatorname{ren}^z_y((\lambda y'.\, u)[a/x]) &= \operatorname{ren}^z_y(\lambda w.\, (\operatorname{ren}^w_{y'} u)[a/x]) && \text{scAbs[ ]}\\
&= \lambda(\operatorname{ren}^z_y w).\ \operatorname{ren}^z_y((\operatorname{ren}^w_{y'} u)[a/x]) && \text{scAbsren}\\
&\sim_\alpha \lambda(\operatorname{ren}^z_y w).\, (\operatorname{ren}^z_y \operatorname{ren}^w_{y'} u)[\operatorname{ren}^z_y a/\operatorname{ren}^z_y x] && \text{ind hyp and Prop 1.7}\\
&= \lambda(\operatorname{ren}^z_y w).\, (\operatorname{ren}^z_y \operatorname{ren}^w_{y'} u)[\operatorname{ren}^z_y a/x] && y \neq x.
\end{aligned}$$

For our second term, for

$$w' = \operatorname{freshv}(\operatorname{vars}(\lambda(\operatorname{ren}^z_y y').\ \operatorname{ren}^z_y u) \cup \operatorname{vars}(\operatorname{ren}^z_y a) \cup \{x\})$$

we have

$$\begin{aligned}(\mathrm{ren}^z_y(\lambda y'.\,u))[\mathrm{ren}^z_y\,a/x] &= (\lambda(\mathrm{ren}^z_y\,y').\ \mathrm{ren}^z_y\,u)[\mathrm{ren}^z_y\,a/x] && \text{scAbsren}\\ &= \lambda w'.\,(\mathrm{ren}^{w'}_{\mathrm{ren}^z_y\,y'}\,\mathrm{ren}^z_y\,u)[\mathrm{ren}^z_y\,a/x] && \text{scAbs[ ]}\end{aligned}$$

We make some case distinctions to establish that these two terms are α-equivalent.

- If $y = y'$ then y cannot be equal to w, and so $\mathrm{ren}^z_y\,w = w$ and also in our first term, having renamed $y' = y$ to w, the second renaming operation, of y to z, has no effect and so this term is equal to

  $$\lambda w.\,(\mathrm{ren}^w_y\,u)[\mathrm{ren}^z_y\,a/x],$$

  while for the second term we note that in this situation $\mathrm{ren}^z_y\,y' = z$, so this term is

  $$\lambda w'.\,(\mathrm{ren}^{w'}_z\,\mathrm{ren}^z_y\,u)[\mathrm{ren}^z_y\,a/x] = \lambda w'.\,(\mathrm{ren}^{w'}_y\,u)[\mathrm{ren}^z_y\,a/x].$$

  These two terms are α-equivalent by the previous part.
- If $y \neq y'$ and $w = y$ then y cannot appear in $\lambda y'.\,u$, and the first term is equal to

  $$\lambda z.\,(\mathrm{ren}^z_w\,\mathrm{ren}^w_{y'}\,u)[\mathrm{ren}^a_y\,/x] = \lambda z.\,(\mathrm{ren}^z_{y'}\,u)[\mathrm{ren}^z_y\,a/x],$$

  while in the second term renaming y to z in u has no effect and this term is equal to

  $$\lambda w'.\,(\mathrm{ren}^{w'}_{y'}\,u)[\mathrm{ren}^z_y\,a/x],$$

  and again the two terms are α-equivalent by the previous part.
- If $y \neq y'$ and $y \neq w$ by Proposition 1.2 we may swap the two inner renaming operations in our first term and it is equal to

  $$\lambda w.\,(\mathrm{ren}^z_y\,\mathrm{ren}^w_{y'}\,u)[\mathrm{ren}^z_y\,a/x] = \lambda w.\,(\mathrm{ren}^w_{y'}\,\mathrm{ren}^z_y\,u)[\mathrm{ren}^x_y\,a/x],$$

  while for the second term we note that in this case $\mathrm{ren}^z_y\,y = y'$ and that it is equal to

  $$\lambda w'.\,(\mathrm{ren}^{w'}_{y'}\,\mathrm{ren}^z_y\,u)[\mathrm{ren}^z_y\,a/x],$$

  and again we may invoke the previous part to deduce that the two are α-equivalent.

scApp If t is an application, say rs, then

$$\begin{aligned}\mathrm{ren}^z_y((rs)[a/x]) &= \mathrm{ren}^z_y((r[a/x])(s[a/x])) && \text{scApp[ ]}\\ &= (\mathrm{ren}^z_y\,r[a/x])(\mathrm{ren}^z_y\,s[a/x]) && \text{scAppren}\\ &= ((\mathrm{ren}^z_y\,r)[\mathrm{ren}^z_y\,a/x])((\mathrm{ren}^z_y\,s)[\mathrm{ren}^z_y\,a/x]) && \text{ind hyp}\\ &= ((\mathrm{ren}^z_y\,r)(\mathrm{ren}^z_y)s)[\mathrm{ren}^z_y\,a/x] && \text{scApp[ ]}\\ &= (\mathrm{ren}^z_y(rs))[\mathrm{ren}^z_y\,a/x] && \text{scAppren}\end{aligned}$$

(f) We prove the statement by induction on the structure of the term.

bcVar If t is a variable, say y, then we have two cases to consider. If $y = x$ then

$$\mathrm{ren}^z_x\,y = \mathrm{ren}^z_x\,x = z,$$

and then

$$\begin{aligned}(\mathrm{ren}^z_x\, x)[a/z] &= z[a/z] && \text{above}\\ &= a && \text{bcVar[ ]}\\ &= x[a/x] && \text{bcVar[ ]}\\ &= y[a/x] && x = y,\end{aligned}$$

and if $y \neq x$ then since $y \in \mathrm{vars}\, y$ we know that $z \neq y$, so

$$\begin{aligned}(\mathrm{ren}^z_x\, y)[a/z] &= y[a/z] && \text{bcVarren}\\ &= y && \text{bcVar[ ]},\ x \neq z\\ &= y[a/x] && \text{bcVar[ ]},\ x \neq y\end{aligned}$$

scAbs If t is a λ-abstraction, say $\lambda y.\, u$, then for

$$w = \mathrm{freshv}(\mathrm{vars}(\lambda y.\ \mathrm{ren}^z_x\, u) \cup \mathrm{vars}\, a \cup \{z\})$$

we have

$$\begin{aligned}(\mathrm{ren}^z_x(\lambda y.\, u))[a/z] &= (\lambda(\mathrm{ren}^z_x\, y).\, (\mathrm{ren}^z_x\, u))[a/z] && \text{scAbsren}\\ &= (\lambda(\mathrm{ren}^z_x\, y).\, (\mathrm{ren}^z_x\, u))[a/z] && \text{bcVarren}\\ &= \lambda w.\, (\mathrm{ren}^w_{\mathrm{ren}^z_x\, y}\, \mathrm{ren}^z_x\, u)[a/z] && \text{scAbsren}\end{aligned}$$

and for

$$w' = \mathrm{freshv}(\mathrm{vars}(\lambda y.\, u) \cup \mathrm{vars}\, a \cup \{x\})$$

we obtain

$$(\lambda y.\, u)[a/x] = \lambda w'.\, (\mathrm{ren}^{w'}_y\, u)[a/x].$$

To show that these two terms are α-equivalent we have to show that for a suitably fresh variable z' we have

$$\mathrm{ren}^{z'}_w((\mathrm{ren}^w_{\mathrm{ren}^z_x\, y}\, \mathrm{ren}^z_x\, u)[a/z]) \sim_\alpha \mathrm{ren}^{z'}_{w'}((\mathrm{ren}^{w'}_y\, u)[a/x]).$$

We know from a previous part that the term on the left hand side may be simplified to an α-equivalent one since

$$\mathrm{ren}^{z'}_w((\mathrm{ren}^w_{\mathrm{ren}^z_x\, y}\, \mathrm{ren}^z_x\, u)[a/z]) \sim_\alpha (\mathrm{ren}^{z'}_{\mathrm{ren}^z_x\, y}\, \mathrm{ren}^z_x\, u)[a/z],$$

and similarly for the term on the right hand side we have

$$\mathrm{ren}^{z'}_{w'}((\mathrm{ren}^{w'}_y\, u)[a/x]) \sim_\alpha (\mathrm{ren}^{z'}_y\, u)[a/x].$$

Our aim is to show that the two simplified terms are α-equivalent. In order to apply the induction hypothesis we need to swap the order of the two renaming operations, so that we may apply it to the term $\mathrm{ren}^{z'}_y$, but we may only do so if we have distinctness of a number of variables involved, see Proposition 1.2. We therefore make some case distinctions to cover the other cases.

- If $y = x$ then $\mathrm{ren}_x^z\, y = \mathrm{ren}_x^z\, x = z$ and so the term on the left is equal to

$$(\mathrm{ren}_z^{z'}\, \mathrm{ren}_x^z\, u)[a/z] = (\mathrm{ren}_x^{z'}\, u)[a/z],$$

  while the term on the right is

$$(\mathrm{ren}_x^{z'})[a/x].$$

  Now we have two substitutions for a variable that we know does not occur in the term we are substituting into: In the first term, z is known not to occur since by assumption it is fresh for t, while in the second, all occurrences of x have been removed. Hence we know by part (b) that

$$(\mathrm{ren}_x^{z'}\, u)[a/z] \sim_\alpha \mathrm{ren}_x^{z'}\, u \sim_\alpha (\mathrm{ren}_x^{z'})[a/x]$$

  as required.
- If $y \neq x$ we might still have $x = z$ since we have not ruled that out. But then the first renaming operation in the first term is ren_x^x and so it has no effect, and in both cases we are substituting for the same variable $x = z$, and so both terms are equal to

$$(\mathrm{ren}_y^{z'}\, u)[a/x],$$

  and so are α-equivalent as required.
- If $y \neq x$ then $\mathrm{ren}_x^z\, y = y$ and if also $x \neq z$ we now know that

$$y \neq x, \qquad y \neq z \qquad \text{and } x \neq z'$$

  which allows us to apply Proposition 1.2 to deduce that for the first term we have

$$(\mathrm{ren}_y^{z'}\, \mathrm{ren}_x^z\, u)[a/z] = (\mathrm{ren}_x^z\, \mathrm{ren}_y^{z'}\, u)[a/z].$$

  Applying the induction hypothesis to $\mathrm{ren}_y^{z'}\, u$ tells us that

$$(\mathrm{ren}_x^z\, \mathrm{ren}_y^{z'}\, u)[a/z] \sim_\alpha \mathrm{ren}_y^{z'}\, u)[a/x]$$

  as required.

scApp If t is an application, say rs, then

$$\begin{aligned}
(\mathrm{ren}_x^z(rs))[a/z] &= ((\mathrm{ren}_x^z\, r)(\mathrm{ren}_x^z\, s))[a/z] && \text{scAppren}\\
&= ((\mathrm{ren}_x^z\, r)[a/z])((\mathrm{ren}_x^z\, s)[a/z]) && \text{scApp[ ]}\\
&\sim_\alpha (r[a/x])(s[a/x]) && \text{ind hyp and scApp}\alpha\\
&= (rs)[a/x] && \text{scApp[ ]}
\end{aligned}$$

Proposition 1.12

Proof. We show each statement.

(a) This is a proof by induction over the structure of the term t. Again we need the induction hypothesis to hold for subterms, were one variable may have been changed via renaming.

bcVar If t is a variable, say y, we have two cases to consider.

- If $y = x$ then

$$\begin{aligned} x[(b[a/x])/x] &= b[a/x] && \text{bcVar[ ]} \\ &= (x[b/x])[a/x] && \text{bcVar[ ]} \end{aligned}$$

- If $y \neq x$ then

$$\begin{aligned} y[(b[a/x])/x] &= y && \text{bcVar[ ]} \\ &= y[a/x] && \text{bcVar[ ]} \\ &= (x[b/x])[a/x] && \text{bcVar[ ]} \end{aligned}$$

scAbs If t is a λ-abstraction, say $\lambda y.\, u$, then for

$$w = \mathrm{freshv}(\mathrm{vars}\ \lambda y.\, u \cup \mathrm{vars}(b[a/x]) \cup \{x\})$$

we have

$$(\lambda y.\, u)[(b[a/x])/x] = \lambda w.\, (\mathrm{ren}_y^w\, u)[b[a/x]/x] \qquad \text{scAbs[ ]}$$

and for

$$w' = \mathrm{freshv}(\mathrm{vars}\ \lambda y.\, u \cup \mathrm{vars}\ b \cup \{x\})$$

and

$$w'' = \mathrm{freshv}(\mathrm{vars}(\lambda w'.\, (\mathrm{ren}_y^w\, u)[b/x]) \cup \mathrm{vars}\ a \cup \{x\})$$

we obtain

$$\begin{aligned} ((\lambda y.\, u)[b/x])[a/x] &= (\lambda w'.\, ((\mathrm{ren}_y^{w'})[b/x]))[a/x] && \text{scApp[ ]} \\ &= \lambda w''.\, (\mathrm{ren}_{w'}^{w''}((\mathrm{ren}_y^{w'}\, u)[b/x]))[a/x] && \text{scApp[ ].} \end{aligned}$$

We know from Proposition 1.10 that

$$\mathrm{ren}_{w'}^{w''}((\mathrm{ren}_y^{w'}\, u)[b/x]) \sim_\alpha (\mathrm{ren}_y^{w''}\, u)[b/x]$$

and so

$$\begin{aligned} (\mathrm{ren}_{w'}^{w''}((\mathrm{ren}_y^{w'}\, u)[b/x]))[a/x] &\sim_\alpha ((\mathrm{ren}_y^{w''}\, u)[b/x])[a/x] && \text{Proposition 1.11} \\ &\sim_\alpha (\mathrm{ren}_y^{w''}\, u)[b[a/x]/x] && \text{ind hyp,} \end{aligned}$$

and sinceProposition 1.7 allow us to abstract the same variable from two α-equivalent terms to get α-equivalent abstractions, this means that our second term is α-equivalent to

$$\lambda w''.\, (\mathrm{ren}_y^{w''}\, u)[b[a/x]/x].$$

We may now invoke Proposition 1.11 to argue that our two terms are α-equivalent since they merely differ by the name of the (fresh) variable which is abstracted after a suitable renaming has taken place.

scApp If t is an application, say rs, then

$$\begin{aligned} (rs)[(b[a/x])/x] &= r[(b[a/x])/x]s[(b[a/x])/x] && \text{scApp[ ]} \\ &\sim_\alpha ((r[b/x])[a/x])((s[b/x])[a/x]) && \text{ind hyp and scApp}\alpha \\ &= ((r[b/x])(s[b/x]))[a/x] && \text{scApp[ ]} \\ &= ((rs)[b/x])[a/x] && \text{scApp[ ]} \end{aligned}$$

(b) This is a proof by induction over the structure of the term t.

bcVar If t is a variable, say z, we have three cases to consider.

- If $z = y$ then we note that z can't also be equal to x since x and y are distinct.

$$\begin{aligned}(y[a/x])[(b[a/x])/y] &= y[(b[a/x])/y] && \text{bcVar}[\,]\\ &= b[a/x] && \text{bcVar}[\,]\\ &= (y[b/y])[a/x] && \text{bcVar}[\,]\end{aligned}$$

- If $z \neq y$ then we have to worry about $z = x$. If that is the case then

$$\begin{aligned}(x[a/x])[(b[a/x])/y] &= a[(b[a/x])/y] && \text{bcVar}[\,]\\ &\sim_\alpha a && y \notin \text{fv}\, a,\ \text{Proposition 1.10}\\ &= x[a/x] && \text{bcVar}[\,]\\ &= (x[b/y])[a/x] && \text{bcVar}[\,], y \neq x\end{aligned}$$

- If $z \neq y$ and $z \neq x$ then

$$\begin{aligned}(z[a/x])[(b[a/x])/y] &= z[(b[a/x])/y] && \text{bcVar}[\,], z \neq x\\ &= z && \text{bcVar}[\,],\ z \neq y\\ &= z[a/x] && \text{bcVar}[\,]\\ &= (z[b/y])[a/x] && \text{bcVar}[\,],\ z \neq y\end{aligned}$$

scAbs If t is a λ-abstraction, say $\lambda z.\, u$ we may calculate, for

$$w = \text{freshv}(\text{vars}\, \lambda z.\, u \cup \text{vars}\, a \cup \{v\})$$

and

$$w' = \text{freshv}(\text{vars}(\lambda w.\, (\text{ren}_z^w\, u)[a/x]) \cup \text{vars}(b[a/x]) \cup \{y\})$$

that

$$\begin{aligned}((\lambda z.\, u)[a/x])[b[a/x]/y] &= (\lambda w.\, (\text{ren}_z^w\, u)[a/x])[b[a/x]/y] && \text{scAbs}[\,]\\ &= \lambda w'.\, (\text{ren}_w^{w'}((\text{ren}_z^w\, u)[a/x]))[b[a/x]/y] && \text{scAbs}[\,].\end{aligned}$$

We know from Proposition 1.10 that

$$\text{ren}_w^{w'}((\text{ren}_z^w\, u)[a/x]) \sim_\alpha (\text{ren}_z^{w'}\, u)[a/x]$$

and so

$$\begin{aligned}(\text{ren}_w^{w'}((\text{ren}_z^w\, u)[a/x]))[b[a/x]/y] &\sim_\alpha ((\text{ren}_z^{w'}\, u)[a/x])[b[a/x]/y] && \text{Proposition 1.11}\\ &\sim_\alpha ((\text{ren}_z^{w'}\, u)[b/y])[a/x] && \text{ind hyp.}\end{aligned}$$

Looking at the other term, for

$$w'' = \text{freshv}(\text{vars}(\lambda z.\, u) \cup \text{vars}\, b \cup \{y\})$$

and

$$w''' = \text{freshv}(\text{vars}() \cup \text{vars}\, a \cup \{x\})$$

we get

$$((\lambda z.\, u)[b/y])[a/x] = (\lambda w''.\, (\text{ren}_z^{w''}\, u)[b/y])[a/x] \qquad \text{scAbs}[\,]$$

$$= \lambda w'''.((\mathrm{ren}^{w'''}_{w''}((\mathrm{ren}^{w''}_{z}\ u)[b/y])))[a/x] \qquad \text{scAbs}[\,],$$

and we may argue similarly by invoking Proposition 1.10 that

$$\mathrm{ren}^{w''}_{w''}((\mathrm{ren}^{w'}_{z}\ u)[b/y]) \sim_\alpha (\mathrm{ren}^{w'''}_{z}\ u)[a/x]$$

and so by Proposition 1.11

$$((\mathrm{ren}^{w'''}_{w''}((\mathrm{ren}^{w''}_{z}\ u)[b/y])))[a/x] \sim_\alpha ((\mathrm{ren}^{w''''}_{z}\ u)[b/y])[a/x]$$

We know from Proposition 1.7 that if we have two α-equivalent terms and abstract the same variable, we get two terms that are also α-equivalent. Hence we know that the two terms of interest are respectively α-equivalent to

$$\lambda w'.((\mathrm{ren}^{w'}_{z}\ u)[b/y])[a/x] \qquad \text{and} \qquad \lambda w'''.((\mathrm{ren}^{w''''}_{z}\ u)[b/y])[a/x].$$

We may now invoke Proposition 1.11 to argue that these terms only differ by the fresh variable they abstract after a suitable renaming has been carried out, and so they are α-equivalent.

scApp If t is an application, say rs then

$$\begin{aligned}
(t[a/x])[b[a/x]/y] &= (rs[a/x])[b[a/x]/y] && t = rs\\
&= ((r[a/x])(s[a/x]))17[b[a/x]/y] && \text{scApp}[\,]\\
&= ((r[a/x])[b[a/x]/y])((s[a/x])[b[a/x]/y]) && \text{scApp}[\,]\\
&\sim_\alpha ((r[b/y])[a/x])((s[b/y])[a/x]) && \text{ind hyp}\\
&= ((r[b/y])(s[b/y]))[a/x] && \text{scApp}[\,]\\
&= (rs)[b/y][a/x] && \text{scApp}[\,]
\end{aligned}$$

Proposition 1.14

Proof. We carry out induction over the structure of t. We know that t β-reduces, which tells us something about its shape, and we require that information in order to carry out our proof. We refer to this kind of induction proof as *induction over the reason that t β-reduces*, and adjust the structure accordingly, using labels that remind us of this. We do not explicitly mention in these proofs, for example, that we know that t cannot be a variable since variables do not reduce. This structure is only possible if we know that for term in question we may carry out at least one β-reduction step.

The induction hypothesis is that the claim holds for proper subterms where at most one variable has been renamed.

bcVarβ If we are in the base case of β-reduction then t is of the form $(\lambda y.u)a$ and

$$t = (\lambda y.u)a \xrightarrow{\beta} u[a/y] = t'.$$

We may now calculate as follows.

$$\begin{aligned}
\mathrm{ren}^z_x\ t &= \mathrm{ren}^z_x((\lambda y.u)a) && t = (\lambda y.u)a\\
&= \mathrm{ren}^z_x(\lambda y.u)(\mathrm{ren}^z_x\ a) && \text{scAppren}\\
&= (\lambda(\mathrm{ren}^z_x\ y).(\mathrm{ren}^z_x\ u))(\mathrm{ren}^z_x\ a) && \text{scAbsren}\\
&\xrightarrow{\beta} \mathrm{ren}^z_x\ u[\mathrm{ren}^z_x\ a/\ \mathrm{ren}^z_x\ y] && \text{bcVar}\beta\\
&\sim_\alpha \mathrm{ren}^z_x(u[a/y]) && \text{Proposition 1.10}\\
&= \mathrm{ren}^z_x\ t' && t' = u[a/y]
\end{aligned}$$

scAbsβ If t is a λ-abstraction, say $\lambda y.\, u$, and $u \xrightarrow{\beta} u'$ is the reason that $t = \lambda y.\, u \xrightarrow{\beta} \lambda y.\, u' = t'$. We use the following instance of the induction hypothesis below.

$$u \xrightarrow{\beta} u' \qquad \text{implies} \qquad \exists s.\ \mathrm{ren}^z_x\, u \xrightarrow{\beta} s \sim_\alpha \mathrm{ren}^z_x\, u'.$$

$$\begin{aligned}
\mathrm{ren}^z_x\, t &= \mathrm{ren}^z_x(\lambda y.\, u) && t = \lambda y.\, u \\
&= \lambda(\mathrm{ren}^z_x\, y).\, (\mathrm{ren}^z_x\, u) && \text{scAbsren} \\
&\xrightarrow{\beta} \lambda(\mathrm{ren}^z_x\, y).\, s && \text{scAbs}\beta \\
&\sim_\alpha \lambda(\mathrm{ren}^z_x\, y).\ \mathrm{ren}^z_x\, u' && \text{ind hyp and Proposition 1.7} \\
&= \mathrm{ren}^z_x(\lambda y.\, u') && \text{scAbsren} \\
&= \mathrm{ren}^z_x\, t' && t' = \lambda y.\, u'
\end{aligned}$$

scAppβ If t is an application, say $t = rs$, and $r \xrightarrow{\beta} r'$ is the reason that t reduces to t', that is $t = rs \xrightarrow{\beta} r's = t'$, then by the induction hypothesis we know we may find a term u with $\mathrm{ren}^z_x\, r \xrightarrow{\beta} u \sim_\alpha \mathrm{ren}^z_x\, r'$ and so we have the following.

$$\begin{aligned}
\mathrm{ren}^z_x\, t &= \mathrm{ren}^z_x(rs) && t = rs \\
&= (\mathrm{ren}^z_x\, r)(\mathrm{ren}^z_x\, s) && \text{scAppren} \\
&\xrightarrow{\beta} u(\mathrm{ren}^z_x\, s) && \text{scApp}\beta \\
&\sim_\alpha (\mathrm{ren}^z_x\, r')(\mathrm{ren}^z_x\, s) && \text{ind hyp and scApp}\alpha \\
&= \mathrm{ren}^z_x(r's) && \text{scAppren} \\
&= \mathrm{ren}^z_x\, t' && t' = r's.
\end{aligned}$$

The case where the reduction takes place in the second term of the application is much the same.

Proposition 1.15

Proof. We show each statement.

(a) Assume we have two λ-terms t and t' with $t \xrightarrow{\beta} t'$. We prove the result by induction.

bcVarβ If t reduces to t' because of bcVarβ, then t is of the form $(\lambda y.\, u)a$ and we reduce to $u[a/y] = t'$. We have that

$$\begin{aligned}
\mathrm{fv}\, u[a/y] &\subseteq (\mathrm{fv}\, u \setminus \{y\}) \cup \mathrm{fv}\, a && \text{Proposition 1.10} \\
&= \mathrm{fv}(\lambda y.\, u) \cup \mathrm{fv}\, a && \text{scAbsfv} \\
&= \mathrm{fv}((\lambda y.\, u)a) && \text{scAppfv} \\
&= \mathrm{fv}\, t && t = (\lambda y.\, u)a
\end{aligned}$$

scAbsβ Assume we have $t \xrightarrow{\beta} t'$ because $u \xrightarrow{\beta} u'$ and $t = \lambda y.\, u$ while $t' = \lambda y.\, u'$. In this situation we get the following:

$$\begin{aligned}
\mathrm{fv}\, t' &= \mathrm{fv}(\lambda y.\, u') && t' = \lambda y.\, u' \\
&= \mathrm{fv}\, u' \setminus \{y\} && \text{scAbsfv} \\
&\subseteq \mathrm{fv}\, u \setminus \{y\} && \text{Ind hyp} \\
&= \mathrm{fv}(\lambda y.\, u) && \text{scAbsfv} \\
&= \mathrm{fv}\, t && t = \lambda y.\, u
\end{aligned}$$

scAppβ Assume that t is of the form rs, that $r \xrightarrow{\beta} r'$ and $t \xrightarrow{\beta} t'$ because of that so $t' = r's$.

$$\begin{aligned}
\operatorname{fv} t' &= \operatorname{fv}(r's) && t' = r's \\
&= \operatorname{fv} r' \cup \operatorname{fv} s && \text{scAppfv} \\
&\subseteq \operatorname{fv} r \cup \operatorname{fv} s && \text{ind hyp} \\
&= \operatorname{fv}(rs) && \text{scAppfv} \\
&= \operatorname{fv} t
\end{aligned}$$

The case where we have a reduction in the second part of an application is essentially the same.

(b) Assume that we have λ-terms t and t' with $t \xrightarrow{\beta} t'$, and let a be another λ-term. We want to show that $t[a/x] \xrightarrow{\beta} t'[a/x]$. This is another case where the induction hypothesis holds for proper subterms where one variable may have been renamed.

bcVarβ If $t = (\lambda y.\, u)b$, and $t' = u[b/y]$ then we are interested in the term

$$t[a/x] = ((\lambda y.\, u)b)[a/x] = ((\lambda y.\, u)[a/x])(b[a/x]),$$

For $w = \operatorname{freshv}(\operatorname{vars}(\lambda y.\, u) \cup \operatorname{vars} a \cup \{x\})$ we get

$$\begin{aligned}
&((\lambda y.\, u)[a/x])(b[a/x]) \\
&\quad = (\lambda w.\, (\operatorname{ren}_y^w u)[a/x])(b[a/x]) && \text{scAbs[ ]} \\
&\quad \xrightarrow{\beta} ((\operatorname{ren}_y^w u)[a/x])[b[a/x]/y] && \text{bcVar}\beta \\
&\quad \sim_\alpha (u[a/x])[b[a/x]/y] && \text{Proposition 1.10} \\
&\quad = (u[b/y])[a/x] && \text{Proposition 1.12} \\
&\quad = t'[a/x] && t' = u[b/y]
\end{aligned}$$

scAbsβ If $t \xrightarrow{\beta} t'$ because of scAbsβ, that is, t is a λ-abstraction, say $t = \lambda y.\, u$, and $u \xrightarrow{\beta} u'$ and $t' = \lambda y.\, u'$ then for $w = \operatorname{freshv}(\operatorname{vars}(\lambda y.\, u) \cup \operatorname{vars} a \cup \{x\})$ we have

$$\begin{aligned}
t[a/x] &= (\lambda y.\, u)[a/x] && t = \lambda y.\, u \\
&= \lambda w.\, (\operatorname{ren}_y^w u)[a/x] && \text{scAbs[ ]} \\
&\xrightarrow{\beta} \lambda w.\, (\operatorname{ren}_y^w u')[a/x] && \text{scAbs}\beta \text{ and ind hyp} \\
&\sim_\alpha t'[a/x],
\end{aligned}$$

where we use $t' \sim_\alpha \lambda w.\, (\operatorname{ren}_y^w u')$ by Proposition 1.7.

scAppβ If $t \xrightarrow{\beta} t'$ because of scAppβ, that is t is an application, say rs, and $r \xrightarrow{\beta} r'$ with $t' = r's$ then

$$\begin{aligned}
t[a/x] &= (rs)[a/x] && t = rs \\
&= (r[a/x])(s[a/x]) && \text{scApp[ ]} \\
&\xrightarrow{\beta} (r'[a/x])(s[a/x]) && \text{scApp}\beta \text{ and ind hyp} \\
&= (r's)[a/x] && \text{scApp[ ]} \\
&= t'[a/x] && t' = r's.
\end{aligned}$$

The case where the β-reduction takes place in the second term of the application is much the same.

(c) Assume we have $a \xrightarrow{\beta} a'$. We want to show that for every λ-term t we have $t[a/x]$ β-reduces to a term that is α-equivalent to $t[a'/x]$. This is another proof by induction, where again we assume that the induction hypothesis holds for proper subterms where one variable may have been renamed.

bcVarβ Assume that t is a variable, say y. There are two cases to consider.
If $y = x$ then $y[a/x] = a$ and by assumption $a \xrightarrow{\beta} a' = y[a'/x]$.
If $y \neq x$ then $y[a/x] = y \xrightarrow{\beta}{\!\!\!\!\rightarrow} y = y[a'/x]$ since $\xrightarrow{\beta}{\!\!\!\!\rightarrow}$ is reflexive.

scAbsβ Assume that t is a λ-abstraction, say $\lambda y.\, u$. For

$$w = \mathrm{freshv}(\mathrm{vars}\, a \cup \mathrm{vars}(\lambda y.\, u) \cup \{x\})$$

we have

$$(\lambda y.\, u)[a/x] = \lambda w.\, (\mathrm{ren}_y^w\, u)[a/x]$$

by scAbs[ ]. By the induction hypothesis we may find a term u' with

$$(\mathrm{ren}_y^w\, u)[a/x] \xrightarrow{\beta}{\!\!\!\!\rightarrow} u' \sim_\alpha (\mathrm{ren}_y^w\, u)[a'/x].$$

We obtain

$$\lambda w.\, u' \sim_\alpha \lambda w.\, (\mathrm{ren}_y^w\, u)[a/x]$$

by Proposition 1.7, and by scAbsβ the required

$$\lambda w.\, (\mathrm{ren}_y^w\, u)[a/x] \xrightarrow{\beta}{\!\!\!\!\rightarrow} \lambda w.\, u' \sim_\alpha \lambda w.\, (\mathrm{ren}_y^w\, u)[a/x].$$

scAppβ Assume that t is an application, say rs. Then

$$\begin{aligned}
t[a/x] &= (rs)[a/x] && t = rs \\
&= (r[a/x])(s[a/x]) && \text{scApp[ ]} \\
&\xrightarrow{\beta} r'(s[a/x]) && \text{scApp}\beta \text{ and ind hyp} \\
&\xrightarrow{\beta} r's' && \text{scApp}\beta \text{ and ind hyp,}
\end{aligned}$$

where $r' \sim_\alpha r[a'/x]$ and $s' \sim_\alpha s[a'/x]$ are the terms we are able to find due to the induction hypothesis, and the final term is α-equivalent to

$$(r[a'/x])(s[a'/x])$$

by scAppα.

Proposition 1.16

Solution. This is a proof by induction over the the term t. Note that this another proof where we require an induction hypothesis which is stronger than merely demanding that we are building a bigger term from terms that satisfy the property, but we need to assume it for terms of the same shape as the terms we are building from, but some of the variables may have different names.

bcVarβ If t is of the form $(\lambda x.\, r)a$ and we have that

$$t = (\lambda x.\, r)a \xrightarrow{\beta}{\!\!\!\!\rightarrow} r[a/x] = t',$$

we also know that u, being α-equivalent to t, must be of the form $(\lambda y.\, s)b$, and we know that

$$\lambda x.\, r \sim_\alpha \lambda y.\, s \qquad \text{and} \qquad a \sim_\alpha b$$

as well as

$$u = (\lambda y.\, s)b \xrightarrow{\beta} s[b/y].$$

The first of these tells us that for $z \notin \mathrm{vars}(xr) \cup \mathrm{vars}(ys)$ we have

$$\mathrm{ren}^z_x\, r \sim_\alpha \mathrm{ren}^z_y\, s.$$

We may now use previously established results to argue that

$$\begin{aligned}
u &\xrightarrow{\beta} s[b/y] && \text{above}\\
&\sim_\alpha (\mathrm{ren}^z_y\, s)[b/z] && \text{Proposition 1.10}\\
&\sim_\alpha (\mathrm{ren}^z_x\, r)[a/z] && \text{Proposition 1.10}\\
&\sim_\alpha r[a/y] && \text{Proposition 1.10}\\
&= t' && \text{above.}
\end{aligned}$$

as required.

scAbsβ In this case t is of the form $\lambda x.\, r$, and the reason that t reduces is because r reduces to some term r', and so

$$t = \lambda x.\, r \xrightarrow{\beta} \lambda x.\, r' = t'.$$

We further know that being α-equivalent to t, the term u must be of the form $\lambda y.\, s$ such that, for all variables z with $z \notin \mathrm{vars}(\lambda x.\, r) \cup \mathrm{vars}(\lambda y.\, s)$ we have

$$\mathrm{ren}^z_x\, r \sim_\alpha \mathrm{ren}^s_y\, s.$$

By Proposition 1.14 we know that there exists a term r'' with

$$\mathrm{ren}^z_x\, r \xrightarrow{\beta} r'' \sim_\alpha \mathrm{ren}^z_x\, r',$$

and by the induction hypothesis and transitivity of $\sim_\alpha$ we may find a term s' with

$$\mathrm{ren}^z_y\, s \xrightarrow{\beta} s' \sim_\alpha \mathrm{ren}^z_x\, r'.$$

We further have from Proposition 1.14 that we may find a term s'' with

$$\mathrm{ren}^y_z\, \mathrm{ren}^z_y\, s \xrightarrow{\beta} s'' \sim_\alpha \mathrm{ren}^y_z\, s',$$

and so

$$\begin{aligned}
s &= \mathrm{ren}^y_z\, \mathrm{ren}^z_y\, s && \text{Proposition 1.2}\\
&\xrightarrow{\beta} s''\\
&\sim_\alpha \mathrm{ren}^y_z\, s' && \text{Proposition 1.14}\\
&\sim_\alpha \mathrm{ren}^z_x\, r'.
\end{aligned}$$

By Propositions 1.2 and 1.14 we also know that we may find s''' with

$$s = \mathrm{ren}^y_z\, \mathrm{ren}^z_y\, s \xrightarrow{\beta} s''' \sim_\alpha \mathrm{ren}^y_z\, s'$$

and so by scAbsβ and Proposition 1.7 we have

$$u = \lambda y.\, s \xrightarrow{\beta} \lambda y.\, s''' \sim_\alpha \lambda y.\, \mathrm{ren}^y_z\, s',$$

and it remains to establish that

$$\lambda y.\, \mathrm{ren}^y_z\, s' \sim_\alpha \lambda x.\, r'.$$

For this it is sufficient to show that there is a variable w which is fresh for $lty?\mathrm{ren}^y_z\, s', \lambda x.\, r'$ and z such that

$$\mathrm{ren}^w_y\, \mathrm{ren}^y_z\, s' \sim_\alpha \mathrm{ren}^w_x\, r$$

which we obtain from

$$\begin{aligned}
\mathrm{ren}^w_y\, \mathrm{ren}^y_z\, s' &= \mathrm{ren}^w_z\, s' && \text{Proposition 1.2}\\
&\sim_\alpha \mathrm{ren}^w_z\, \mathrm{ren}^z_x\, r' && \text{Proposition 1.7 applied to } s' \sim_\alpha \mathrm{ren}^z_x\, r'\\
&= \mathrm{ren}^w_x\, r' && \text{Proposition 1.2.}
\end{aligned}$$

scAppβ In this case t is of the form rr' and there are two symmetric cases to consider, so we only write out one of them. If t reduces to t' because we have $r \xrightarrow{\beta} r''$ and so

$$t = rr' \xrightarrow{\beta} r''r'$$

then $t \sim_\alpha u$ tells us that u must be of the form ss' with

$$r \sim_\alpha s \qquad \text{and} \qquad r' \sim_\alpha s'.$$

By the induction hypothesis we know that there is a term s'' with $s \xrightarrow{\beta} s'' \sim_\alpha r''$, and since by scApp$\alpha$ we have

$$u = ss' \xrightarrow{\beta} s''s' \sim_\alpha r''r' = t'$$

we have provided the required witness.

The second statement is a straightforward induction over the number of β-reductions involved:

In the base case we have $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ t$ in 0 steps by reflexivity of $\xrightarrow{\beta}\!\!\!\!\!\rightarrow$, and then if $t \sim_\alpha u$ we know that $u \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ u \sim_\alpha t$ gain by reflexivity.

In the step case assume that $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ t'$ in $n+1$ steps. That means we may find a term t'' so that $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ t''$ in n steps, and also $t'' \xrightarrow{\beta} t'$. If $t \sim_\alpha u$ then by the induction hypothesis there is u'' with $u \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ u'' \sim_\alpha t''$. Since $\sim_\alpha$ is symmetric we may apply the first statement to $t'' \sim_\alpha u''$ and so $t'' \xrightarrow{\beta} t$ tells us that we may find u' with $u'' \xrightarrow{\beta} u' \sim_\alpha t'$. Hence overall we obtain that $u \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ u'' \xrightarrow{\beta} u' \sim_\alpha t'$, and u' is the requires witness to show that the statement holds.

Proposition 1.19

Proof. We show each statement.

(a) This is a proof by induction over the number of steps of β-reductions.

bcNat If there are 0 steps involved then $t = t'$ and also $\lambda y.\, t = \lambda y.\, t'$.

scNat Assume we know the statement holds if we β-reduce n times. Then reducing one more time gives us

$$t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ t' \xrightarrow{\beta} t'',$$

where we carry out n reductions to get from t to t'. By the induction hypothesis we know that

$$\lambda x.\, t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ \lambda x.\, t' \qquad \text{and we have} \qquad \lambda x.\, t' \xrightarrow{\beta} \lambda x.\, t''$$

by scAbsβ, so $\lambda x.\, t \xrightarrow{\beta}\!\!\!\!\!\rightarrow\ \lambda x.\, t''$ as required.

(b) This proof can be carried out in an identical manner to the previous one, merely replacing the rule by scAppβ.

(c) This proof is the same as the previous one part from changing the term in the application to which one applies the reduction.

(d) This is another proof by induction over the number of reduction steps carried out.

bcNat If there are 0 steps involved then $t = t'$ and also $\operatorname{fv} t = \operatorname{fv} t'$.

scNat If we know that the statement holds when we reduce n times we may show that it holds for $n + 1$ reductions. Assume we have

$$t \xrightarrow{\beta}{\!\!\!\!\rightarrow} t' \xrightarrow{\beta} t'',$$

where n reductions take us from t to t'. By the induction hypothesis we get that $\operatorname{fv} t' \subseteq \operatorname{fv} t$, and by Proposition 1.15 we have

$$\operatorname{fv} t'' \subseteq \operatorname{fv} t' \subseteq \operatorname{fv} t$$

as required.

(e) Again this is a proof by induction over the number of β reductions involved. We note that from Proposition 1.15 we have that if $t \xrightarrow{\beta} t'$ and $a \xrightarrow{\beta} a'$ then we can find λ-terms u and u' with

$$t[a/v] \xrightarrow{\beta}{\!\!\!\!\rightarrow} u \sim_\alpha t'[a/v] \xrightarrow{\beta}{\!\!\!\!\rightarrow} u' \sim_\alpha t'[a'/v].$$

We further get from Proposition 1.16 that we may find u'' with $u \xrightarrow{\beta}{\!\!\!\!\rightarrow} u''$ and $u'' \sim_\alpha t'[a'/v]$, and so the required result follows from a straightforward induction over the number of β-reductions involved.

Proposition 1.21

Proof. This is a proof by induction over the structure of the term t.

bcVar If t is a variable, say y, then there are two cases.

If $y = x$ then

$$\Diamond \operatorname{ren}_x^z y = \Diamond z = z = \operatorname{ren}_x^z y = \operatorname{ren}_x^z \Diamond y$$

and similarly if $y \neq x$ then

$$\Diamond \operatorname{ren}_x^z y = \Diamond y = y = \operatorname{ren}_x^z y = \operatorname{ren}_x^z \Diamond y.$$

scAbs If t is a λ-abstraction, say $\lambda y.\, u$ then

$$\Diamond \operatorname{ren}_x^z(\lambda y.\, u) = \Diamond \lambda(\operatorname{ren}_x^z y).\, (\operatorname{ren}_x^z u) = \lambda(\operatorname{ren}_x^z y).\, (\Diamond \operatorname{ren}_x^z u),$$

while

$$\operatorname{ren}_x^z \Diamond(\lambda y.\, u) = \operatorname{ren}_x^z(\lambda y.\, \Diamond u) = \lambda(\operatorname{ren}_x^z y).\, (\operatorname{ren}_x^z \Diamond u).$$

By the induction hypothesis we have $\Diamond \operatorname{ren}_x^z u \sim_\alpha \operatorname{ren}_x^z \Diamond u$, and by Proposition 1.7 we get the required result.

scApp If t is an application, say rs, then we have to make a case distinction.

If r is a λ-abstraction, say $\lambda y.\, u$, then

$$\begin{aligned} \Diamond \operatorname{ren}_x^z((\lambda y.\, u)s) &= \Diamond((\lambda(\operatorname{ren}_x^z y).\, (\operatorname{ren}_x^z u))s) && \text{scAppren, scAbsren} \\ &= \Diamond \operatorname{ren}_x^z u[\Diamond \operatorname{ren}_x^z s / \operatorname{ren}_x^z y] && \text{scApp}\Diamond \end{aligned}$$

while

$$\begin{aligned} \mathrm{ren}^z_x\,\Diamond((\lambda y.\,u)s) &= \mathrm{ren}^z_x(\Diamond u[\Diamond s/y]) && \text{scApp}\Diamond \\ &\sim_\alpha (\mathrm{ren}^z_x\,\Diamond u)[\mathrm{ren}^z_x\,\Diamond s/\,\mathrm{ren}^z_x\,y] && \text{Proposition 1.10} \end{aligned}$$

The induction hypothesis gives us

$$\Diamond\,\mathrm{ren}^z_x\,u \sim_\alpha \mathrm{ren}^z_x\,\Diamond u \qquad \text{and} \qquad \Diamond\,\mathrm{ren}^z_x\,s \sim_\alpha \mathrm{ren}^z_x\,\Diamond s.$$

We make a case distinction. If $x \neq y$ then $\mathrm{ren}^z_x\,y = y$, and the two substitutions are for the same variable. In that case the two expressions are α-equivalent by Proposition 1.11.

If $x = y$ then $\mathrm{ren}^z_x\,y = z$ and so the first term is $\Diamond\,\mathrm{ren}^z_x\,u[\Diamond\,\mathrm{ren}^z_x\,s/z]$, which by the induction hypothesis and Proposition 1.11 is α-equivalent to $\mathrm{ren}^z_x\,\Diamond u[\mathrm{ren}^z_x\,\Diamond s/z]$, which by Proposition 1.10 is α-equivalent to $\Diamond u[\mathrm{ren}^z_x\,\Diamond s/y]$.

If r is not a λ-abstraction then

$$\begin{aligned} \Diamond\,\mathrm{ren}^z_x(rs) &= \Diamond((\mathrm{ren}^z_x\,r)(\mathrm{ren}^z_x\,s)) && \text{scAppren} \\ &= (\Diamond\,\mathrm{ren}^z_x\,r)(\Diamond\,\mathrm{ren}^z_x\,s) && \text{scApp}\Diamond \\ &\sim_\alpha (\mathrm{ren}^z_x\,\Diamond r)(\mathrm{ren}^z_x\,\Diamond s) && \text{ind hyp and scApp}\alpha \\ &= \mathrm{ren}^z_x(\Diamond r)(\Diamond s) && \text{scAppren} \\ &= \mathrm{ren}^z_x\,\Diamond(rs) && \text{scApp}\Diamond. \end{aligned}$$

Proposition 1.22

Proof. The proof is by induction over the reason that the two terms are α-equivalent.

bcVarα If t is a variable then so is t', and they must be the same variable. The parallel reduct of a variable is that variable again, so in this case $\Diamond t = \Diamond t'$.

scAbsα If t is a λ-abstraction, say $\lambda x.\,u$ then so must t' be, say $\lambda x'.\,u'$ and we further know from t being α-equivalent to t' that for for a sufficiently fresh variable z we have $\mathrm{ren}^z_x\,u \sim_\alpha \mathrm{ren}^z_{x'}\,u'$. The induction hypothesis gives us that $\Diamond\,\mathrm{ren}^z_x\,u \sim_\alpha \Diamond\,\mathrm{ren}^z_{x'}\,u'$.

In order to show that

$$\Diamond(\lambda x.\,u) = \lambda x.\ \Diamond u$$

is α-equivalent to

$$\Diamond(\lambda x'.\,u') = \lambda x'.\ \Diamond u'$$

it is sufficient to show that for a sufficiently fresh variable z' we have

$$\mathrm{ren}^{z'}_x\,\Diamond u \sim_\alpha \mathrm{ren}^{z'}_{x'}\,\Diamond u'.$$

Using Proposition 1.21 we have for z'' satisfying the freshness requirement for both z and z' that the given statement gives us the required one:

$$\mathrm{ren}^{z''}_x\,\Diamond u \sim_\alpha \Diamond\,\mathrm{ren}^{z''}_x\,u \sim_\alpha \Diamond\,\mathrm{ren}^{z''}_{x'}\,u'' \sim_\alpha \mathrm{ren}^{z''}_{x'}\,u'$$

scAppα If t is an application, say rs, then so is t', say $r's'$, and $r \sim_\alpha r'$ as well as $s \sim_\alpha s'$. We have to make a case distinction based on the result of applying the parallel reduct.

If r is a λ-abstraction, say $r = \lambda x.\,u$, then r' must also be an application, say $r' = \lambda x'.\,u'$, and we know that for a sufficiently fresh variable z we have $\mathrm{ren}^z_x\,u \sim_\alpha \mathrm{ren}^z_{x'}\,u'$. By the induction hypothesis we know that $\Diamond\,\mathrm{ren}^z_x\,u \sim_\alpha \Diamond\,\mathrm{ren}^z_{x'}\,u$ and $\Diamond s \sim_\alpha \Diamond s'$.

We have

$$\begin{aligned} \Diamond t &= \Diamond((\lambda x.\, u)s) && t = (\lambda x.\, u)s \\ &= \Diamond u[\Diamond s/x] && \text{scApp}\Diamond \end{aligned}$$

and

$$\begin{aligned} \Diamond t' &= \Diamond((\lambda x'.\, u')s') && t = (\lambda x'.\, u')s' \\ &= \Diamond u'[\Diamond s'/x'] && \text{scApp}\Diamond \end{aligned}$$

For z' being a sufficiently fresh variable we have the following.

$$\begin{aligned} \Diamond u[\Diamond s/x] &\sim_\alpha (\text{ren}_x^{z'}\, \Diamond u)[\Diamond s/z'] && \text{Proposition 1.10} \\ &= (\text{ren}_{x'}^{z'}\, \Diamond u')[\Diamond s'/z'] && \text{ind hyp and Prop 1.11} \\ &\sim_\alpha \Diamond u'[\Diamond s'/x'] && \text{Proposition 1.10} \end{aligned}$$

In the case where r is not a λ-abstraction we obtain

$$\begin{aligned} \Diamond t &= \Diamond(rs) && t = rs \\ &= \Diamond r\, \Diamond s && \text{scApp}\Diamond \\ &\sim_\alpha \Diamond r'\, \Diamond s' && \text{ind hyp and scApp}\alpha \\ &= \Diamond(r's') && \text{scApp}\Diamond \\ &= \Diamond t' && t' = r's' \end{aligned}$$

Proposition 1.25

Proof. We may calculate generally

$$\Diamond((\lambda x.\, t)a) = \Diamond t[\Diamond a/x] \qquad \text{scApp}\Diamond$$

but we need to put in some work to calculate $\Diamond(t[a/x])$, so we carry out a proof by induction over the structure of the term t, which will allow us to calculate the result of applying the substitution.

Hence the statement we prove by induction is that we may find t' with

$$\Diamond t[\Diamond a/x] \xrightarrow{\beta}\!\!\!\!\rightarrow\, t' \sim_\alpha \Diamond(t[a/x]),$$

where the induction hypothesis applies to subterms where one variable may have been renamed.

bcVar[] If t is a variable, say y, then there are two cases to consider.
If $y = x$ then $y[a/x]$ is a and so

$$\begin{aligned} \Diamond((\lambda x.\, t)a) &= \Diamond y[\Diamond a/x] && \text{scApp}\Diamond,\ t = y \\ &= y[\Diamond a/x] && \text{bcVar}\Diamond \\ &= \Diamond a && \text{bcVar}[\,],\ y = x \\ &= \Diamond(y[a/x]) && \text{bcVar}\Diamond,\ y = x \end{aligned}$$

and so we obtain the desired result by reflexivity of $\xrightarrow{\beta}{\!\!\!\!\!\rightarrow}$.
If $y \neq x$ then

$$\begin{aligned}
\Diamond((\lambda x.\, t)a) &= \Diamond y[\Diamond a/x] && \text{scApp}\Diamond,\ t = y\\
&= y[\Diamond a/x] && \text{bcVar}\Diamond\\
&= y && \text{bcVar}[\,],\ y \neq x\\
&= \Diamond y && \text{bcVar}\Diamond\\
&= \Diamond(y[a/x]) && \text{bcVar}\Diamond,\ y \neq x
\end{aligned}$$

Again we obtain the desired result by reflexivity of $\xrightarrow{\beta}{\!\!\!\!\!\rightarrow}$.

scAbs[] If t is a λ-abstraction, say $\lambda y.\, u$, then for $w = \mathrm{freshv}(\mathrm{vars}(\lambda y.\, \Diamond u) \cup \mathrm{vars}\, a \cup \{x\})$ and a variable z which s fresh for all terms

$$\begin{aligned}
\Diamond((\lambda x.\, t)a) &= \Diamond((\lambda x.\, \lambda y.\, u)a) && t = \lambda y.\, u\\
&= \Diamond(\lambda y.\, u)[\Diamond a/x] && \text{scApp}\Diamond\\
&= (\lambda y.\ \Diamond u)[\Diamond a/x] && \text{scAbs}\Diamond\\
&= \lambda w.\, (\mathrm{ren}^w_y\, \Diamond u)[\Diamond a/x] && \text{scAbs}[\,]\\
&\sim_\alpha \lambda z.\ \mathrm{ren}^z_w((\mathrm{ren}^w_y\, \Diamond u)[\Diamond a/x]) && \text{Proposition 1.7}\\
&\sim_\alpha \lambda z.\, (\mathrm{ren}^z_y\, \Diamond u)[\Diamond a/x] && \text{Propositions 1.10 and 1.7}
\end{aligned}$$

while for $w' = \mathrm{freshv}(\mathrm{vars}(\lambda y.\, u) \cup \mathrm{vars}\, a \cup \{x\})$

$$\begin{aligned}
\Diamond(t[a/x]) &= \Diamond(\lambda y.\, u[a/x]) && t = \lambda y.\, u\\
&= \Diamond(\lambda w'.\ \mathrm{ren}^w_y\, u[a/x]) && \text{scAbs}[\,].\\
&= \lambda w'.\ \Diamond(\mathrm{ren}^{w'}_y\, u[a/x]) && \text{scAbs}\Diamond
\end{aligned}$$

We get from the induction hypothesis that for an arbitrary variable z we may find a term u'' with

$$(\Diamond(\mathrm{ren}^z_y\, u))[a/x] \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} u'' \sim_\alpha \Diamond((\mathrm{ren}^z_y\, u)[a/x])$$

and we know from Proposition 1.21 that

$$\Diamond(\mathrm{ren}^z_y\, u) \sim_\alpha \mathrm{ren}^z_y\, \Diamond u$$

and from Proposition 1.11 that this implies that

$$(\mathrm{ren}^z_y\, \Diamond u)[\Diamond a/x] \sim_\alpha ((\Diamond(\mathrm{ren}^z_y\, u))[\Diamond a/x].$$

We may now invoke Proposition 1.16 to argue that we may find u' such that

$$(\mathrm{ren}^z_y\, \Diamond u)[\Diamond a/x] \xrightarrow{\beta}{\!\!\!\!\!\rightarrow} u' \sim_\alpha \Diamond((\mathrm{ren}^z_y\, u)[a/x]),$$

and it is sufficient to show that

$$\mathrm{ren}^z_{w'}(\Diamond((\mathrm{ren}^{w'}_y\, u)[a/x]) \sim_\alpha \Diamond((\mathrm{ren}^z_y\, u)[a/x]$$

to obtain the desired result with the help of scAbsβ, Proposition 1.7 and Proposition 1.16. We observe that

$$\begin{aligned}
\mathrm{ren}^z_{w'}(\Diamond((\mathrm{ren}^{w'}_y\, u)[a/x])) &\sim_\alpha \Diamond(\mathrm{ren}^z_{w'}((\mathrm{ren}^{w'}_y\, u)[a/x] && \text{Proposition 1.21}\\
&\sim_\alpha \Diamond((\mathrm{ren}^z_y\, u)[a/x]) && \text{Proposition 1.11 and Proposition 1.22}
\end{aligned}$$

and so we are done.

scApp[] If t is an application, say uu', then

$$\begin{aligned} \Diamond((\lambda x.\, t)a) &= \Diamond((\lambda x.\, uu')a) && t = uu' \\ &= \Diamond(uu')[\Diamond a/x] && \text{scApp}\Diamond \end{aligned}$$

and we have to make another case distinction. If u is a λ-abstraction, say $\lambda y.\, r$ then we continue

$$\begin{aligned} \ldots &= \Diamond((\lambda y.\, r)u')[\Diamond a/x] && u = \lambda y.\, r \\ &= (\Diamond r[\Diamond u'/y])[\Diamond a/x] && \text{scApp}\Diamond \\ &= \Diamond(r[a/x])[\Diamond(u'[a/x])/y] && \text{Proposition 1.12} \\ &= \Diamond((\lambda y.\, r[a/x])(u'[a/x])) && \text{scApp}\Diamond \\ &= \Diamond((\lambda y.\, r)u'[a/x]) && \text{scApp}[\,] \\ &= \Diamond(uu'[a/x]) && u = \lambda x.\, r \\ &= \Diamond(t[a/x]) && t = uu' \end{aligned}$$

If u is not a λ-abstraction then we continue as follows.

$$\begin{aligned} \ldots &= \Diamond u\, \Diamond u'[\Diamond a/x] && \text{scApp}\Diamond \\ &= (\Diamond u[\Diamond a/x])(\Diamond u'[\Diamond a/x]) && \text{scApp}[\,] \\ &\xrightarrow{\beta}\!\!\!\!\rightarrow\ rr' && \text{ind hyp and scApp}\beta \\ &\sim_\alpha (\Diamond u[a/x])(\Diamond u'[a/x]) && \text{ind hyp} \\ &= \Diamond(u[a/x]u'[a/x]) && \text{scApp}\Diamond \\ &= \Diamond(uu'[a/x]) && \text{scApp}[\,] \\ &= \Diamond(t[a/x]) && t = uu' \end{aligned}$$

Lemma 1.26

Proof. This proof is an induction based on the reason why t β-reduces to t'

bcVarβ Assume that $t = (\lambda x.\, r)a$ and $t \xrightarrow{\beta} r[b/y] = t'$. We know by Proposition 1.25 that there is a term u with

$$\Diamond t = \Diamond(\lambda x.\, ra) \xrightarrow{\beta}\!\!\!\!\rightarrow u \sim_\alpha \Diamond r[a/x] = \Diamond t'.$$

scAbsβ If $t = \lambda x.\, r$ and $r \xrightarrow{\beta} r'$ is the reason that $t = \lambda x.\, r \xrightarrow{\beta} \lambda x.\, r' = t'$ then we know from the induction hypothesis that we may find u' with $\Diamond r \xrightarrow{\beta}\!\!\!\!\rightarrow u' \sim_\alpha \Diamond r'$ and so by scAbsβ that

$$\Diamond t = \Diamond(\lambda x.\, r) = \lambda x.\ \Diamond r \xrightarrow{\beta}\!\!\!\!\rightarrow \lambda x.\, u' \sim_\alpha \lambda x.\ \Diamond r' = \Diamond(\lambda x.\, r') = \Diamond t',$$

where we owe the justification that $\lambda x.\, u' \sim_\alpha \lambda x.\ \Diamond r'$, which follows from $u' \sim_\alpha \Diamond r'$ by Proposition 1.7. Hence $u = \lambda x.\, u'$ satisfies the given claim.

scAppβ If t is an application, say $t = rs$, and we have $t \xrightarrow{\beta} t'$ because there is r' with $r \xrightarrow{\beta} r'$ and so $t = rs \xrightarrow{\beta} r's = t'$ we have to make a case distinction to be able to calculate the parallel reduct of t.

Assume that r is is a λ-abstraction, say $\lambda x.\, r''$. In this situation we must have $r \xrightarrow{\beta} r'$ because there is r''' with $r'' \xrightarrow{\beta} r'''$, and

$$t = rs = (\lambda x.\, r'')s \xrightarrow{\beta} (\lambda x.\, r''')s = r's = t'.$$

We know by scAbs$\Diamond$ that $\Diamond t = \Diamond((\lambda x.\, r'')s) = \Diamond r''[\Diamond s/x]$. The induction hypothesis gives us u' with

$$\Diamond r'' \xrightarrow{\beta}\!\!\!\!\!\rightarrow u' \sim_\alpha \Diamond r'''.$$

Hence we may use Proposition 1.19 to argue that there is a term u such that

$$\Diamond t = \Diamond r''[\Diamond s/x] \xrightarrow{\beta}\!\!\!\!\!\rightarrow u \sim_\alpha u'[\Diamond s/x] \sim_\alpha \Diamond r'''[\Diamond s/x] = \Diamond t',$$

and that u satisfies the claim.

If r is not a λ-abstraction we have $\Diamond t = \Diamond(rs) = \Diamond r\, \Diamond s$, and the induction hypothesis tells us that there is u' with $\Diamond r \xrightarrow{\beta}\!\!\!\!\!\rightarrow u' \sim_\alpha \Diamond r'$ and so

$$\Diamond t = \Diamond r\, \Diamond s \xrightarrow{\beta}\!\!\!\!\!\rightarrow u'\, \Diamond s \sim_\alpha \Diamond r'\, \Diamond s.$$

However, we can't be sure that $\Diamond t' = \Diamond(r's)$ is equal to $\Diamond r'\, \Diamond s$ since we can't be sure that r' is not an abstraction. So we need to make another case distinction here.

If r' is not an abstraction then we are done and $u = u'\, \Diamond s$ witnesses our claim.

If r' is an abstraction, say $\lambda y'.\, r''$, then since $u' \sim_\alpha \Diamond r' = \Diamond(\lambda y'.\, r'') = \lambda y'.\, \Diamond r''$ we know that u' is also an abstraction, say $\lambda y.\, u''$, and so we have

$\Diamond t = \Diamond r \Diamond s$	
$\xrightarrow{\beta}\!\!\!\!\!\rightarrow u'\, \Diamond s$	as before
$= (\lambda y.\, u'')\, \Diamond s$	$u' = \lambda y.\, u''$
$\xrightarrow{\beta}\!\!\!\!\!\rightarrow u''[\Diamond s/y]$	bcVarβ
$\sim_\alpha \Diamond r''[\Diamond s/y']$	Proposition 1.11
$= \Diamond((\lambda y'.\, r'')s)$	scApp $\Diamond$
$= \Diamond(r's)$	$r' = \lambda y\varepsilon.\, r''$
$= \Diamond t'$	$t'r's$.

In this situation our witness is $u''[\Diamond s/y]$.

Corollary 1.28

Proof. The given statement follows from Corollary 1.27 by a straightforward induction over the number $n \in \mathbb{N}$.

If $n = 0$ and so $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow t$ then we know that $t = \Diamond^0 t \xrightarrow{\beta}\!\!\!\!\!\rightarrow \Diamond^0 t = t$.

Assume we know the statement holds for n, and that we have $t \xrightarrow{\beta}\!\!\!\!\!\rightarrow t'$. By the induction hypothesis there exists u' with $\Diamond^n t \xrightarrow{\beta}\!\!\!\!\!\rightarrow u' \sim_\alpha \Diamond^n t'$.

Since $\Diamond^n t \xrightarrow{\beta}\!\!\!\!\!\rightarrow u'$ we know from Corollary 1.27 that there exists u with

$$\Diamond^{n+1} t = \Diamond\, \Diamond^n t \xrightarrow{\beta}\!\!\!\!\!\rightarrow u \sim_\alpha \Diamond u',$$

and from $u' \sim_\alpha \Diamond^n t'$ we deduce by Proposition 1.22 that $\Diamond u' \sim_\alpha \Diamond\, \Diamond^n t' = \Diamond^{n+1} t'$, so overall we have

$$\Diamond^{n+1} t \xrightarrow{\beta}\!\!\!\!\!\rightarrow u \sim_\alpha \Diamond u' \sim_\alpha \Diamond^{n+1} t',$$

so u is a suitable witness.

Theorem 1.30

Proof. By induction over the number m of β-reduction steps involved in $t \xrightarrow{\beta}\!\!\!\!\rightarrow t'$.

bcNat Base: If $m = 0$ then $t' = t \xrightarrow{\beta}\!\!\!\!\rightarrow t = \Diamond^0 t$.

scNat Suppose the result holds for m, and that $t \xrightarrow{\beta}\!\!\!\!\rightarrow t'$ in $m+1$ steps. That means there is a term r such that $t \xrightarrow{\beta} r$ and $r \xrightarrow{\beta}\!\!\!\!\rightarrow t'$ in m steps. By the induction hypothesis there exists $n \in \mathbb{N}$ and s with $t' \xrightarrow{\beta}\!\!\!\!\rightarrow s \sim_\alpha \Diamond^n r$. We further may apply Proposition 1.29 to $t \xrightarrow{\beta} r$ to find a term r' with $r \xrightarrow{\beta}\!\!\!\!\rightarrow r' \sim_\alpha \Diamond t$, and Proposition 1.22 tells us that $\Diamond^{n+1} t \sim_\alpha \Diamond^n r'$. By Corollary 1.28 we know that $r \xrightarrow{\beta}\!\!\!\!\rightarrow r'$ implies the existence of a term s' with $\Diamond^n r \xrightarrow{\beta}\!\!\!\!\rightarrow s' \sim_\alpha \Diamond^n r'$. We summarize this situation in a diagram.

$$\begin{array}{ccccccc} t' & & & & & & \\ {\scriptstyle\beta}\downarrow & & & & & & \\ s & \sim_\alpha & \Diamond^n r & & & & \\ & & {\scriptstyle\beta}\downarrow & & & & \\ & & s' & \sim_\alpha & \Diamond^n r' & \sim_\alpha & \Diamond^{n+1} t \end{array}$$

By Corollary 1.17 (using transitivity of α-equivalence) we know that there exists a term u satisfying the properties given in the following diagram.

$$\begin{array}{ccccc} t' & & & & \\ {\scriptstyle\beta}\downarrow & & & & \\ u & \sim_\alpha & \Diamond^{n+1} t & & \end{array}$$

Hence t' β-reduces to a term u which is α-equivalent to $\Diamond^{n+1} t$ as required.

Technical Proofs from Chapter 2

Proposition 2.1

Proof. If $\Gamma = \Gamma'$ and $\Delta = \Delta'$ then

$$\Gamma, \Delta = \Gamma', \Delta'.$$

If $\Gamma \approx \Gamma'$ then $\Gamma, \Delta \approx \Gamma', \Delta$, and similarly if $\Delta \approx \Delta'$ then $\Gamma, \Delta \approx \Gamma, \Delta'$ by definition of $\approx$.

The proof is now a simple induction over the number of steps required to move from Γ to Γ' and from Δ to Δ'.

Proposition 2.2

Proof. We look at each part

(a) Assume that we have a type environment Δ with $\Gamma, \Gamma' = \Delta, x{:}\sigma$. Then we may find type environments $\Delta_0, \Delta_1, \ldots, \Delta_n$ such that

$$\Gamma, \Gamma' = \Delta_0 \approx \Delta_1 \approx \cdots \approx \Delta_n = \Delta, x{:}\sigma$$

This gives us the following properties:

- The typing assumption $x:\sigma$ must occur in at least one of Γ or Γ'.
- The type x is different from all the types mentioned in any of the typing assumptions that occur to the right of the (right-most) typing assumption $x:\sigma$ in Γ, Γ'.

Hence we must be in one of two cases:

- it might be the case that $x:\sigma$ occurs in Γ' and it can be moved past all the typing assumptions that occur to its right, and so we may find a type environment Γ''' with $\Gamma' = \Gamma''', x:\sigma$.
- Otherwise $x:\sigma$ must occur in Γ, and it may be moved past all the typing assumptions that occur to its right, and so we get both, the existence of a type environment Γ''' such that $\Gamma = \Gamma''', x:\sigma$ as well as $x:\sigma, \Gamma' = \Gamma', x:\sigma$.

Assume that we know that one of the two cases holds.

- If we have $\Gamma' = \Gamma'', x:\sigma$ then by Proposition 2.1 we have $\Gamma, \Gamma' = \Gamma, \Gamma'', x:\sigma$ and we may use $\Delta = \Gamma, \Gamma''$.
- If we are in the second case then by Proposition 2.1 the assumption gives us have that

$$\Gamma, \Gamma' = \Gamma'', x:\sigma, \Gamma' = \Gamma'', \Gamma, x:\sigma$$

and so we may use $\Delta = \Gamma'', \Gamma$.

(b) We invoke the previous part with Δ in the role of $'Gamma$ and $y':\rho$ in the role of Γ' . Since x and y are distinct we must be in the second case, so we may find a type environment Δ' with $\Delta = \Delta', x:\sigma$.

Proposition 2.7

Proof. This is once again a proof by induction over the derivation. We assume that that the statement holds for subterms of the given term for all variables.

bcTAx If the derivation consists of the axiom rule then t must be a variable because of the form of that rule. Let us use x for this variable. Then the derivation is of the form

$$\frac{}{\Gamma, y:\alpha \vdash x:\tau}\,\mathrm{ax}$$

and there are two cases:

- If $x = y$ then $\tau = \alpha$ by the axiom rule. Now $\mathrm{ren}^z_x\, x = z$ and

$$\frac{}{\Gamma, z:\alpha \vdash z:\tau}\,\mathrm{ax}$$

is a valid derivation with the required conclusion.

- If $x \neq y$ then z being fresh for t gives us that $x \neq z$. Since $\Gamma, y:\alpha \vdash x:\tau$ is derivable we know that there exists a type environment Δ with $\Gamma, y:\alpha = \Delta, x:\tau$, and by Proposition 2.2 we know that there exists a type environment Γ' with $\Gamma = \Gamma', x:\tau$. Now

$$\Gamma, x:\tau = \Gamma', x:\tau, z:\alpha = \Gamma', z:\alpha, x:\tau$$

and so

$$\frac{}{\Gamma, z:\alpha \vdash x:\tau}\,\mathrm{ax}$$

is a valid instance of the axiom rule.

scTAbs If the derivation finishes with the abstraction rule then there must be types ρ and σ with $\tau = \rho \to \sigma$ and t must be of the form $\lambda x : \rho.\, u$, while we have a derivation which concludes with the rule

$$\frac{\Gamma, y : \alpha, x : \rho \vdash u : \sigma}{\Gamma, y : \alpha \vdash (\lambda x : \rho.\, u) : \rho \to \sigma}\,\text{abs}$$

. Again there are two cases:

- If $y \neq x$ then
$$\Gamma, y : \alpha, x : \rho = \Gamma, x : \rho, y : \alpha$$
and we may apply the induction hypothesis to obtain a derivation of $\Gamma, x : \rho, z : \alpha \vdash \mathrm{ren}^z_y\, u : \sigma$. Since z is fresh for t we know that $x \neq z$ and so
$$\Gamma, x : \rho, z : \alpha = \Gamma, z : \alpha, x : \rho$$
and so we may apply the abstraction rule to this derivation to obtain one of $\Gamma, z : \alpha \vdash (\lambda x : \rho.\, u) : \rho \to \sigma$, which is as required since
$$\mathrm{ren}^z_y(\lambda x : \rho.\, u) = \lambda x : \rho.\, \mathrm{ren}^z_y\, u$$
in this situation.
- If $y = x$ then we have an instance of shadowing. In this case the given derivation ends with
$$\Gamma, x : \alpha, x : \rho \vdash u : \sigma.$$
We may apply the induction hypothesis to this statement to obtain a derivation of the judgement
$$\Gamma, x : \alpha, z : \rho \vdash (\mathrm{ren}^z_x\, u) : \sigma.$$
Since z is fresh for t we know that $x \neq z$ and so $x \notin \mathrm{vars}(\mathrm{ren}^z_x\, u)$, and hence in particular $x \notin \mathrm{fv}(\mathrm{ren}^z_x\, u)$, which means that we may invoke Proposition 2.4 to obtain a derivation of
$$\Gamma, z : \rho \vdash (\mathrm{ren}^z_x\, u) : \sigma$$
and by Proposition 2.3 we may turn this into a derivation of
$$\Gamma, z : \alpha, z : \rho \vdash (\mathrm{ren}^z_x\, u) : \sigma.$$
We may now extend that derivation with an instance of the abstraction rule to obtain a derivation of $\Gamma, z : \alpha \vdash (\lambda z : \rho.\, \mathrm{ren}^z_x\, u) : \rho \to \sigma$, and since in this situation we do indeed have
$$\mathrm{ren}^z_x(\lambda x : \rho.\, u) = \lambda z : \rho.\, \mathrm{ren}^z_x\, u$$
this is indeed the required judgement.

scTApp If the derivation finishes with the application rule, then t must be an application, say rs and we must be able to find a type ρ such that this must be of the form

$$\frac{\Gamma, y : \alpha \vdash r : \rho \to \tau \quad \Gamma, y : \alpha \vdash s : \rho}{\Gamma, y : \alpha \vdash rs : \tau}\,\text{app}$$

and we also know that z is fresh for both r and s. Applying the induction hypothesis to the given derivations of the two premises, we obtain derivations of

$$\Gamma, z : \alpha \vdash \mathrm{ren}^z_y\, r : \rho \to \tau \qquad \text{and} \qquad \Gamma, z : \alpha \vdash \mathrm{ren}^z_y\, s : \rho$$

which we can combine with the application rule to obtain a derivation whose last rule is

$$\frac{\Gamma, z : \alpha \vdash \mathrm{ren}^z_y\, r : \rho \to \tau \quad \Gamma, z : \alpha \vdash \mathrm{ren}^z_y\, s : \rho}{\Gamma, z : \alpha \vdash (\mathrm{ren}^z_y\, r)(\mathrm{ren}^z_y\, s) : \tau}\,\text{app}$$

which is what we want to show since $\mathrm{ren}^z_y(rs) = (\mathrm{ren}^z_y\, r)(\mathrm{ren}^z_y\, s)$.

Proposition 2.10

Proof. The proof is by induction on the term t, which by inversion we may turn into one on the derivation of $\Gamma, x:\alpha \vdash t:\tau$. Intuitively, we traverse the derivation until we find a place were the axiom rule is applied to make use of the assumption $x:\alpha$, and get rid of this assumption by pasting the given derivation of $a:\alpha$ there.

bcTAx If this derivation is by the axiom rule, then $t = y$ for some variable y. Then there are two cases:

- If $y = x$ then $\tau = \alpha$ and we have $t[a/x] = x[a/x] = a$. This means that we must supply a derivation that $\Gamma \vdash a:\alpha$, but we are given just such a derivation by our first assumption.
- If $y \neq x$ then $t[a/x] = y[a/x] = y = t$ then the assumption $x:\alpha$ is not required to type $t:\tau$, and thus $\Gamma \vdash t:\tau$ is a valid instance of the axiom rule.

scTAbs In this case the derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ \Gamma, x:\alpha, y:\rho \vdash u:\sigma\end{array}}{\Gamma, x:\alpha \vdash (\lambda y:\rho.\, u):\rho \to \sigma}\,\text{abs}$$

where $\tau = \rho \to \sigma$.
We have to derive the judgement

$$\Gamma, x:\alpha \vdash (\lambda y:\rho.\, u)[a/x]:\rho \to \sigma.$$

Now $(\lambda y:\rho.\, u)[a/x] = \lambda w:\rho.\, (\mathrm{ren}_y^w\, u)[a/x]$ where

$$w = \mathrm{freshv}(\mathrm{vars}\, a \cup \mathrm{vars}\, \lambda y:\rho.\, u \cup \{x\}).$$

We may apply Proposition 2.7 to the given derivation of $\Gamma, x:\alpha, y:\rho \vdash u:\sigma$ to obtain a derivation of

$$\Gamma, x:\alpha, w:\rho \vdash (\mathrm{ren}_y^w\, u):\sigma.$$

Since in particular $w \notin \mathrm{fv}\, a$ we can again weaken the derivation of $\Gamma \vdash a:\alpha$ using Proposition 2.4 to obtain a derivation of $\Gamma, w:\rho \vdash a:\alpha$. The induction hypothesis provides a derivation of

$$\Gamma, w:\rho \vdash (\mathrm{ren}_y^w\, u)[a/x]:\sigma$$

which we can use along with the abstraction rule to construct a derivation ending in

$$\frac{\Gamma, w:\rho \vdash (\mathrm{ren}_y^w\, u)[a/x]:\sigma}{\Gamma \vdash (\lambda w:\rho.\, (\mathrm{ren}_y^w\, u)[a/x]):\rho \to \sigma}\,\text{abs}$$

as required.

scTApp Assume the given derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ \Gamma, x:\alpha \vdash r:\sigma \to \tau\end{array}\quad\begin{array}{c}\vdots\\ \Gamma, x:\alpha \vdash s:\sigma\end{array}}{\Gamma, x:\alpha \vdash rs:\tau}\,\text{app}$$

Then the induction hypothesis means we have derivations of

$$\Gamma \vdash r[a/x]:\sigma \to \tau \qquad \text{and} \qquad \Gamma \vdash s[a/x]:\sigma$$

so we can use the application rule

$$\frac{\Gamma \vdash r[a/x] : \sigma \to \tau \quad \Gamma \vdash s[a/x] : \sigma}{\Gamma \vdash r[a/x]s[a/x] : \tau}\,\text{app}$$

to extend these derivations to obtain one ending in the above conclusion. Since $(rs)[a/x] = r[a/x]s[a/x]$ this gives the required derivation of

$$\Gamma \vdash ((rs)[a/x]) : \tau.$$

Proposition 2.13

Proof. We show each part.

(a) Assume that we know that $\Gamma\Gamma' \vdash x : \sigma$ is derivable. We know from the previous part that there are two cases that may arise here.

- If $\Gamma' \vdash x : \sigma$ is derivable then $\Gamma\Delta\Gamma' \vdash x : \sigma$ is derivable by Proposition 2.4..
- Otherwise we know that $\Gamma \vdash x : \sigma$ is derivable and by compatibility of Δ with Γ that gives $\Gamma\Delta \vdash x : \sigma$ being derivable, which means that we may find Δ' with.$\Gamma\Delta = \Delta' x : \sigma$, and so by Proposition 2.1 we may reason that

$$\Gamma\Delta\Gamma' = \Delta' x : \sigma\Gamma' = \Delta'\Gamma' x : \sigma$$

which implies $\Gamma\Delta\Gamma' \vdash x : \sigma$ by the axiom rule.

(b) This is a proof by induction over the given derivation. We quantify over all type‘ environments.

bcTAx The statement required here is precisely the previous part.

scTabs If the given derivation concludes with the abstraction rule then t must be of the form $\lambda x : \rho.\, u$, τ must be $\rho \to \sigma$ and the given derivation of the form

$$\frac{\begin{array}{c}\vdots\\ \Gamma\Gamma', x : \rho \vdash u : \sigma\end{array}}{\Gamma\Gamma' \vdash \lambda x : \rho.\, u : \rho \to \sigma}\,\text{abs}$$

We apply the induction hypothesis for type environments Γ and $\Gamma', x : \rho$ as well as Δ, which tells us that $\Gamma\Delta\Gamma', x : \rho$ is derivable and we may use the application rule to obtain the desired derivation of

$$\Gamma\Delta\Gamma' \vdash \lambda x : \rho.\, u : \rho \to \sigma.$$

scTapp If the give derivation concludes with the application rule then the given term must be an application, say $f\,u$, and there must be a type σ such that the given derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ \Gamma\Gamma' \vdash f : \sigma \to \tau\end{array} \quad \begin{array}{c}\vdots\\ \Gamma\Gamma' \vdash u : \sigma\end{array}}{\Gamma\Gamma' \vdash f\,u : \tau}\,\text{app}$$

The induction hypothesis tells us that we are able to derive

$$\Gamma\Delta\Gamma' \vdash f : \sigma \to \tau \qquad \text{and} \qquad \Gamma\Delta\Gamma' \vdash u : \sigma$$

and we may take the corresponding derivations and apply the application rule to get the required derivation of

$$\Gamma\Delta\Gamma' \vdash f\,u : \tau.$$

(c) This follows from the preceding part in the case where Γ' is the empty type environment.

Proposition 2.14

Proof. This is a proof by induction over the type.

bc→ This holds by assumption.

sc→ Assume that $(\Gamma, t) \in \mathcal{R}_{\rho\to\sigma}$ and that Δ is compatible with Γ. Assume that Δ' is compatible with $\Gamma\Delta$. Then Δ' is compatible with Γ. If we now hat $(\Delta', u) \in \mathcal{R}_\rho$ we know that

$$(\Gamma\Delta', tu) \in \mathcal{R}_\sigma$$

which by the induction hypothesis implies that

$$(\Gamma\Delta\Delta', tu) \in \mathcal{R}_\sigma,$$

and since $\mathcal{R}$ is a logical predicate we obtain that

$$(\Gamma\Delta, t) \in \mathcal{R}_{\rho\to\sigma}$$

as required.

Proposition 2.16

Proof. The second statement follows from the first by a straightforward induction on the number of β-reduction steps involved: in the base case we already know that $t = t'$, and the step case is essentially given by the first statement. So we turn out attention to the first statement.

We proceed by induction for the reason why $t \xrightarrow{\beta} t'$. The induction hypothesis is that the claim holds for subterms where one variable may have been renamed.

bcVarβ Assume that $t = (\lambda x\colon\rho.\, u)a$ and $t \xrightarrow{\beta} u[a/x]$. We further know by inversion that we may derive $\Gamma, x\colon\rho \vdash u\colon\tau$ and $\Gamma \vdash a\colon\rho$. Since $t \sim_\alpha u[a/x]$ we know that the reduced term must be of the same form as t, and since substitution does not affect the overall structure of a term until we get to the leaves of its parse tree there are only three possibilities here:

- The term u might be a variable. If this variable, say y, is not equal to x then $y[a/x] = y$ which certainly is not α-equivalent to $(\lambda x\colon\rho.\, x)a$. If, on the other hand, $u = x$ then $x[a/x] = a$ and we must have that $a \sim_\alpha (\lambda x\colon\rho.\, u)a$, but then the term on the right would have to be α-equivalent to one of its proper subterms, namely a, which is not possible.
- If u is not a variable it must be an application in order for $u[a/x]$ to match the structure of the original term, and the first term of that application, after substituting a for x, must be α-equivalent to $\lambda x\colon\rho.\, u$, so either it is an abstraction, which is the case we look at below, or it is a variable, which is the case we treat here.
  So assume that u is of the form ys, then

  $$u[a/x] = (ys)[a/x] = (y[a/x])(s[a/x]) \sim_\alpha (\lambda x\colon\rho.\, u)a.$$

  This means we must have

  $$y[a/x] \sim_\alpha \lambda x\colon\rho.\, ys,$$

  and for this to be possible we must have $y = x$. However, the type of x (and so the type of a) is ρ while type of $\lambda x\colon\rho.\, ys$ is $\rho \to \tau$, so this case is not be possible.

- If u is an application whose first term is not a variable then in order to match the structure of the original term we must be able to find a variable y and terms s and b such that $u = (\lambda y\colon \rho.\, s)b$, where we know that the type annotation for y must be the same as for x by the definition of α-equivalence of type-annotated terms.
We note that if we had $y = x$ then

$$t = (\lambda x\colon \rho.\, (\lambda x\colon \rho.\, s)b)a \xrightarrow{\beta} (\lambda x\colon \rho.\, s)b[a/x] = (\lambda x\colon \rho.\, s)b,$$

and for the reduced term to be α-equivalent to t we would need a term to be α-equivalent to one of its proper subterms, which cannot occur. We use this fact below when we apply renaming of x on the term y, which has to give y, or renaming y in x, which has to give x.
Now

$$u[a/x] = (\lambda y\colon \rho.\, s)b[a/x] = (\lambda w\colon \rho.\, \mathrm{ren}_y^w\, s[a/x])b[a/x] \sim_\alpha t = (\lambda x\colon \rho.\, (\lambda y\colon \rho.\, s)b)a,$$

where

$$w = \mathrm{freshv}(\mathrm{vars}\, s \cup \mathrm{vars}\, a \cup \{x\}),$$

which means we must have

$$\lambda w\colon \rho.\, (\mathrm{ren}_y^w\, s)[a/x] \sim_\alpha \lambda x\colon \rho.\, (\lambda y\colon \rho.\, s)b \qquad \text{and} \qquad b[a/x] \sim_\alpha a.$$

Hence for z a fresh variable for both terms we have,

$$\mathrm{ren}_w^z((\mathrm{ren}_y^w\, s)[a/x]) \sim_\alpha \mathrm{ren}_x^z((\lambda y\colon \rho.\, s)b).$$

and we know from Proposition 1.10 that

$$\mathrm{ren}_w^z((\mathrm{ren}_y^w\, s)[a/x]) \sim_\alpha \mathrm{ren}_y^z\, s[a/x],$$

and so s must satisfy

$$\mathrm{ren}_y^z\, s[a/x] \sim_\alpha (\lambda y\colon \rho.\, \mathrm{ren}_x^z\, s)\mathrm{ren}_x^s\, b.$$

We claim that a term s satisfying this property cannot exist. In particular, given a term a that there is no term s with the property that there are variables x and y and a term b such that for a variable z fresh as given above we have the given instance of α-equivalence.
We assume that s is a term of minimal length with this property. We look at the possible structure of s.
  - If s is a variable then unless it is the variable x it cannot possibly be a term that (after renaming) is α-equivalent to an application.
    If it is x then the condition becomes

$$a \sim_\alpha (\lambda y\colon \rho.\, \mathrm{ren}_x^z\, x)(\mathrm{ren}_x^z\, b) = (\lambda y\colon \rho.\, z)(\mathrm{ren}_x^z\, b).$$

    Now z occurs free in the term on the right, which means that it has to occur free in the term on the left, but it was chosen to be fresh for

$$\lambda w\colon \rho.\, (\mathrm{ren}_y^w\, x)[a/x]) = \lambda w\colon \rho.\, a$$

    which is a contradiction.

– If s is an abstraction then it cannot satisfy the given condition since after renaming and substitution the resulting term is still an abstraction and so cannot be α-equivalent to an application.
– If s is an application, say $s = pc$ the condition becomes

$$((\mathrm{ren}_y^z\, p)[a/x])(\mathrm{ren}_y^z\, c[a/x]) \sim_\alpha (\lambda y{:}\rho.\ \mathrm{ren}_x^z(pc))(\mathrm{ren}_x^z\, b),$$

and we need the first two terms of these applications to be α-equivalent, that is

$$(\mathrm{ren}_y^z\, p)[a/x] \sim_\alpha \lambda y{:}\rho.\ \mathrm{ren}_x^z\,(pc).$$

If p is a variable then it must be x for the term resulting from renaming and substitution to be α-equivalent to an abstraction. But once again if $p = x$ it (and a) must have type ρ while the term it is α-equivalent to has type $\rho \to \rho$, and so this case cannot occur.
Hence p must be an abstraction, say $\lambda y' : \rho.\, q$ and then our condition becomes

$$(\mathrm{ren}_y^z(\lambda y' : \rho.\, q))[a/x] \sim_\alpha \lambda y{:}\rho.\ \mathrm{ren}_x^z((\lambda y' : \rho.\, q)c).$$

We look at the original term t and the result of β-reducing it.

$$\begin{aligned}&(\lambda x{:}\rho.\,(\lambda y{:}\rho.\,(\lambda y' : \rho.\, q)c)b)a\\ &\quad\xrightarrow{\beta} (\lambda w{:}\rho.\,(\lambda w' : \rho.\,((\mathrm{ren}^{w'}_{\mathrm{ren}^w_y\, y'}\,\mathrm{ren}^w_y\, q)[a/x]))(\mathrm{ren}^w_y\, c)[a/x])b[a/x],\end{aligned}$$

where w and w' are the variables provided by the freshv function for the appropriate input..
We once more work out whether y' might be equal to one of the other variables known to occur in t, namely x or y.
If $y' = y$ then we know it must be distinct from x, the original term t is and the reduction under consideration is

$$(\lambda x{:}\rho.\,(\lambda y{:}\rho.\,(\lambda y{:}\rho.\, q)c)b)a \xrightarrow{\beta} (\lambda w{:}\rho.\,(\lambda w' : \rho.\ \mathrm{ren}^{w'}_y\, q)(\mathrm{ren}^w_y\, c)[a/x])b[a/x].$$

In our condition therefore renaming y to z has no effect and said condition becomes

$$\lambda w' : \rho.\, q[a/x] \sim_\alpha \lambda y{:}\rho.\,(\lambda y{:}\rho.\ \mathrm{ren}^z_x\, q)\,\mathrm{ren}^z_x\, c.$$

We can see that if x occurs in q as a free variable then z occurs free in the term on the right, but not in the term on the left, which is impossible. But if x does not occur in q as a free variable then up to α-equivalence the substitution has no effect on the term on the left and then the condition becomes

$$\lambda w' : \rho.\, q \sim_\alpha \lambda y{:}\rho.\,(\lambda y{:}\rho.\, q)\,\mathrm{ren}^z_x\, c,$$

and so we are again in a situation where a term would have to be α-equivalent to one where it appears (up to renaming a variable) a proper subterm, which is not possible.
If $y' = x$ then it is distinct from y and the result of β-reducing t is

$$(\lambda w{:}\rho.\,(\lambda w' : \rho.\,((\mathrm{ren}^{w'}_x\,\mathrm{ren}^w_y\, q)[a/x]))(\mathrm{ren}^w_y\, c)[a/x])b[a/x],$$

and we know that since x does not occur in $\mathrm{ren}^{w'}_x\,\mathrm{ren}^w_y\, q$, the result of substituting a for x in that turn is α-equivalent to $\mathrm{ren}^{w'}_x\,\mathrm{ren}^w_y\, q$.

Our condition becomes, since x and y are known to be distinct, and freely using results from the previous chapter,

$$\lambda w' : \rho.\ \mathrm{ren}_x^{w'}\ \mathrm{ren}_y^z\ q \sim_\alpha \lambda y : \rho.\ (\lambda z : \rho.\ \mathrm{ren}_x^z\ q)\ \mathrm{ren}_x^z\ c.$$

This says, for z' a variable fresh for both terms, that

$$\mathrm{ren}_x^{z'}\ \mathrm{ren}_y^z\ q \sim_\alpha \lambda z : \rho.\ (\mathrm{ren}_y^{z'}\ \mathrm{ren}_x^z\ q)(\mathrm{ren}_y^{z'}\ \mathrm{ren}_x^z\ c),$$

and so a term of the shape of q would have to be α-equivalent to a term that contains a term of the shape of q as a proper subterm, which is not possible.
So we know that y' must be distinct from both x and y, which means our condition is, for the variable w'' provided by the freshv function for the appropriate inputs, that

$$\lambda w'' : \rho.\ (\mathrm{ren}_{y'}^{w''}\ \mathrm{ren}_y^z\ q)[a/x] \sim_\alpha \lambda y : \rho.\ (\lambda y' : \rho.\ \mathrm{ren}_x^z\ q)\ \mathrm{ren}_x^z\ c.$$

This means that z' fresh for both terms we must have

$$\mathrm{ren}_{w''}^{z'}(\mathrm{ren}_{y'}^{w''}\ \mathrm{ren}_y^z\ q[a/x]) \sim_\alpha (\lambda y' : \rho.\ \mathrm{ren}_y^{z'}\ \mathrm{ren}_x^z\ q)\ \mathrm{ren}_y^{z'}\ \mathrm{ren}_x^z\ c.$$

We know from Proposition 1.10 that the term on the left is α-equivalent to

$$(\mathrm{ren}_{y'}^{z'}\ \mathrm{ren}_y^z\ q)[a/x].$$

But now $\mathrm{ren}_x^z\ q$ is a shorter term that, with variables y and y' and the fresh variable z', satisfies the condition for which we assumed s to be a term of minimal length. Hence a term s satisfying the given property can indeed not exist as claimed, and so the this base case cannot occur.

scAbsβ If $t = \lambda x : \rho.\ u \xrightarrow{\beta} \lambda x : \rho.\ u'$ where $u \xrightarrow{\beta} u'$ and $\lambda x : \rho.\ u \sim_\alpha \lambda x : \rho.\ u'$ then we know that for $w = \mathrm{freshv}(\mathrm{vars}(\lambda x : \rho.\ u) \cup \mathrm{vars}(\lambda x : \rho.\ u'))$ we have $\mathrm{ren}_x^w\ u \sim_\alpha \mathrm{ren}_x^w\ u'$. We also know by Proposition 1.14 that $\mathrm{ren}_x^w\ u \xrightarrow{\beta} \mathrm{ren}_x^w\ u'$ and so we may apply the induction hypothesis to tell us that $\mathrm{ren}_x^w\ u = \mathrm{ren}_{x'}^w\ u'$, and by Proposition 1.2 this implies, by renaming w back to x, that $u = u'$ and so $\lambda x : \rho.\ u = \lambda x : \rho.\ u'$ as required.

scAppβ If $t = rs \xrightarrow{\beta} r's$ and $r \xrightarrow{\beta} r'$ then $rs \sim_\alpha r's$ implies that $r \sim_\alpha r'$ and $s \sim_\alpha s'$. Then we may employ the induction hypothesis to establish that $r = r'$ and so $rs \xrightarrow{\beta} r's$. The case where the reduction occurs in the other term is much the same.

Lemma 2.18

Proof.

(a),(b) This is a proof by induction on the type τ. We treat the two statements as a big conjunction—in other words we prove the properties *simultaneously* by induction. The induction hypothesis is that both claims hold for the immediate subtypes of the given type.

bc$\to$ In the base case, property (i) is straightforward from the definition of $\mathcal{S}$ and (ii) is a property of strong normalization proved in 2.17.

sc$\to$ We assume that we have a type of the form $\rho \to \sigma$ and prove both step cases.

(i) Assume that $(\Gamma, t) \in \mathcal{S}_{\rho\to\sigma}$. Let z be a variable which Γ does not assign any type to. Then $z : \rho$ is compatible with Γ and $\Gamma z : \rho \vdash z : \rho$ is derivable by the axiom rule, so by the induction hypothesis for part (ii) for type ρ (with $k = 0$), we have $(\Gamma z : \rho, z) \in \mathcal{S}_\rho$. Using that $\mathcal{S}$ is a logical predicate we may deduce from the assumption $(\Gamma, t) \in \mathcal{S}_{\rho\to\sigma}$ that $(\Gamma z : \rho, tz) \in \mathcal{S}_\sigma$. Then by part (i) of the induction hypothesis for the type σ, we know that tz is strongly normalizing. Since t is a subterm of tz we know by Proposition 2.17 that t is also strongly normalizing.

(ii) Assume we have terms as in the statement, and that

$$\Gamma \vdash x t_1 t_2 \cdots t_n : \rho \to \sigma \text{ is derivable.}$$

We need to prove that $(\Gamma, x t_1 \cdots t_n) \in \mathcal{S}_{\rho\to\sigma}$, for which we need to establish the property defining logical predicates. Given

$$(\Gamma\Delta, t_{n+1}) \in \mathcal{S}_\rho$$

such that Δ is compatible with Γ, part (i) of the induction hypothesis for the type ρ tells us that t_{n+1} is strongly normalizing. We would like to apply part (ii) of the induction hypothesis for the type σ. We know that all terms $t_1, \ldots, t_n, t_{n+1}$ are strongly normalizing, and that Δ is compatible with Γ, and so we obtain

$$(\Gamma\Delta, x t_1 \ldots t_{n+1}) \in \mathcal{S}_\sigma.$$

Since $\mathcal{S}$ is a logical predicate we may conclude the required

$$(\Gamma, x t_1 \cdots t_n) \in \mathcal{S}_{\rho\to\sigma}.$$

(c) This is also a proof by induction on the type τ.

bc→ This is proved in Proposition 2.17.

sc→ We assume that we have a type of the form $\rho \to \sigma$. Assume we have terms as required. We want to show that

$$(\Gamma, (\lambda x : \alpha.\, s) t_0 t_1 \cdots t_k) \in \mathcal{S}_{\rho\to\sigma}.$$

Again we invoke the definition of a logical predicate, so assume that Δ be compatible with Γ and that we also have

$$(\Gamma\Delta, t_{k+1}) \in \mathcal{S}_\rho.$$

Then since by assumption

$$(\Gamma, s[t_0/x] t_1 \cdots t_k) \in \mathcal{S}_{\rho\to\sigma},$$

the definition of $\mathcal{S}_{\rho\to\sigma}$ implies that $(\Gamma\Delta, s[r_0/x] t_1 \cdots t_k t_{k+1}) \in \mathcal{S}_\sigma$. By the induction hypothesis for the type σ, we have

$$(\Gamma, (\lambda x : \alpha.\, s) t_0 t_1 \ldots t_k t_{k+1}) \in \mathcal{S}_\sigma,$$

and so by definition of $\mathcal{S}_{\rho\to\sigma}$ we have shown the required statement.

Lemma 2.19

Proof. This is a proof by induction on the term t. The induction hypothesis is that the claim holds for all immediate subterms, but we quantify over type environments Γ, Δ and terms a_i that are suitable to replace the variables that occur in Γ (and which therefore must have the appropriate type).

bcPVar If t is a variable, then it must be one of the variables x_i in order for $\Gamma \vdash t : \tau$ to be derivable, and then we have $\tau = \alpha_i$. In that case $t[a_1/x_1] \cdots [a_n/x_n] = a_i$ and so by assumption $(\Delta, a_i) \in \mathcal{S}_{\alpha_i} = \mathcal{S}_\tau$ as required.

scPAbs If t is an abstraction, say $\lambda x : \rho.\, u$, then τ is of the form $\rho \to \sigma$ for some type σ and and

$$\Gamma, x : \sigma \vdash u : \rho \to \sigma$$

is derivable.

We have to show that

$$(\Delta, (\lambda x : \rho.\, u)[a_1/x_1] \cdots [a_k/x_k]) \in \mathcal{S}_{\rho\to\sigma}.$$

We aim to use the definition of a logical predicate here, so assume that Δ' is compatible with Δ and that $(\Delta\Delta', r) \in \mathcal{S}_\rho$. Since $\mathcal{S}$ is a logical predicate it is sufficient to show that

$$(\Delta\Delta', ((\lambda x : \rho.\, u)[a_1/x_1] \cdots [a_k/x_k])r) \in \mathcal{S}_\sigma.$$

Since $(\Delta\Delta', r) \in \mathcal{S}_\rho$, by part (i) of Lemma 2.18 we know that r is strongly normalizing, and so we may apply part (iii) of the same lemma to argue that it is sufficient to prove that

$$(\Delta\Delta', u[a_1/x_1] \cdots [a_k/a_k][r/x]) \in \mathcal{S}_\sigma.$$

In order to do that we apply the induction hypothesis for the term u, for the terms a_i from before and additionally for r. We have ensured that the type environment $\Delta\Delta'$ paired with any of the substituted terms gives an element of $\mathcal{S}$ at the appropriate type, and so we obtain the required statement from the induction hypothesis.

scPApp Assume that t is an application, say $f\,u$. then by inversion there is a type σ such that

$$\Gamma \vdash f : \sigma \to \tau \qquad \text{and} \qquad \Gamma \vdash u : \sigma$$

are derivable. This allows us to invoke the induction hypothesis to deduce that

$$(\Delta, f[a_1/x_1]...[a_n/x_n]) \in \mathcal{S}_{\sigma\to\tau}$$

and

$$(\Delta, u[a_1/x_1] \cdots [a_n/x_n]) \in \mathcal{S}_\sigma.$$

Since $\mathcal{S}$ is a logical predicate this means that

$$(\Delta, f[a_1/x_1] \cdots [a_n/x_n]u[a_1/x_1] \cdots [a_n/x_n] \in \mathcal{S}_\tau.$$

Since

$$(f[a_1/x_1] \cdots [a_n/x_n])(u[a_1/x_1] \cdots [a_n/x_n]) = (fu)[a_1/x_1] \cdots [a_n/x_n]$$

this gives us

$$(\Delta, fu[a_1/x_1]...[a_n/x_n]) \in \mathcal{S}_\tau$$

as required.

Proposition 2.28

Proof. This is a proof by induction over the structure of t, where we want to keep track of why $t \xrightarrow{\eta} t'$. The induction hypothesis is that the statement holds for proper subterms.

bcVarη If $t = \lambda x.\, sx \xrightarrow{\eta} s = t'$ and $x \notin \mathrm{fv}\, s$ then to compute $\mathrm{ren}^z_y\, t$ we have to make a case distinction.

- If $y = x$ then $\mathrm{ren}^z_y\, t = \mathrm{ren}^z_x(\lambda x.\, sx) = \lambda z.\ \mathrm{ren}^z_x\, sz$ and by bcVarη we have

$$\lambda z.\ \mathrm{ren}^z_x\, sz \xrightarrow{\eta} \mathrm{ren}^z_x\, s = \mathrm{ren}^z_x\, t'$$

  as required.
- If $y \neq x$ then $\mathrm{ren}^z_y\, t = \mathrm{ren}^z_y(\lambda x.\, sx) = \lambda x.\ \mathrm{ren}^z_y\, sx$ and again by bcVarη we have the required $\lambda x.\ \mathrm{ren}^z_y\, sx \xrightarrow{\eta} \mathrm{ren}^z_y\, s = \mathrm{ren}^z_y\, t'$.

scAbsη If $t = \lambda x.\, s \xrightarrow{\eta} \lambda x.\, s' = t'$ and $s \xrightarrow{\eta} s'$ then by the induction hypothesis we know that $\mathrm{ren}^z_y\, s \xrightarrow{\eta} \mathrm{ren}^z_y\, s'$. We again make a case distinction.

- If $y = x$ then $\mathrm{ren}^z_y\, t = \mathrm{ren}^z_x\, \lambda x.\, s = \lambda z.\, \mathrm{ren}^z_x\, s$ which by scAbsη is η-equivalent to $\mathrm{ren}^z_y\, t' = \mathrm{ren}^z_x\, \lambda x.\, s' = \lambda z.\, \mathrm{ren}^z_x\, s'$.
- If $y \neq x$ then $\mathrm{ren}^z_y\, t = \mathrm{ren}^z_y\, \lambda x.\, s = \lambda x.\, \mathrm{ren}^z_y\, s$ which by once again by scAbsη is η-equivalent to $\mathrm{ren}^z_y\, t' = \mathrm{ren}^z_y\, \lambda x.\, s' = \lambda x.\, \mathrm{ren}^z_y\, s'$.

scAppη If $t = su \xrightarrow{\eta} s'u' = t'$ and $s \xrightarrow{\eta} s'$ as well as $u = u'$ then we know by the induction hypothesis that $\mathrm{ren}^z_y\, s \xrightarrow{\eta} \mathrm{ren}^z_y\, s'$ and so by scAppη we may conclude that

$$\mathrm{ren}^z_y(su) = \mathrm{ren}^z_y\, s\mathrm{ren}^z_y\, u \xrightarrow{\eta} \mathrm{ren}^z_y\, s'\mathrm{ren}^z_y\, u = \mathrm{ren}^z_y(s'u) = \mathrm{ren}^z_y(s'u').$$

The other case is much the same.

Proposition 2.31

Proof. We have done most of the work required by showing that this works for untyped λ-terms in Proposition 2.28 and we just have to worry about the typing requirements that are required for η-conversion in this setting.

bcVarη In the base case we need to ensure that $\Gamma, z : \sigma \vdash \mathrm{ren}^z_y\, t : \sigma$ is derivable which follows immediately from Proposition 2.7.

scAbsη In this case there are no typing requirements.

scAppη In this case we know, for example, that $t \xrightarrow{\eta}_{\Gamma, x:\sigma} t'$ and $u = u'$, and that there are types σ and τ such that both $\Gamma, y : \sigma \vdash t : \sigma \to \tau$ and $\Gamma, y : \sigma \vdash u : \sigma$ are derivable. By Proposition 2.7 we know that this means that $\Gamma, z : \sigma \vdash \mathrm{ren}^z_y\, t : \sigma \to \tau$ and $\Gamma, z : \sigma \vdash \mathrm{ren}^z_y\, u : \sigma$ are both derivable which is sufficient to allow us to conclude that

$$\mathrm{ren}^z_y(tu) = (\mathrm{ren}^z_y\, t)(\mathrm{ren}^z_y\, u) \xrightarrow{\eta}_{\Gamma, z:\sigma} (\mathrm{ren}^z_y\, t')(\mathrm{ren}^z_y\, u) = \mathrm{ren}^z_y(t'u) = \mathrm{ren}^z_y(t'u')$$

as required.

The other case is much the same.

Proposition 2.37

Proof. We observe first of all that we know from Proposition 2.7 that under the given assumption $\Gamma z : \sigma \vdash \mathrm{ren}^z_y\, t$ is derivable and so the required denotation of $\mathrm{ren}^z_y\, t$ is defined.

The proof proceeds by induction over the structure of t.

bcVar If t is a variable, say x, then there are two cases to consider.

- If $x = y$ then $\mathrm{ren}^z_y\, t = \mathrm{ren}^z_x\, x = z$ and bcVar[[ ]] tells us that

$$[\![(\Gamma y : \sigma, x)]\!]^X_\phi = [\![(\Gamma x : \sigma, x)]\!]^X_\phi = \phi x$$

  while

$$[\![(\Gamma z : \sigma, z)]\!]^X_{\phi[z \mapsto \phi y]} = [\![(\Gamma z : \sigma, z)]\!]^X_{\phi[z \mapsto \phi x]} = (\phi[z \mapsto \phi x])(z) = \phi x.$$

  Since the two agree we have established this case.
- If $x \neq y$ then $\mathrm{ren}^z_y\, t = \mathrm{ren}^z_y\, x = x$ and bcVar[[ ]] tells us that

$$[\![(\Gamma y : \sigma, x)]\!]^X_\phi = \phi x$$

  while

$$[\![(\Gamma z : \sigma, x)]\!]^X_{\phi[z \mapsto \phi y]} = (\phi[z \mapsto \phi y])(x) = \phi x.$$

  Since the two agree once again we have established this case as well.

scAbs If t is an abstraction, say $\lambda x : \rho.\, u$, then we know there is a type γ that, given Γ, is derivable for u and we have $\tau = \rho \to \gamma$. By scAbs[[ ]] we have that $[\![(\Gamma y : \sigma, \lambda x : \rho.\, u)]\!]^X_\phi$ is a function from $[\![\rho]\!]^X$ to $[\![\gamma]\!]^X$ given by the assignment

$$a \mapsto [\![(\Gamma y : \sigma x : \rho, u)]\!]^X_{\phi[x \mapsto a]}.$$

We make another case distinction to calculate $\mathrm{ren}^z_x(\lambda x : \rho.\, u)$.

- If $x = y$ then $\mathrm{ren}^z_y(\lambda x : \rho.\, u) = \lambda z : \rho.\ \mathrm{ren}^z_x\, u$, and again by scAbs[[ ]]

$$[\![(\Gamma z : \sigma, \lambda z : \rho.\ \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi x]} : [\![\sigma]\!]^X \longrightarrow [\![\gamma]\!]^X$$

  is given by the assignment

$$a \mapsto [\![(\Gamma z : \sigma z : \rho, \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi x]} = [\![(\Gamma z : \rho, \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi x]},$$

  where we invoke Proposition 2.35 for the final equality, and using the same result we know that in this situation

$$[\![(\Gamma y : \sigma x : \rho, u)]\!]^X_{\phi[x \mapsto a]} = [\![(\Gamma x : \sigma x : \rho, u)]\!]^X_{\phi[x \mapsto a]} = [\![(\Gamma x : \rho, u)]\!]^X_{\phi[x \mapsto a]}.$$

  Since we know that z is distinct from $y = x$ we may invoke the induction hypothesis to conclude that

$$[\![(\Gamma z : \rho, \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi x]} = [\![(\Gamma x : \rho, u)]\!]^X_{\phi[x \mapsto a]}$$

  as required.
- If $x \neq y$ then $\mathrm{ren}^z_y(\lambda x : \rho.\, u) = \lambda x : \rho.\ \mathrm{ren}^z_y\, u$ and we also know that $\Gamma y : \sigma x : \rho = \Gamma x : \rho y : \sigma$, which we use below, and we also know that we may swap the order of x and z in the type environment. By scAbs[[ ]] we know that

$$[\![(\Gamma z : \sigma, \lambda x : \rho.\ \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi y]} : [\![\rho]\!]^X \longrightarrow [\![\gamma]\!]^X$$

  is given by the assignment

$$a \mapsto [\![(\Gamma z : \sigma, x : \rho, \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi y]} = [\![(\Gamma x : \rho, z : \sigma, \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi y]}.$$

  By the induction hypothesis we know that

$$[\![(\Gamma x : \rho z : \sigma, \mathrm{ren}^z_x\, u)]\!]^X_{\phi[z \mapsto \phi y]} = [\![(\Gamma x : \rho y : \sigma, u)]\!]^X_\phi = [\![(\Gamma y : \sigma x : \rho, u)]\!]^X_\phi$$

  as required.

scApp If t is an application, say rs, we may show the required statement directly from the induction hypothesis:

$$\begin{aligned}
[\![(\Gamma y:\sigma, rs)]\!]^X_\phi &= [\![(\Gamma y:\sigma, r)]\!]^X_\phi([\![(\Gamma y:\sigma, s)]\!]^X_\phi) && \text{scApp}[\![\,]\!] \\
&= [\![(\Gamma z:\sigma, \mathrm{ren}^z_y\, r)]\!]^X_{\phi[z\mapsto \phi y]}([\![(\Gamma y:\sigma, s)]\!]^X_{\phi[z\mapsto\phi y]}) && \text{ind hyp} \\
&= [\![(\Gamma z:\sigma, \mathrm{ren}^z_y\, r\, \mathrm{ren}^z_y\, s)]\!]^X_{\phi[z\mapsto\phi y]} && \text{scApp}[\![\,]\!] \\
&= [\![(\Gamma z:\sigma, \mathrm{ren}^z_y(rs))]\!]^X_{\phi[z\mapsto\phi y]} && \text{scAppren.}
\end{aligned}$$

Proposition 2.39

Proof. We know from Proposition 2.10 that under the assumptions stated we know that $\Gamma \vdash t[a/x]$ is also derivable so the Γ-denotation with respect to X and ϕ of $t[a/x]$ is defined.

The proof is by induction over the structure of the term t. We quantify over all type environments and valuations, and in the induction hypothesis we assume the property holds for proper subterms where one variable may have been renamed.

bcVar If t is a variable, say y, we have to make a case distinction.

- If $y = x$ then $t[a/x] = a$. We note that in this situation we must have $\sigma = \tau$ for $\Gamma, x:\sigma \vdash x:\tau$ to be derivable. We may use bcVar$[\![\,]\!]$ to calculate $[\![(\Gamma, t)]\!]^X_{\phi[x\mapsto [\![(\Gamma x:\sigma, a)]\!]^X_\phi]}$ is that element of $[\![\sigma]\!]^X$ given by
$$(\phi[x \mapsto [\![(\Gamma, a)]\!]^X_\phi])(x) = [\![(\Gamma, a)]\!]^X_\phi$$
which establishes the claim.
- If $y \neq x$ then $t[a/x] = y = t$. We check the denotations of the two given terms from the claim.
$$\begin{aligned}
[\![(\Gamma x:\sigma, y)]\!]^X_{\phi[x\mapsto [\![(\Gamma,a)]\!]^X_\phi]} &= \phi y && \text{bcVar}[\![\,]\!] \\
&= [\![(\Gamma, y)]\!]^X_\phi && \text{bcVar}[\![\,]\!] \\
&= [\![(\Gamma, y[a/x])]\!]^X_\phi && y[a/x] = y
\end{aligned}$$

scAbs If t is an abstraction, say $\lambda y:\rho.\, u$ we note that there is a type γ such that $\tau = \rho \to \gamma$ and $\Gamma, x:\rho \vdash u:\gamma$ is derivable. We have $(\lambda y:\rho.\, u)[a/x] = \lambda w:\rho.\, (\mathrm{ren}^w_y\, u)[a/x]$ where

$$w = \mathrm{freshv}(\mathrm{vars}(\lambda y.\, u) \cup \mathrm{vars}\, u \cup \{x\}).$$

Again we check the relevant denotations. For t we have by scAbs$[\![\,]\!]$ that

$$[\![(\Gamma x:\sigma, \lambda y:\rho.\, u)]\!]^X_{\phi[x\mapsto [\![(\Gamma,a)]\!]^X_\phi]}$$

is the function from $[\![\rho]\!]^X$ to $[\![\gamma]\!]^X$ given by the assignment

$$b \mapsto [\![(\Gamma x:\sigma y:\rho, u)]\!]^X_{\phi[x\mapsto [\![(\Gamma,a)]\!]^X_\phi][y\mapsto b]}.$$

On the other hand $[\![(\Gamma, \lambda w:\rho.\, (\mathrm{ren}^w_y\, u)[a/x])]\!]^X_\phi$ is a function with the same source and target given by the assignment

$$b \mapsto [\![(\Gamma w:\rho, (\mathrm{ren}^w_y)[a/x])]\!]^X_{\phi[w\mapsto b]}.$$

We may calculate, using $w \neq x$,

$$\begin{aligned}
&[\![(\Gamma w:\rho, (\mathrm{ren}^w_y)[a/x])]\!]^X_{\phi[w\mapsto b]} \\
&\quad = [\![(\Gamma w:\rho x:\sigma, \mathrm{ren}^w_y\, u)]\!]^X_{\phi[w\mapsto b][x\mapsto [\![(\Gamma w:\rho,a)]\!]^X_{\phi[w\mapsto b]}]} && \text{ind hyp} \\
&\quad = [\![(\Gamma x:\sigma w:\rho, \mathrm{ren}^w_y\, u)]\!]^X_{\phi[x\mapsto [\![(\Gamma w:\rho,a)]\!]^X_{\phi[w\mapsto b]}][w\mapsto b]} && \Gamma, x:\sigma, w:\rho = \Gamma, w:\rho, x:\sigma \\
&\quad = [\![(\Gamma x:\sigma y:\rho, u)]\!]^X_{\phi[x\mapsto [\![(\Gamma w:\rho,a)]\!]^X_{\phi[w\mapsto b]}][y\mapsto b]} && \text{Proposition 2.37}
\end{aligned}$$

We know that $w \notin \mathrm{fv}\, a$ and so from Proposition 2.36 that

$$[\![(\Gamma w:\rho, a)]\!]^X_{\phi[w\mapsto b]} = [\![(\Gamma, a)]\!]^X_{\phi},$$

so we have that the second assignment under consideration maps $b \in [\![\rho]\!]^X$ to

$$[\![(\Gamma x:\sigma y:\rho, u)]\!]^X_{\phi[x\mapsto [\![(\Gamma,a)]\!]^X_{\phi}][y\mapsto b]}.$$

Hence the two assignments agree.

scApp If t is an application, say rs then $rs[a/x] = (r[a/x])(s[a/x])$. In this situation we have

$$\begin{aligned}
&[\![(\Gamma x:\sigma, rs)]\!]^X_{\phi[x\mapsto [\![(\Gamma,a)]\!]^X_{\phi}]} \\
&\quad = [\![(\Gamma, r)]\!]^X_{\phi[x\mapsto [\![(\Gamma x:\sigma,a)]\!]^X_{\phi}]}([\![(\Gamma, s)]\!]^X_{\phi[x\mapsto [\![(\Gamma,a)]\!]^X_{\phi}]}) && \mathrm{scApp}[\![\,]\!] \\
&\quad = [\![(\Gamma, r[a/x])]\!]^X_{\phi}([\![(\Gamma, s[a/x])]\!]^X_{\phi}) && \text{ind hyp} \\
&\quad = [\![(\Gamma, (r[a/x])(s[a/x]))]\!]^X_{\phi} && \mathrm{scApp}[\![\,]\!] \\
&\quad = [\![(\Gamma, rs[a/x])]\!]^X_{\phi} && \mathrm{scApp}[\,].
\end{aligned}$$

Proposition 2.46

Proof. Both relations of interest are equivalence relations, and therefore it is sufficient to show the following for t, t', τ and Γ as given and an arbitrary (Γ, τ, ι)-context C:

- If $t \sim_\alpha t'$ then $t \simeq t'$. A simple proof by induction establishes that in this situation we have $C\{t\} \sim_\alpha C\{t'\}$ and so $t \simeq t'$.
- If $t \xrightarrow{\beta} t'$ then $t \simeq t'$. Again a simple proof by induction shows that in this situation we have $C\{t\} \xrightarrow{\beta} C\{t'\}$ and so $t \simeq t'$.
- If $t \xrightarrow{\eta} \Gamma t'$then $t \simeq t'$. This is established by Fact 1.

Technical Proofs from Chapter 3

Proposition 3.3

Proof. We only show that the statement works for a single step of β-reduction, the remainder is an easy proof by induction over the number of steps involved.

That statement is a proof by induction over the reason why β-reduction occurs, so it is once again a proof by induction over the structure of the term, but where we make more case distinctions than given by the term forming rules.

bcVarβ If we have a term of the form $\lambda x : \sigma.\, ta \xrightarrow{\beta} t[a/x]$ then we may proceed much as we did for Corollary 2.12 by first proving a result for substitution, which now has more cases since we have more rules for term formation, but none of the new cases are problematic.

bcPredβ If $\mathsf{pred}\,\bar{0} \xrightarrow{\beta} \bar{0}$ or $\mathsf{pred}\,\bar{s}n \xrightarrow{\beta} n$ then $\tau = \mathsf{nat}$, and we know that $\bar{0} : \mathsf{nat}$, while for the second case we know that in order for $\bar{s}n$ to be typeable we must have $n : \mathsf{nat}$ as well.

bcIfzβ If $\mathsf{ifz}\,0tu \xrightarrow{\beta} t$ or $\mathsf{ifz}\,\bar{s}ntu \xrightarrow{\beta} u$ then we know that in order for the starting terms to be typeable we must have that t and u are both of type nat,

bcRecβ If $\mathsf{rec}\,t \xrightarrow{\beta} t\,\mathsf{rec}\,t$ then in order for $\mathsf{rec}\,t$ to be typeable it must be of type $\tau \to \tau$. We then have that $\mathsf{rec}\,t$ is of type τ, as is $t\,\mathsf{rec}\,t$ by the application rule.

scPredβ If $t \xrightarrow{\beta} t'$ is the reason that $\mathsf{pred}\,t \xrightarrow{\beta} \mathsf{pred}\,t'$ then t must be of type nat, and by the induction hypothesis so is t'.

scsuccβ If $t \xrightarrow{\beta} t'$ is the reason that $\bar{s}t \xrightarrow{\beta} \bar{s}t'$ then we know once again that t must be of type nat, and once again the same is true for t.

scPredβ If $t \xrightarrow{\beta} t'$ is the reason that $\mathsf{ifz}\,trs \xrightarrow{\beta} \mathsf{ifz}\,t'rs$ then we know that t is of type τ and by the induction hypothesis so is t'.

scAppβ If $t \xrightarrow{\beta} t'$ is the reason that $tu \xrightarrow{\beta} t'u$. This case is shown in the same way as for Corollary 2.12.

Proposition 3.5

Proof. We show each part.

(a) The property follows more or less immediately from the corresponding property for the applicative preorder.

(b) If $t \trianglelefteq t'$ and both terms are of type nat then

$$(\lambda x : \mathsf{nat}.\, x)t \preceq (\lambda x : \mathsf{nat}.\, x)t'$$

by definition of $\trianglelefteq$. Since $(\lambda x : \mathsf{nat}.\, x)t \xrightarrow{\beta} t$, and the corresponding statement for t' in place of t we may use Proposition 3.4 to conclude that

$$t \preceq (\lambda x : \mathsf{nat}.\, x)t \preceq (\lambda x : \mathsf{nat}.\, x)t' \preceq t'.$$

If $t \trianglelefteq t'$ and both terms are of type $\sigma \to \tau$ we would like to invoke Proposition 3.4 and decompose this type into

$$\tau_1 \to \tau_2 \to \ldots \to \tau_n \to \mathsf{nat}.$$

By Proposition 3.4 it is sufficient to show that

$$tu_1 \ldots u_n \preceq t'u_1 \ldots u_n$$

for all terms $u_1, \ldots, u_n$ such that we may derive $\vdash u_i : \tau_i$ for $1 \leq i \leq n$. Assume we have terms u_i satisfying the assumptions just given. Then we have

$$\vdash \lambda f : \tau.\, f u_1 u_2 \ldots u_n : \tau \to \mathsf{nat},$$

and so $t \trianglelefteq t'$ implies

$$(\lambda f : \tau.\, f u_1 u_2 \ldots u_n)t \preceq (\lambda f : \tau.\, f u_1 u_2 \ldots u_n)t'.$$

But

$$(\lambda f : \tau.\, f u_1 u_2 \dots u_n) t \xrightarrow{\beta} t u_1 u_2 \dots u_n$$

and similarly for the terms where t is replaced by t' and so by

$$\begin{aligned} t u_1 \dots u_n &\preceq (\lambda f : \tau.\, f u_1 u_2 \dots u_n) t && \text{Proposition 3.4} \\ &\preceq (\lambda f : \tau.\, f u_1 u_2 \dots u_n) t' && \text{above} \\ &\preceq t' u_1 \dots u_n && \text{Proposition 3.4} \end{aligned}$$

This completes the proof.

Proposition 3.6

Proof. We prove this by induction over the structure of t, making use of the fact that we know that t is typeable. We quantify over all type environments and suitable valuation.

bcVar If t is a variable, say y, then there are two cases to consider. If $x \neq y$ then

$$d \longmapsto [\![(\Gamma, y)]\!]_{\phi[x \mapsto d]} = \phi y$$

is a constant function which is Scott-continuous by Exercise 47. Otherwise

$$d \longmapsto [\![(\Gamma, x)]\!]_{\phi[x \mapsto d]} = d$$

is the identity function which the same result identifies as being Scott-continuous.

bcZero If $t = \bar{0}$ then the function in question is also constant and so once again Scott-continuous.

scSucc If $t = \bar{\mathsf{s}} u$ then

$$d \longmapsto [\![(\Gamma, \bar{\mathsf{s}} u)]\!]_{\phi[x \mapsto d]} = \mathrm{succ}\, [\![(\Gamma, u)]\!]_{\phi[x \mapsto d]},$$

which is the composite of two Scott-continuous functions and so Scott-continuous once again by Exercise 47.

scPred If $t = \mathsf{pred}\, u$ we may argue in much the same way as in the previous case.

scIfz If $t = \mathsf{ifz}\, u r s$ then

$$d \longmapsto [\![(\Gamma, \mathsf{ifz}\, u r s)]\!]_{\phi[x \mapsto d]} = \mathrm{ifz}\, [\![(\Gamma, u)]\!]_{\phi[x \mapsto d]} [\![(\Gamma, r)]\!]_{\phi[x \mapsto d]} [\![(\Gamma, s)]\!]_{\phi[x \mapsto d]}.$$

This is a continuous assignment by Proposition 4.12.

scRec If $t = \mathsf{rec}\, u$ then

$$d \longmapsto [\![(\Gamma, \mathsf{rec}\, u)]\!]_{\phi[x \mapsto d]} = \mathsf{fix}\, [\![(\Gamma, u)]\!]_{\phi[x \mapsto d]},$$

and by the induction hypothesis this is again this is the composite of two Scott-continuous functions, where Proposition 4.14 gives us that fix is Scott-continuous, and so once again a Scott-continuous function by Exercise 47.

scAbs If $t = \lambda y : \sigma.\, u$ then for $e \in [\![s]\!]$ we have

$$d \longmapsto [\![(\Gamma, \lambda y : \sigma.\, u)]\!]_{\phi[x \mapsto d]} = (e \mapsto [\![(\Gamma, u)]\!]_{\substack{\phi[x \mapsto d] \\ [y \mapsto e]}}).$$

If $x = y$ then the assignment is

$$d \longmapsto (e \mapsto [\![(\Gamma, u)]\!]_{\phi[x \mapsto e]}) \ ,$$

which is a constant function and so Scott-continuous.
If $x \neq y$ then the assignment is equal to

$$d \longmapsto (e \mapsto [\![(\Gamma, u)]\!]_{\substack{\phi[y \mapsto e] \\ [x \mapsto d]}}),$$

which is continuous by the induction hypothesis.

scApp If $t = rs$ then

$$d \longmapsto [\![(\Gamma, rs)]\!]_{\phi[x \mapsto d]} = [\![(\Gamma, r)]\!]_{\phi[x \mapsto d]}([\![(\Gamma, s)]\!]_{\phi[x \mapsto d]}),$$

and since we know by the induction hypothesis that

$$d \longmapsto [\![(\Gamma, r)]\!]_{\phi[x \mapsto d]}$$

and

$$d \longmapsto [\![(\Gamma, s)]\!]_{\phi[x \mapsto d]}$$

are both Scott-continuous the result follows once again from Proposition 4.12.

Proposition 3.8

Proof. This is a proof by induction over the structure of the term t, where we quantify over all type environments and valuations, and in the induction hypothesis we assume the property holds for proper subterms where one variable my have been renamed.

We have significantly more cases to worry about than in Proposition 2.39, but the proof for the three cases that we are already familiar with, bcVar, scAbs and scApp are essentially the same as in the proof of that result, so here we only cover the cases that are new.

bcZero Since $\bar{0}$ does not contain any free variables substitution does not have any effect, and indeed

$$[\![(\Gamma, \bar{0})]\!]_{\phi[x \mapsto [\![(\Gamma, a)]\!]_\phi]} = 0 = [\![(\Gamma, \bar{0}[a/x])]\!]_\phi.$$

scSucc We have that

$$\begin{aligned}
[\![(\Gamma, \bar{s}t)]\!]_{\phi[x \mapsto [\![(\Gamma, a)]\!]_\phi]} &= \text{succ}\ [\![(\Gamma, t)]\!]_{\phi[x \mapsto [\![(\Gamma, a)]\!]_\phi]} && \text{scSucc}[\![\ ]\!] \\
&= \text{succ}\ [\![(\Gamma, \bar{s}(t[a/x]))]\!]_\phi && \text{ind hyp} \\
&= [\![(\Gamma, (\bar{s}t)[a/x])]\!]_\phi && \text{scSucc}[\ ].
\end{aligned}$$

scPred We have that

$$\begin{aligned}
[\![(\Gamma, \mathsf{pred}\, t)]\!]_{\phi[x \mapsto [\![(\Gamma, a)]\!]_\phi]} &= \text{pred}\ [\![(\Gamma, t)]\!]_{\phi[x \mapsto [\![(\Gamma, a)]\!]_\phi]} && \text{scPred}[\![\ ]\!] \\
&= \text{pred} = [\![(\Gamma, \mathsf{pred}\,(t[a/x]))]\!]_\phi && \text{ind hyp} \\
&= [\![(\Gamma, (\mathsf{pred}\, t)[a/x])]\!]_\phi && \text{scPred}[\ ].
\end{aligned}$$

scIfz We have that

$$\begin{aligned}
&[\![(\Gamma, \mathsf{ifz}\, trs)]\!]_{\phi[x\mapsto[\![]\!]]_{\phi(\Gamma,a)}} \\
&\quad = \mathrm{ifz}\, [\![(\Gamma,t)]\!]_{\phi[x\mapsto[\![]\!]]_{\phi(\Gamma,a)}} [\![(\Gamma,r)]\!]_{\phi[x\mapsto[\![]\!]]_{\phi(\Gamma,a)}} [\![(\Gamma,t)]\!]_{\phi[x\mapsto[\![]\!]]_{\phi(\Gamma,a)}} && \text{scIfz}[\![\,]\!] \\
&\quad = \mathrm{ifz}\, [\![(\Gamma,t[a/x])]\!]_{\phi} [\![(\Gamma,r[a/x])]\!]_{\phi} [\![(\Gamma,s[a/x])]\!]_{\phi} && \text{ind hyp} \\
&\quad = [\![(\Gamma, \mathsf{ifz}\, t[a/x] r[a/x] s[a/x])]\!]_{\phi} && \text{scIfz}[\![\,]\!] \\
&\quad = [\![(\Gamma, (\mathsf{ifz}\, trs)[a/x])]\!]_{\phi} && \text{scIfz}[\,].
\end{aligned}$$

scRec We have that

$$\begin{aligned}
[\![(\Gamma, \mathsf{rec}\, t)]\!]_{\phi[x\mapsto[\![(\Gamma,a)]\!]_{\phi}]} &= \mathsf{fix}\, [\![(\Gamma,t)]\!]_{\phi[x\mapsto[\![(\Gamma,a)]\!]_{\phi}]} && \text{scRec}[\![\,]\!] \\
&= \mathsf{fix}\, [\![(\Gamma,t[a/x])]\!]_{\phi} && \text{ind hyp} \\
&= [\![(\Gamma, \mathsf{rec}\,(t[a/x]))]\!]_{\phi} && \text{scRec}[\![\,]\!] \\
&= [\![(\Gamma, (\mathsf{rec}\, t)[a/x])]\!]_{\phi} && \text{scRec}[\,].
\end{aligned}$$

Lemma 3.15

Proof. We show each part

(a) We note that we have this property for $\tau = \mathsf{nat}$ since the definition of $\mathcal{A}$ at type nat means the condition is vacuously satisfied.
If τ is of the form ρs then we know that $k_\perp$ is the least element of $[\![\rho]\!] \Rightarrow [\![\sigma]\!]$, and we know that for all $e \in [\![\rho]\!]$ and all $u \in \mathsf{PCF\bar{T}rm}_\rho$ with $e\ \mathcal{A}_\rho\ u$ we have

$$\perp = k_\perp e\ \mathcal{A}_\sigma\ tu$$

by the induction hypothesis which implies that

$$k_\perp\ \mathcal{A}_{\rho\to\sigma}\ t$$

as required.

(b) This is a proof by induction over the type τ.

bcnat At the base type we have to show that if $d\ \mathcal{A}_{\mathsf{nat}}\ t$ then $\perp\ \mathcal{A}_{\mathsf{nat}}\ t$ since $\perp$ is the only element of $\mathbb{N}_\perp$ other than d itself that satisfies the given criterion. But that holds by part (i).

sc→ If τ is of the form $\rho \to \sigma$ then we know that d and d' are functions, and we use $f = d$ and $f' = d'$ to remind us of this fact. To show that $f'\ \mathcal{A}_{\rho\to\sigma}\ t$ we have to show that for all $e \in [\![\rho]\!]$ and all $u \in \mathsf{PCF\bar{T}rm}_\rho$ with $e\ \mathcal{A}_\rho\ u$ we have

$$f' e\ \mathcal{A}_\sigma\ tu.$$

We know that $f\ \mathcal{A}_{\rho\to\sigma}\ t$ and that means that we know that

$$f e\ \mathcal{A}_\sigma\ tu,$$

and since

$$f' e \leq f e$$

by the definition of the order on $[\![\rho]\!] \Rightarrow [\![\sigma]\!]$ we may use the induction hypothesis to conclude that

$$f' e\ \mathcal{A}_\sigma\ tu$$

as required.

(c) This is a proof by induction over the type τ.

bcnat Every directed subset of nat has a maximal element which gives its supremum and so there is nothing to show here.

sc→ If τ is of the form $\rho \to \sigma$ assume that $\mathcal{F}$ is a directed subset of $[\![\rho]\!] \Rightarrow [\![\sigma]\!]$ such that $f\ \mathcal{A}_{\rho\to\sigma}\ t$ for all $f \in \mathcal{F}$. We have to show that for all $e \in [\![\rho]\!]$ and all $u \in \mathsf{PCFTrm}_\rho$ with $e\ \mathcal{A}_\rho\ u$ we have

$$(\bigvee\!\uparrow \mathcal{F})e\ \mathcal{A}_\sigma\ tu.$$

Now

$$(\bigvee\!\uparrow \mathcal{F})e = \bigvee_{f\in\mathcal{F}}\!\uparrow f\,e$$

by Proposition 4.11 and

$$\bigvee_{f\in\mathcal{F}}\!\uparrow f\,e\ \mathcal{A}_\sigma\ tu$$

by the induction hypothesis, and so the required statement holds.

(d) We prove this also by induction over the type τ.

bcnat In the base case if $x \in \mathbb{N}_\perp$ and $x\ \mathcal{A}_{\mathsf{nat}}\ t$ then by definition of $\mathcal{A}$ we have that $x \in \mathbb{N}$ implies $t \xrightarrow{\beta\!\!\!\!\rightarrow} \bar{\mathsf{s}}^n\bar{0}$, and since $t \preceq t'$ we must also have $t' \xrightarrow{\beta\!\!\!\!\rightarrow} \bar{\mathsf{s}}^n\bar{0}$, and so $x_{\mathsf{nat}}{}'$ as required.

sc→ If we are in the step case, say the type $\rho \to \sigma$, then if $f \in [\![\rho \to \sigma]\!]$ and $d\ \mathcal{A}_{\rho\to\sigma}\ t$ then we know that for every e and u with $e\ \mathcal{A}_\rho\ u$ we have $f e \mathcal{A}_\sigma tu$. By the induction hypothesis that implies $f\,e\ \mathcal{A}_\sigma\ t'u$ and so by the definition of a logical relation we have $f\ \mathcal{A}_{\rho\to\sigma}\ t'$ as required.

Lemma 3.17

Proof. The proof is by induction over the term t. It bears some resemblance to the proof of Lemma 2.19 and we give a terser account where we would repeat justifications already provided there.

bcVar If t is a variable then it must be one of the variables x_i and then the result of carrying out the substitutions is the term a_i. In that situation we have that $[\![(\Gamma, x_i)]\!]_\phi = \phi x_i$ and we know $\phi x_i = d_i\ \mathcal{A}_{\alpha_i}\ a_i$ by assumption.

bcZero If $t = \bar{0}$ then the substitution has no effect and we have $[\![(\Gamma, \bar{0})]\!]_\phi = 0$ and we know that $0\ \mathcal{A}_{\mathsf{nat}}\ \bar{0}$ as required.

scSucc If $t = \bar{\mathsf{s}}u$ then we have that $\tau = \mathsf{nat}$ and

$$[\![(\Gamma, \bar{\mathsf{s}}u)]\!]_\phi = \mathrm{succ}\ [\![(\Gamma, u)]\!]_\phi,$$

and by the induction hypothesis we know that

$$[\![(\Gamma, u)]\!]_\phi\ \mathcal{A}_{\mathsf{nat}}\ u[a_1/x_1]\cdots[a_k/x_k].$$

By Lemma 3.14 this implies

$$[\![(\Gamma, \bar{\mathsf{s}}u)]\!]_\phi\ \mathcal{A}_{\mathsf{nat}}\ \bar{\mathsf{s}}(u[a_1/x_1]\cdots[a_k/x_k]) = \bar{\mathsf{s}}u[a_1/x_1]\cdots[a_k/x_k]$$

as required.

scPred If $t\,\mathsf{pred}\,u$ then we are in a similar situation as before and by much the same argument we obtain the required statement.

scIfz If $t = \mathsf{ifz}\,urs \in \mathsf{PCFTrm}$ we have another case where we may argue much as in the previous two cases, just noting now that the induction hypothesis applies to all three terms r, s and u.

scRec If $t = \mathsf{rec}\, u$ then we know that $\Gamma \vdash u : \tau \to \tau$ and the induction hypothesis tells us that

$$[\![(\Gamma, u)]\!]_\phi \; \mathcal{A}_{\tau\to\tau}\; u[a_1/x_1] \cdots [a_k/x_k].$$

By Lemma 3.16 this implies

$$[\![(\Gamma, \mathsf{rec}\, u)]\!]_\phi \; \mathcal{A}_\tau\; \mathsf{rec}\, u[a_1/x_1] \cdots [a_k/x_k] = \mathsf{rec}\, u[a_1/x_1] \cdots [a_k/x_k]$$

as required.

scAbs If $t = \lambda x : \rho.\, u$ then we know that $\tau = \rho \to \sigma$ for some types ρ, σ, and $\Gamma, x : \rho t \vdash u : \sigma$ is derivable and if we have that $d\; \mathcal{A}_\rho\; a$ then the induction hypothesis tells us that

$$[\![(\Gamma x : \rho, u)]\!]_{\phi[x\mapsto d]} \; \mathcal{A}_\sigma\; u[a_1/x_1] \cdots [a_k/x_k][a/x].$$

We obtain

$$[\![(\Gamma, \lambda x : \rho.\, u)]\!]_\phi \; \mathcal{A}_{\rho\to\sigma}\; (\lambda x : \rho.\, u)[a_1/x_1] \cdots [a_k/x_k][a/x]$$

from Lemma 3.16.

scApp If $t = rs$ then we know we can find a type σ such that $\Gamma \vdash r : \sigma \to \tau$ and $\Gamma \vdash s : \sigma$ are derivable, and the induction hypothesis tells us that

$$[\![(\Gamma, r)]\!]_\phi \; \mathcal{A}_{\sigma\to\tau}\; r[a_1/x_1] \cdots [a_k/x_k]$$

as well as

$$[\![(\Gamma, s)]\!]_\phi \; \mathcal{A}_\sigma\; r[a_1/x_1] \cdots [a_k/x_k].$$

By the definition of logical relation this means that

$$[\![(\Gamma, r)]\!]_\phi([\![(\Gamma, s)]\!]_\phi) \; \mathcal{A}_\tau\; r[a_1/x_1] \cdots [a_k/x_k] r[a_1/x_1] \cdots [a_k/x_k],$$

and we may rewrite each side to obtain

$$[\![(\Gamma, rs)]\!]_\phi \; \mathcal{A}_\tau\; (rs)[a_1/x_1] \cdots [a_k/x_k]$$

as required.

Technical Proofs from Chapter 4

Proposition 4.4

Proof. We note that the first statement is a special case of the second since given s in S we may use itself as a witness to obtain an element of S' that is greater than or equal to the given element, and so it is sufficient to show the second statement.

Assume we have two sets satisfying the given condition. Every upper bound of S' is also an upper bound of S since if we have $p \in P$ such that $s' \leq p$ for all $s' \in S$ we have that for all $s \in S$ we may find $s' \in S'$ with $s \leq s' \leq p$.

Hence if p is the least upper bound of S we know that it must in particular be less than or equal to every upper bound of S', which includes the least upper bound of S'.

Proposition 4.5

Proof. For the first statement we note that for all $S \in \mathcal{S}$ we have $\bigvee S \leq \bigvee \bigcup_{S \in \mathcal{S}} S$ by Proposition 4.4. Hence $\bigvee \bigcup_{S \in \mathcal{S}} S$ is an upper bound for the set consisting of all the $\bigvee S$. But if p is an upper bound for all the $\bigvee S$ then it is also an upper bound for all the elements of $\bigcup_{S \in \mathcal{S}} S$, and so $\bigvee \bigcup_{S \in \mathcal{S}} S \leq p$, which establishes that $\bigvee \bigcup_{S \in \mathcal{S}} S \leq p$ is the least upper bound of all the $\bigvee S$ for which $S \in \mathcal{S}$.

The second statement follows from the first since

$$\bigvee_{S \in \mathcal{S}} \bigvee S = \bigvee \bigcup_{S \in \mathcal{S}} S = \bigvee \bigcup_{T \in \mathcal{T}} T = \bigvee_{T \in \mathcal{T}} \bigvee T.$$

Proposition 4.8

Proof. We have to show that every directed subset has a least upper bound. Let S be a directed subset of P. Pick two elements $s, s' \in S$. We form a sequence of elements of S as follows:

- $s_0 = s$ and $s_1 = s'$.
- For $i \geq 1$ we form s_{i+1} as follows: If

  $$S \setminus \{p \in P \mid \exists t \in j \in \{0, 1, \ldots, i\}.\, p \leq s_j\}$$

  is empty then stop. Else pick an element s'' of that set, and set s_{i+1} to be any element of S with $s_i, s'' \leq s_{i+1}$. We know that at least one such element exists by directedness of S.

We know that this process has to stop after at most $n + 1$ steps by the assumption for P. Assume that k is the highest index we reached in the process. By construction we know that

- If $i \leq j$ then $s_i \leq s_j$.
- For all $s \in S$ we have $s \leq s_k$.

Hence s_k is the greatest element of S, and so equal to the supremum of S. Therefore every directed subset of P has a supremum.

Proposition 4.9

Proof. Assume that S is a directed subset of P and that q and q' are elements of its image under f. Then we may find p and p in S such that $q = f\,p$ and $q' = f\,p'$. Since S is directed we may find $p'' \in S$ with $p, p' \leq p''$ and since f is order-preserving we must have $f\,p, f\,p' \leq f\,p''$ which establishes the claim.

Proposition 4.10

Proof. In the proof of Proposition 4.8 we have seen that in a poset satisfying the given conditions every directed subset has a greatest element.

If S is a directed subset of P we know that it has a greatest element, say p, and so $\bigvee\!\uparrow S = p$.

We may now calculate $f \bigvee\!\uparrow S = f\,p$, and we claim that $\bigvee\!\uparrow_{s \in S} f\,s = f\,p$. Since f is order-preserving, for all $s \in S$ we have $f\,s \leq f\,p$ and so $\bigvee\!\uparrow_{s \in S} \leq f\,p$. But the element p occurs in in the set S and so we must have $f \bigvee\!\uparrow S = f\,p = \bigvee\!\uparrow_{s \in S} f\,s$ which means that f is indeed Scott-continuous.

Proposition 4.11

Proof. We show each part.

(a) Assume we have entities satisfying the given assumptions. Then for all $s \in S$ we have $s \leq \bigvee^{\uparrow} S$ and since f is order-preserving $f\,s \leq f \bigvee^{\uparrow} S$. Hence $f \bigvee^{\uparrow} S$ is an upper bound for the set $\{f\,s \mid s \in S\}$ and so the least upper bound of that set must be less than or equal to $f \bigvee^{\uparrow} S$.

(b) Strictly speaking our notation is a bit cheeky since it suggests that the function defined in the hint is indeed the desired least upper bound.

A priori it is not clear that said function is even well-defined. To start with we need to ensure that the set given in the definition is indeed directed to ensure that its least upper bound exists.

Given two elements of that set, say $f\,d$ and $f'd$, we know that since $\mathcal{F}$ is directed that we may find $f'' \in \mathcal{F}$ with $f, f' \leq f''$. In particular this tells us that for the given d we have $f\,d, f'd \leq f''d$ and so the set is indeed directed.

Secondly it is not clear that the given function is indeed an element of $D \Rightarrow E$ and so we have to show that it is order-preserving and Scott-continuous.

Assume that $d \leq d'$ for two elements from D. Then we have that for every element $f\,d$ of

$$\{f\,d \mid f \in \mathcal{F}\}$$

there exists $f\,d'$ in

$$\{f\,d' \mid f \in \mathcal{F}\}$$

with $f\,d \leq f\,d'$ and so by Proposition 4.4 we know that

$$\bigvee^{\uparrow}\{f\,d \mid f \in \mathcal{F}\} \leq \bigvee^{\uparrow}\{f\,d' \mid f \in \mathcal{F}\}.$$

Hence the proposed least upper bound of $\mathcal{F}$ is an order-preserving function.

To show that this function is Scott-continuous we have to show that it preserves least upper bounds of directed sets. Let S be a directed subset of D.

We calculate

$$\begin{aligned}
(\bigvee^{\uparrow}\mathcal{F})(\bigvee^{\uparrow} S) &= \bigvee^{\uparrow}\{f \bigvee^{\uparrow} S \mid f \in \mathcal{F}\} && \text{def } \bigvee^{\uparrow}\mathcal{F} \\
&= \bigvee_{f\in\mathcal{F}}^{\uparrow} f \bigvee^{\uparrow} S && \text{notational convention} \\
&= \bigvee_{f\in\mathcal{F}}^{\uparrow} \bigvee_{s\in S}^{\uparrow} f\,s && f \text{ Scott-continuous} \\
&= \bigvee_{s\in S}^{\uparrow} \bigvee_{f\in\mathcal{F}}^{\uparrow} f\,s && \text{Proposition 4.5} \\
&= \bigvee_{s\in S}^{\uparrow} (\bigvee^{\uparrow}\mathcal{F})(s) && \text{def } \bigvee^{\uparrow}\mathcal{F}
\end{aligned}$$

It remains to establish that the given function is indeed the least upper bound of $\mathcal{F}$. It is easy to see that for all $f \in \mathcal{F}$ we have $f \leq \bigvee^{\uparrow}\mathcal{F}$ since for every $d \in D$ we have

$$f\,d \leq \bigvee_{f'\in\mathcal{F}}^{\uparrow} f'd.$$

Hence $\bigvee^{\uparrow}\mathcal{F}$ is indeed an upper bound for $\mathcal{F}$. If g is an upper bound for $\mathcal{F}$ then we must have that for every $d \in D$ and every $f \in \mathcal{F}$ we have

$$f\,d \leq g\,d,$$

which tells us that gd is an upper bound for all the fd and so implies

$$\bigvee^{\uparrow}_{f \in \mathcal{F}} fd \leq gd,$$

and therefore according to the definition of the pointwise order we have

$$\bigvee^{\uparrow} \mathcal{F} = \bigvee^{\uparrow}_{f \in \mathcal{F}} f \leq g.$$

Hence $\bigvee^{\uparrow} \mathcal{F}$ is indeed the least upper bound of $\mathcal{F}$.

Proposition 4.12

Proof. Assume that we have dcpos and a function f as described and that S is a directed subset of D and $e \in E$. We use g for the given assignment.

$$\begin{aligned}
g \bigvee^{\uparrow} S &= (f(\bigvee^{\uparrow} S)) \bigvee^{\uparrow} S && \text{def } g \\
&= (\bigvee^{\uparrow}_{s \in S} fs) \bigvee^{\uparrow} S && f \text{ Scott-continuous} \\
&= \bigvee^{\uparrow}_{s \in S} (fs \bigvee^{\uparrow} S) && \text{Proposition 4.11} \\
&= \bigvee^{\uparrow}_{s \in S} \bigvee^{\uparrow}_{s' \in S} fss' && fs \in E \Rightarrow F \\
&= \bigvee^{\uparrow}_{s \in S} fss && \text{Proposition 4.5, S directed} \\
&= \bigvee^{\uparrow}_{s \in S} gs && \text{def } g
\end{aligned}$$

The second statement follows from the first by setting g to the constant function that maps every element of D to e.

Proposition 4.14

Proof. We start by showing that `fix` is order preserving. If $f, g : D \longrightarrow D$ are Scott-continuous functions with $f \leq g$ then for every $d \in D$ we have

$$fd \leq gd,$$

from which we obtain

$$f^2 d \leq fgd \leq g^2 d,$$

and a simple induction shows that

$$f^n d \leq g^n d$$

for all $n \in \mathbb{N}$. So in particular for each element $f^i \bot$ of

$$\{f^n \bot \mid n \in \mathbb{N}\}$$

there exists an element $g^i \bot$ of

$$\{g^n \bot \mid n \in \mathbb{N}\}$$

with

$$f^i \bot \leq g^i \bot$$

and therefore

$$\mathsf{fix}\, f = \bigvee\!\!\uparrow_{n\in\mathbb{N}} f^n \bot \leq \bigvee\!\!\uparrow_{n\in\mathbb{N}} g^n \bot = \mathsf{fix}\, g$$

by Proposition 4.4, which establish that fix is indeed order-preserving.

Let $\mathcal{F}$ be a directed subset of $D \Rightarrow D$. We next show that for $n \in \mathbb{N}$ and $d \in D$ we have that

$$(\bigvee\!\!\uparrow \mathcal{F})^n d = \bigvee\!\!\uparrow_{f\in\mathcal{F}} f^n d.$$

This is a proof by induction. If $n = 0$ then the result follows immediately from Proposition 4.11. For the step case we note that

$$\begin{aligned}
(\bigvee\!\!\uparrow \mathcal{F})^{n+1} d &= \bigvee\!\!\uparrow \mathcal{F}((\bigvee\!\!\uparrow \mathcal{F})^n d) && \text{def composition}\\
&= \bigvee\!\!\uparrow \mathcal{F}(\bigvee\!\!\uparrow_{f\in\mathcal{F}} f^n d) && \text{ind hyp}\\
&= \bigvee\!\!\uparrow_{f'\in\mathcal{F}} f'(\bigvee\!\!\uparrow_{f\in\mathcal{F}} f^n d) && \text{Proposition 4.11}\\
&= \bigvee\!\!\uparrow_{f'\in\mathcal{F}} \bigvee\!\!\uparrow_{f\in\mathcal{F}} f' f^n d && f' \text{ Scott-continuous}\\
&= \bigvee\!\!\uparrow_{f\in\mathcal{F}} f^{n+1} d && \text{Proposition 4.4}
\end{aligned}$$

The last equality deserves an explanation. Given f' and f in $\mathcal{F}$ by directedness of that set we know that there exists $f'' \in \mathcal{F}$ with $f, f' \leq f''$. Hence we have

$$f' f^n d \leq f''^{n+1} d,$$

so by Proposition 4.4 we have

$$\bigvee\!\!\uparrow_{f'\in\mathcal{F}} \bigvee\!\!\uparrow_{f\in\mathcal{F}} f' f^n d \leq \bigvee\!\!\uparrow_{f\in\mathcal{F}} f^{n+1} d.$$

We obtain the other inequality also from Proposition 4.4 since every $f^{n+1} d$ appears in the set over which the directed supremum is taken in the line above.

We have to show that

$$\bigvee\!\!\uparrow_{f\in\mathcal{F}} \mathsf{fix}\, f = \mathsf{fix}(\bigvee\!\!\uparrow \mathcal{F}).$$

We know how these are calculated and we make use of this information.

$$\begin{aligned}
\mathsf{fix}(\bigvee\!\!\uparrow \mathcal{F}) &= \bigvee\!\!\uparrow_{n\in\mathbb{N}} (\bigvee\!\!\uparrow \mathcal{F})^n \bot && \text{Exercise 49}\\
&= \bigvee\!\!\uparrow_{n\in\mathbb{N}} \bigvee\!\!\uparrow_{f\in\mathcal{F}} f^n \bot && \text{above}\\
&= \bigvee\!\!\uparrow_{f\in\mathcal{F}} \bigvee\!\!\uparrow_{n\in\mathbb{N}} f^n \bot && \text{Proposition 4.5}\\
&= \bigvee\!\!\uparrow_{f\in\mathcal{F}} \mathsf{fix}\, f && \text{Exercise 49}
\end{aligned}$$

as required, which establishes that the fixed point operator is Scott-continuous.