Alpha Avoidance

master thesis in computer science

by

Samuel Frontull

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Dr. Vincent van Oostrom
Department of Computer Science

Innsbruck, 6 September 2021

Master Thesis

Alpha Avoidance

Samuel Frontull (01418157)
samuel.frontull@student.uibk.ac.at

6 September 2021

Supervisor: Dr. Vincent van Oostrom

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum | Unterschrift

Abstract

When substitutions and bindings interact, there is a risk of undesired side-effects if the substitution is applied naïvely. This can be observed in different domains; the λ-calculus captures this phenomenon in an abstract setting. Its computation rule called β-reduction may require the renaming of bound variables via α-conversion, in order to avoid a variable capture. Various restrictions of the λ-calculus in which there is no need for α-conversion are known from the literature. This holds for finite developments with no redex creation, for affine λ-calculi with no duplication, for the weak λ-calculus with weak reduction, and for the safe λ-calculus, where the occurrence of free variables is restricted according to their type-theoretic order. This thesis presents a characterization the need for α-conversion by so-called α-paths. These α-paths predict potential name collisions by relying on the predictive power of legal paths. The existence of α-paths can be excluded for the aforementioned λ-calculi, if a suitable variable naming convention is adopted. This does not hold for the simply typed λ-calculus, where the need for α-conversion sometimes might be unavoidable, nor for the untyped λ-calculus, for which the question about α-avoidance is shown to be undecidable for the leftmost–outermost reduction strategy via a reduction from Post's correspondence problem.

Acknowledgments

First and foremost, I would like to thank you, *Vincent*, for giving me the opportunity to work on this exciting topic that allowed me to move on the front lines of a research field, develop something of my own and thus broaden my horizon. In the countless meetings and conversations, you always had the patience to give profound answers and explanations and showed me how to look at things from different perspectives. This has enabled me to develop myself enormously. Thank you for your wise guidance.

I would also like to thank you, *William*, for the pleasant and forthright communication, even though my inquiries concerned work you did quite some time ago. You sacrificed some of your valuable time because of me. Thank you.

My family was and is a source of energy for me in several respects. Each member in his own way. *Albert*† and *Carmela*, thank you for your unconditional support and belief in my abilities. *Michela*, thank you for always taking care of me and for making sure I could treat myself every once in a while. *Elias*, thank you for your exemplary attitude which also helped me to overcome my own challenges. This thesis is dedicated to you. *Magdalena*, thank you for allowing me pleasant rest periods while working on this project, for your regular stops in Innsbruck, and for helping me to compose the examples in Figure 1.1 and 1.2.

Contents

1 Introduction **1**

2 Preliminaries **8**
2.1 Terms . . . 8
2.2 Terms as graphs . . . 11
2.3 Occurrences, scopes and bindings . . . 12
2.4 Substitution and α-conversion . . . 13
2.5 β-reduction . . . 17

3 The need for α-conversion characterized by α-paths **22**
3.1 Intuition . . . 22
3.2 Legal paths . . . 30
3.3 α-paths . . . 35

4 Finite Developments **41**

5 The affine λ-calculus **44**

6 The weak λ-calculus **47**

7 The safe λ-calculus **49**

8 The simply typed λ-calculus **61**

9 The untyped λ-calculus **63**

10 Conclusion **70**

Bibliography **71**

A Appendix **73**
A.1 Alpha avoidance in the μ-calculus . . . 73
A.2 Naive replacement and α-equivalence . . . 76
A.3 De Bruijn indices and α-avoidance . . . 78
A.4 PCP encoded in the λ-calculus . . . 80

1 Introduction

The substitution of fungibles[1] is a key concept in different areas. In computer science, substitution is a fundamental concept. It is, for example, the core operation for computation in λ-calculus, applied by compilers to optimize programs and, in general, the key for reasoning with logical expressions. In music theory, the substitution of chords is a technique that composers use to add variety to their arrangements. In linguistics, anaphors substitute the subject they refer to and are used to avoid repetitions.

Even if these notions of substitution have different domains, they all share the following property: undesired side-effects may arise if the substitution is applied naïvely. To get an idea of what can happen, let's look at the following examples.

Program optimization It is part of a good coding style to write readable code. This often induces outsourcing code sections into their own functions. The downside of having additional functions is that each function call comes with some overhead. Functions operate on their own stack frame, which needs to be set up at each function call. A good compiler should therefore inline many of these calls to recover an efficiently executable program [25]. Inlining means to replace the function call by its body, avoiding the call-overhead mentioned above. This substitution can go wrong if done in a naïve way. Listing 1.1 shows the code of a function that computes the least significant digit of the scaling of a number by a factor `a`. Listing 1.2 shows an inlined version where the function call to `scale` was replaced by its body.

```
let a = 3;;

let scale x =
  let b = abs(x) in
    a * b

let scale_lsd x =
  let a = 10 in
    (scale x) mod a;;
```

Listing 1.1: Example of an Ocaml program.

```
let a = 3;;

let scale_lsd_inlined x =
  (* variable shadowing *)
  let a = 10 in
    (
      let b = abs(x) in
      a * b
    ) mod a;;
```

Listing 1.2: Inlined version (incorrect).

Unfortunately, this optimization attempt went wrong. Why? Because at the moment when the modulo is computed, the value of the scaling factor `a` is not `3` anymore, but `10` (it got captured). Hence, this transformation does not preserve the original semantics of the program. Such name capture is a major issue for any compiler [25] and a solution to

[1]We intend substitutions that are supposed to preserve certain properties.

it is variable *renaming*. By renaming specific variables appropriately, inlining can safely be applied. Variable renaming requires the supply of *fresh* variables. By renaming the global `a` to a fresh variable `n`, as shown in Listing 1.3 and 1.4, we can overcome this issue.

```
let n = 3;;

let scale x =
  let b = abs(x) in
    a * b

let scale_lsd x =
  let a = 10 in
    (scale x) mod n;;
```

Listing 1.3: Program equivalent to Listing 1.1

```
(* variable renamed *)
let n = 3;;

let scale_lsd_inlined x =
  let a = 10 in
    (
      let b = abs(x) in
      a * b
    ) mod n;;
```

Listing 1.4: Inlined version (correct).

Predicate logic As already mentioned, side-effects caused by a naïve substitution do not only occur in program optimizations but also in other environments and formalisms with bindings. In predicate logic, we distinguish between syntactic and semantic substitution. The difference between these different types of substitution can be illustrated with the following example inspired by Lamport [21]:

$$P = \forall y.\exists x.x > y \qquad\qquad y = 2x \tag{1.1}$$

Statement P claims that for all y we can find a bigger x. Assume we would like to prove this statement by natural deduction. The first step would be to take an instance for y to eliminate the $\forall$-quantifier. In principle, y could also be replaced by $2x$. The syntactic substitution would lead to the following conclusion:

$$\exists x.x > 2x \tag{1.2}$$

and this clearly is unsatisfiable. This wrong conclusion is due to a variable capture. The existential quantifier is meant to bind only the first occurrence of the variable x and not both. In this case, renaming is required before applying the substitution. A correct semantic substitution could be $\exists z.z > 2x$, which still has the chance of being satisfiable.

Music Theory This phenomenon also occurs in music theory within the concept of chord-substitution[2]. Chord-substitutions are used to add some variety to the music. This technique consists of replacing a chord with another one that fits the context. Advanced knowledge is required to understand this technique fully, but the idea is easily explainable if basics concepts are known.

Accidentals[3] are used in musical notes to raise ($\sharp$) and lower ($\flat$) notes or to cancel ($\natural$) previous ones. Their scope ranges from the immediately following note to the end of the bar. As accidentals control the pitch of occurrences of notes, they act as binders. As we

[2] https://en.wikipedia.org/wiki/Chord_substitution
[3] https://en.wikipedia.org/wiki/Accidental_(music)

have already seen, it can get dangerous if substitutions and bindings interfere and it is no different here. Figure 1.1 shows an example of a cadence in C major. The C major (first note) of this cadence could be substituted by a Cmaj7, as shown in Figure 1.2.

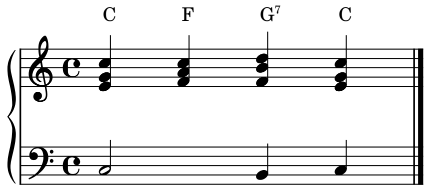


Figure 1.1: Cadence in C major

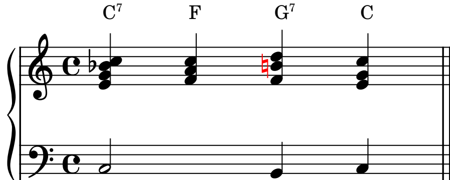


Figure 1.2: Chord-substitution

If we would naïvely apply this substitution, we would have an unintended side-effect. The Cmaj7 introduces an accidental (♭) that would also control the H-note in the third chord Gmaj7. The scope of this accidental has, therefore, to end before that chord. This is why the additional accidental (♮) marked in red in Figure 1.2 is needed. In this way we can preserve the right harmony.

Linguistics In linguistics, there is also a notion of binding. Subjects act as binders and words like *it, she, he, they, . . .*, called *anaphors*, are references to them. An illustrative example for the notion of binding in linguistics is visualized in Figure 1.3 (lhs). In the sentence *Bob asks Alice whether he can sing*, the word *he* refers back to its binder Bob, as denoted by the common index 1.

Bob$_{1}$ asks Alice$_{2}$ whether he$_{1}$ can sing.

Bob$_{1}$ asks John$_{2}$ whether he$_{2(1)}$ can sing.

Figure 1.3: Naïve name substitution can change the meaning of a sentence.

If we have to express, that Bob asks the same question to John, the instinctive approach would be to simply substitute John for Alice in the previous sentence, as done in Figure 1.3 (rhs). But Alice cannot simply be replaced by a male name like John, as we then have ambiguous pronouns. In *Bob asks John whether he can sing* he could refer to both John and Bob. This permits different interpretations and can lead to misunderstandings. This is the reason why the binding theory [10] was established for linguistics. If in the sentence *Bob asks Alice whether he can sing* we want to make the name of the person being asked (Alice) fungible, then one solution could be to reformulate it to something like *Bob wants to know whether he can sing and asks `<NAME>`*.

λ-calculus All the examples presented have illustrated problematic side-effects of naïve substitutions interfering with bindings. In all these situations, some effort was needed to allow a safe substitution (renaming of variables, ending of scope, using a different formulation) but it happily was always possible. The λ-calculus captures this problem in

an abstract setting. Its computation rule called β-reduction is nothing but the application of substitutions. We can have binding connections between λ-abstractions and variables.

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

Figure 1.4: A binding illustrated by the red arrow.

β-reduction has also to be applied in a way that avoids undesired name collisions with free occurrences of variables (*variable capture*). The binding connections could be broken by the naïve substitution, precisely as in the examples shown above. This would lead to inconsistent results, from which anything could be deduced [3, Fallacy 2.1.10]. To avoid a variable capture, we rename bound variables via α-conversion, before applying a substitution.

Figure 1.5 shows two reduction sequences, an incorrect one $\boxed{1}$, where the substitution is naïvely applied and the initially free variable y gets captured (this naïve substitution creates a binding connection that did not exist before), and a correct one $\boxed{2}$, where α-conversion is applied before the actual substitution.

$$\begin{array}{lll} (\lambda x.\lambda y.x)\ y & \boxed{1} & \not\to_{\beta} \lambda y.y \\ & \boxed{2} & \to_{\alpha} \lambda x.(\lambda z.x)\ y \to_{\beta} \lambda z.y \end{array}$$

Figure 1.5: Naïve substitution can lead to a variable capture.

The need for α-conversion seems to be random, unpredictable and in some cases it is even unavoidable – but this is not always the case. Various restrictions of the λ-calculus in which the need for α-conversion is avoidable are known from the literature. By adhering to a variable naming convention and limiting the reduction length, restricting the occurrence of variables, modifying the reduction rule, or by introducing an explicit notion of scope we can allow α-free computations. This holds for example for developments [3, Definition 11.2.11] like the underlined λ-calculus with no redex-creation, for affine λ-calculi with no duplication, for the weak λ-calculus with weak reduction, for the safe λ-calculus [9] where the occurrence of free variables is restricted according to their type-theoretic order and for the *adbmal*-calculus[4] with an explicit symbol for ending scopes [18]. Since all these different calculi share this common property, an interesting research question is whether there is a uniform characterization to it. Our answer to this question are the so-called *α-paths* presented in this thesis.

These α-paths predict potential name collisions and thus the need for α-conversion. They are a generalization of the self-capturing chains[5] introduced by Vincent van Oostrom in [17] for the μ-calculus (Appendix A.1) and rely on a notion of *legal paths* due to Asperti and Guerrini [2]. Via the α-paths, we can prove for the above mentioned calculi that no α-conversion is ever needed if a suitable variable naming convention is adopted.

[4] Not covered in this thesis.
[5] which were in turn inspired by Melliès' notion of gripping

Why α-avoidance? There are multiple reasons why α-free computations are preferable. One important reason is that it enables a more efficient computation based on *naïve* substitutions. *Naïve* substitutions or *capture-permitting* substitutions, in contrast to *capture-avoiding* substitutions, do not care about potential name collisions. It is in the nature of things that this results in a faster computation.

Another benefit will be noticeable at the latest as soon as one thinks about how to implement β-reduction for the ordinary lambda calculus. α-conversion requires the supply of fresh variables, which is a non-trivial problem (see for example [31]). The need for fresh variables itself is not the real issue. Rather it is the implicit requirement to have an unbounded supply. In plain terms, this means that one is forced to either provide a large list of fresh variables, as the exact needs cannot be estimated, or to introduce some impurity, as for example done in [24]. A function generating fresh variable names could, for example, make use of a counter that is incremented at each function call. Both approaches are not ideal, the former will overestimate the resource usage (and still can fail), and the latter is inconsistent with the requirement of stateless and deterministic computations.

The number of α-conversions required to reduce a term to normal form can be quadratic in the size of the input. For example, the exponentiation of Church numerals [3, Definition 6.4.4] reduced using the leftmost–outermost reduction strategy requires $n/2$ α-conversions where n is the number of β-conversions, if both numerals use the same variable name for the inner abstraction. A single initial α-conversion would allow an α-free reduction to normal form and save much effort in this case. This is another strong argument that shows that the initial manipulation of terms is not only a shift of a dynamic problem to a static one, but a substantial retrenchment.

b^N	1	2	3	4	5	6	7	...
1	1/2	1/2	1/2	1/2	1/2	1/2	1/2	
2	2/4	3/6	4/8	5/10	6/12	7/14	8/16	
3	3/6	7/14	13/26	21/42	31/62	43/86	57/114	
4	4/8	15/30	40/80	85/170	156/312	259/518	400/800	
5	5/10	31/62	121/242	341/682	781/1562	1555/3110	2801/5602	
6	6/12	63/126	364/728	1365/2730	3906/7816	9331/18662	19608/39216	
...								

Table 1.1: Number of α-conversions/β-conversions needed for the exponentiation b^N

Example 1.1. The exponentiation 2^2 in λ-calculus using Church numerals can be computed by applying 2 to itself (2 2). The 2 in church numerals corresponds to $\lambda fx.f\ (f\ x)$. Figure 1.6 shows the computation of 2^2 using leftmost-outermost reduction. On the left-hand side α-conversion is needed three times in the six β-conversions (written as a separate step), for the α-equivalent term on the right-hand side, no α-conversion is needed.

Last but not least, the importance of the relation of the input and the output of a

$$
\begin{array}{ll}
 & \underline{(\lambda f x.f\ (f\ x))\ (\lambda f x.f\ (f\ x))} \\
\to_\beta & \lambda x.\underline{(\lambda f x.f\ (f\ x))\ ((\lambda f x.f\ (f\ x))\ x)} \\
\to_\alpha & \lambda x.\underline{(\lambda f x'.f\ (f\ x'))\ ((\lambda f x.f\ (f\ x))\ x)} \\
\to_\beta & \lambda x.\lambda x'.(\underline{(\lambda f x.f\ (f\ x))\ x})\ (((\lambda f x.f\ (f\ x))\ x)\ x') \\
\to_\alpha & \lambda x.\lambda x'.(\underline{(\lambda f x'.f\ (f\ x'))\ x})\ (((\lambda f x.f\ (f\ x))\ x)\ x') \\
\to_\beta & \lambda x.\lambda x'.\underline{(\lambda x'.x\ (x\ x'))\ (((\lambda f x.f\ (f\ x))\ x)\ x')} \\
\to_\beta & \lambda x.\lambda x'.x\ (x\ ((\underline{(\lambda f x.f\ (f\ x))\ x})\ x')) \\
\to_\alpha & \lambda x.\lambda x'.x\ (x\ ((\underline{(\lambda f x'.f\ (f\ x'))\ x})\ x')) \\
\to_\beta & \lambda x.\lambda x'.x\ (x\ (\underline{(\lambda x'.x\ (x\ x'))\ x'})) \\
\to_\beta & \lambda x.\lambda x'.x\ (x\ (x\ (x\ x')))
\end{array}
$$

$$
\begin{array}{ll}
 & \underline{(\lambda f y.f\ (f\ y))\ (\lambda f x.f\ (f\ x))} \\
\to_\beta & \lambda y.\underline{(\lambda f x.f\ (f\ x))\ ((\lambda f x.f\ (f\ x))\ y)} \\
\to_\beta & \lambda y.\lambda x.(\underline{(\lambda f x.f\ (f\ x))\ y})\ (((\lambda f x.f\ (f\ x))\ y)\ x) \\
\to_\beta & \lambda y.\lambda x.\underline{(\lambda x.y\ (y\ x))\ (((\lambda f x.f\ (f\ x))\ y)\ x)} \\
\to_\beta & \lambda y.\lambda x.y\ (y\ ((\underline{(\lambda f x.f\ (f\ x))\ y})\ x)) \\
\to_\beta & \lambda y.\lambda x.y\ (y\ (\underline{(\lambda x.y\ (y\ x))\ x})) \\
\to_\beta & \lambda y.\lambda x.y\ (y\ (y\ (y\ x)))
\end{array}
$$

Figure 1.6: Different computations of 2^2 using leftmost-outermost reduction.

computation should not be underestimated. Proofs often only work out if they argue modulo α. In α-free reductions, one can argue with syntactic equality instead of α-equivalence which is a stronger statement. This can facilitate proofs. The relation to the input is also important for debugging. Assume some α-conversion is needed at some point during computation or that a compiler decides to rename variables in order to prevent the need for α-conversion. In both cases, it will be hard for the programmer to trace back errors concerning computations with the renamed variables, as they do not appear in the source code. Proof assistants also have to deal with this problem. Isabelle[6], for example, uses additional labels (colors) for variables to distinguish between user-defined and internal variables to avoid name collisions.

de Bruijn's namefree calculus There is an alternative representation of λ-terms due to de Bruijn [14]. In its namefree calculus indices are used instead of variables and there is therefore no need to rename variables. One may therefore think this also means to avoid α-conversion, but that's not the case as the problem of name collisions is only transposed to the problem of index collisions. Indices may need to be updated when contracting redexes. To get a deeper understanding why that is the case, interested readers are referred to Appendix A.3.

From a dynamic to a static problem This thesis is about α-avoidance, but what does it mean to avoid α? What we aim for in this thesis is to characterize the need for α-conversion in different λ-calculi and to find out more about which circumstances lead to naming problems. An interesting question, for example, is whether we could define naming conventions or reduction strategies that allow reducing λ-terms in a naïve way. In the end, this can allow moving a dynamic problem (the ad hoc renaming in a β-step) to a static one (finding an α-equivalent λ-term for which every reduction sequence from it is α-free). α-avoidance, as we use it in this thesis, does not mean completely getting

[6] `https://isabelle.in.tum.de/`

rid of this operation but avoiding it in the dynamic process of β-reduction.

Outlook In Chapter 2 we present some basic concepts of the λ-calculus, based on the unrestricted, untyped λ-calculus. We also give a precise definition of what it means to avoid α or that α can be avoided. In Chapter 3 we introduce the so-called α-paths that can be used to characterize the need for α-conversion. These α-paths will then be instantiated to different λ-calculi: developments (Chapter 4), the affine λ-calculus (Chapter 5), the weak λ-calculus (Chapter 6), the safe λ-calculus (Chapter 7), the simply-typed λ-calculus and finally for the untyped λ-calculus (Chapter 9) for which we show that the question about α-avoidance is undecidable for the leftmost–outermost reduction strategy.

2 Preliminaries

This chapter presents some basic concepts of the λ-calculus, based on the unrestricted, untyped λ-calculus. This calculus is well-known and well-studied. In this chapter, we will only introduce the main concepts relevant for this topic. We refer the interested readers to [3, 11] and [30, Chapter 10] for a deeper understanding. We will conclude with giving the definition of what it means to avoid α or that α can be avoided, the main notion investigated in this thesis. Understanding the different concepts for the untyped λ-calculus should give a good foundation for the rest of this thesis, as the more restrictive calculi discussed in this thesis add only slight modifications of some definitions.

2.1 Terms

Definition 2.1. The set of λ-*terms* Λ is inductively defined as:

$$\begin{array}{rcl} x \in \mathcal{V} & \to & x \in \Lambda, \\ M, N \in \Lambda & \to & (M\,N) \in \Lambda, \\ M \in \Lambda, x \in \mathcal{V} & \to & (\lambda x.M) \in \Lambda. \end{array}$$

where $\mathcal{V}$ is a countably infinite set of variables. Sometime we also consider λ-terms over a set of constants Γ [30]. In that case we write Λ_Γ and have the additional case

$$\Gamma \in \Lambda_\Gamma$$

If Γ is a singleton we write Λ_c, where c is the constant.

Convention 2.2. For readability purposes some conventions are used:

- outermost brackets are omitted
- applications associate to the left
- nested abstractions can be combined

Example 2.3. The following are valid λ-terms. The right hand side shows the same term, where the above mentioned notational conventions are applied:

$$\begin{array}{rcl} x & = & x \\ (x\,y) & = & x\,y \\ (\lambda x.(x\,y)) & = & \lambda x.x\,y \\ ((\lambda x.(x\,y))\,x) & = & (\lambda x.x\,y)\,x \\ (((\lambda x.(\lambda y.(x\,y)))\,y)\,x) & = & (\lambda xy.x\,y)\,y\,x \end{array}$$

Definition 2.4. The set of *subterms* $\mathcal{S}ub(M)$ of a λ-term M is inductively defined as:

$$\mathcal{S}ub(M) = \begin{cases} \{x\} & \text{if } M = x \\ \mathcal{S}ub(N) \cup \{M\} & \text{if } M = \lambda x.N \\ \mathcal{S}ub(N_1) \cup \mathcal{S}ub(N_2) \cup \{M\} & \text{if } M = N_1\,N_2 \end{cases}$$

Example 2.5. Let $M = (\lambda xy.x\,y)\,y\,x$. Then

$$\mathcal{S}ub(M) = \{M, (\lambda xy.x\,y)\,y, (\lambda xy.x\,y), (\lambda y.x\,y), x\,y, x, y\}$$

As we can see in Example 2.5, x and y are subterms of the term M, but if we talk about these subterms, then we do not know exactly which subterm is intended as both occur twice. This is why we need positions. Positions allow us to talk about specific subterms in a λ-term M.

Definition 2.6. A *position* in a λ-term is a finite sequence of 1s and 2s. The set of positions $\mathcal{P}os(M)$ of a λ-term M is inductively defined as:

$$\mathcal{P}os(M) = \begin{cases} \{\epsilon\} & \text{if } M \text{ is a variable} \\ \{\epsilon\} \cup \{1 \cdot \mathcal{P}os(N)\} & \text{if } M = \lambda x.N \\ \{\epsilon\} \cup \{1 \cdot \mathcal{P}os(N_1)\} \cup \{2 \cdot \mathcal{P}os(N_2)\} & \text{if } M = N_1\,N_2 \end{cases}$$

where $\cdot$ denotes string concatenation (to all positions in the set).

Example 2.7. Let $M = (\lambda xy.x\,y)\,y\,x$. Then

$$\mathcal{P}os(M) = \{\epsilon, 1, 11, 111, 1111, 11111, 11112, 2, 12\}$$

Definition 2.8. A position p is said to be a *strict prefix* of a position q, if $q = p \cdot q'$ where q' is non-empty. We use the notation $p \prec q$ to denote that p is a prefix of q.

Example 2.9. $\epsilon \prec 1$ and $11 \prec 1121$ but $11 \not\prec 1221$ and $1 \not\prec 1$.

Definition 2.10. A position p is said to be a *prefix* of a position q, if $p = q$ or $p \prec q$. We use the notation $p \preceq q$ to denote that p is a prefix of q.

Example 2.11. $1 \preceq 1$.

Definition 2.12. Two positions p, q are said to be *parallel*, denoted by $p \parallel q$, if $p \not\preceq q \wedge q \not\preceq p$.

Example 2.13. $12 \parallel 21$ and $1211 \parallel 2$ but $2211 \nparallel 2$

Proposition 2.14. *Positions that start with a distinct number are parallel.*

Proof. Assume $p = c_1 \cdot p'$ and $q = c_2 \cdot q'$ where $c_1 \neq c_2$. Then $p \neq c_2 \cdot q' \cdot s = q \cdot s$ and $q \neq c_1 \cdot p' \cdot s = p \cdot s$. $\square$

Proposition 2.15. *If* $p \parallel q$, *then* $s \cdot p \parallel s \cdot q$.

Proof. Assume $p \parallel q$. This means that $p \not\leq q$ and $q \not\leq p$.

$$\begin{array}{rlcl} & p \not\leq q & \wedge & q \not\leq p \\ \Longrightarrow & \neg\exists q'.q = p \cdot q' & \wedge & \neg\exists p'.p = q \cdot p' \\ \Longrightarrow & \neg\exists q'.s \cdot q \neq s \cdot p \cdot q' & \wedge & \neg\exists p'.s \cdot p \neq s \cdot q \cdot p' \\ \Longrightarrow & s \cdot p \not\leq s \cdot q & \wedge & s \cdot q \not\leq s \cdot p \\ \Longrightarrow & s \cdot p \parallel s \cdot q & & \end{array}$$

□

Proposition 2.16. *Let $M\,N$ be a λ-term. Then the positions in M and N are pairwise parallel. Stated formally: $\forall p \in \mathcal{P}os(M), q \in \mathcal{P}os(N).p \parallel q$.*

Proof. Let $p \in \mathcal{P}os(M)$ and $q \in \mathcal{P}os(N)$ in a λ-term $M\,N$, then $p = s \cdot 1 \cdot p'$ and $q = s \cdot 2 \cdot q'$, where s is the position of the application. We know that $1 \cdot p' \parallel 2 \cdot q'$ as they start with a different number and by Proposition 2.15 we can conclude that $p \parallel q$. □

Definition 2.17. A position p is set to be *left* of a position q, written as $p \parallel_l q$, if $p = s \cdot 1 \cdot p'$ and $q = s \cdot 2 \cdot q'$ where s is a common prefix and p', q' are arbitrary.

Proposition 2.18. *If $p \parallel_l q$, then $p \parallel q$.*

Proof. Since we have $p \parallel_l q$, we can write $p = s \cdot 1 \cdot p'$ and $q = s \cdot 2 \cdot q'$. From Proposition 2.14 we know that $1 \cdot p' \parallel 2 \cdot q'$ and by Proposition 2.15 we can conclude that $s \cdot 1 \cdot p' \parallel s \cdot 2 \cdot q'$, so $p \parallel q$. □

Proposition 2.19. *$\parallel_l$ is transitive.*

Proof. Assume we have $p \parallel_l q$ and $q \parallel_l r$. By Definition 2.18 we have $p = s \cdot 1 \cdot p'$ and $q = s \cdot 2 \cdot q'$ for some s, p', q'. Since we have $q \parallel_l r$, we must have that $s = t \cdot 1 \cdot s'$ and $r = t \cdot 2 \cdot s'$, for some t, s'. This results in $p = t \cdot 1 \cdot s' \cdot 1 \cdot p'$. By Definition 2.18 it follows that $p \parallel_l r$. □

Definition 2.20. We write $M|_p$ for the subterm of a λ-term M at position p.

Example 2.21. $(\lambda xy.x\,y)\,y\,x|_1 = (\lambda xy.x\,y)\,y$, $(\lambda xy.x\,y)\,y\,x|_{12} = y$

Definition 2.22. We write $M(p)$ for the *symbol* of a λ-term M at position p. It is defined as:

$$M(p) = \begin{cases} x & \text{if } M|_p = x \\ \lambda x & \text{if } M|_p = \lambda x.N \\ @ & \text{if } M|_p = N_1\,N_2 \end{cases}$$

Example 2.23. $\lambda y.z(\epsilon) = \lambda y$

Definition 2.24. The set of all symbols of a λ-term M is defined as:

$$\mathcal{S}ym(M) = \{M(p) \text{ where } p \in \mathcal{P}os(M)\}$$

Example 2.25. $\mathcal{S}ym(\lambda x.(\lambda y.x\,y)\,z) = \{\lambda x, \lambda y, x, y, z\}$.

2.2 Terms as graphs

λ-terms can also be represented via a labeled graph. The set of vertices and edges of this graph are introduced next.

Definition 2.26. The set of *vertices* $\mathcal{V}(M)$ of a λ-term M is defined as:

$$\mathcal{V}(M) = \mathcal{P}os(M)$$

Definition 2.27. The *edges* $E(M, p)$ in a λ-term M prefixed by p is a set of pairs inductively defined as:

$$E(M,p) = \begin{cases} \{\} & \text{if } M = x \\ \{(p, p \cdot 1), (p, p \cdot 2)\} \cup E(N_1, p \cdot 1) \cup E(N_2, p \cdot 2) & \text{if } M = N_1\, N_2 \\ \{(p, p \cdot 1)\} \cup E(N, p \cdot 1) & \text{if } M = \lambda x.N \end{cases}$$

where p serves as accumulator. We write $E(M)$ as abbreviation for $E(M, \epsilon)$.

Definition 2.28. The set of labeled symbols $\mathcal{LS}(M)$ of a λ-term M is defined as:

$$\mathcal{LS}(M) = \{(v, M(v)) \mid v \in \mathcal{V}(M)\}$$

where $M(v)$ is the symbol at position v. We write $M(v)^v$ for short to denote such symbols.

Remark 2.29. We may call labeled symbols *nodes* and more specifically *@-nodes*, if the label is a @, λ-*nodes* if the label is a λx or *v-nodes* if the label is a variable.

Definition 2.30. A *labeled graph* $G(M)$ of a λ-term M is a pair consisting of a set of labeled symbols $\mathcal{LS}(M)$ and a set of edges $E(M)$ in M.

$$G(M) = (\mathcal{LS}(M), E(M))$$

Example 2.31. Figure 2.1 is a visualization of the graph $G(M)$ where $M = (\lambda xy.x\, y)\, y\, x$.

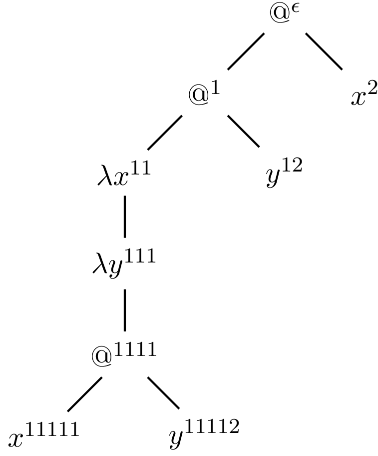


Figure 2.1: $G((\lambda xy.x\, y)\, y\, x)$ visualized.

Definition 2.32. A *path* $\sigma = [(p_1, q_1), (p_2, q_2), \ldots, (p_n, q_n)]$ in a λ-term M is a sequence of edges in $G(M)$, where $n \geq 1$ and $q_i = p_{i+1}$ (adjacent). We abbreviate $[(p_1, p_2), (p_2, p_3), \ldots, (p_{n-1}, p_n)]$ as $[p_1, p_2, \ldots, p_n]$.

Example 2.33. $\sigma = [\epsilon, 1, 11, 111, 1111, 11112]$ is a path in the term shown in Figure 2.1. More precisely, it is the path $@^{\epsilon} \to @^{1} \to \lambda x^{11} \to \lambda y^{111} \to @^{1111} \to y^{11112}$.

A *path* does not necessarily have to start at the root (position ϵ).

Definition 2.34. The *length* of a path σ, written as $|\sigma|$, is defined to be the number of edges in it.

Example 2.35. Let σ be the path from Example 2.33. Then $|\sigma| = 5$.

Definition 2.36. The *subpath* of a path σ in a λ-term M starting with the edge at index i and ending with the edge at index $|\sigma| - 1$ is written as $\sigma_{\llcorner i}$.

Example 2.37. Let σ be the path from Example 2.33, then $\sigma_{\llcorner 3} = \lambda y^{111} \to @^{1111} \to y^{11112}$.

Definition 2.38. The *subpath* of a path σ in a λ-term M starting with the edge at index 0 and ending with the edge at index $|\sigma| - i - 1$ is written as $\sigma_{\lrcorner i}$.

Example 2.39. Let σ be the path from Example 2.33, then $\sigma_{\lrcorner 3} = @^{\epsilon} \to @^{1} \to \lambda x^{11}$.

2.3 Occurrences, scopes and bindings

Definition 2.40. Let M be a λ-term. If $M(p) = s$ for some symbol s, then p is an *occurrence* of s.

Definition 2.41. Let M be a λ-term. If a λx occurs at position p in M, then the subterm $M|_p$ is called the *scope* of the λx.

We can have *nesting* of scopes. The nesting of scopes is visualized in Figure 2.2.

Definition 2.42. An occurrence of a variable x in a λ-term M is said to be a *bound* occurrence (bound for short), if it is in the scope of a λx. The innermost such λx is said to be the *binder* of x.

Example 2.43. Let $M = \lambda x.\lambda x.x$. The λ-node x at position 1 is the binder of the bound variable x. Let $N = \lambda x.(\lambda x.y)\, x$. The λ-node x at position ϵ is the binder of the bound variable x.

Definition 2.44. An occurrence of a variable x in a λ-term M is said to be a *free* occurrence (free for short), if it is not a bound occurrence.

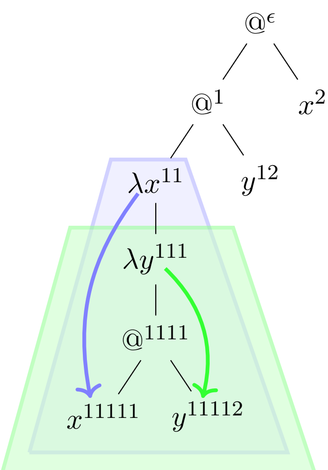


Figure 2.2: Scopes and bindings

Definition 2.45. (Free variables) The set of *free variables* $\mathcal{F}V(M)$ of a term M is the set of variables with a free occurrence in M.

Example 2.46. $\mathcal{F}V(\lambda xx.x) = \{\}, \mathcal{F}V(\lambda xz.x\,y) = \{y\}, \mathcal{F}V(\lambda x.z\,y) = \{y, z\}$

The variable y in the term M represented in Figure 2.2 at position 11112 is bound, and the one at position 12 is free. The variable x at position 11111 is bound in M, but free in $M|_{111}$. The blue and green arrows highlight the binding connection between the λ-nodes and the v-nodes.

A λ-term with no free variables is called *closed* or a *combinator*.

Definition 2.47. A *context* is a λ-term in $\Lambda_{\square}$ (having some occurrences of the constant $\square$, also called *hole*) and is denoted by C. If C is a context and M a λ-term, then $C[M]$ denotes the result of the replacing the holes in C by M [30, Section 10.1].

Example 2.48. Let $C = \lambda x.(\lambda z.z)\,\square$, then $C[\lambda y.x\,y] = \lambda x.(\lambda z.z)\,\lambda y.x\,y$. This example shows that variables may become bound (we apply naïve substitution).

2.4 Substitution and α-conversion

Definition 2.49. Let M and N be terms, x and y variables. The *naïve* substitution $M[x\backslash N]$ is defined as:

$$
\begin{array}{rcl}
x[x\backslash N] & := & N \\
y[x\backslash N] & := & y \\
(M_1\, M_2)[x\backslash N] & := & M_1[x\backslash N]\, M_2[x\backslash N] \\
(\lambda x.M')[x\backslash N] & := & \lambda x.M' \\
(\lambda y.M')[x\backslash N] & := & \lambda y.M'[x\backslash N]
\end{array}
$$

where we may have a so-called *capture* of a variable if in the last rule we have that $x \in \mathcal{FV}(M')$ and $y \in \mathcal{FV}(N)$. This substitution is therefore also called *capture-permitting* substitution.

We don't want variables to get captured in the substitution process as this would make the whole calculus inconsistent (in Example 2.55 we will see the undesired effect of a variable capture). A variable capture has therefore to be avoided. This can be done via ad-hoc renaming bound variables. Next we will see how this renaming, called α-conversion, is defined. For that we first need the notion of fresh variables.

Definition 2.50. Let M be a λ-term. A variable x having no free or bound occurrence and not abstracted in M is called *fresh for* M.

Example 2.51. x is fresh for y and $\lambda z.z\, y$ but not for $\lambda x.y$ and $\lambda y.x$.

Definition 2.52. The single α-*step*, performing the renaming of a bound variable in some context C, is defined as follows:

$$
C[\lambda x.M] \to_\alpha C[\lambda y.M[x\backslash y]], \text{ where } y \text{ is fresh for } M
$$

$$
(\mathit{refl})\ \frac{}{M \equiv_\alpha N} \qquad (\mathit{cong})\ \frac{M \equiv_\alpha M' \quad N \equiv_\alpha N'}{M\, N \equiv_\alpha M'\, N'} \qquad (\mathit{symm})\ \frac{M \equiv_\alpha N}{N \equiv_\alpha M}
$$

$$
(\xi)\ \frac{M \equiv_\alpha M'}{\lambda x.M \equiv_\alpha \lambda x.M'} \qquad (\mathit{trans})\ \frac{M \equiv_\alpha N \quad N \equiv_\alpha P}{M \equiv_\alpha P} \qquad (\alpha)\ \frac{y \notin \mathcal{V}ar(M)}{\lambda x.M \equiv_\alpha \lambda y.M[x\backslash y]}
$$

Table 2.1: The rules for α-equivalence [28].

Definition 2.53. M is α-*convertible* to N, $M \equiv_\alpha N$, if N results from M by a series of back- or forward α-steps (Definition 2.52) performed in an arbitrary context (not necessarily at the root of a λ-term).

α-convertibility can be characterized by the six rules listed in Table 2.1 [18].

Example 2.54. The λ-terms $M = \lambda xy.x\, y$ and $N = \lambda yx.y\, x$ are α-equivalent as $M \to_\alpha \lambda xz.x\, z \to_\alpha \lambda yz.y\, z \to_\alpha N$.

```
let succ x = x + 1        let succ y = y + 1
```

Figure 2.3: Different implementations of the successor function.

From a simplistic point of view, α-equivalent expressions are the same [14]. It doesn't matter whether we call the argument of the successor function x or y (see Figure 2.3). We therefore want α-equivalent terms to be interchangeable. More precisely, if we substitute a free variable x in a λ-term M or in an α-equivalent one M', by a λ-term N, then we want the following property to be true: $M[x\backslash N] \equiv_\alpha M'[x\backslash N]$. Unfortunately this property is not true for the capture-permitting substitution as shown in Example 2.55.

Example 2.55. Consider the λ-terms $M = \lambda y.x$ and $M' = \lambda z.x$. We have $M \equiv_\alpha M'$, but $M[x\backslash N] \not\equiv_\alpha M'[x\backslash N]$ for $N = y$, as $M[x\backslash N] = \lambda y.y$ and $M[x\backslash N'] = \lambda z.y$.

What happened in Example 2.55 is what we call a variable capture. This is the reason why the variable y in Definition 2.52 is required to be fresh. To avoid such variable capture, we may have to apply α-conversion before applying the actual substitution. We therefore define a capture-avoiding substitution next.

Definition 2.56. Let M and N be terms, x and y variables. The *capture-avoiding* substitution $M[\![x/N]\!]$ (”*safely substitute N for x in M*”) is defined as:

$$\begin{aligned}
x[\![x\backslash N]\!] &:= N \\
y[\![x\backslash N]\!] &:= y \\
(M_1\, M_2)[\![x\backslash N]\!] &:= M_1[\![x\backslash N]\!]\, M_2[\![x\backslash N]\!] \\
(\lambda x.M')[\![x\backslash N]\!] &:= \lambda x.M' \\
(\lambda y.M')[\![x\backslash N]\!] &:= \lambda y.M'[\![x/N]\!] \\
& \quad\ \text{if } y \notin \mathcal{FV}(N) \\
(\lambda y.M')[\![x\backslash N]\!] &:= \lambda z.M'[\![y\backslash z]\!][\![x\backslash N]\!] \\
& \quad\ \text{if } y \in \mathcal{FV}(N), \text{ where } z \text{ is fresh for } M' \text{ and } N
\end{aligned}$$

In contrast to Example 2.55 where we observed a variable capture when applying capture-permitting substitution, we have no variable capture when applying capture-avoiding substitution, as shown in Example 2.57.

Example 2.57. The terms $M = \lambda y.x$ and $M' = \lambda z.x$. We have $M \equiv_\alpha M'$, and $M[\![x\backslash N]\!] \equiv_\alpha M'[\![x\backslash N]\!]$ for $N = y$. $M[\![x\backslash N]\!] \equiv_\alpha \lambda y'.y$ and $M[\![x\backslash N']\!] = \lambda z.y$.

Lemma 2.58. *If $M[\![x\backslash N]\!] \equiv_\alpha M[x\backslash N]$, then there is no variable capture in $M[x\backslash N]$.*

Proof. Assume, for the sake of contradiction, that there is a variable capture in $M[x\backslash N]$. This is the case, if we have a subterm $\lambda y.M'$ in M, where $x \in \mathcal{FV}(M')$ and $y \in \mathcal{FV}(N)$ (Definition 2.49). In this case we have $(\lambda y.M')[x\backslash N] = \lambda y.M'[x\backslash N]$ and $(\lambda y.M')[\![x\backslash N]\!] = \lambda z.M'[\![y\backslash z]\!][\![x\backslash N]\!]$. However, $\lambda z.M'[\![y\backslash z]\!][\![x\backslash N]\!] \not\equiv_\alpha \lambda y.M'[x\backslash N]$ as $y \in \mathcal{FV}((\lambda y.M')[\![x\backslash N]\!])$ but $y \notin \mathcal{FV}((\lambda y.M')[x\backslash N])$. It follows from Definition 2.53 that such terms are not α-convertible, giving the contradiction. $\square$

Definition 2.59. The substitution $[\![x\backslash N]\!]$ is said to be α-*free* for M, if $M[\![x\backslash N]\!]$ satisfies the condition of Lemma 2.58.

Remark 2.60. Being α-free for a substitution means that we can apply it naïvely (there is no variable capture in $M[x\backslash N]$ that would break the equivalence $M[\![x\backslash N]\!] \equiv_\alpha M[x\backslash N]$). It does not mean that no α-renaming is applied in $M[\![x\backslash N]\!]$. In the $M[\![x\backslash N]\!]$, we might apply α-renaming even when not needed, in the sense that not doing it will not cause a variable capture. So changing the condition in Lemma 2.58 to $M[\![x\backslash N]\!] = M[x\backslash N]$ would result in a statement that, in general, is not true. Consider the term $M = \lambda y.y$. Then $M[\![x\backslash y]\!] = \lambda z.z$ and $M[x\backslash y] = \lambda y.y$. In this example, we α-renamed in $M[\![x\backslash y]\!]$, even if, according to Definition 2.49, do not have a variable capture as $x \notin \mathcal{FV}(M)$. Whenever we say α *is needed* or refer to *the need for α-conversion* in this thesis, we intend α-conversions that avoid variable captures.

There are some hidden complexity aspects for α-conversion. We must be able to check for α-equivalence, but what is the most efficient way to do that? What's the smallest number of distinct variables we need to represent a specific λ-term. How many fresh variables do we need to α-convert a λ-term M to a λ-term N? The *adbmal*-paper by Hendriks and van Oostrom [18] discusses three distinct definitions of α-conversion and their complexity aspects. In this paper they show that one fresh variable is enough to α-convert a λ-term M to a λ-term N (if they are α-convertible). It is not immediately clear that replacing a variable x by a fresh variable v across a whole term M preserves α-equivalence. A proof to it is given in Appendix A.2. α-conversion corresponds to the graph recoloring problem [29], Figure 2.4 gives an intuition for this correspondence.

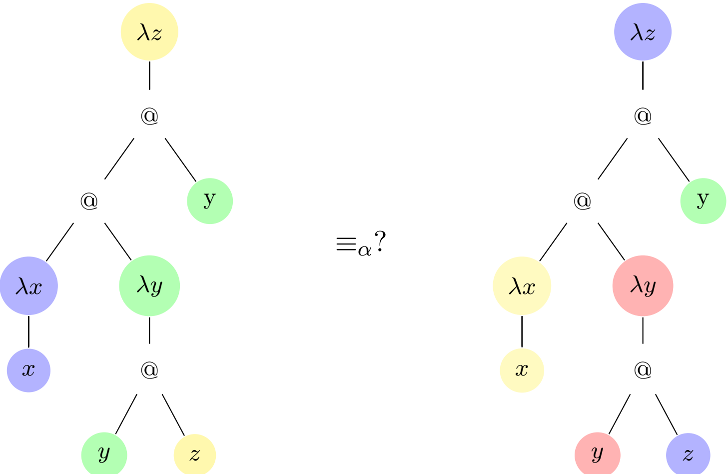


Figure 2.4: α-conversion and the graph-recoloring problem.

2.5 β-reduction

Definition 2.61. A term of shape $(\lambda x.M)\,N$ is called a reducible expression or *redex*.

Example 2.62. $(\lambda x.x)\,y$ is a redex, $\lambda x.x\,y$ is not a redex.

Definition 2.63. The *single-step β-reduction* $\to_\beta$, contracting a redex $(\lambda x.M)\,N$ in some context C, is defined as follows:

$$C[(\lambda x.M)\,N] \;\to_\beta\; C[M[\![x\backslash N]\!]]$$

$M[\![x\backslash N]\!]$ denotes the capture-avoiding substitution. We write $\to_{\beta p}$ to indicate the position p of the redex that is contracted. We call $C[(\lambda x.M)\,N]$ the *source* and $C[M[\![x\backslash N]\!]$ the *target* of $\to_\beta$. The *naïve single-step β-reduction* $\to_{\beta_{\text{ï}}}$ can be defined analogously, except that we apply the capture-permitting substitution $M[x\backslash N]$.

Convention 2.64. (*Variable convention* [3, 2.1.13]) If $M_1,\ldots,M_m$ occur in a certain mathematical context (e.g. definition, proof) then in these terms all bound variables are chosen to be different from the free variables.

A redex that satisfies Convention 2.64 can be contracted by naïve single-step β-reduction [3].

Definition 2.65. A β-reduction step $C[(\lambda x.M)\,N] \to_\beta C[M[\![x\backslash N]\!]]$ is said to be *α-free*, if the applied substitution $[\![x\backslash N]\!]$ is α-free for M (Definiton 2.59). In such a case, we have $M{\to_{\beta_{\text{ï}}}}\,N'$ and $N \equiv_\alpha N'$.

Next, we will define the *static trace* relation that allows relating positions in the source and the target of a $\to_\beta$-step. Later on, we will see that when reasoning with paths, this relation is essential for showing that α can be avoided. For more details on *tracing*, the reader is referred to [30, Section 8.6].

Definition 2.66. Assume $M \to_{\beta o} N$ where we contract a redex in M at position o. We can *trace* the positions of the source term M to positions in the target term N and vice-versa with the *static trace* relation, denoted by $\blacktriangleright$ [30, Section 8.6.1]. We distinguish three *components*:

- (context) $\quad p \blacktriangleright p \quad$ if o is not prefix of p
- (body) $\quad o11p \blacktriangleright op \quad$ if $p \neq \epsilon$ and $p \neq q$
- (arg) $\quad o2p \blacktriangleright oqp \quad$ for all positions q, such that $o11q$ is bound by $o1$.

If $p \blacktriangleright q$, then we say p is an *origin* of q, and q is a *copy* of p. Positions which do not have copies in N are said to be *erased* [30, Notation 8.6.9].

Proposition 2.67. *Let $M \in \Lambda$ and $M \to_\beta N$. If a variable at position p is bound in M then the copies of p in N are also bound. In other words, bound variables are not released.*

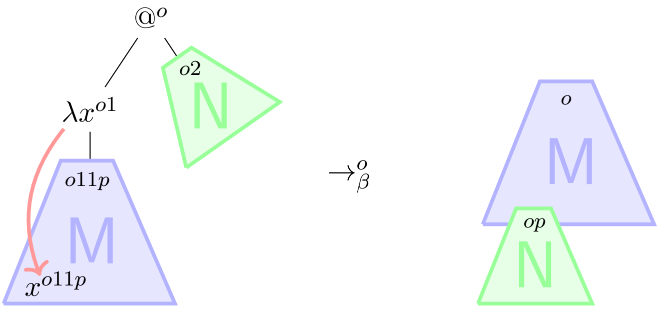


Figure 2.5: β-step a position o

Proof. This can be shown via the trace relation mapping positions in the source term M to the target term N where the contracted redex is at position o in M (Definition 2.66). Assume we have a bound variable occurrence at position v and its binder at position b with $b \prec v$ in M. Then we have the following cases:

i) v in the context: then we have $v \triangleright v$. Its binder is also in the context so $b \triangleright b$ and the assumption holds.

ii) v is in the body: then we have $v = o11v' \triangleright ov'$. Its binder can be either (a) in the context or (b) in the body. (a) if b is in the context, then it follows that $b \prec o11v'$ (as it binds the variable) and $b \prec o$ (because it is in the context) which implies $b \prec ov'$. (b) If the binder is in the body, then $b = o11b' \triangleright ob'$ and since we know that $b \prec v$ we have $b' \prec v'$ and therefore also $ob' \prec ov'$.

iii) v is in the argument: then we have $v = o2v' \triangleright oqv'$ for all q such that $o11q$ is bound by the abstraction at position $o1$. Its binder can be either (a) in the context or (b) in the argument. (a) if b is in the context, then we have $b \prec o2v'$ (as it binds the variable) and $b \prec o$ (because it is in the context) which implies $b \prec oqv'$. (b) If the binder is in the argument, then $b = o2b' \triangleright oqb'$ and since we know that $b \prec v$ we have $b' \prec v'$ and therefore also $oqb' \prec oqv'$.

□

Definition 2.68. Assume $M \to_\beta N$. A redex in N at position q is called a *residual* of a redex in M if $p \blacktriangleright q$ and $M|_p$ is a redex.

Example 2.69. Let $M = (\lambda x.x)\,((\lambda xy.x)\,z)$. Then $M \to_\beta \underline{(\lambda xy.x)\,z}$. The underlined redex is a residual of a redex at position 2 in M.

Example 2.70. Let $M = ((\lambda xy.x)\,z)\,y$. Then $M \to_\beta \underline{(\lambda y.z)\,y}$. The underlined redex is not a residual of a redex in M.

Definition 2.71. Assume $M \to_\beta N$. We call a redex in N at position q a *created* redex if it is not a residual of M.

Example 2.72. $(\lambda x.x\,y)\,(\lambda z.z) \to_\beta \underline{(\lambda z.z)\,y}$. The underlined λ-term is a created redex.

Proposition 2.73. *Let $M \to_{\beta o} N$. A redex at position p' in N is a created redex, iff the origins p of p' and q of $p'1$ are from a different component.*

Proof. The *if*-direction follows from the fact that if we have $p \blacktriangleright p'$ and $q \blacktriangleright p'1$, then this can only be if $M(p1)$ is the variable that gets substituted in contraction of the redex at position o. Therefore there is no redex at position p in M and consequently the redex at position p' in N is a created redex. For the *only-if*-direction we just have to look at the trace relation (Definition 2.66) and we see that if two positions are from the same component, then their are just transposed. This means that the redex at position p' in N is a residual of the redex at position p in M. □

Definition 2.74. A λ-term M is in *normal form* if no $\to_\beta$-step is possible from it.

Definition 2.75. We write $M \to_\beta^n N$ to denote that M reduces to N in n $\to_\beta$-steps. We write $M \twoheadrightarrow_\beta N$ if M can be transformed to N by zero or multiple $\to_\beta$-steps.

Example 2.76. $(\lambda x.x\,x)\,(\lambda yz.y\,z) \twoheadrightarrow_\beta \lambda zz'.z\,z'$ or, more precisely, $(\lambda x.x\,x)\,(\lambda yz.y\,z) \to_\beta^3 \lambda zz'.z\,z'$

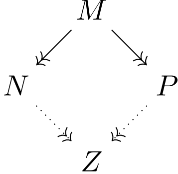
Example 2.77. x, $\lambda x.x$ and $y\,(\lambda x.x)$ are in normal form, $\lambda z.(\lambda x.x)\,y$ is not.

A λ-term can have multiple *redex*-occurrences for which β-reduction can be applied.

Definition 2.78. *A* reduction strategy *specifies which steps are allowed among the possible steps.* A strategy for $\to_\beta$ is a sub-relation of $\to_\beta$ (of the rules given in Definition 2.63) having the same objects and normal forms [30, Definition 9.1.1].

For example, the *leftmost–outermost* reduction and the *rightmost–innermost* reduction are strategies. They only allow, as the name already suggests, respectively to contract the leftmost, outermost and the rightmost, innermost redex in a λ-term M. In contrast, *call-by-value* reduction is not a strategy as it reduces λ-terms only to weak head normal form (defined in Chapter 6).

Theorem 2.79. $\twoheadrightarrow_\beta$ *satisfies the Church–Rosser property. Suppose M, N, and P are λ-terms such that $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta P$. Then there exists a λ-term Z such that $N \twoheadrightarrow_\beta Z$ and $P \twoheadrightarrow_\beta Z$ [28]. This property is also called* confluence.



What follows from Theorem 2.79 is that a λ-term has at most one normal form.

Definition 2.80. A *reduction sequence* with respect to $\to_\beta$ is a finite or infinite sequence $M \to_\beta M_1 \to_\beta M_2 \to_\beta \ldots$. A *reduction step* is a specific occurrence of in a reduction sequence [30, Definition 1.1.2].

Example 2.81. $(\lambda x.x\,x)\,(\lambda yz.y\,z) \to_\beta (\lambda yz.y\,z)\,(\lambda yz.y\,z) \to_\beta (\lambda z.(\lambda yz.y\,z)\,z)$ is a reduction sequence.

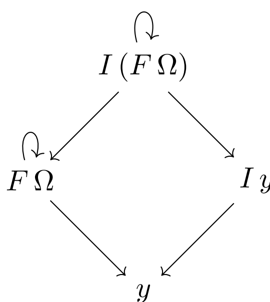
Definition 2.82. We call a reduction sequence $M \to_\beta M_1 \to_\beta \ldots$ a *development* of M, if in each β-step we only contract residuals of redexes in M.

Example 2.83. The reduction sequence of Example 2.81 is not a development in $(\lambda x.x\,x)\,(\lambda yz.y\,z)$. The one in Example 2.72 is a development of $(\lambda x.x\,y)\,(\lambda z.z)$.

Definition 2.84. A reduction sequence is said to be *α-free*, if each reduction step is α-free (Definition 2.65).

It may be, that a term has an infinite reduction sequence. The illustration in Example 2.85 shows the *reduction graph* (see [30, Definition 1.1.7]) of the λ-term $M = I\,(F\,\Omega)$. Depending on which redex of M is contracted, we reduce to different λ-terms. By Theorem 2.79, from any of these terms we can evaluate to the normal form y, but if we, for example, always reduce the rightmost–innermost redex, then we will get an infinite reduction sequence and never evaluate to the normal form (because Ω has no normal form).

Example 2.85. Let $I = \lambda x.x$, $F = \lambda x.y$, and $\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$.



Definition 2.86. A λ-term M is said to be *weakly normalizing* if it has a reduction sequence ending at a term N, where N is in normal form.

Example 2.87. $I\,(F\,\Omega)$ is weakly normalizing, Ω is not weakly normalizing.

Definition 2.88. A λ-term M is said to be *strongly normalizing* if it has no infinite reduction sequence.

Definition 2.89. A reduction strategy is said to be *α-free*, if each reduction sequence in this strategy (where each β-step is allowed by the strategy) is α-free.

In ordinary β-reduction, we α-rename ad hoc on the fly (Figure 2.6 (1)). We could, however, also first α-convert a term to comply with Convention 2.64 and then apply naïve β-reduction (Figure 2.6 (2)), as described in [3]. From this definition, it follows that an ordinary β-step corresponds to a naïve β-step which is preceded by some α-renaming steps. In this thesis, we investigate under which circumstances we can α-convert M to a λ-term M'', such that we can naïvely reduce M'' to normal form (Figure 2.6 (3)).

(1) $M \to_\beta N_1 \to_\beta \ldots \to_\beta N_k$

(2) $M \to_\alpha \ldots \to_\alpha M' \to_{\beta_{\text{ï}}} N_1 \to_\alpha \ldots \to_\alpha N'_{j-1} \to_\alpha \ldots \to_\alpha N''_{j-1} \to_\beta N_j$

(3) $M \to_\alpha M' \to_\alpha \ldots \to_\alpha M'' \to_{\beta_{\text{ï}}} N_1 \to_{\beta_{\text{ï}}} \ldots \to_{\beta_{\text{ï}}} N_l$

$N_k \equiv_\alpha N_l \equiv_\alpha N_j$? ($\alpha$-avoidance problem)

Figure 2.6: The α-avoidance problem.

Definition 2.90. We say that a λ-term M *avoids* α, if every reduction sequence from it is α-free (Definition 2.84).

Example 2.91. Ω avoids α, $(\lambda x.y)\,((\lambda xz.x)\,z)$ does not.

Definition 2.92. A λ-term M has α-free *simulations*, if there exists a λ-term N such that $M \equiv_\alpha N$ and N avoids α.

We say that α can be avoided for a specific λ-term M if it has α-free simulations.

Example 2.93. Ω and $(\lambda x.y)\,((\lambda xz.x)\,z)$ have α-free simulations, $(\lambda x.x\,x)\,(\lambda yz.y\,y)$ does not.

Definition 2.94. We say that we can avoid α (or the need of α-conversion) in a λ-calculus, if every λ-term in this calculus has α-free simulations (for its specific β-conversion rule).

We have just defined what it means to avoid α. Definition 2.94 is the main definition in this thesis. In the rest of it, we investigate for which calculi this definition applies (and for which ones not). We want to understand which restrictions of the untyped λ-calculus allow this property to hold. We do it via the so-called α-paths that are introduced in the next chapter. These paths characterize the need for α-conversion and thus give a different perspective on α.

3 The need for α-conversion characterized by α-paths

In this chapter, we introduce what we consider the major contribution of this thesis: α-paths. α-paths predict a potential variable capture and thus the need for α-conversion. Potential because it depends on the reduction whether the name collision indicated by an α-path will appear or not. Moreover, these paths also indicate the variables and binders that must be named distinctly to prevent the predicted variable capture from occurring in any reduction sequence. α-paths are a generalization of the self-capturing chains introduced by Vincent van Oostrom in [17] for the μ-calculus (Appendix A.1). They rely on a notion of *legal* paths due to Asperti and Guerrini [2]. We will use some basic concepts of Graph Theory taught in the *Diskrete Mathematik* bachelor course. Please refer to the lecture notes [23] for thing that are not defined here.

3.1 Intuition

We start by giving an intuition for how the need for α-conversion can be characterized by paths. To do that, let's enumerate the conditions that have to be true to make a variable capture occur (here we assume that $x \neq y$) in a β-step from some λ-term:

1. naïve contraction of a redex $(\lambda x.M)\ N$ at position p
2. $y \in FV(N)$ at position $p2q$
3. $x \in FV(M)$ at position $p1s1r$ (bound by the abstraction in the redex)
4. the x at position $p1s1r$ is captured by a λ-node λy at position $p1s$.

The above described situation is illustrated in Figure 3.1 on the left. Each of the four conditions enumerated above can be expressed via an edge.

Definition 3.1. We introduce four additional types of *edges* for a λ-term M:

- (a-edge ⇢) An $a$pplication-edge (p, q) connects a variable at position p that is free in the argument of an application to the corresponding application node at position q.
- (r-edge ⟶) A $r$edex-edge $(p, p1)$ connects the application node at position p to its left son at position $p1$, if that is a λ-node (so we have a redex).

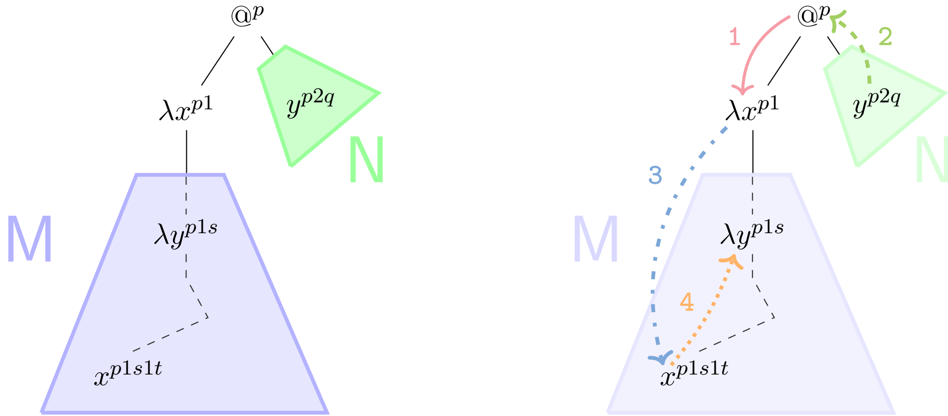


Figure 3.1: Constellation leading to a variable capture.

- (b-edge) A $b$inding-edge (p, q) connects a binder at position p to a variable it binds within its body at position q.
- (c-edge) A $c$apturing-edge (p, q) connects a variable at position p to a λ-node at position q, if the variable at position p occurs free in the subterm at position q (so, in particular, is distinct from the binder).

The illustration on the right in Figure 3.1 shows how these edges can be drawn. We have an a-edge $(p2q, p)$, an r-edge $(p, p1)$, a b-edge $(p1, p1s1t)$ and a c-edge $(p1s1t, p1s)$. We will add the a-, r-, b- and c-edges as actual edges to the graph of a λ-term. We call such a graph the α-graph of a λ-term M.

Corollary 3.2. *Let M be a λ-term and $(p, p1)$ an* r*-edge in $G_\alpha(M)$. Then $M|_p$ is a redex.*

Proof. Follows from the definition of an r-edge (Definition 3.1). □

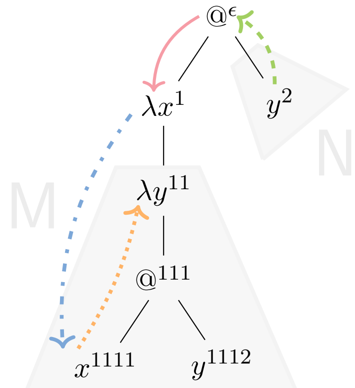
Definition 3.3. An α-graph $G_\alpha(M)$ of a λ-term M is defined as:

$$G_\alpha(M) = (\mathcal{LS}(M), E_\alpha(M), l)$$

where $E_\alpha(M) = E(M) \cup E_a(M) \cup E_b(M) \cup E_c(M) \cup E_r(M)$ and where $E_a(M)$ is the set of the a-edges, $E_b(M)$ is the set of the b-edges, $E_c(M)$ is the set of the c-edges and $E_r(M)$ is the set of the r-edges in M. l is a mapping describing the labeling of the edges in $E_\alpha(M)$ as follows:

label	set	edge
0	$E(M)$	
1	$E_a(M)$	
2	$E_b(M)$	
3	$E_c(M)$	
4	$E_r(M)$	

We can have parallel edges (with a different label) between two vertices, so formally, an α-graph is a directed multigraph with labeled edges [23]. For simplicity, we will use different colors and arrows to distinguish between edges with different labels, as shown in the table above.



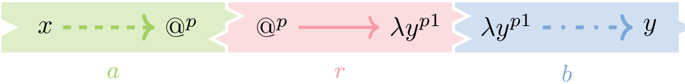
1. the @-node at position $p = 1$ forms a redex-pattern with the λx-node at position 11
2. $N := y$ and $y \in FV(N)$ at position 12 ($q = \epsilon$)
3. $M := \lambda y.x\, y$, $x \in FV(M)$ and occurs at position 11111 ($s = 1, r = \epsilon$)
4. there is a λy-node at position 111

Figure 3.2: $G_\alpha((\lambda x.\lambda y.x\, y)\, y)$.

When contracting the redex in $(\lambda x.\lambda y.x\, y)\, y$, α-renaming is needed to avoid that the y occurring free gets captured. The α-graph of this term is illustrated in Figure 3.2 (edges irrelevant for this example were omitted in the figure). We have designed edges such that capturing is captured by a path in the α-graph of a λ-term comprising the four types of edges. In Figure 3.2 it would be the path $[2, \epsilon, 1, 1111, 11]$. This is a special instance of a basic *arbic*-path we will define down below, where the variable at position 2 is abstracted in the λ-node at position 11[1].

We can also have λ-terms where α, depending on the reduction, is needed only after multiple β-steps. This is, for example, the case for the λ-term illustrated in Figure 3.3. It depends on the reduction order whether α-conversion is needed for the λ-term depicted in Figure 3.3. In this example, an outermost reduction will require one α-conversion, whereas an innermost reduction is α-free.

Definition 3.4. Let M be λ-term. We call a path $\sigma_{arb} = a \cdot r \cdot b$ an *arb-path* of M, if a is an a-edge, r an r-edge and b a b-edge in $G_\alpha(M)$.



Proposition 3.5. *Let p be the position of the starting* v*-node y and q the position of the ending* v*-node of an* arb*-path. Then $q \parallel_l p$.*

[1] we will call such paths α-paths

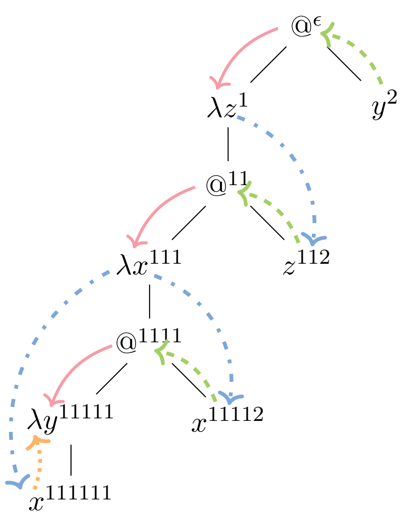


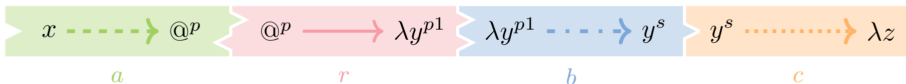
Figure 3.3: An outermost reduction needs α-conversion.

Proof. Let $a = (p, a_2)$ be the a-edge. a_2 is the position of the @-node of the redex, therefore we know that $p = a_2 \cdot 2 \cdot q$. The λ-node of the succeeding r-edge is at position $a_2 \cdot 1$ and can only bind a variable in its scope at position $a_2 \cdot 1 \cdot q' = q$. By Definition 2.17 we have $q \parallel_l p$. □

The example term from Figure 3.3 illustrates why we have to combine multiple *arb*-paths to check whether or not α-conversion is needed after multiple reduction steps. To characterize also the need for α-conversions that require the contraction of one or multiple redexes of a λ-term M, the so-called *arbic*-paths are introduced next.

Definition 3.6. The set of *arbic*-paths of a λ-term M is inductively defined as:

- (base case) Let σ_{arb} be an *arb*-path of M and c a c-edge in $G_\alpha(M)$. Then the path $\sigma_{arb} \cdot c$ is an *arbic*-path of M.



- (*arb*-composition) let σ_{arb} be an *arb*-path and ψ an *arbic*-path of M. Then the path $\sigma_{arb} \cdot \psi$ is an *arbic*-path of M.

From Definition 3.6 we see that *arbic*-paths are nonempty sequences of *arb*-paths followed by a c-edge $(\sigma^+_{arb} \cdot c)$. By Proposition 3.5 we know that we cannot form cycles with the *arb*-composition rule. Therefore, the set of *arbic*-paths of a λ-term M is finite.

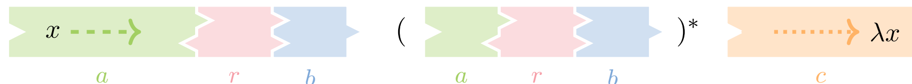
Example 3.7. The paths $\sigma_0 = 112 \to 11 \to 111 \to 111111 \to 11111$, and $\sigma_1 = 2 \to \epsilon \to 1 \to \sigma_0$ are *arbic*-paths for the λ-term illustrated in Figure 3.3.

Lemma 3.8. *Let p be the position of the starting* v*-node y and q the position of the ending λ-node of an* arbic*-path ψ. Then $q \parallel_l p$.*

Proof. We have $\psi = \sigma^+_{arb} \cdot c$, where σ^+_{arb} is a non-empty sequence of *arb*-paths, from p to a position t, and c a c-edge from t to q (Definition 3.6). By Proposition 2.19 (transitivity of $\parallel_l$) and Proposition 3.5 it immediately follows that $t \parallel_l p$. In the proof of Proposition 3.5 we can see that this property also holds for the λ-node binding the v-node at position t. Since the λ-node at position q must be in the scope of that binder, we have $q \parallel_l p$. □

Above, we have seen that the path $[2, \epsilon, 1, 1111, 11]$ in the α-graph of the term illustrated in Figure 3.2 was a special instance of an *arbic*-path. It was "special" because the variable at position 2 is also abstracted in the λ-node at position 11. Such paths characterize the need for α-conversion, precisely because of that condition. We, therefore, call such paths *arbic* α-paths.

Definition 3.9. (*arbic α-path*) Let M be a λ-term and ψ an *arbic*-path of M. If ψ starts with a variable x and ends with a λ-node λy where $x = y$, then ψ is called a arbic α-path.



Example 3.10. The path σ_2 as defined in Example 3.7 is an arbic α-path for the λ-term illustrated in Figure 3.3.

Proposition 3.11. *Let $G_\alpha((\lambda x.M)\ N)$ contain no arbic α-path of length 4. If M contains an x occurring free in the scope of some λy (with $x \neq y$), then y is not free in N.*

Proof. Otherwise there is an arbic α-path $y \to @ \to \lambda x \to x \to \lambda y$ of length 4 in $G_\alpha((\lambda x.M)\ N)$, contradicting the assumption. □

Redexes that fulfill Proposition 3.11 can be contracted by means of capture-permitting substitution.

Lemma 3.12. *(α-free). Suppose that there is no arbic α-path in $G_\alpha((\lambda x.M)\,N)$ of length 4 starting at a free variable in N and ending in M. Then $M[\![x\backslash N]\!] \equiv_\alpha M[x\backslash N]$.*

Proof. By induction and cases on M, the only interesting case being when the term is an abstraction $\lambda y.M'$ with $x \neq y$. Then either x is not free in M' and N gets erased or else by the assumption, y is not free in N (shown in Proposition 3.11), hence

$$\begin{array}{rcll}(\lambda y.M)[\![x\backslash N]\!] & = & \lambda y.M[\![x\backslash N]\!] & \text{as } y \notin \mathcal{FV}(N) \\ & \equiv_\alpha & \lambda y.M[x\backslash N] & \text{I.H.} \\ & = & (\lambda y.M)[x\backslash N] & \square\end{array}$$

In Lemma 3.12 we only claimed α-equivalence, and not the syntactic equivalence $(=)$ of $M[\![x\backslash N]\!]$ and $M[x\backslash N]$. Example 3.13 illustrates why (for the same reasons given in Remark 2.60).

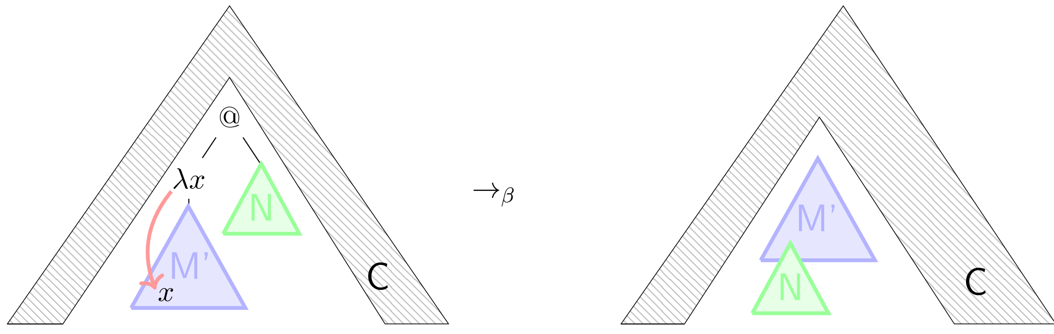
Example 3.13. Let $(\lambda x.M)\,N$ be a redex with $M = \lambda y.y$ and $N = y$. Then $M[\![x\backslash N]\!] = \lambda z.z$ and $M[\![x\backslash N]\!] = \lambda y.y$. We have $M[\![x\backslash N]\!] \equiv_\alpha M[x\backslash N]$, but $M[\![x\backslash N]\!] \neq M[x\backslash N]$.

Definition 3.14. Let $M \to_\beta N$. An r-edge $(p, p1)$ in $G_\alpha(N)$ denotes a residual of a redex in M, if, according to Definition 2.68, $N|_p$ is a residual of a redex in M.

Lemma 3.15. *Let $s \to_{\beta o} t$. If $G_\alpha(s)$ contains no arbic α-path, then the sub-graph of $G_\alpha(t)$, where we restrict the set of* r*-edges only to those denoting residuals of redexes in s, also does not.*

Proof. We write $\langle G_\alpha(t)\rangle$ for the described sub-graph of $G_\alpha(t)$. Assume there are no arbic α-paths in $G_\alpha(s)$. By Lemma 3.12 we have no variable capture if we contract a redex in s by means of capture-permitting substitution. We have $s = C[(\lambda x.M)\,N]$ and $t \equiv_\alpha C[M[x\backslash N]]$ for some context C, body M and argument N, with $(\lambda x.M)\,N$ being the contracted redex at position o. We prove the lemma by relating the edges in $\langle G_\alpha(t)\rangle$ to edges and paths in $G_\alpha(s)$. As done in [17], we use primed variables (e.g. p') to range over positions in the target term t, indicating the positions they trace back to in the source term s, by unpriming (e.g. p). Let's recall the trace relation from Definition 2.66 for the contraction of a redex at position o where we do case distinction on the following three components:

- (context) $\quad p \blacktriangleright p \quad$ if o is not prefix of p
- (body) $\quad o11p \blacktriangleright op \quad$ if $p \neq \epsilon$ and $p \neq q$
- (arg) $\quad o2p \blacktriangleright oqp \quad$ for all positions q, such that $o11q$ is bound by $o1$.



Given an a- or a c-edge from p' to q' in $\langle G_\alpha(t)\rangle$. p' denotes the position of a variable y, q' the position of an application (in the case of an a-edge) or an absraction (in the case of a c-edge). We have $q' \prec p'$ and the variable y at $t(p')$ occurs free in $t|_{q'}$. We distinguish following cases:

- p', q' are both in the same component: then $s(p)$ occurs free in $s|_q$ and therefore we have the same edge from p to q in $G_\alpha(s)$.
- q' is in the context and p' in the body: then $x \neq y$ (otherwise the y would have been replaced by N) and we have the same edge in $G_\alpha(s)$ with 11 inserted at o.
- q' is in the context and p' in the argument: there is no variable capture so $s(p)$ must occur free in $s|_q$. Therefore we have the same edge from p to q in $G_\alpha(s)$.
- q' is in the body and p' in the argument:
  - a-edge: the origin of such a-edge is an arb-path from p to qq', for some q', followed by an a-edge from qq' to q in $G_\alpha(s)$.
  - c-edge: the origin of such c-edge is an arb-path from p to qq', for some q', followed by a c-edge from qq' to q in $G_\alpha(s)$.

Given a b-edge from q' to p' in $\langle G_\alpha(t)\rangle$. p' denotes the position of the bound variable y, q' the position of the binder λy. We have $q' \prec p'$ and distinguish following cases:

- p', q' are both in the same component: then we have the a b-edge from q to p in $G_\alpha(s)$.
- q' is in the context and p' in the body: we have $x \neq y$ and a b-edge in $G_\alpha(s)$ with 11 inserted at o.
- q' is in the context and p' in the argument: there is no variable capture so $s(p)$ must occur free in $s|_q$. Therefore, we have a b-edge from q to p in $G_\alpha(s)$.
- q' is in the body and p' in the argument: such a b-edge would map back to an arbic α-path from p to q in $G_\alpha(s)$, which is excluded by the assumption (Figure 3.4 illustrates an example).

For the r-edges $(p', p'1)$ in $\langle G_\alpha(t)\rangle$ we make the following case distinction:

- p' and q' are in the same component: then we have an r-edge from p to q in $G_\alpha(s)$.
- in all other cases: by Proposition 2.73 such a r-edge would denote a created redex in t. We have no such r-edge in $\langle G_\alpha(t)\rangle$.

We have seen that an r-edge and a b-edge in $\langle G_\alpha(t)\rangle$ maps back to an edge of the same type in $G_\alpha(s)$. a-edges and c-edges map back to a path of shape $\sigma^*_{arb} \cdot e$, where e denotes an edge of the same type and σ_{arb} an arb-path in $G_\alpha(s)$. An arbic α-path in $G_\alpha(t)$ has the following shape $(a_1, r_1, b_1, \ldots, a_n, r_n, b_n, c)$, where x_i denotes an x-edge (p_i, q_i). If we replace a-edges and c-edges by the path the map back to we get $(\sigma^*_{arb_1} \cdot e_1, r_1, b_1, \ldots, \sigma^*_{arb_n} \cdot e_n, r_n, b_n, \sigma^*_{arb_c} \cdot e_c)$, where $\sigma^*_{arb_i} \cdot e_i$ in $G_\alpha(s)$ connects the same positions as the corresponding x-edge in $\langle G_\alpha(t)\rangle$. It follows that if we have an arbic α-path in $\langle G_\alpha(t)\rangle$, then we have an arbic α-path $G_\alpha(s)$. □

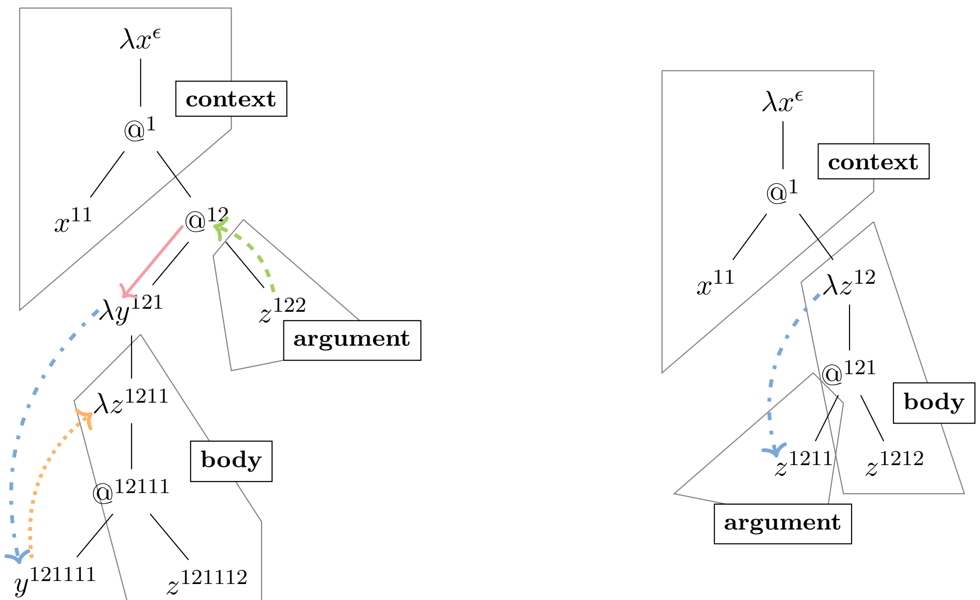


Figure 3.4: A body–argument b-edge maps to an arbic α-path.

Lemma 3.16. *Let M be a λ-term. If M contains no arbic α-path, then every development from M is α-free.*

Proof. Suppose for a proof by contradiction, some development from M is not α-free. Then it would contain a first non-α-free step. That step would contain an arbic α-path. Then by induction and Lemma 3.15, M would contain an arbic α-path which gives the contradiction. □

Note that with these arbic α-paths we do not characterize variable captures that result from the contraction of *created* redexes (discussed in the next section). For example, the λ-term $M = (\lambda z.z\, y)\, (\lambda x.\lambda y.x)$ in Figure 3.5 reduces in one β-step to $N = (\lambda x.\lambda y.x)\, y$ for which we need α to reduce it further. However, we do not have any arbic α-path in $G_\alpha(M)$ predicting this name-collision.

Lemma 3.17. *For every λ-term M there exists a λ-term N where $M \equiv_\alpha N$, such that N does not contain any arbic α-paths.*

Proof. Assume we α-convert M to N such that all binders are named distinctly and distinct from the free variables. By Lemma 3.8 we have $q \parallel_l p$, for p being the position of the first v-node and q the position of the last λ-node in an *arbic*-path. By construction of N, we know that for the variable x at position p and the binder λy we have $x \neq y$, if $q \parallel_l p$. □

Arbic α-paths can also characterize the need for α-conversion in such terms as the one shown in Figure 3.3, where α-conversion is needed after multiple reduction steps. However, we also saw that arbic α-paths only work for calculi that do not allow redex

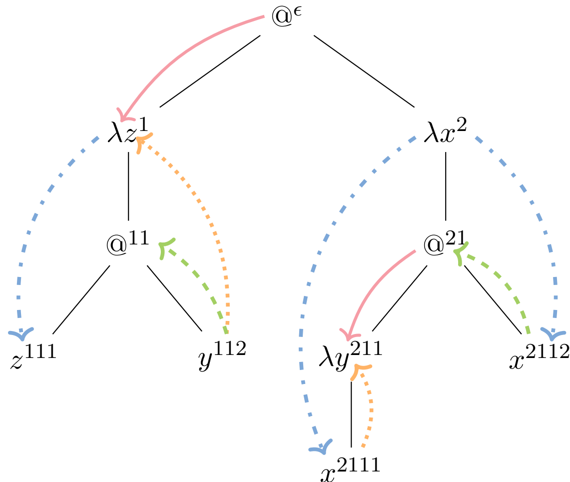


Figure 3.5: $G_\alpha((\lambda z.z\,y)\,(\lambda x.\lambda y.x))$.

creation (Figure 3.5). In calculi that allow redex creation, a more sophisticated approach is needed to characterize also the need for α-conversion for created redexes. This can be done by using the so-called *legal* paths we will introduce next.

3.2 Legal paths

A configuration inside a term, which is not a redex yet, but might become one along reduction, is called a *virtual* redex [2]. The legal paths due to Asperti and Guerrini allow characterizing virtual redexes. They can answer the following question: *"Can a given application node ever be involved in a reduction?"* [2]. For a better understanding, look at the left term in Figure 3.6. The application at position 11 has a variable as left son, so it is not a redex. However, since the variable is not free, it might become one, if a λ-node will substitute it. To determine whether this can happen, we have to check if the binder of x, which in this case is the λ-node at position 1, will ever be involved in a β-reduction. To answer that, we have to travel towards the root and search for a matching application, which we immediately find at position ϵ. We also know that the term that will be substituted for x is the subterm at position 2 ($\lambda zy.z$). Since this term starts with a λ-node it will form a redex with the application at 11. We, therefore, can answer the initial question positively. While trying to find the answer, we created the following path $11 \to 111 \to 1 \to \epsilon \to 2$, as drawn in Figure 3.6. This is an example of a *legal* path.

This example was just meant to give an idea of how a path can characterize a virtual redex. A similar example, where the whole process of finding a redex is explained in more detail can be found in [2, Section 6.2].

Legal paths are defined via the so-called *well-balanced* paths. Their generation idea corresponds to the approach of nesting questions about nodes as described in the example

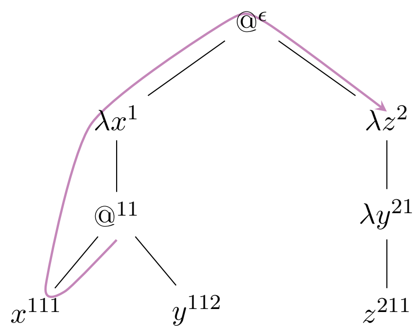


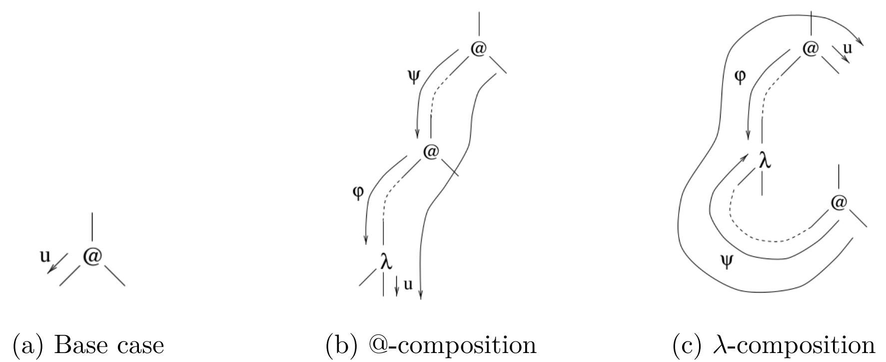
Figure 3.6: A legal path characterizing a virtual redex.

above. As this naive procedure does not consider any context (the path we came from) but only ensures that the endpoints match, we may construct paths that do not correspond to a virtual redex[2]. Therefore, the set of *well-balanced* paths *gives raise to a set of paths wider than the ones corresponding to virtual redexes* [2, Section 6.2.4]. These paths are sorted out by the *legality* constraint that checks if the call- and the return-path of the ”nested questions” coincide.

Definition 3.18. A path starting at an application node @ is of type

- `@-v`, if it ends with a `v`-node.
- `@-`λ, if it ends with a λ-node.
- `@-@`, if it ends with an `@`-node.

Definition 3.19. The set of *well-balanced* paths (abbreviated as *wbp*) of a term M is inductively defined as follows [2]:



(a) Base case (b) @-composition (c) λ-composition

Figure 3.7: Well-balanced paths [2].

[2]We will see an example later on (Figure 3.8)

- (base case) The function edge of any application of M is a wbp. In Figure 3.7a denoted by the edge u.
- (*@-composition*) Let ψ be a wbp of type `@-@` and φ a wbp of type `@-`λ be two composable paths. Then $\psi \cdot \varphi \cdot u$ is a wbp, where u outgoes the final abstraction of φ to the root of the body.
- (λ-*composition*) Let ψ be a wbp of type `@-v` whose ending variable is bound by the ending abstraction of a wbp φ of type `@-`λ. Then $\psi \cdot (\varphi)^r \cdot u$ is a wbp, where u is the argument edge of the initial application of φ.

From Definition 3.19 we can see why these paths are called *well-balanced* – in well-balanced paths of type `@-`λ, the number of `@`-nodes and λ-nodes is balanced. We can perceive well-balanced paths of type `@-@` and `@-v` as "prefixes" of balanced paths (closing λ-nodes missing). The construction of *well-balanced* paths corresponds to the procedure described before. The goal was to determine whether the application at position 11 (Figure 3.6) will ever have a matching λ-node at its function port. In fact, in the definition of wbp's when reaching a variable, we continue with the term the variable will potentially be substituted for. When reaching an application node, we have to find out what is inside its matching λ-node, because that is what remains when that redex is reduced. Following this logic, wbp's of type `@-`λ should all characterize a *virtual* redex that is created at some point along β-reduction, but, as already mentioned, this is not the case. That is because composition rules of wbp's do not take into account the history of a path which could lead to *illegal* paths of type `@-`λ. Illegal in the sense that they do not correspond to a *virtual* redex.

Figure 3.8 illustrates for the λ-term $(\lambda x.(x\,(\lambda z.z))\,(x\,(\lambda z.z)))\,(\lambda y.y)$ the iterative computation of the wbp's to give an example where the composition of wbp's leads to an illegal path. The first tree in Figure 3.8 (labeled by 1) illustrates the base case: each function edge of an application is a wbp, as denoted by the red arrows. In the second tree, the wbp's (highlighted in blue) are created by λ-composition. In the third iteration step, a new wbp (highlighted in green) can be created by @-composition. This wbp of type `@-v` ending at the variable y at position 21 enables additional λ-compositions with the paths created in the second iteration step. These new *well-balanced* paths, visualized respectively in tree 4.1 and 4.2 are both of type `@-`λ, but only the one in 4.1 characterizes a *virtual redex*, whereas the one in 4.2 is illegal. The application node at position 11 will never interact with the λ-node at position 1122. The reason why such illegal paths may arise is the occurrence of cycles. In the example above this is the case in 4.2, where we jump back to the wrong x. A legal path must follow back the same path traversed to access N before the cycle [2] (N is, in our case, the subterm at position 2). Cycles, like the one in Figure 3.8, that are internal to the argument and start and end with an application's argument edge are called *elementary.*

Definition 3.20. Let φ be a wbp. A subpath ψ of φ is an *elementary* @-cycle of ψ (over an @-node) when:

- starts and ends with the argument edge of the @-node;

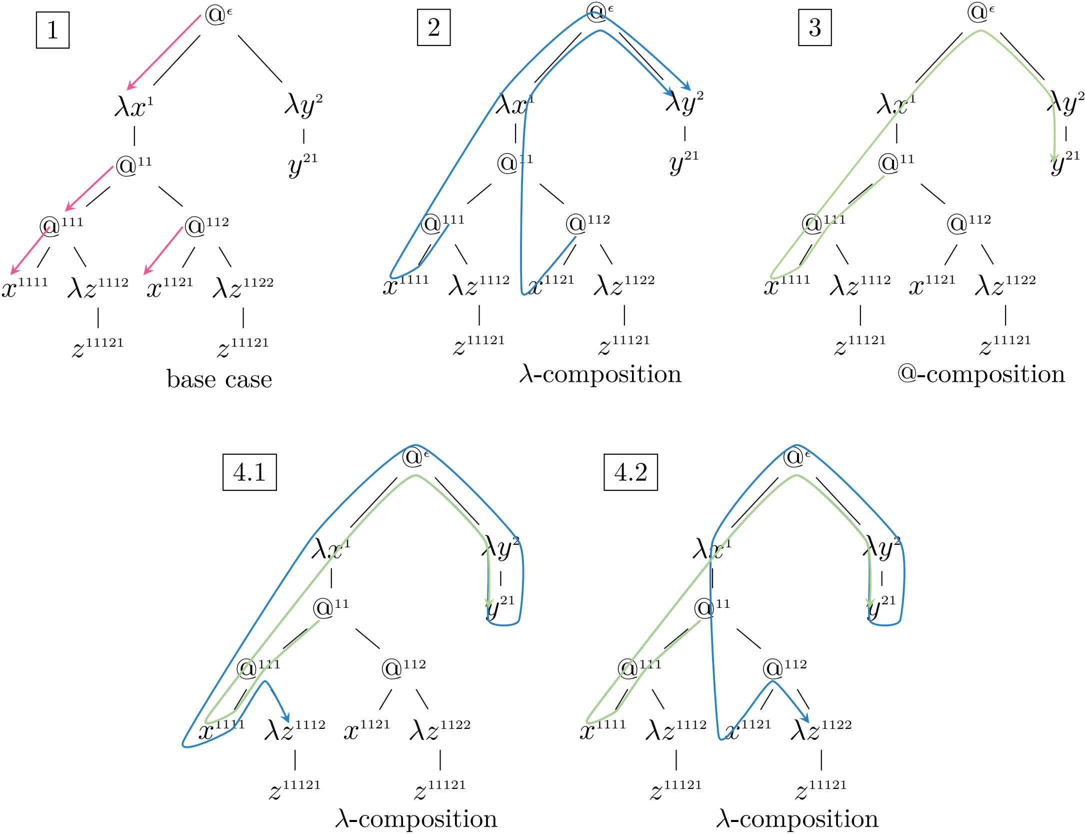


Figure 3.8: *Well-balanced* paths in $(\lambda x.(x\,(\lambda z.z))\,(x\,(\lambda z.z)))\,(\lambda y.y)$

- is internal to the argument N of the application corresponding to the @-node (i.e., does not traverse any variable that occurs free in N).

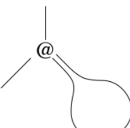


Figure 3.9: An elementary cycle.

Unfortunately, the cycles that may occur are not always *elementary*. In particular, @-cycles may contain occurrences of other cycles and pairing the last point in which a path entered an argument of an application to the first point in which it exited from the argument is generally incorrect [2]. This leads to the following definition of @-cycles:

Definition 3.21. Let ζ be a wbp. The set of the *@-cycles* of ζ (over an @-node) and of the v-cycles of ζ (over the occurrence v of a variable) is inductively defined as follows:

- Elementary: Every elementary @-cycle of ζ is an @-cycle.
- v-cycle : Every cyclic subpath of ζ of the form $v\ \lambda\ (\phi)^r\ @\ \psi\ @\ \phi\ \lambda\ v$, where ϕ is a wbp, is an @-cycle and v is a binding edge, is a v-cycle (over v).
- @-cycle : Every subpath ψ of ζ that starts and ends with the argument edge of a given @-node, and that is composed of subpaths internal to the argument N of @and v-cycles over free variables of N is an @-cycle (over the @-node).

@-cycles are always surrounded by two wbp's of type @-λ as following lemma states:

Lemma 3.22. *Let ψ be an @-cycle of ϕ over an @-node. The wbp ϕ can be uniquely decomposed as*

$$\phi = \zeta_1\ \lambda\ (\zeta_2)^r\ @\ \psi\ @\ \zeta_3\ \lambda\ \zeta_4$$

*where ζ_2 (*call-path*) and ζ_3 (*return-path*) are wbp's of type $@ - \lambda$. The last label of ζ_1 and the first label of ζ_4 are the* discriminants*.*

Definition 3.23. A wbp is a *legal* path if and only if the call and return paths of any @-cycle are one the reverse of the other and their discriminants are equal.

Lemma 3.24. *All (virtual) redexes of a λ-term M are characterized by some legal path in M of type @-λ.*

Proof. This was proven by Asperti in [2, Section 6.2.5] by showing a bijective correspondence between legal paths and paths yielded by degrees (see [2, Theorem 6.2.42] and [2, Fact 6.2.3]). ☐

Corollary 3.25. *M is a strongly normalizing term iff the set of legal paths of type @-λ in M is finite [1, Lemma 38].*

Proof. Immediately follows from Lemma 3.24. ☐

Example 3.26. The following *well-balanced* paths were created in the last iteration step for the term illustrated in Figure 3.8:

4.1 : $@^{11} \to @^{111} \to x^{1111} \to \lambda x^1 \to @^\epsilon \to \lambda y^2 \to y^{21} \to \lambda y^2 \to @^\epsilon \to \lambda x^1 \to x^{1111} \to @^{111} \to \lambda z^{1112}$

✔ legal — ζ_1, ζ_2, @-cycle, ζ_3, ζ_4

4.2 : $@^{11} \to @^{111} \to x^{1111} \to \lambda x^1 \to @^\epsilon \to \lambda y^2 \to y^{21} \to \lambda y^2 \to @^\epsilon \to \lambda x^1 \to x^{1121} \to @^{112} \to \lambda z^{1122}$

✘ legal — $\zeta_1^{'}$, $\zeta_2^{'}$, @-cycle, $\zeta_3^{'}$, $\zeta_4^{'}$

The path 4.1 is legal, because $(\zeta_2)^r = \zeta_3$ and $\zeta_1 = \zeta_4$ whereas the path 4.2 is not, because $\zeta_1^{'} \neq \zeta_4^{'}$.

Figure 3.10 illustrates the legal paths in the λ-term $(\lambda x.x\,x)\,(\lambda yz.y\,z)$.

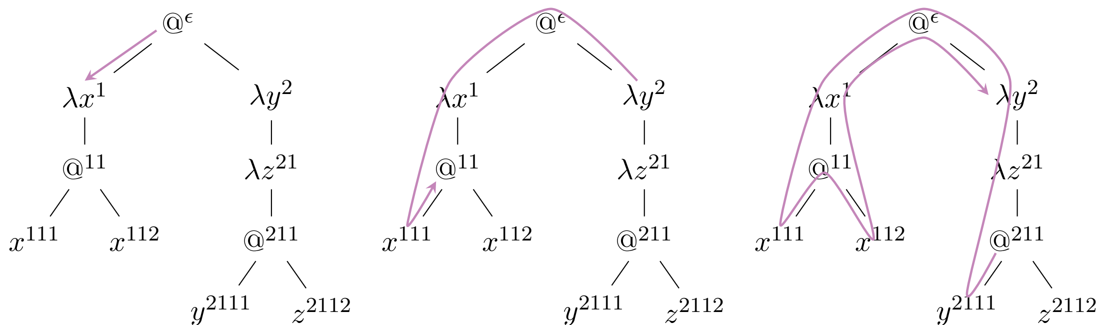


Figure 3.10: Legal paths of type `@`-λ in $(\lambda x.x\,x)\,(\lambda yz.y\,z)$

3.3 α-paths

We have already seen arbic α-paths that predict the need for α-conversion for λ-calculi without redex creation. The α-paths presented in this section are an extension of them and allow to predict the need for α-conversion in λ-calculi with redex creation. α-paths are defined on the so-called albic-paths that rely on *legal* paths.

Definition 3.27. Let M be a λ-term. We call a non-empty path $\sigma_{alb} = a \cdot l \cdot b$ an *alb-path* of M, if a is an a-edge starting at position p', l a *legal* path of type `@`-λ and b a b-link ending at position q' in $G_\alpha(M)$, where $p' \neq q'$.

$y^{p'} \dashrightarrow @^p$ | $@^p \longrightarrow \lambda x^q$ | $\lambda x^q \dashrightarrow x^{q'}$

a | l | b

A legal path may potentially connect any `@`-node to any λ-node in a λ-term M. This may lead to a setting like the one shown in Figure 3.11 where an *alb*-path starts and ends at the same `v`-node. Such *alb*-paths are not valid and therefore excluded by the side-condition $p' \neq q'$.

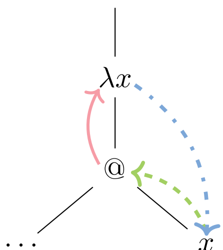
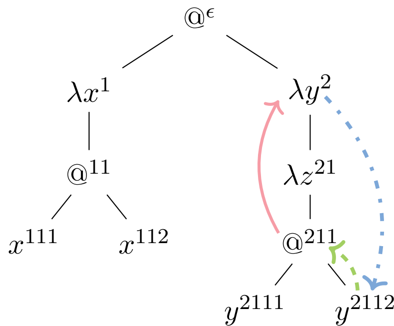


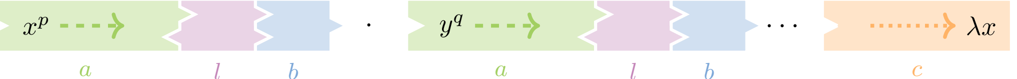
Figure 3.11: An *alb*-path cannot start and end at the same `v`-node.

Example 3.28. The following example shows that without side-condition in Definition 3.27 it would be possible to create infinite compositions of links in $(\lambda x.x\,x)\,(\lambda yz.y\,y)$.



Definition 3.29. The set of *albic-paths* of a λ-term M is inductively defined as:

- (base case) Let σ_{alb} be an *alb*-path and c a c-edge in M. Then the path $\sigma_{alb}\cdot$ is an albic-path of M.
- (*alb*-composition) let σ_{alb} be an *alb*-path of M starting at position p and ψ an albic-path of M starting at position q. Then the path $\sigma_{alb} \cdot \psi$ an albic-path of M.



Proposition 3.30. *Let M be a λ-term. Then each* arbic*-path of M is also an albic-path of M.*

Proof. Follows from the fact that each r-edge is a legal path (base case, see Definition 3.19). ☐

Definition 3.31. (α-path) Let M be a λ-term and ψ an albic-path of M. If ψ starts at a variable x and ends at a λ-node λy where $x = y$ and the binder of x is not in $\psi_{\lrcorner 1}$, then ψ is called an α-path.

Not every *albic*-path starting with a variable x and ending at its binder λx is problematic (variable capture). It could be that free variables in the argument of a virtual redex are substituted before that redex exists. For example, this is the case in the term shown in Figure 3.12.

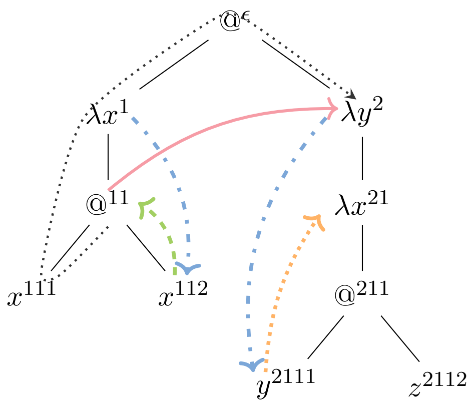


Figure 3.12: Harmless *albic*-path.

The *albic*-path $112 \to 11 \to 2 \to 2111 \to 21$ is harmless, as the variable x is substituted by $\lambda y.\lambda x.y\ z$ before reducing the redex at 11. That is why we have the side-condition that *"the binder of x is not in $\psi_{\lrcorner 1}$"* in Definition 3.31.

Lemma 3.32. *A arbic α-path is an instance of an α-path.*

Proof. Follows from Proposition 3.30. □

Proposition 3.33. *Let $G_\alpha((\lambda x.M)\ N)$ contain no α-path. Suppose a free occurrence of x in M is connected to a binder λy through a* c*-edge. Then y is not free in N.*

Proof. Otherwise there is an α-path $y \to @ \to \lambda x \to x \to \lambda y$, contradicting the assumption. □

Redexes that fulfill Proposition 3.33 can be contracted by means of capture-permitting substitution.

Lemma 3.34. *(α-free). Suppose that there is no α-path in $G_\alpha((\lambda x.M)\,N)$ starting at a free variable in N and ending in M. Then $M[\![x\backslash N]\!] \equiv_\alpha M[x\backslash N]$.*

Proof. By induction and cases on the formation of the term substituted in (which is M), the only interesting case being when the term is an abstraction $\lambda y.M'$ with $x \neq y$. Then either x is not free in M' and N gets erased or else by the assumption, y is not free in N (shown in Proposition 3.33), hence

$$
\begin{array}{rcll}
(\lambda y.M)[\![x\backslash N]\!] & = & \lambda y.M[\![x\backslash N]\!] & \text{as } y \notin \mathcal{F}V(N) \\
 & \equiv_\alpha & \lambda y.M[x\backslash N] & \text{I.H.} \\
 & = & (\lambda y.M)[x\backslash N] & \qquad\qquad \square
\end{array}
$$

Lemma 3.35. *$\to_\beta$ preserves α-path-freeness.*

Proof. Let $s \to_\beta t$. By assumption, if s contains no α-path, then also t doesn't. This proof works precisely as the proof of Lemma 3.15. The only difference is that we can also relate r-edges of created redexes and legal paths to the source term s. Therefore, we only consider r-edges and legal paths in this proof. We use primed variables (e.g. p') to range over positions in the target term t, indicating the positions they trace back to in the source term s, by unpriming (e.g. p). Let $(p', p'1)$ be an r-edge in $G_\alpha(t)$. There is an r-edge from p to $p1$ in $G_\alpha(s)$, if p' and q' are in the same component. Otherwise, by Lemma 3.24, there is a legal path connecting their origins p and q. Moreover, if we have a legal path in N from p' to q', we also have a legal path in M between the origins p, q.

Since we can relate every type of edge and every legal path from p' to q' in $G_\alpha(t)$ to an edge or path from p and q in $G_\alpha(s)$ and via the same final reasoning done in the proof of Lemma 3.15 (legal paths in t just map to legal paths in s), it follows that if we have an arbic α-path in $G_\alpha(t)$, then we have an arbic α-path in $G_\alpha(s)$. □

Lemma 3.36. *Let M be a λ-term. If M contains no α-path, then M avoids α.*

Proof. By Lemma 3.35 we know that α-path-freeness is preserved by β-reduction. Since by Lemma 3.34 we know that we can contract such terms by means of capture-permitting substitution, no α is needed in the whole reduction sequence. □

We have already seen that we can get rid of arbic α-paths by naming all binders distinctly and distinct from free variables. This was possible because the starting and the ending position of these paths are always parallel. For α-paths this is not always the case anymore, as the Figure 3.13a and Figure 3.13b show. The α-paths in these graphs both start at variable z and end at the binder λz (Irrelevant edges were omitted in the illustrations. In Figure 3.13b also the legal paths were simplified to a single edge). Such paths are *unremovable* since we cannot since we cannot α-rename only the λ-node (without renaming the variable).

Definition 3.37. An α-path is called *unremovable*, if it starts at variable occurrence at position $p1q$ and ends at its binder at position p ($p \prec p1q$).

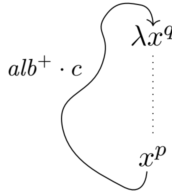
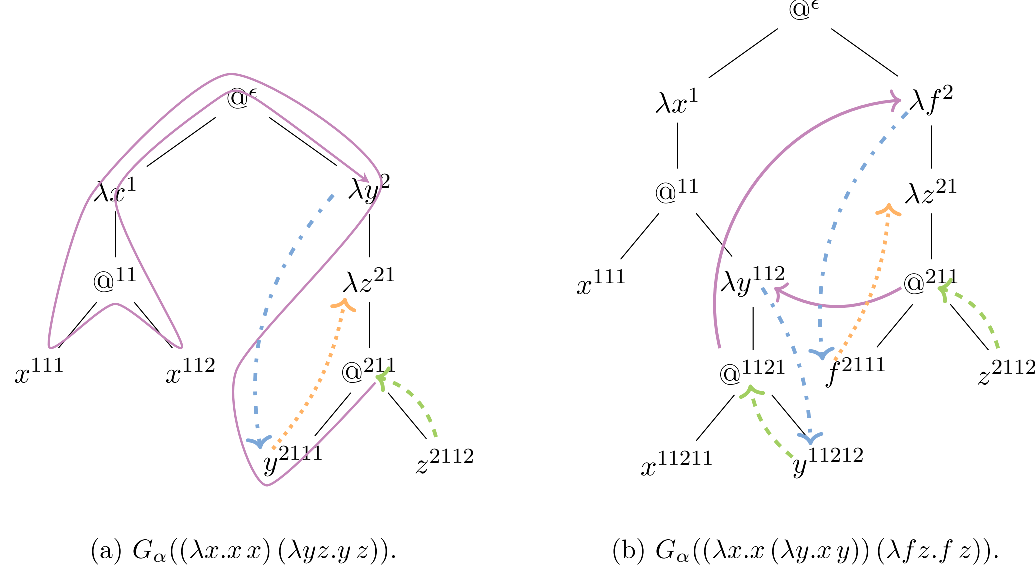


Figure 3.14: Illustration of an unremovable α-path.

The example term in Figure 3.13b illustrates that an unremovable α-path does not necessarily have to contain *albic*-paths or legal paths from a position p to a position q with $q \prec p$.



(a) $G_\alpha((\lambda x.x\,x)\,(\lambda yz.y\,z))$.

(b) $G_\alpha((\lambda x.x\,(\lambda y.x\,y))\,(\lambda fz.f\,z))$.

Figure 3.13: Examples of λ-terms with unremovable α-paths.

Lemma 3.38. *For every λ-term M containing no unremovable α-paths, there exists a λ-term N where $M \equiv_\alpha N$, such that N does not contain any α-paths.*

Proof. Assume we α-convert M to N such that all binders are named distinctly and distinct from the free variables. Since we exclude unremovable α-paths, we know that for any other *albic*-path in M we have $q \not\prec p$, for p being the position of the first v-node and q the position of the last λ-node in an *albic*-path. By construction of N, we know that for the variable x at position p and the binder λy we have $x \neq y$, if $q \not\prec p$. □

Since we can only guarantee α-free reductions when we have no α-paths in a λ-term, it is interesting to analyze in which λ-calculi they can occur and in which ones not. If we can prove that unremovable α-paths cannot exist in a specific λ-calculus, then we know that we can always avoid α. We will see in the subsequent chapters that we can indeed exclude their existence in the underlined λ-calculus (Chapter 4) and the affine λ-calculus (Chapter 5). For the weak λ-calculus (Chapter 6) and the safe λ-calculus (Chapter 7) we show that unremovable α-paths characterize redexes that will not be contracted or only simultaneously with other redexes (such that α-conversion can be avoided). In the simply-typed λ-calculus (Chapter 8) and in the untyped λ-calculus (Chapter 9) unremovable α-paths can occur. We will give examples of simply typed λ-terms and untyped λ-terms that have no α-free simulations.

Remark 3.39. In some cases, we could ignore unremovable α-paths, in the sense that we can still allow naïve substitutions. For example, when we know that the result will not be affected by a potential name collision. Consider the term $(\lambda x.y)\,((\lambda x.x\,x)\,(\lambda yz.y\,z))$. As we just saw in Figure 3.13, we have an unremovable α-path in $(\lambda x.x\,x)\,(\lambda yz.y\,z)$. However, we can reduce to the normal form y independently of whether we apply α-conversion

when needed or not. That is because the subterm $\lambda x.y$ erases its argument (we call such terms *erasing* terms). When asking the question about α-avoidance, we could therefore allow the naïve substitution in this case. That would require the capability of deriving a safe reduction sequence. This, however, is a different problem not covered in this thesis. Still, it is an interesting problem to investigate in future work.

4 Finite Developments

Reductions of residuals, also known as *developments*, are finite. This was proved already in 1936 by Church–Rosser for the λI-calculus [11] and then generalized to the full λ-calculus by Schroer [27] and independently by Hindley [19]. In developments, there is no redex-creation. We can impose this restriction by underlining initial redexes and allowing only the contraction of labeled redexes. This chapter discusses the underlined λ-calculus and shows that we can always avoid the need for α-conversion can always in this calculus.

Definition 4.1. The set $\underline{\Lambda}$ of underlined λ-terms is inductively defined as follows [32]:

($\underline{\text{var}}$) $x \in \underline{\Lambda}$, for all variables x

($\underline{\text{app}}$) $M, N \in \underline{\Lambda} \Longrightarrow M\,N \in \underline{\Lambda}$

($\underline{\text{abs}}$) $M \in \underline{\Lambda} \Longrightarrow \lambda x.M \in \underline{\Lambda}$

($\underline{\text{beta}}$) $M, N \in \underline{\Lambda} \Longrightarrow (\underline{\lambda} x.M)\,N \in \underline{\Lambda}$

Example 4.2. $(\lambda x.x)\,y \in \underline{\Lambda}$, $(\underline{\lambda} x.x)\,y \in \underline{\Lambda}$ and $(\underline{\lambda} x.x) \notin \underline{\Lambda}$

Definition 4.3. The underlined β-reduction ($\underline{\beta}$) contracts only underlined redexes:

$$(\underline{\lambda} x.M)\,N \quad \to_{\underline{\beta}} \quad M[\![x \backslash N]\!]$$

where $M[\![x \backslash N]\!]$ denotes the capture-avoiding substitution.

According to Definition 4.3 terms with no underlined lambdas are in $\underline{\beta}$-normal form.

Example 4.4. $(\underline{\lambda} x.x\,x)\,(\lambda x.x\,x) \to_{\underline{\beta}} (\lambda x.x\,x)\,(\lambda x.x\,x)$ and $(\lambda x.x\,x)\,(\lambda x.x\,x)$ is in $\underline{\beta}$-normal form

Example 4.5. $(\underline{\lambda} x\lambda y.x\,y)\,y \to_{\alpha} (\underline{\lambda} x\lambda z.x\,z)\,y \to_{\underline{\beta}} \lambda z.yz$

Proposition 4.6. *We have no redex creation in the underlined λ-calculus.*

Proof. Assume $M \to_{\underline{\beta}} N$ and that we have a created redex in N. From Proposition 2.73 we know that a redex at position p' in N is a created redex, iff the origins p of p' and q of $p'1$ in M are from a different component. This however would imply that we must have a subterm of shape $\underline{\lambda} x.M'$ in M. From Definition 4.1 we know that $\underline{\lambda} x.M' \notin \underline{\Lambda}$ and therefore we have $M \notin \underline{\Lambda}$ giving the contradition. □

Corollary 4.7. *A reduction sequence of underlined β-steps $M \to_{\underline{\beta}} M_1 \to_{\underline{\beta}} M_2 \to_{\underline{\beta}} \ldots$ is a development of M.*

Proof. Since by Proposition 4.6 we know that $\rightarrow_{\underline{\beta}}$ does not create redex, in $M \rightarrow_{\underline{\beta}} M_1 \rightarrow_{\underline{\beta}} M_2 \rightarrow_{\underline{\beta}} \ldots$ in every reduction step we only contract residuals of redexes in $\overline{M}$. By Definition 2.82 such a reduction sequence is a development. □

We use the arbic α-paths (Definition 3.9) to characterize the need for α-conversion in finite developments.

Definition 4.8. The mapping $T_{\underline{\lambda}}(M)$ of an underlined λ-term M to an ordinary λ-term N is the obvious homomorphic mapping forgetting the underlining.

Definition 4.9. The set of r-edges $E_{\underline{r}}(M)$ of an underlined λ-term M is inductively defined as:

$$E_{\underline{r}}(M,p) = \begin{cases} \{\} & \text{if } M = x \\ E_{\underline{r}}(N, p\cdot 1) & \text{if } M = \lambda x.N \\ E_{\underline{r}}(N_1, p\cdot 1) \cup E_{\underline{r}}(N_2, p\cdot 2) & \text{if } M = N_1\, N_2 \\ \{[p, p1]\} \cup E_{\underline{r}}(N_1, p\cdot 11) \cup E_{\underline{r}}(N_2, p\cdot 2) & \text{if } M = (\underline{\lambda} x.N_1)\, N_2 \end{cases}$$

where p serves as accumulator. We write $E_{\underline{r}}(M)$ as abbreviation for $E_{\underline{r}}(M, \epsilon)$.

Definition 4.10. The α-graph of an underlined λ-term M is the α-graph of $T_{\underline{\lambda}}(M)$, $G_\alpha(T_{\underline{\lambda}}(M))$, where we restrict the set of r-edges $E_{\underline{r}}(M)$.

Proposition 4.11. *Let $G_\alpha((\underline{\lambda} x.M)\, N)$ contain no arbic α-path. Suppose a free occurrence of x in M is connected to a binder λy either through a* c*-edge or through an* arbic*-path. Then y is not free in N.*

Proof. Otherwise there is a arbic α-path $y \rightarrow @ \rightarrow \lambda x \rightarrow x \rightarrow \cdots \rightarrow \lambda y$, contradicting the assumption. □

Redexes that fulfill Proposition 4.11 can be contracted by means of capture-permitting substitution.

Lemma 4.12. *(α-free). Suppose that if in $G_\alpha((\underline{\lambda} x.M)\, N)$ there is an* arbic *α-path from a free occurrence of x in M to a binder λy, then y is not free in N. Then $M[\![x\backslash N]\!] = M[x\backslash N]$.*

Proof. By induction and cases on the formation of the term substituted in (which is M), the only interesting case being when the term is an abstraction $\lambda y.M'$ with $x \neq y$. Then either x is not free in M' or else by the assumption, y is not free in N, hence

$$\begin{aligned} (\lambda y.M)[\![x\backslash N]\!] = {} & \lambda y.M[\![x\backslash N]\!] && \text{as } y \notin \mathcal{FV}(N) \\ = {} & \lambda y.M[x\backslash N] && \text{I.H.} \\ = {} & (\lambda y.M)[x\backslash N] \end{aligned}$$

□

Lemma 4.13. *Arbic α-path-freeness is preserved by $\underline{\beta}$-reduction.*

Proof. We have already shown in Lemma 3.16 that this property holds for developments. Assume $M \to_{\underline{\beta}} N$ and $T_{\underline{\lambda}}(M) \to_{\beta} N'$ for an arbitrary underlined λ-term M. The set of r-edges $E_{\underline{r}}(N)$ exactly correspond to the set of r-edges $E_r(N')$ restricted only to those denoting residuals of redexes in $T_{\underline{\lambda}}(M)$. In particular, $\langle G_\alpha(N) \rangle = G_\alpha(N)$. The argumentation of Lemma 3.15 therefore also applies for $\to_{\underline{\beta}}$. □

Lemma 4.14. *Let M be an arbitrary underlined λ-term. If M contains no α-path, then every reduction sequence from M is α-free.*

Proof. By Lemma 4.13 we know that *arbic* α-path-freeness is preserved by $\underline{\beta}$-reduction. Since by Lemma 4.12 we know that we can contract such terms by means of capture-permitting substitution, no α is needed in the whole reduction sequence. □

Lemma 4.15. *For every underlined λ-term M there exists an underlined λ-term N such that $M \equiv_\alpha N$ and N avoids α.*

Proof. By naming the λ-nodes at the end of arbic α-paths distinctly and distinct from any variable occurring free in some argument, we can get rid of any arbic α-path. □

Corollary 4.16. *In the underlined λ-calculus we can avoid α (Definition 2.94).*

5 The affine λ-calculus

In an affine λ-term M, in each subterm each variable has at most one free occurrence.

Definition 5.1. The set Λ_{AFF} of *affine* λ-terms is a subset of Λ and inductively defined as follows:

(var) $x \in \Lambda \implies x \in \Lambda_{AFF}$, for all variables x

(app) $M\ N \in \Lambda \implies M\ N \in \Lambda_{AFF}$, if $M, N \in \Lambda_{AFF}$ and $\mathcal{FV}(M) \cap \mathcal{FV}(N) = \emptyset$

(abs) $\lambda x.M \in \Lambda \implies \lambda x.M \in \Lambda_{AFF}$, if $M \in \Lambda_{AFF}$

Example 5.2. $\lambda x.(\lambda y.y)\ x \in \Lambda_{AFF}$, $\lambda x.(\lambda y.x\ y)\ x$ and $x\ x \notin \Lambda_{AFF}$

Proposition 5.3. *($\to_\beta$ preserves affinity) Let $M \in \Lambda_{AFF}$. If $M \to_\beta N$, then $N \in \Lambda_{AFF}$.*

Proof. Assume $N \notin \Lambda_{AFF}$. Then there must exist multiple free occurrences of a variable y in the scope of a binder λy in N. Since $M = C[(\lambda x.M')\ M'']$ (with $(\lambda x.M')\ M''$ being the contracted redex in the context C) is affine, by Definition 5.1 also each subterm of M is affine and we can only form N, if one free occurrence is duplicated by the β-step. But that would imply that $M \notin \Lambda_{AFF}$ as x would have multiple occurrences in M'. □

Definition 5.4. The size of a λ-term M is defined recursively as:

$$\begin{aligned} \mathit{size}(x) &= 1 \\ \mathit{size}(\lambda x.M) &= 1 + \mathit{size}(M) \\ \mathit{size}(M\ N) &= 1 + \mathit{size}(M) + \mathit{size}(N) \end{aligned}$$

Proposition 5.5. *Let M, N be λ-terms. Then $\mathit{size}(M[\![x\backslash N]\!]) = \mathit{size}(M) + k \cdot (\mathit{size}(N) - 1)$, where k is the number of free occurrences of x in M.*

Proof. We prove it by induction on M.

- $M = x$. Then $x[\![x\backslash N]\!] = N$ and $\mathit{size}(x[\![x\backslash N]\!]) = \mathit{size}(x) + 1 \cdot (\mathit{size}(N) - 1) = 1 + \mathit{size}(N) - 1 = \mathit{size}(N)$.
- $M = y$. Then $y[\![x\backslash N]\!] = y$ and $\mathit{size}(y[\![x\backslash N]\!]) = \mathit{size}(y) + 0 \cdot (\mathit{size}(N) - 1) = \mathit{size}(y)$.
- $M = \lambda x.M_1$. Then $M[\![x\backslash N]\!] = M$ and $\mathit{size}(M[\![x\backslash N]\!]) = \mathit{size}(M) + 0 \cdot (\mathit{size}(N) - 1) = \mathit{size}(M)$.
- $M = \lambda y.M_1$. Then $M[\![x\backslash N]\!] = \lambda y.M_1[\![x\backslash N]\!]$ and $\mathit{size}((\lambda y.M_1)[\![x\backslash N]\!]) = 1 + \mathit{size}(M_1[\![x\backslash N]\!]) = 1 + \mathit{size}(M_1) + k \cdot (\mathit{size}(N) - 1) = \mathit{size}(\lambda y.M_1[\![x\backslash N]\!])$.

- $M = M_1\ M_2$. Then $M[\![x\backslash N]\!] = M_1[\![x\backslash N]\!]\ M_2[\![x\backslash N]\!]$ and $\mathit{size}(M[\![x\backslash N]\!]) = \mathit{size}(M_1\ M_2) + k \cdot (\mathit{size}(N) - 1) = 1 + \mathit{size}(M_1) + \mathit{size}(M_2) + k \cdot (\mathit{size}(N) - 1) = 1 + \mathit{size}(M_1) + \mathit{size}(M_2) + (k_{M_1} + k_{M_2}) \cdot (\mathit{size}(N) - 1) = \mathit{size}(M_1[\![x\backslash N]\!]) + \mathit{size}(M_2[\![x\backslash N]\!]) + 1 = \mathit{size}(M_1[\![x\backslash N]\!]\ M_2[\![x\backslash N]\!])$

□

Proposition 5.6. *Let $M \in \Lambda_{AFF}$. If $M \to_\beta N$, then $\mathit{size}(N) < \mathit{size}(M)$.*

Proof. If $M \to_\beta N$, then M contains a redex in some context C, therefore we can write M as $C[(\lambda x.M_1)\ M_2]$ and N as $C[M_1[\![x\backslash M_2]\!]]$. x occurs at most once free in M_1, so we have $\mathit{size}(M) = \mathit{size}(C[(\lambda x.M_1)\ M_2]) = \mathit{size}(C) + 1 + \mathit{size}(\lambda x.M_1) + \mathit{size}(M_2) > \mathit{size}(C) + \mathit{size}(M_1) + k \cdot (\mathit{size}(M_2) - 1) = \mathit{size}(C[M_1[\![x\backslash M_2]\!]]) = \mathit{size}(N)$, where k is 0 if x does not occur free in M_1 and 1 otherwise. □

Proposition 5.7. *Affine λ-terms are strongly normalizing.*

Proof. Trivially follows from Proposition 5.6. We can only apply finitely many β-steps, as the size continuously decreases. □

Proposition 5.8. *Let $M \in \Lambda_{AFF}$ and $M \to_\beta N$. If in M all binders are named distinctly and distinct from the free variables, then this name property also holds for N.*

Proof. Trivially follows by the fact that λ-nodes cannot be duplicated in a β-step. The number of λ-nodes even decreases by one. □

Lemma 5.9. *Let $M \in \Lambda_{AFF}$, $M \to_\beta N$ and $q \prec p$ for some positions p, q in M. If $p \blacktriangleright p'$ and $q \blacktriangleright q'$, then $q' \prec p'$.*

Proof. Since we have no duplication, each symbol has at most one copy in N. Let's recall the trace relation for the contracted redex $(\lambda x.M_1)\ M_2$ at position o in M, where the abstraction λx binds (at most) one variable x in M_1 at position $o11t$:

- (context) $p \blacktriangleright p$ if o is not prefix of p
- (body) $o11p \blacktriangleright op$ if $p \neq \epsilon$ and $p \neq q$
- (arg) $o2p \blacktriangleright otp$ for some t such that $o11t$ is bound by $o1$.

We distinguish the following cases where we have $p \prec q$, with $p \blacktriangleright p'$ and $q \blacktriangleright q'$:

1. p, q both in the context: Then as $p' = p$ and $q' = q$ so by assumption we have $p' \prec q'$.

2. $p = o11s_1$ and $q = o11s_2$ both in the body: Then from $p \prec q$ we know that $s_1 \prec s_2$ and we have $os_1 = p' \prec q' = os_2$.

3. $p = o2s_1$ and $q = o2s_2$ both in the argument: Then from $p \prec q$ we know that $s_1 \prec s_2$ and we have $ots_1 = p' \prec q' = ots_2$.

4. p is in the context and $q = o11s$ in the body. Then $p' = p$ and $q' = os$ and since $p \prec q$ we also have $p' \prec q'$.

5. p is in the context and $q = o2s$ in the argument. Then $p' = p$ and $q' = oqs$. Since we know that $p \prec o$ (because it is in the context), we also have $p' \prec q'$.

The other cases can be omitted because they violate the assumption that $p \prec q$.

□

In the affine λ-calculus, there are no unremovable α-paths. α can therefore always be avoided by choosing appropriate variable names, as proven next.

Lemma 5.10. *Let M be an arbitrary term in Λ_{AFF}. There are no unremovable α-paths in $G_\alpha(M)$.*

Proof. Since each β-step preserves the property proven in Lemma 5.9, we cannot have a reduct of M where for the copy of p (the position of a variable), p', and the copy q (the position of an abstraction), q', we have $p' \parallel q'$, if for the origins we have $q \prec p$. This would temporarily be needed to form a redex whose contraction causes a variable capture. Moreover, by Lemma 3.35 we know that we could map back such a situation to an (unremovable) α-path in $G_\alpha(M)$. We conclude that no such path can exist in M. □

Lemma 5.11. *For every affine λ-term M there exists an affine λ-term N such that $M \equiv_\alpha N$ and N avoids α.*

Proof. By Lemma 5.10 we know that no unremovable α-paths can exist so it follows by Lemma 3.38. □

Corollary 5.12. *In the affine λ-calculus we can avoid α (Definition 2.94).*

6 The weak λ-calculus

In the weak lambda calculus, the λ-terms are only reduced to weak head normal form (via the weak β-reduction). In this chapter, we show that also in the weak λ-calculus, we can allow α-free computations if we choose appropriate variable names.

Definition 6.1. A λ-term is in *weak head* normal form (WHNF), if and only if it is of the form

$$M\, N_1 \, \ldots \, N_k$$

where $k \geq 0$ and M either is a variable or an abstraction if $k = 0$ and a variable otherwise [24].

Example 6.2. $\lambda x.(\lambda y.y)\ x$ is in *WHNF*, $z\ ((\lambda y.y)\ x)$ is in *WHNF*, $(\lambda y.y)\ x$ is not in *WHNF*

What follows from Definition 6.1 is that we do not reduce inside abstractions. The weak β-reduction therefore restricts the ordinary β-reduction:

Definition 6.3. The *weak β-reduction* is the smallest relation $\rightarrow_{\beta l}$ on λ-terms satisfying the following rules:

$$(\beta_w)\ \frac{(\lambda x.M)\, N_1\, N_2\, \ldots\, N_k\ \rightarrow_{\beta}\ M[\![x \backslash N_1]\!]\, N_2\, \ldots\, N_k}{(\lambda x.M)\, N_1\, N_2\ \ldots\, N_k\ \rightarrow_{\beta w}\ M[\![x \backslash N_1]\!]\, N_2\, \ldots\, N_k} \quad \text{for } k \geq 1$$

The *naïve single-step $\rightarrow_{\beta w}$-reduction* denoted as $\rightarrow_{\beta_{l\ddot{\imath}}}$ is defined identically, except for the $\rightarrow_{\beta l}$-rule that uses capture-permitting substitution.

To show that we can always avoid α-conversion, we rely on the fact that bound variables are never released, as shown in Proposition 2.67. This property also holds for the weak β-reduction.

Lemma 6.4. *For every λ-term M there exists a λ-term N such that $M \equiv_\alpha N$ and any $\rightarrow_{\beta w}$-reduction from N is α-free.*

Proof. We prove it by showing that the name-collision characterized by an unremovable α-paths will not arise. Suppose we have an unremovable α-path in a λ-term M. From Definition 3.37 it follows that such a path has the shape $\sigma^{+}_{alb} \cdot c$, as illustrated in Figure 3.14. Assume, that at some point along the reduction sequence of M we reach a λ-term N, containing a redex whose contraction lead to the predicted name-collision. Let q be the position of the variable y occurring free in the argument of that redex in N. Since the position q originates from position p in M ($p \blacktriangleright p' \blacktriangleright \ldots \blacktriangleright q$) and the variable occurrence

at position p in M was bound, by Proposition 2.67 we know that also the variable y at position q in N is bound. This redex, therefore, occurs in the scope of the binder of y and will not be contracted. Any other α-path can be removed by naming each binder distinctly and distinct from the free variables, as proven in Lemma 3.38. □

Corollary 6.5. *In the weak λ-calculus we can avoid α (Definition 2.94).*

In [24, Section 11.3.2], Peyton Jones stated that the name-capture problem can never arise when reducing a valid functional program to weak head normal form. This is because a valid functional program has no free variables. Therefore, the argument of the top-level redex has no free variables either. We will use this result later on when proving the undecidability of α-avoidance for the leftmost–outermost reduction strategy in the untyped λ-calculus (Chapter 9).

Remark 6.6. We have shown that we can avoid α in the weak λ-calculus. However, we had to argue why a name-collision predicted by an unremovable α-path will not occur. It follows that α-paths only overapproximate the need for α-conversion in the weak λ-calculus. For example, consider the term in Figure 6.1. The α-path depicted in that figure predicts a variable capture that will never occur with a naïve weak β-reduction $\to_{\beta_{\ddot{\text{l}}\ddot{\text{i}}}}$. This is because the legal paths were developed for ordinary β-reduction. The weak β-reduction, however, does not contract redexes in the scope of abstractions. It remains an open question whether we could compute a set of weak legal paths that characterize only the redexes contracted in a weak β-reduction sequence. This could allow proving Peyton Jones' result via *weak* α-paths [1].

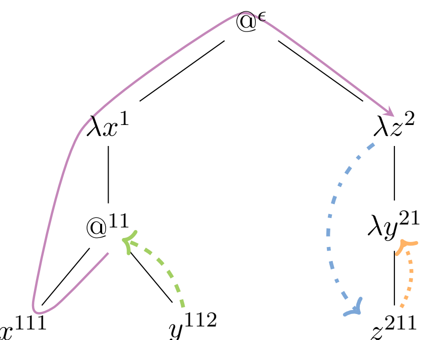


Figure 6.1: α-paths overapproximate the need for α-conversion for weak β-reduction.

[1]only a name suggestion

7 The safe λ-calculus

This chapter is about α-avoidance in the safe λ-calculus as developed by Blum and Ong [9]. In this calculus, a variable capture can never occur. The fundamental concept allowing α-free computations is known as the *safety* restriction. This syntactic restriction restricts the free occurrences of variables according to their type-theoretic order[1]. It was initially introduced for higher-order grammars, first implicitly by Damm [12] and then reformulated by Knapik [20]. Knapik showed that no fresh variables are needed when computing the value tree of a safe recursion scheme. This nice algorithmic property was then transposed to a homogeneous safe λ-calculus by Miranda, Ong and Aehlig [15] and then further simplified by Blum [9] (in this thesis, Miranda only gave the idea, Blum then formulated it in detail). Moreover, Blum showed that the restriction of types being homogeneous is not needed.

Definition 7.1. The set of *simple types* over the atomic type o is generated from following grammar:

$$T ::= o \mid T \to T.$$

By convention, $\to$ associates to the right and $A_1 \to \cdots \to A_n \to o$ is abbreviated as $(A_1, \ldots, A_n, o)$ and (o) as o.

Definition 7.2. (*type order* [5, p. 11]) The order of a type is given by

$$\begin{aligned} ord\, o &= 0 \\ ord\,(A \to B) &= max(1 + ord\, A, ord\, B) \end{aligned}$$

The order of a typed term or symbol is defined to be the order of its type. $ord\,\Gamma$, with Γ being a set of type assignments, denotes the lowest value in the set $\{ord\, A \mid x : A \in \Gamma\}$.

Example 7.3. o has order 0, (o, o, o) has order 1, $((o, o), o)$ has order 2

Example 7.4. $\lambda x^{(o,o)}.x : ((o, o), o, o)$ has order 2, $x : o$ has order 0.

Example 7.5. $ord\,\{x : (o, o), y : ((o, o), o)\} = 1$.

Figure 7.1 shows the system of rules of the safe λ-calculus by Blum [9]. This calculus is a sub-calculus of the simply-typed calculus à la Church. The conditions on the context in the *app* and *abs* cases ensure that the variables occurring free have order at least the order of the term they occur in. This is what is known as the *safety* restriction. The *asa* stands for *almost-safe* (application). The idea of an almost safe term is that it can be

[1] it also works with other rank functions [5, 3.1.5]

turned into a safe term via further applications or further abstractions[2]. For example, the application $(\lambda x^o y^o.x)\, z$ (with z of type o) is not safe but almost it is almost safe. As part of the application $(\lambda x^o y^o.x)\, z\, f$ (with f, z of type o) it forms a safe application.

$$(var)\ \frac{}{x : A \vdash_s x : A} \qquad (const)\ \frac{}{\vdash_s f : A}\ f : A \in \Xi \qquad (wk)\ \frac{\Gamma \vdash_s M : A}{\Delta \vdash_s M : A}\ \Gamma \subset \Delta$$

$$(app_{as})\ \frac{\Gamma \vdash_{asa} M : A \to B \quad \Gamma \vdash_s N : A}{\Gamma \vdash_{asa} M\, N : B} \qquad (\delta)\ \frac{\Gamma \vdash_s M : A}{\Gamma \vdash_{asa} M : A}$$

$$(app)\ \frac{\Gamma \vdash_{asa} M : A \to B \quad \Gamma \vdash_s N : A}{\Gamma \vdash_s M\, N : B}\ \ ord\, B \leq ord\, \Gamma$$

$$(abs)\ \frac{\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash_{asa} M : B}{\Gamma \vdash_s \lambda x_1^{A_1} \ldots x_n^{A_n}.M : (A_1, \ldots, A_n, B)}\ \ ord\, (A_1, \ldots, A_n, B) \leq ord\, \Gamma$$

Figure 7.1: The safe λ-calculus by Blum [5]

In the safe λ-calculus we have *combined* abstractions. This means that we can abstract a list of variables $\lambda x_1 \ldots x_n.M$, provided that they are distinct (*abs*-rule).

Example 7.6. $\lambda.x$ and $\lambda x^o.\lambda x^o.x$ are valid λ-terms of the safe λ-calculus, $\lambda x^o x^o.x$ is not.

Definition 7.7. A term M of type A is said to be *safe*, if the following statement is derivable with the inference rules of the safe λ-calculus listed in Figure 7.1:

$$\mathcal{FV}(M) \vdash_s M : A$$

Example 7.8. The statement $\vdash_s \lambda f^{((o,o),o)}.f\,(\lambda x^o.f\,(\lambda y^o.y)) : (((o,o),o),o)$ can be derived with the rules of the safe λ-calculus, as shown in derivation tree of Figure 7.2. This term is therefore safe.

Example 7.9. The statement $\vdash_s \lambda f^{((o,o),o)}.f\,(\lambda x^o.f\,(\lambda y^o.x)) : (((o,o),o),o)$ cannot be derived with the rules of the safe λ-calculus, as shown in Figure 7.3. The variable x has order 0 which is smaller than the order of the subterm $\lambda y^o.x$ (order 1) it occurs free in. This subterm, and consequently the whole term, are therefore unsafe.

A term $M = \lambda \overline{x}.M'$ is safe if M' is safe and for each variable y free in M we have $ord\, x_i < ord\, y$ for all $x_i \in \overline{x}$ (the abstracted variable must all have order smaller than $ord\, y$).

The safety restriction was, as already mentioned, initially introduced for higher-order grammars. In higher-order grammars, we have simultaneous contraction of nested redex,

[2]We will see later on, that allowing the derivation almost safe terms allows to derive safe terms that get stuck (they would be further reducible in the simply-typed λ-calculus)

$$\frac{}{y : o \vdash_s y : o}\ var$$

$$\frac{y : o \vdash_s y : o}{f : ((o,o),o), y : o \vdash_s y : o}\ wk$$

$$\frac{f : ((o,o),o), y : o \vdash_s y : o}{f : ((o,o),o), y : o \vdash_{asa} y : o}\ \delta$$

$$\frac{f : ((o,o),o), y : o \vdash_{asa} y : o}{f : ((o,o),o) \vdash_s \lambda y^o.y : (o,o)}\ abs$$

$$\frac{f : ((o,o),o) \vdash_s \lambda y^o.y : (o,o)}{f : ((o,o),o), x : o \vdash_s \lambda y^o.y : (o,o)}\ wk$$

$$\frac{}{f : ((o,o),o) \vdash_s f : ((o,o),o)}\ var$$

$$\frac{f : ((o,o),o) \vdash_s f : ((o,o),o)}{f : ((o,o),o), x : o \vdash_s f : ((o,o),o)}\ wk$$

$$\frac{f : ((o,o),o), x : o \vdash_s f : ((o,o),o)}{f : ((o,o),o), x : o \vdash_{asa} f : ((o,o),o)}\ \delta$$

$$\frac{f : ((o,o),o), x : o \vdash_{asa} f : ((o,o),o) \qquad f : ((o,o),o), x : o \vdash_s \lambda y^o.y : (o,o)}{f : ((o,o),o), x : o \vdash_{asa} f\,(\lambda y^o.y) : o}\ app_{as}$$

$$\frac{f : ((o,o),o), x : o \vdash_{asa} f\,(\lambda y^o.y) : o}{f : ((o,o),o) \vdash_s \lambda x^o.f\,(\lambda y^o.y) : (o,o)}\ abs$$

$$\frac{}{f : ((o,o),o) \vdash_s f : ((o,o),o)}\ var$$

$$\frac{f : ((o,o),o) \vdash_s f : ((o,o),o)}{f : ((o,o),o) \vdash_{asa} f : ((o,o),o)}\ \delta$$

$$\frac{f : ((o,o),o) \vdash_{asa} f : ((o,o),o) \qquad f : ((o,o),o) \vdash_s \lambda x^o.f\,(\lambda y^o.y) : (o,o)}{f : ((o,o),o) \vdash_{asa} f\,(\lambda x^o.f\,(\lambda y^o.y)) : o}\ app_{as}$$

$$\frac{f : ((o,o),o) \vdash_{asa} f\,(\lambda x^o.f\,(\lambda y^o.y)) : o}{\vdash_s \lambda f^{((o,o),o)}.f\,(\lambda x^o.f\,(\lambda y^o.y)) : (((o,o),o),o)}\ abs$$

Figure 7.2: The derivation tree of a safe, simply-typed λ-term.

$$\frac{\lightning}{\vdash_s x : o}$$

$$\frac{\vdash_s x : o}{f : ((o,o),o), y : o \vdash_s x : o}\ wk$$

$$\frac{f : ((o,o),o), y : o \vdash_s x : o}{f : ((o,o),o), y : o \vdash_{asa} x : o}\ \delta$$

$$\frac{f : ((o,o),o), y : o \vdash_{asa} x : o}{f : ((o,o),o) \vdash_s \lambda y^o.x : (o,o)}\ abs$$

$$\frac{f : ((o,o),o) \vdash_s \lambda y^o.x : (o,o)}{f : ((o,o),o), x : o \vdash_s \lambda y^o.x : (o,o)}\ wk$$

$$\frac{}{f : ((o,o),o) \vdash_s f : ((o,o),o)}\ var$$

$$\frac{f : ((o,o),o) \vdash_s f : ((o,o),o)}{f : ((o,o),o), x : o \vdash_s f : ((o,o),o)}\ wk$$

$$\frac{f : ((o,o),o), x : o \vdash_s f : ((o,o),o)}{f : ((o,o),o), x : o \vdash_{asa} f : ((o,o),o)}\ \delta$$

$$\frac{f : ((o,o),o), x : o \vdash_{asa} f : ((o,o),o) \qquad f : ((o,o),o), x : o \vdash_s \lambda y^o.x : (o,o)}{f : ((o,o),o), x : o \vdash_{asa} f\,(\lambda y^o.x) : o}\ app_{as}$$

$$\frac{f : ((o,o),o), x : o \vdash_{asa} f\,(\lambda y^o.x) : o}{f : ((o,o),o) \vdash_s \lambda x^o.f\,(\lambda y^o.x) : (o,o)}\ abs$$

$$\frac{}{f : ((o,o),o) \vdash_s f : ((o,o),o)}\ var$$

$$\frac{f : ((o,o),o) \vdash_s f : ((o,o),o)}{f : ((o,o),o) \vdash_{asa} f : ((o,o),o)}\ \delta$$

$$\frac{f : ((o,o),o) \vdash_{asa} f : ((o,o),o) \qquad f : ((o,o),o) \vdash_s \lambda x^o.f\,(\lambda y^o.x) : (o,o)}{f : ((o,o),o) \vdash_{asa} f\,(\lambda x^o.f\,(\lambda y^o.x)) : o}\ app_{as}$$

$$\frac{f : ((o,o),o) \vdash_{asa} f\,(\lambda x^o.f\,(\lambda y^o.x)) : o}{\vdash_s \lambda f^{((o,o),o)}.f\,(\lambda x^o.f\,(\lambda y^o.x)) : (((o,o),o),o)}\ abs$$

Figure 7.3: Example of an unsafe, simply typed λ-term.

whereas in the ordinary λ-calculus we contract them one by one. Ordinary β-reduction however does not preserve safety, as shown in Example 7.10.

Example 7.10. Let $\Gamma = \{f : (o, o, o), v : o, w : o\}$ and $M = (\lambda x^o y^o . f\, x\, y)\, v\, w$. We have $\Gamma \vdash_s M : ((o, o, o), o, o, o)$, but $\underline{(\lambda x^o y^o . f\, x\, y)\, v}\, w \rightarrow_\beta (\lambda y^o . f\, v\, y)\, w$ which is unsafe as $\lambda y^o . f\, v\, y$ is unsafe. With an additional β-step we get $f\, v\, w$ which is safe again [5].

As Example 7.10 suggests, we may have to contract nested redexes simultaneously if we want to preserve safety. This requires a notion for simultaneous substitution. The following definitions of simultaneous substitution were taken from [5, Definition 2.6 and 2.7] and adapted to deal with combined abstractions. In the case of a singleton substitution and ordinary abstractions (a single variable) these definitions simplify to the definitions of the substitutions we already know (Definition 2.56 and 2.49, respectively).

Definition 7.11. (*simultaneous* capture-avoiding substitution) The simultaneous capture-avoiding substitution of $N_1, \ldots, N_n$ for the (distinct) variables $x_1, \ldots, x_n$ in M, written $M\{x_1 \backslash N_1, \ldots, x_n \backslash N_n\}$ and abbreviated as $M\{\overline{x} \backslash \overline{N}\}$ is inductively defined as follows:

$$
\begin{aligned}
M\{\emptyset \backslash \emptyset\} &= M \\
x_i\{\overline{x} \backslash \overline{N}\} &= N_i \\
y\{\overline{x} \backslash \overline{N}\} &= y \\
M_1\, M_2\{\overline{x} \backslash \overline{N}\} &= M_1\{\overline{x} \backslash \overline{N}\}\, M_2\{\overline{x} \backslash \overline{N}\} \\
(\lambda \overline{y}.M)\{\overline{x} \backslash \overline{N}\} &= (\lambda \overline{y}.M)\{x_1 \backslash N_1, \ldots, x_{i-1} \backslash N_{i-1}, x_{i+1} \backslash N_{i+1}, \ldots, x_n \backslash N_n\} \\
&\quad \text{if } x_i \in \overline{y} \\
(\lambda \overline{y}.M)\{\overline{x} \backslash \overline{N}\} &= \lambda \overline{y}.M\{\overline{x} \backslash \overline{N}\} \\
&\quad \text{where } \overline{y} \cap FV(N_i') = \emptyset \text{ for all } N_i' \in \overline{N'}. \\
(\lambda \overline{y}.M)\{\overline{x} \backslash \overline{N}\} &= \lambda \overline{y'}.M\{\overline{y} \backslash \overline{y'}\}\{\overline{x} \backslash \overline{N}\} \\
&\quad \text{where } y_i' = y_i \text{ if } \exists i . y_i \notin FV(N_i) \\
&\quad \text{otherwise } y_i \text{ is fresh for } \lambda \overline{y}.M \text{ and } N.
\end{aligned}
$$

In contrast to the simultaneous capture-avoiding substitution we have the simultaneous capture-permitting substitution.

Definition 7.12. (*simultaneous* capture-permitting substitution) The simultaneous capture-permitting substitution is inductively defined as follows:

$$
\begin{aligned}
M[\emptyset \backslash \emptyset] &= M \\
x_i[\overline{x} \backslash \overline{N}] &= N_i \\
y[\overline{x} \backslash \overline{N}] &= y \\
M_1\, M_2[\overline{x} \backslash \overline{N}] &= M_1[\overline{x} \backslash \overline{N}]\, M_2[\overline{x} \backslash \overline{N}] \\
(\lambda \overline{y}.M)[\overline{x} \backslash \overline{N}] &= \lambda \overline{y}.M[\overline{x'} \backslash \overline{N'}] \\
&\quad \text{where } x_i' \notin \overline{y} \text{ for all } x_i' \in \overline{x}'.
\end{aligned}
$$

Remark 7.13. Note that the statement $M[x_1\backslash N_1][x_2\backslash N_2] = M[x_1x_2\backslash N_1N_2]$ is not true in general, as x_2 may be free in N_1. For example, consider the λ-term $M = x$. Then $M[x\backslash y][y\backslash z] = z$ and $M[xy\backslash yz] = y$.

Definition 7.14. (*almost safe application* [5]) A term is an *almost safe application* if it is safe or if it is of the form $N_1 \ldots N_m$ for some $m \geq 1$ where N_1 is not an application and for every $1 \leq i \leq m, N_i$ is safe.

Definition 7.15. (*safe redex* [5, Definition 3.21]) In the safe lambda calculus, since we have simultaneous substitution, a redex is a succession of several standard redexes. An untyped safe redex is an untyped almost safe application of the form $(\lambda x_1 \ldots x_n.M)\, N_1 \ldots N_l$ for some $l, n \geq 1$ such that M is an almost safe application.

Consequently $\lambda x_1 \ldots x_n.M$ is safe and each N_i is safe for $1 \leq i \leq n$. Additionally following property is satisfied for all $x_i \in FV(N_1) \cup \cdots \cup FV(N_l)$:

$$ord\, A \leq ord\, x_i$$

where $M : A$, $\lambda x_1 \ldots x_n.M :$ and $(\lambda x_1 \ldots x_n.M)\, N_1 \ldots N_m$.

Definition 7.16. (*safe redex contraction*) The relation β_s is defined on the set of safe redexes as follows:

$$(\lambda x_1^{A_1} \ldots x_n^{A_n}.M)\, N_1\, \ldots\, N_l \quad \beta_s \quad (\lambda x_{k+1} \ldots x_n.M)\langle x_1\backslash N_1, \ldots, x_k\backslash N_k\rangle$$

where $\lambda.M = M$ and $M\langle\overline{x}\backslash\overline{N}\rangle$ denotes the simultaneous capture-permitting substitution, provided that $\lambda x_{k+1} \ldots x_n.M$ is safe.

Definition 7.17. The *safe β-reduction*, written as $\rightarrow_{\beta_s}$, is the compatible closure of the relation β_s with respect to the formation rules of the safe lambda calculus.

Blum proved that $\rightarrow_{\beta_s}$ preserves safety [5, Lemma 3.24]. From Definition 7.16 it follows that a safe redex can only be contracted if enough arguments are provided. More precisely, if an argument of order k is provided, all arguments of order k and higher must be provided. Moreover, we have to apply simultaneous substitution. This is highlighted in Example 7.18.

Example 7.18. Let $M = \lambda y^o.(\lambda x^o y^o.x)\, y$. M is safe and the subterm $(\lambda x^o y^o.x)\, y$ is a safe redex however by Definition 7.16 it cannot be contracted because $\lambda y^o.x$ is not safe. The variables in the abstraction $\lambda x^o y^o$ have equal order and therefore we need to supply an argument for both. This would allow to contract them simultaneously.

As we can see in Example 7.18, even though the term $\lambda y^o.(\lambda x^o y^o.x)\, y$ is safe, it cannot be further reduced to reach a normal form. Safe terms derivable in the safe λ-calculus by Blum may therefore get *stuck*. Stuck in the sense that in the simply-typed λ-calculus we could further reduce the term. Consequently, we may have different normal forms. In the final part of this chapter, we will come back to this problem.

Example 7.19 shows that the safety restriction and the safe β-reduction are not enough to avoid variable capture.

Example 7.19. Let $A = ((o,o),o)$, $B = (A,(o,o),o,A)$, $\Gamma = \{y : (((o,o),o),o), z : ((o,o),o)\}$ and $M = (\lambda x^A y^{(o,o)} z^o.x)\,(\lambda q^{(o,o)}.y\,z)$. M is safe and we have $\Gamma, x : A \vdash_s \lambda y^{(o,o)} z^o.x : ((o,o),o,A)$, but safe β-reduction still would require α-conversion, as $M\ \beta_s\ \lambda y^{(o,o)} z^o q^{(o,o)}.y\,z$ where the y got captured.

One additional constraint is therefore needed to ensure that α-conversion is never needed – it's the order consistency restriction discussed next.

Definition 7.20. (*order consistency* [5, p. 53]) We say that a set Γ of typing assumptions of the form $x : A$, for some variable x and simple type T, is *order-consistent* if all the types assigned to a given variable are of the same order

$$x : A_1 \in \Gamma \wedge x : A_2 \in \Gamma \implies ord\,A_1 = ord\,A_2$$

Example 7.21. The set $\{x : o, y : (o,o)\}$ is order-consistent, the set $\{x : o, x : (o,o)\}$ is not order-consistent.

Definition 7.22. Let $M \in \Lambda_T$ be an annotated term. We define the set $Ass(M)$ as the *set of type-assignments* induced by the type annotations in M:

$$\begin{aligned} Ass(x) &= \emptyset \\ Ass(M\,N) &= Ass(M) \cup Ass(N) \\ Ass(\lambda x^T.M) &= \{x : T\} \cup Ass(M) \\ Ass(\Gamma \vdash_{Ch} M : T) &= \Gamma \cup Ass(M). \end{aligned}$$

A type-annotated term M is said to be *order-consistent* just if the set $Ass(M)$ is [5, p. 53].

The term shown in Example 7.19 is not order-consistent as the set of type-assignments is $\{y : (((o,o),o),o), z : ((o,o),o), x : ((o,o),o), y : (o,o), z : o, q : (o,o)\}$ where y has two types of different order. To make the term of Example 7.19 order-consistent, we need to rename the λy.

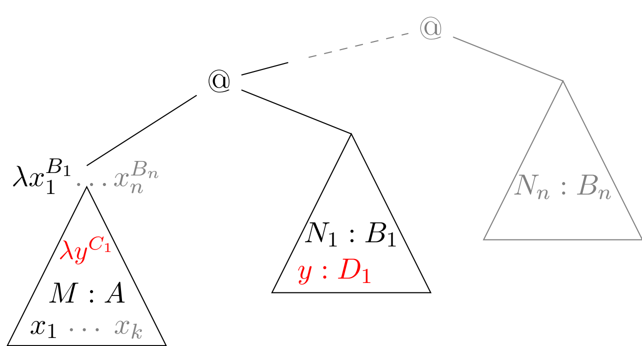
Definition 7.23. (*safe variable typing convention* [5]) In any definition, theorem or proof involving countably many terms, it is assumed that the set of terms involved is order-consistent.

Lemma 7.24. *(No-variable-capture lemma [5, Lemma 3.15]) In the safe lambda calculus à la Church, there is no variable capture when performing simultaneous capture-permitting substitution provided that we adopt the safe variable typing convention: If* $\Gamma, \overline{x} : \overline{B} \vdash_s M : A$*,* $\Gamma \vdash_s N_1 : B_1, \ldots, \Gamma \vdash_s : N_n : B_n$*, where* $|\overline{x}| = n$ *then*

$$M\{\overline{N}\backslash\overline{x}\} = M[\overline{N}\backslash\overline{x}]$$

Proof. This was proven by Blum in [5]. Suppose we have $y \in \mathcal{FV}(N_1)$ as visualized in the figure below. Because N_1 is safe we know that $ord\,y \geq ord\,N_1$. We would have a variable capture if x_1 occurs free in M in the scope of a λy. In that case, the subterm

$\lambda y.M'$ must be safe, therefore $ord\, x_1 \geq ord\,(\lambda y.M') > ord\, y$. Because we substitute N_1 for x_1 we also know that $ord\, N_1 = ord\, x_1$. This leads to a contradiction. The same argumentation works for the other arguments N_i. We can therefore exclude variable capture.



□

In [5], Blum also proposed an alternative to Lemma 7.24 that does not rely on Convention 7.23, which we recall next.

Lemma 7.25. *(No-variable-capture lemma 2 [5, Lemma 3.17]) Let* $\Gamma, x : B \vdash_s M : A$, $\Gamma \vdash_s N_1 : B_1, \dots, \Gamma \vdash_s N_n : B_n$, *with* $|x| = n$, *be valid judgments of the safe lambda calculus à la Church. Then if further* $\Gamma \vdash_{Ch} M\{N/x\} : A$ *is a valid Church simply-typed term-in-context (not-necessarily safe) then:*

$$M[\![\overline{x}\backslash\overline{N}]\!] \equiv M[\overline{x}\backslash\overline{N}]$$

Counterexample 7.26. Consider the following safe λ-term:

$$\{y : (((o,o),o),o), z : ((o,o),o)\} \vdash_s \underline{(\lambda x^{((o,o),o)} y^{(o,o)} z^{o}.x)\ (\lambda q^{(o,o)}.y\ z)}$$

There is a variable capture when β_s reducing the underlined safe redex. Moreover, we have for $\Gamma = \{y : (((o,o),o),o), z : ((o,o),o)\}$:

- $\Gamma, x : ((o,o),o) \vdash_s \lambda y^{(o,o)} z^{o}.x$
- $\Gamma \vdash_s \lambda q^{(o,o)}.y\ z$
- $\Gamma \vdash_{Ch} \lambda y^{(o,o)} z^{o}.\lambda q^{(o,o)}.y\ z$
  This is a valid judgment in the simply-typed lambda calculus à la Church.

This is, therefore, a counterexample to Lemma 7.25.

Counterexample 7.26 shows that the alternative proposed in Lemma 7.25 does not work. This was also confirmed by Blum [6]. It leaves open whether or not there are alternatives that do work. However, we assume and rely on the safe variable typing convention in the following Lemma 7.27, where we show that the safe λ-calculus avoids α by reasoning with α-paths.

Lemma 7.27. *In the safe λ-calculus no variable capture can occur, provided that the safe variable typing convention is adopted.*

Proof. Suppose we have an α-path in a safe λ-term. Then this path would start at a variable y occurring free in the argument N of some application, which is connected via legal path to an abstraction λx binding a variable x in the scope of a λy, as illustrated below.

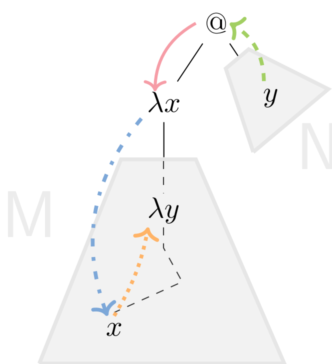


Figure 7.4: Simultaneous contraction can avoid the need for α-conversion.

In such case, by definition of safe terms, we know that $\lambda x.M$ and N are both safe. Moreover, we know that $ord\, y \geq N$ and $ord\, N = ord\, x$. We can therefore have the following two cases: (i) $ord\, y > ord\, x$ or (ii) $ord\, y = ord\, x$.

In any case, as the subterm $\lambda y.M'$ would be unsafe in isolation, we conclude that the λy must occur combined with the λx. By definition of safe β-reduction, we know that combined abstractions of same order are contracted simultaneously. Therefore we cannot have a variable capture. □

As we have seen, the safety restriction allows avoiding a potential variable capture by reasoning with types. Types are preserved under β-reduction, which is the key property enabling this reasoning.

However, as we can see in Example 7.18, we may not be allowed to evaluate safe terms to a normal form. Consider (again) the following terms: $M = \lambda y^o.(\lambda x^o y^o.x)\, y$ and $N = \lambda z^o.(\lambda x^o y^o.x)\, z$. Both M and N cannot be further reduced in the safe λ-calculus because contracting the redex they contain would break the safety of the whole term.

We expect safe terms to evaluate to normal form, so we need to forbid such constructs. Miranda did this in the first version of the safe λ-calculus he formulated jointly with Ong and Aehlig and presented in [15]. In his calculus, he assumes homogeneous types, as Knapik did for higher-order grammars [20].

Definition 7.28. A type $(\tau_1, \tau_2, \ldots, \tau_n, o)$ with $n \geq 0$ is said to be *homogeneous* if each τ_i is homogeneous and $ord\, \tau_1 \geq ord\, \tau_2 \geq \cdots \geq ord\, \tau_n$ [20][3].

[3] Knapik used a different terminology and called what we presented as order of a type, the *level* of a type

Example 7.29. $(o \to o) \to o$ and $o \to o$ are homogeneous, $o \to (o \to o)$ is not homogeneous.

Miranda used for homogeneous types $A = (A_{11}, \ldots, A_{1l_1}, \ldots, A_{r1}, \ldots, A_{rl_r}, o)$ the following abbreviation:

$$A = (\overline{A_1} \mid \ldots \mid \overline{A_r} \mid o)$$

to mean that all types in each partition $\overline{A_i} = A_{i1}, \ldots, A_{il_1}$ have the same level and $\forall i, j.i < j \iff ord\, A_{ia} > ord\, A_{ib}$.

We adopt the same notation for the order restriction on the set of types variables as done in [15]: Let Γ be a set of typed variables, then Γ_P denotes the restriction of Γ to those elements satisfying predicate P. We write $\Gamma_{\geq k}$, for the restriction of Γ to variables of order k and above. $\Gamma_{>k}$ is the restriction of Γ to the variable with orders strictly greater than k and $\Gamma_{=k}$ is the restriction of Γ to only the variables of order k.

Example 7.30. Let $\Gamma = \{A : ((o, o), o, o), B : (o, o, o), x : o, y : o\}$, then $\Gamma_{\geq 1} = \{A, B\}$, $\Gamma_{>1} = \{A\}$ and $\Gamma_{=1} = \{B\}$

Figure 7.5 lists the rules of the first version of the safe λ-calculus (Δ is a set of constants) due to Miranda, Ong and Aehlig [15].

$$(var)\ \frac{}{\{x : A\} \vdash x : A} \qquad (const)\ \frac{}{\vdash f : A}\ f \in \Delta \qquad (wk)\ \frac{\Gamma' \vdash M : A}{\Gamma \vdash M : A}\ \Gamma' \subset \Gamma$$

$$(app)\ \frac{\Gamma \vdash M : (B_{11}, \ldots, B_{1n} \mid \overline{B_2} \mid \ldots \mid \overline{B_n} \mid o) \quad \Gamma \vdash N_1 : B_{11} \quad \ldots \quad \Gamma \vdash N_n : B_{1n}}{\Gamma \vdash M\, N_1 \ldots N_n : (\overline{B_2} \mid \ldots \mid \overline{B_n} \mid o)}$$

$$(app{+})\ \frac{\Gamma \vdash M : (B_{11}, \ldots, B_{1j}, \overline{B} \mid \overline{B_2} \mid \ldots \mid \overline{B_n} \mid o) \quad \Gamma_{\geq m} \vdash N_1 : B_{11} \quad \ldots \quad \Gamma_{\geq m} \vdash N_j : B_{1j}}{\Gamma \vdash M\, N_1 \ldots N_j : (\overline{B} \mid \overline{B_2} \mid \ldots \mid \overline{B_n} \mid o)}$$
$$m = ord\,(\overline{B} \mid \overline{B_2} \mid \ldots \mid \overline{B_n} \mid o)$$

$$(abs)\ \frac{\Gamma \vdash M : A \quad \Gamma_{=k} = \{x_1 : B_{11}, \ldots, x_n : B_{1n}\}}{\Gamma_{>k} \vdash \lambda x_1 \ldots x_n.M : (B_{11}, \ldots, B_{1n} \mid A)}\quad k = ord\, A$$

Figure 7.5: The first version of the safe λ-calculus [15].

In this system, we can only build abstractions with variables of equal order (homogeneous). For example, the term $\lambda f^{(o,o)} x^o.f\, x$ is not derivable in Miranda's system. This term can only be derived if we split the combined abstraction as follows $\lambda f^{(o,o)}.\lambda x^o.f\, x$. Moreover, we can see that we can only derive applications if we either provide all arguments of a partition (*app*-rule) or in case we do not provide all arguments of a partition (we have a term s of order m and want to create an applicative term $s\, t_1 \ldots t_j$ of order m), then the free variables in the applied arguments must all have at least the order of the resulting term. With these restrictions, statements such as $\{y : o\} \vdash (\lambda x^o y^o.x)\, y$ are not derivable and, in contrast to the safe λ-calculus by Blum and Ong, safe redexes can

always be contracted. This is the crucial observation for solving the problem of terms getting stuck.

In [5, Lemma 3.26 (iii)] it was wrongly assumed that $\rightarrow_{\beta_s}$ has the unique normal form property. After reporting this problem to Blum [7], he immediately worked on it, and we independently came up with the same solution. As solution, we propose the system shown in Figure 7.6 (Δ is a set of constants).

$$(var)\ \frac{}{\{x : A\} \vdash x : A} \qquad (const)\ \frac{}{\vdash f : A}\ f : A \in \Delta \qquad (wk)\ \frac{\Gamma' \vdash M : A}{\Gamma \vdash M : A}\ \Gamma' \subset \Gamma$$

$$(app)\ \frac{\Gamma \vdash M : (A_1, \ldots, A_n, B) \quad \Gamma_{\geq m} \vdash N_1 : A_1 \quad \ldots \quad \Gamma_{\geq m} \vdash N_j : B_j}{\Gamma \vdash M\, N_1\, \ldots\, N_j : B}\quad m = ord\, B$$

$$(abs)\ \frac{\Gamma_{\geq m} \cup \{x_1 : A_1, \ldots, x_n : A_n\} \vdash M : B}{\Gamma \vdash_s \lambda x_1 \ldots x_n.M : (A_1, \ldots, A_n, B)}\quad m = ord\, (A_1, \ldots, A_n, B)$$

Figure 7.6: A strongly normalizing safe λ-calculus.

It turned out that these rules exactly correspond to the rules of the safe λ-calculus published in [8] and to the typing rules for *long-safe* terms (without constants) listed in [5, Table 3.2]. This system can be seen as a mix of the two systems by Blum and Miranda (Figure 7.1 and 7.5). It does not make assumptions on types and forbids the construction of safe redexes that cannot be contracted. The system is, therefore, less restrictive than Miranda's system (types not homogeneous) but more restrictive than Blum's one (*app*-rule).

Example 7.31. The simply-typed term $(\lambda f^{(o,o,o)} y^o.f\, y)\, (\lambda x^o y^o.x)$ is derivable in the safe λ-calculus by Blum, but not in our system (and in Miranda's) because of the application $f\, y$. As shown below, we could not evaluate this term to normal form without breaking the safety and also would need to apply α-conversion.

$$\begin{aligned} & (\lambda f^{(o,o,o)} y^o.f\, y)\, (\lambda x^o y^o.x) \\ \underset{\beta}{\rightarrow}\ & \lambda y^o.(\lambda x^o y^o.x)\, y \\ \underset{\alpha}{\rightarrow}\ & \lambda y^o.(\lambda x^o z^o.x)\, y \\ \underset{\beta}{\rightarrow}\ & \lambda y^o.(\lambda z^o.y) \end{aligned}$$

Example 7.32. The simply-typed term $(\lambda x^o y^{(o,o)}.y)$ is derivable in our system (and in Blum's), but not in Miranda's system because it is not homogeneously typed.

Lemma 7.33. *Given a simply typed lambda term M of type B. If in each subterm M' every free variable x has order at least the order M', then M is safe according to the inference rules of Figure 7.6.*

Proof. By well-founded induction on the structure of M. In the following proof $P(M)$ is used to denote that in each safe subterm M' of M every free variable x has order at least order M'.

1. $M = x$. Trivial.

2. $M = \lambda x_1^{A_1} \dots x_n^{A_n}.N : (A_1 \to \dots \to A_n \to B)$.

$$(abs)\ \frac{\Gamma_{\geq m} \cup \{x_1 : A_1, \dots, x_n : A_n\} \vdash M : B}{\Gamma \vdash_s \lambda x_1 \dots x_n.M : (A_1, \dots, A_n, B)} \quad m = ord\,(A_1, \dots, A_n, B)$$

- $N = y$. To prove the statement for M we assume $P(M)$. Trivially we have $P(y)$ so by the induction hypothesis we have that y is safe.
  - $y \in \{x_1, \dots, x_n\}$. Then $\Gamma = \emptyset$ and by applying the *wk* rule followed by the *abs*-rule we can conclude that M is safe.

$$\cfrac{\cfrac{\cfrac{}{y : A_k \vdash_s y : A_k}\ var}{x_1 : A_1, \dots, y : A_k, \dots, x_n : A_n \vdash_s y : A_k}\ wk}{\lambda x_1^{A_1} \dots y^{A_k} \dots x_n^{A_n}.y : A_1 \to \dots \to A_n \to A_k}\ abs$$

  - $y \notin \{x_1, \dots, x_n\}$. Since we assumed that $P(M)$, we know that $ord\,M \leq ord\,y$ and we can apply the *abs*-rule also in this case.

$$\cfrac{\cfrac{\cfrac{}{y : B \vdash_s y : B}\ var}{x_1 : A_1, \dots, x_n : A_n, y : B \vdash_s y : B}\ wk}{y : B \vdash_s \lambda x_1^{A_1} \dots x_n^{A_n}.y : A_1 \to \dots \to A_n \to B}\ abs$$

- $N = \lambda y_1^{B_1} \dots y_n^{B_n}.N'$. We assume $P(M)$. Since N is a subterm of M, we also have $P(N)$. By induction hypothesis we have that N is safe. We can apply the *abs*-rule and conclude that M is safe.

$$\cfrac{\cfrac{\dots}{x_1 : A_1, \dots, x_n : A_n, \Gamma \vdash_s \lambda y_1^{B_1} \dots y_n^{B_n}.N' : B_1 \to \dots \to B_n \to C}\ \dots}{\Gamma \vdash_s \lambda x_1^{A_1} \dots x_n^{A_n}.\lambda y_1^{B_1} \dots y_n^{B_n}.N' : A_1 \to \dots \to A_n \to (B_1 \to \dots \to B_n \to C)}\ abs$$

- $N = N_0\, N_1\, \dots\, N_m$. We assume $P(N)$. By induction hypothesis each N_i for $i \in \{1, \dots, m\}$ is safe.

$$\Delta\ \cfrac{\Gamma' \vdash_s N_0 : B_1 \to \dots \to B_m \to C}{x_1 : A_1, \dots, x_n : A_n, \Gamma \vdash_s N_0 : B_1 \to \dots \to B_m \to C}\ wk$$

$$\cfrac{\cfrac{\Delta \quad \cfrac{\Gamma'' \vdash_s N_1 : B_1 \quad \dots}{x_1 : A_1, \dots, x_n : A_n, \Gamma \vdash_s N_1 : B_1 \quad \dots}\ wk \quad \cfrac{\Gamma''' \vdash_s N_m : B_m}{x_1 : A_1, \dots, x_n : A_n, \Gamma \vdash_s N_m : B_m}\ wk}{x_1 : A_1, \dots, x_n : A_n, \Gamma \vdash_s N_0\, N_1\, \dots\, N_m : C}\ app}{\Gamma \vdash_s \lambda x_1^{A_1} \dots x_n^{A_n}.N_0\, N_1\, \dots\, N_m : A_1 \to \dots \to A_n \to C}\ abs$$

By assumption we have that $ord\,(A_1 \to \dots \to A_n \to C) \leq ord\,\Gamma$ and we are allowed to apply the *abs*-rule

3. $M = N_0\, N_1\, \dots\, N_m : B$. We assume that we have $P(M)$. Therefore we also have $P(N_i)$ for each N_i and by induction hypothesis each N_i is safe. By assumption we have that $ord\, B \leq ord\, \Gamma$ and we are allowed to apply the *app*-rule

$$(app)\ \frac{\Gamma \vdash M : (A_1, \dots, A_n, B) \quad \Gamma_{\geq m} \vdash N_1 : A_1 \quad \dots \quad \Gamma_{\geq m} \vdash N_j : B_j}{\Gamma \vdash M\, N_1\, \dots\, N_j : B} \quad m = ord\, B$$

□

Lemma 7.33 from above gives a procedure for checking whether a term is safe or not. If variables occurring free have order at least the order of the subterms they occur free in, then the whole term is safe. To check it, we label each node at (any) position p with the order of the subterm $M|_p$. We then can verify whether a term is safe by checking whether the order of the variables is the minimum value on their path towards the root. For example, the term depicted in Figure 7.7 is unsafe because the subterm $\lambda f.f\ (f\ c)$ is unsafe.

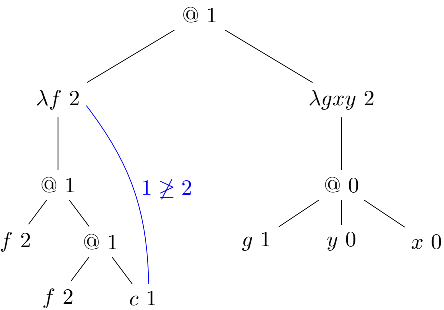


Figure 7.7: Safety check.

8 The simply typed λ-calculus

The simply typed λ-calculus ($\Lambda^{\rightarrow}$) is strongly normalizing. The need for α-conversion can therefore always be predicted, however sometimes it may be unavoidable.

Definition 8.1. The set of *simple types* over the atomic type o is generated from following grammar:

$$T ::= o \mid T \rightarrow T.$$

By convention, $\rightarrow$ associates to the right and $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow o$ is abbreviated as $(A_1, \ldots, A_n, o)$.

Definition 8.2. A typing *context* (or typing *environment*) Γ is a set of typing assumptions of the form $x : A$ where A is a (simple) type.

Definition 8.3. This is the system of rules for the Simply Typed λ-Calculus:

$$(var)\,\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad (app)\,\frac{\Gamma \vdash N_1 : A \rightarrow B \quad \Gamma \vdash N_2 : A}{\Gamma \vdash N_1\, N_2 : B} \qquad (abs)\,\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \rightarrow B}$$

Terms derivable in the system of the simply typed λ-calculus are said to be *well-typed*.

Example 8.4. Let $A = (o, o, o)$. The term $(\lambda f^{A \rightarrow A} c^A . f\,(f\,c))\,(\lambda z^A x^o y^o . z\,y\,x)$ is well-typed, but still requires α-conversion is needed when reducing it to normal form. All binders are already named distinct, α-conversion is therefore unavoidable.

$$\begin{array}{ll}
 & \underline{(\lambda f c . f\,(f\,c))\,(\lambda z x y . z\,y\,x)} \\
\underset{\beta}{\rightarrow} & (\lambda z x y . z\,y\,x)\,\underline{((\lambda z x y . z\,y\,x)\,c)} \\
\underset{\beta}{\rightarrow} & \underline{(\lambda z x y . z\,y\,x)\,(\lambda x y . c\,y\,x)} \\
\underset{\beta}{\rightarrow} & (\lambda x y . \underline{(\lambda x y . c\,y\,x)\,y}\,x) \\
\underset{\alpha}{\rightarrow} & \lambda x y . \underline{(\lambda x y' . c\,y'\,x)\,y}\,x \\
\underset{\beta}{\rightarrow} & \lambda x y . \underline{(\lambda y' . c\,y'\,y)\,x} \\
\underset{\beta}{\rightarrow} & \lambda x y . c\,x\,y
\end{array}$$

We can use the α-paths to predict the need for α-conversion in the simply typed λ-calculus. Since well-typed terms are strongly normalizing, also the set of α-paths in a well-typed term is finite. In the term in Example 8.4 we have the following paths:

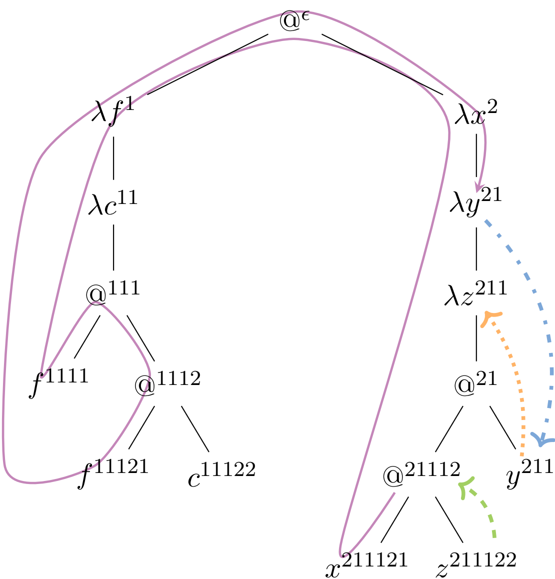


Figure 8.1: An unremovable α-path in $(\lambda fc.f\,(f\,c))\,(\lambda xyz.(x\,z)\,y)$.

Example 8.5 shows that simultaneous reduction is not enough to avoid the need for α-renaming in the simply typed λ-calculus. Vincent found this term by relying on Example 8.4 and applying the trick of temporarily blocking a redex (by inserting the identity function) [33].

Example 8.5. Applying simultaneous β-reduction is not enough to allow α-avoidance in the simply-typed λ-calculus.

$$
\begin{aligned}
& \underline{(\lambda fc.f\,(f\,c))\,(\lambda gx.(\lambda i.i)\,(\lambda y.g\,y\,x))} \\
\xrightarrow[\beta_{sim}]{} \quad & \lambda c.\underline{(\lambda gx.(\lambda i.i)\,(\lambda y.g\,y\,x))\,((\lambda gx.(\lambda i.i)\,(\lambda y.g\,y\,x))\,c)} \\
\xrightarrow[\beta_{sim}]{} \quad & \lambda c.(\lambda x.\underline{(\lambda i.i)\,(\lambda y.((\lambda gx.(\lambda i.i)\,(\lambda y.g\,yx))\,c)\,y\,x)}) \\
\xrightarrow[\beta_{sim}]{} \quad & \lambda c.(\lambda x.(\lambda y.(\underline{(\lambda gx.(\lambda i.i)\,(\lambda y.g\,yx))\,c)\,y\,x}) \\
\not\xrightarrow[\beta_{sim}]{} \quad & \lambda c.(\lambda x.(\lambda y.((\lambda i.i)\,(\lambda y.y\,y\,x))\,c)) \\
\xrightarrow[\alpha]{} \quad & \lambda c.(\lambda x.(\lambda y.(\underline{(\lambda gx.(\lambda i.i)\,(\lambda y'.g\,y'x))\,c)\,y\,x})) \\
\xrightarrow[\beta_{sim}]{} \quad & \lambda c.(\lambda x.(\lambda y.(\underline{(\lambda i.i)\,(\lambda y'.y\,y'\,x)})\,c)) \\
\xrightarrow[\beta_{sim}]{} \quad & \lambda c.(\lambda x.(\lambda y.\underline{(\lambda y'.y\,y'\,x)\,c})) \\
\xrightarrow[\beta_{sim}]{} \quad & \lambda c.(\lambda x.(\lambda y.y\,c\,x))
\end{aligned}
$$

9 The untyped λ-calculus

In the untyped λ-calculus, also called the full λ-calculus, there are no restrictions. We have seen that already that in the simply typed λ-calculus, α-renaming may be unavoidable. The simply typed λ-calculus, however, is strongly normalizing. Untyped terms may not have a normal form, the set of legal paths in an untyped λ-term can potentially be infinitely large. For such terms, predicting the need for α-conversion via α-paths is impossible since we rely on the legal paths (which is an infinite set for such terms). In this chapter, we show that in the untyped λ-calculus, the question about α-avoidance is undecidable for the leftmost–outermost reduction strategy by giving a reduction from Post's Correspondence Problem [26].

Definition 9.1. The set Λ of λ-terms is inductively defined as follows:

(var) $x \in \Lambda$, for all variables x

(app) $M, N \in \Lambda \Longrightarrow M\,N \in \Lambda$

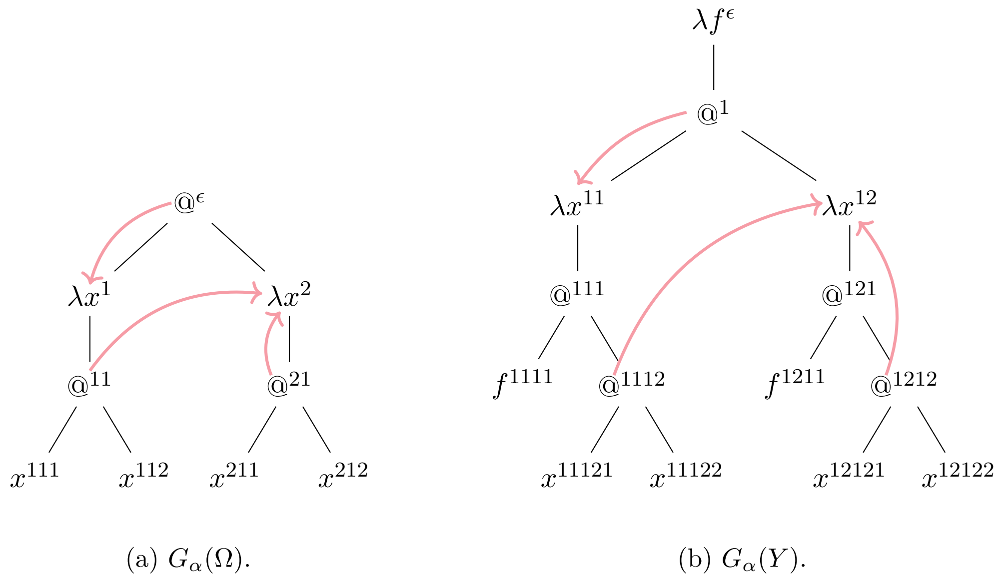
(abs) $M \in \Lambda \Longrightarrow \lambda x.M \in \Lambda$

Example 9.2 gives an example for an untyped term where the need for α-conversion cannot be avoided, no matter how variables are named initially. This term is interesting because its reduction sequence combines all restrictions we analyzed so far. There is a duplication (forbidden in the linear λ-calculus), a redex creation (restricted in the underlined λ-calculus), and a redex contraction in the scope of a λ-node in the last β-step (not allowed in the lazy λ-calculus). Moreover, this term is not typable and therefore also not safe.

Example 9.2. An unavoidable α-conversion.

$$
\begin{array}{lll}
 & (\lambda x.x\,x)\,(\lambda y\lambda z.y\,z) & \\
\underset{\beta}{\rightarrow} & (\lambda y\lambda z.yz)\,(\lambda y\lambda z.y\,z) & \text{duplication} \\
\underset{\beta}{\rightarrow} & \lambda z.(\lambda y\lambda z.y\,z)\,z & \text{redex creation} \\
\underset{\alpha}{\rightarrow} & \lambda z.(\lambda y.\lambda z'.y\,z')\,z & \\
\underset{\beta}{\rightarrow} & \lambda z\lambda z'.z\,z' & \text{contraction under lambda}
\end{array}
$$

Even if the need for α-conversion in Example 9.2 was unavoidable, it still was predictable. However, in the untyped λ-calculus terms may not be strongly normalizing (for example $(\lambda x.y)\ \Omega$). In Remark 9.3 we explain the idea of how we could over-approximate the need for α-conversion for such terms.



(a) $G_\alpha(\Omega)$.

(b) $G_\alpha(Y)$.

Figure 9.1: Overapproximation for non-normalizing -terms.

Remark 9.3. For λ-terms that are not strongly-normalizing we have an infinite set of legal paths, all characterizing a different virtual redex. For such terms, a prediction about the need for α-conversion via the α-paths is therefore impossible. Already for $(\lambda x.x\,x)\,(\lambda x.x\,x)$, the standard example of a non-normalizing λ-term ($\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \dots$), we will run into that problem. However, for Ω we know that we cannot have any α-path, because there is no c-edge in $G_\alpha(\Omega)$ (see Figure 9.1a).

For the Y-combinator $Y = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$, also non-normalizing, this question is more difficult to answer (see Figure 9.1b). But, even if λ-terms are not normalizing, we could try to overapproximate the need for α-conversion. An interesting question, therefore, is whether such an overapproximation is possible in general. We propose to compute only finitely many legal paths (would be an infinite set) and only consider those legal paths, that build new connections between two nodes. For example, suppose we already have a legal path from an @-node at position p and a λ-node at position q, and we generate another legal path between these positions. In that case, we discard it and do not add it to our set of legal paths. This way, we know that the procedure terminates (we have only finitely many nodes in an α-graph), and can search for α-paths. With this approach we only generate three legal paths for both Ω and the Y-combinator as depicted in Figure 9.1 (we only illustrated the connection between the starting and the ending node, not the whole legal path). We would conclude that in both term there is no need for dynamic α-conversion as there are no α-paths. Since we do not prove that this indeed corresponds to an overapproximation, this question remains open and is part of future work.

When restricting $\to_\beta$ to the leftmost–outermost reduction strategy, α-avoidance is, in general, undecidable for the untyped λ-calculus. This can be shown via a reduction from

Post's correspondence problem (PCP) that is already known to be *recursively unsolvable* i.e., undecidable [26].

The correspondence decision problem asks whether for an arbitrary finite set of string pairs $(s_1, s'_1), (s_2, s'_2), \ldots, (s_n, s'_n)$ on the characters a and b, there is a solution to the equation

$$s_{i_1} s_{i_2} \ldots s_{i_k} = s'_{i_1} s'_{i_2} \ldots s'_{i_k} \qquad k \geq 1, i_j \in \{1, 2, \ldots, n\} \tag{9.1}$$

Example 9.4. Let $n = 2$, $(s_1, s'_1) = (a, abb)$ and $(s_2, s'_2) = (bb, b)$. Then $s_1 s_2 s_2 = abbbb = s'_1 s'_2 s'_2$ is a solution to the correspondence decision problem.

Example 9.4 shows an easy instance of PCP and its solution, for $n = 2$ PCP is decidable [16]. However, as already mentioned above, it is undecidable in general. Next, we will show how we can encode an algorithm for the *PCP*-problem in the λ-calculus and that α-conversion can indeed be avoided in the evaluation of *PCP PAIRS*, with *PAIRS* being an arbitrary list of pairs of strings if we adopt a weak β-reduction.

Boolean Logic We start by showing how boolean logic can be encoded in the λ-caclulus.

$$\begin{aligned}
TRUE =& \lambda xy.x \\
FALSE =& \lambda xy.y \\
ITE =& \lambda cab.c\,a\,b \\
AND =& \lambda ab.a\,b\,FALSE \\
OR =& \lambda ab.a\;TRUE\,b \\
CT =& \lambda x.TRUE \\
CF =& \lambda x.FALSE
\end{aligned}$$

Example 9.5. $AND\;TRUE\;FALSE \twoheadrightarrow_\beta TRUE\;FALSE\;FALSE \twoheadrightarrow_\beta FALSE$

Pairs Next we define the encoding of pairs.

$$\begin{aligned}
PAIR =& \lambda abc.c\,a\,b \\
FST =& \lambda p.p\;TRUE \\
SND =& \lambda p.p\,FALSE
\end{aligned}$$

Example 9.6. $FST\,(PAIR\;TRUE\,FALSE) \rightarrow_\beta (PAIR\;TRUE\,FALSE)\;TRUE \twoheadrightarrow_\beta TRUE\;TRUE\,FALSE \twoheadrightarrow_\beta TRUE$

Recursion The Y-combinator allows to define recursive functions in the λ-calculus:

$$Y = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

Example 9.7. $Y\,f \rightarrow_\beta (\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)) \rightarrow_\beta f\,((\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))) \equiv f\,(Y\,f)$

Strings We define strings over the alphabet $\Sigma = \{a, b\}$.

$$\begin{aligned}EMPTY =& \lambda abx.x\\ ISEMPTY =& \lambda s.s\; CF\; CF\; TRUE\\ CONCAT =& \lambda yzabx.y\, a\, b\, (z\, a\, b\, x)\end{aligned}$$

Example 9.8. The string aa can be defined as $\lambda abx.a\,(a\,x)$

Example 9.9. The string bb can be defined as $\lambda abx.b\,(b\,x)$

Example 9.10. $ISEMPTY\; EMPTY \rightarrow_\beta EMPTY\; CF\; CF\; TRUE \twoheadrightarrow_\beta TRUE$

Example 9.11. $ISEMPTY\; \lambda abx.a\,(b\,(b\,x)) \twoheadrightarrow_\beta CF\,(CF\,(CF\; TRUE)) \twoheadrightarrow_\beta FALSE$

Next, we define functions that prepend a character to a string ($PREP_A$, $PREP_B$), one to check whether the first character is an a (HD_A) or a b (HD_B) and one to check whether the first character of two strings is equal (HD_EQ).

$$\begin{aligned}PREP_A =& \lambda sabx.a\,(s\,a\,b\,x)\\ PREP_B =& \lambda sabx.b\,(s\,a\,b\,x)\\ HD_A =& \lambda s.s\; CT\; CF\; FALSE\\ HD_B =& \lambda s.s\; CF\; CT\; FALSE\\ HD_EQ =& \lambda xy.OR\,(AND\,(HD_A\,x)\,(HD_A\,y))\,(AND\,(HD_B\,x)\,(HD_B\,y))\end{aligned}$$

Defining the tail of a string in the λ-calculus is non-trivial. It can be implemented by using pairs by implementing the idea visualized in Figure 9.2.

str	fst	snd
a	ϵ	ϵ
ab	a	ϵ
abb	ba	a
$abba$	bba	ba
	$abba$	bba
		bba

Figure 9.2: How to compute the tail of string using pairs.

Let $\lambda abx.s$ be our string where $s = a(s) \mid b(s) \mid x$. If we, as illustrated above in Figure 9.2, replace in s the characters a, b by a function that copies the first element of a pair to the second element and prepends the corresponding character to the first

element of the pair ($NEXT_A, NEXT_B$) and x by a pair of empty strings, then we can just return the second element of the final pair, and we have the tail (TL_STR).

$$
\begin{aligned}
NEXT_A &= \lambda x.PAIR\,(PREP_A\,(FIRST\,x))\,(FIRST\,x)\\
NEXT_B &= \lambda x.PAIR\,(PREP_B\,(FIRST\,x))\,(FIRST\,x)\\
TL_STR &= \lambda s.SECOND\,(s\,NEXT_A\,NEXT_B\,(PAIR\,EMPTY\,EMPTY))
\end{aligned}
$$

EQ checks whether two strings are equal by checking whether the first characters are equal and, if so, recursively whether the tails are equal. $PREFIX$ checks whether a string x is a prefix of another string y.

$$
\begin{aligned}
EQ &= Y\,(\lambda fxy.ITE\,(OR\,(IS_EMPTY\,x)\,(IS_EMPTY\,y))\\
&\quad (AND\,(IS_EMPTY\,x)\,(IS_EMPTY\,y))\\
&\quad (AND\,(HD_EQ\,x\,y)\,(f\,(TL\,x)\,(TL\,y))))\\
PREFIX &= Y\,(\lambda fxy.ITE\,(IS_EMPTY\,x)\,TRUE\\
&\quad (AND\,(HD_EQ\,x\,y)\,(f\,(TL\,x)\,(TL\,y))))
\end{aligned}
$$

Lists The last data structure we need to encode is a list. We only need basic functions on lists: HD_L extracts the head element of a list, TL_L the tail of a list, and $APPEND$ appends a list y to another list x. The TL_L function follows the same idea illustrated in Figure 9.2. To make HD_L typable, we return a pair of empty strings whenever we try to extract the head element of an empty list[1].

$$
\begin{aligned}
NIL &= \lambda cn.n\\
CONS &= \lambda ht.\lambda cn.c\,h\,(t\,c\,n)\\
IS_NIL &= \lambda l.l\,(\lambda ht.FALSE)\,TRUE\\
HD_L &= \lambda l.l\,(\lambda ht.h)\,(PAIR\,EMPTY\,EMPTY)\\
NEXT_L &= \lambda xp.PAIR\,(CONS\,x\,(FIRST\,p))\,(FIRST\,p)\\
TL_L &= \lambda l.SECOND\,(l\,NEXT_L\,(PAIR\,NIL\,NIL))\\
APPEND &= \lambda xy.\lambda cn.x\,c\,(y\,c\,n)
\end{aligned}
$$

PCP With booleans, pairs, lists, and strings we can finally implement the PCP algorithm. The $SIMP$ function simplifies a pair of strings by cutting off the first n characters from both strings, where n is the length of the shorter string (e.g. $simp(abba, bab) = (a, \epsilon)$). P_VALID is true, if in a pair of strings one string is a prefix of the other (e.g. $pvalid(abba, bab) = false$). The $FIND_EQ$ function checks whether in a list of pairs of strings, we have a pair with two equal strings (in case we have a solution).

[1] In our program we will always check if a list is empty or not before calling the HD_L function.

$$
\begin{aligned}
SIMP =& \lambda p. Y\,(\lambda fxy.ITE\,(OR\,(IS_EMPTY\,x)\,(IS_EMPTY\,y))\,(PAIR\,x\,y)\\
&(f\,(TL\,x)\,(TL\,y)))\,(FIRST\,p)\,(SECOND\,p)\\
P_VALID =& \lambda p.OR\,(PREFIX\,(FIRST\,p)\,(SECOND\,p))\\
&(PREFIX\,(SECOND\,p)\,(FIRST\,p))\\
FIND_EQ =& Y\,(\lambda fx.ITE\,(IS_NIL\,x)\,FALSE\\
&(OR\,(EQ\,(FIRST\,(HD_L\,x))\,(SECOND\,(HD_L\,x)))\,(f\,(TL_L\,x))))
\end{aligned}
$$

The *CMB* function, given two pairs (s_1, t_1) and (s_2, t_2), returns a pair with concatenated elements $(s_1 \cdot s_2, t_1 \cdot t_2)$. The *MAP_CMB* function maps the *CMB* with a pair x over each element of a list y. It also simplifies the resulting list of pairs and filters out invalid ones. The *CROSS_CMB* function combines each element in a list x with each element in a list y.

$$
\begin{aligned}
CMB =& \lambda pq.PAIR\,(CONC\,(FIRST\,p)\,(FIRST\,q))\\
&(CONC\,(SECOND\,p)\,(SECOND\,q))\\
MAP_CMB =& Y\,(\lambda fxy.ITE\,(IS_NIL\,y)\,NIL\,(ITE\,(PVALID\,(CMB\,x\,(HD_L\,y)))\\
&(CONS\,(SIMP\,(CMB\,x\,(HD_L\,y)))\,(f\,x\,(TL_L\,y)))\,(f\,x\,(TL_L\,y))))\\
CROSS_CMB =& Y\,(\lambda fxy.ITE\,(IS_NIL\,x)\,NIL\\
&(APPEND\,(MAP_CMB\,(HD_L\,x)\,y)\,(f\,(TL_L\,x)\,y)))
\end{aligned}
$$

Finally, we can define the *PCP* function that takes a list of pairs as input and recursively combines them. *PCP* is *semi-decidable*. It only stops if a solution is found (created at least one pair with equal strings) or if no new combination can be created (no solution).

$$
\begin{aligned}
PCP =& \lambda x. Y\,(\lambda fxy.ITE\,(IS_NIL\,x)\,FALSE\\
&(ITE\,(FIND_EQ\,x)\,TRUE\,(f\,(CROSS_CMB\,x\,y)\,y)))\,x\,x
\end{aligned}
$$

It remains to show how we can encode a specific problem. We chose the following PCP problem [22, Problem 12][2]:

$$
\begin{aligned}
T_1 &= \lambda abx.b\,(a\,(a\,x)) & B_1 &= \lambda abx.b\,x\\
T_2 &= \lambda abx.a\,x & B_2 &= \lambda abx.b\,(a\,(a\,x))\\
T_3 &= \lambda abx.b\,x & B_3 &= \lambda abx.a\,x
\end{aligned}
$$

$$
P_1 = PAIR\,T_1\,B_1 \qquad P_2 = PAIR\,T_2\,B_2 \qquad P_3 = PAIR\,T_3\,B_3
$$

[2] we encode the 0 as a and the 1 as b

$PAIRS = CONS\, P_1\, (CONS\, P_2\, (CONS\, P_3\, NIL))$ is the list containing the above defined pairs of strings. Since this problem has a solution of length 75 [22, Problem 12], we know that at some point, $PCP\ PAIRS$ will stop and return true. However, as already mentioned, in general, we do not know whether the PCP algorithm will stop at all. In some instances, the algorithm will not terminate.

The encodings presented in this chapter were all implemented in Haskell. Everything is typable, so we know that PCP will evaluate to either $TRUE$ or $FALSE$ (and nothing else). The source code can be found in Appendix A.4. Since Haskell is lazy, it never reduces under λ-abstractions. Moreover, there are no free variables in $PCP\ PAIRS$. We have seen in Chapter 6, that α-conversion can be avoided for closed terms when we apply weak β-reduction. So we can conclude that for $PCP\ PAIRS$ the leftmost–outermost reduction strategy is α-free (for any $PAIRS$ instance).

Consider the following program:

$$(ITE\,(PCP\ PAIRS)\ AA\ BB)\,(\lambda xyz.(x\,z)\,y)$$

where AA is the encoding of the string aa and BB the encoding of the string bb, according to Example 9.8 and 9.9. If the problem has a solution, it will reduce to the λ-term $AA\,(\lambda xyz.(x\,z)\,y)$, which is α-equivalent to the simply-typed λ-term from Example 8.1 (with α unavoidable). Otherwise, it will reduce to the λ-term $BB\,(\lambda xyz.(x\,z)\,y)$ from which we get with one β-step to $\lambda bx.b\,(b\,x)$. Moreover, we know that if we always contract the top-level redex (it is also the leftmost, outermost one), the reduction sequence to these terms is α-free. In that case, if we further reduce these terms to normal form, then we need α-conversion if and only if the PCP problem has a solution. It depends on the result of $PCP\ PAIRS$ whether the described reduction strategy is α-free or not. Since we know that PCP is in general undecidable, we conclude that the question about α-avoidance is, in general, undecidable for the leftmost–outermost reduction strategy.

Remark 9.12. We have proven the undecidability of α-avoidance for the leftmost–outermost reduction strategy in the untyped λ-calculus. We restricted to this specific strategy because there are many reduction sequences from the program PCP program we encoded. However, we think that we could prove the general undecidability of α-avoidance. The approach we propose is to translate the PCP program from above into continuation-passing style [13]. This way, by definition of the program, we can make the reduction deterministic and therefore enforce the reduction sequence we have shown to be problematic. This would allow us to conclude the general undecidability of α-avoidance. This question, however, remains open and is part of future work.

10 Conclusion

In this thesis, we investigated what it means to avoid α or that α can be avoided. We presented a general characterization for the need for α-conversion via the so-called α-paths (Chapter 3). α-paths exploit the predictive power of the legal paths due to Asperti and Guerrini [2]. These legal paths characterize virtual redexes, i.e. all redexes occurring in some reduction sequence from a λ-term. By reasoning on the structure of the initial term, we estimated whether α-conversion might be needed when contracting these virtual redexes. The α-paths were instantiated to different λ-calculi: developments (Chapter 4), the affine λ-calculus (Chapter 5), the weak λ-calculus (Chapter 6), the safe λ-calculus (Chapter 7), the simply-typed λ-calculus, and finally for the untyped λ-calculus (Chapter 9). We have proven that if a λ-term does not contain any α-path, then any reduction sequence from it is α-free.

We have seen that forbidding redex creation, duplication, or the contraction of redexes in the scope of abstractions is enough to allow α-avoidance. In the safe λ-calculus, we have seen how α can be avoided by reasoning with types. In the calculus by Blum and Ong [9], however, we have seen that terms may get stuck. We proposed an alternative system that forbids the construction of "unsafe" applications and therefore is strongly normalizing. In the simply-typed λ-calculus, we have seen that α cannot always be avoided. However, since this calculus is strongly normalizing, the problem is still decidable. The untyped λ-calculus, however, is not strongly normalizing. For this unrestricted calculus, we showed that the question about α-avoidance is undecidable for the leftmost–outermost reduction strategy.

For all calculi where we can avoid α, we have seen that adopting the right naming conventions enables α-free computations. In the end, this means moving a dynamic problem to a static one. We motivated in the introduction (Chapter 1) why this shift can be relevant for different aspects like efficiency (naive substitution), traceability (relation input–output) and referential transparency (no supply of fresh variables needed).

There are still some open problems. It remains to show whether we can overapproximate the need for α-conversion (Remark 9.3) and whether we can prove the general undecidability of α-avoidance (Remark 9.12). We already know that we do not have any name collision in valid functional programs if we apply lazy β-reduction [24]. However, it would be interesting to know whether we can characterize this result via paths. This would require the characterization of virtual redexes via paths for reduction rules that differ from the ordinary β-reduction (Remark 6.6). Another open problem we described is whether we can derive a safe reduction sequence, in which no α is needed, for an arbitrary λ-term (Remark 3.39). These problems are left for future work.

Bibliography

[1] S. Alves and M. Florido. Weak linearization of the lambda calculus. *Theor. Comput. Sci.*, 342(1):79–103, Sept. 2005.

[2] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, USA, 1999.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.

[4] H. P. Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. H. Barendregt. Lambda calculi with types. 2000.

[5] W. Blum. *The Safe Lambda Calculus*. PhD thesis, Oxford University, UK, 2009.

[6] W. Blum. personal communication, May 2020.

[7] W. Blum. personal communication, August 2021.

[8] W. Blum and C. H. L. Ong. The Safe Lambda Calculus. In S. R. Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 39–53, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] W. Blum and C. H. L. Ong. The Safe Lambda Calculus. *Logical Methods in Computer Science*, Volume 5, Issue 1, Feb. 2009.

[10] D. Büring. *Binding Theory*. Cambridge Textbooks in Linguistics. Cambridge University Press, 2005.

[11] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[12] W. Damm. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20(2):95–207, 1982.

[13] O. Danvy and A. Filinski. Representing control: a study of the cps transformation, 1992.

[14] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[15] J. G. de Miranda. *Structures generated by higher-order grammars and the safety constraint*. PhD thesis, University of Oxford, UK, 2006.

[16] A. Ehrenfeucht, J. Karhumäki, and G. Rozenberg. The (generalized) post correspondence problem with lists consisting of two words is decidable. *Theoretical Computer Science*, 21(2):119–144, 1982.

[17] J. Endrullis, C. Grabmayer, J. W. Klop, and V. van Oostrom. On equal μ-terms. *Theoretical Computer Science*, 412(28):3175 – 3202, 2011. Festschrift in Honour of Jan Bergstra.

[18] R. Hendriks and V. van Oostrom. adbmal. *Lecture Notes in Computer Science*, 2741:136–150, 2003. Proceedings title: Conference on Automated Deduction.

[19] J. Hindley. Reductions of residuals are finite. *Transactions of The American Mathematical Society - TRANS AMER MATH SOC*, 240:345–361, 06 1978.

[20] T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 205–222, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[21] L. Lamport. Substitution: Syntactic versus semantic. 1998.

[22] R. J. Lorentz. Creating Difficult Instances of the Post Correspondence Problem. In T. Marsland and I. Frank, editors, *Computers and Games*, pages 214–228, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[23] G. Moser. Diskrete Mathematik, Ein Skriptum zur Vorlesung. University of Innsbruck, 2019.

[24] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, January 1987.

[25] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, 2002.

[26] E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264 – 268, 1946.

[27] D. E. Schroer. *The Church–Rosser Theorem.* PhD thesis, Cornell University, 1963.

[28] P. Selinger. Lecture notes on the lambda calculus. *CoRR*, abs/0804.3434, 2008.

[29] R. Statman. On the complexity of alpha conversion. *J. Symb. Log.*, 72:1197–1203, 12 2007.

[30] Terese. *Term rewriting systems.* Cambridge University Press, 2003.

[31] C. Urban, A. Pitts, and M. Gabbay. Nominal unification. 2004.

[32] V. van Oostrom. FD à la Church, June 1996.

[33] V. van Oostrom. personal communication, 2021.

A Appendix

A.1 Alpha avoidance in the μ-calculus

This section relies on the concepts and definitions introduced in [17]. The μ-calculus is the calculus of recursive types [4]. As in untyped λ-calculus, terms are formed by variables, applications and abstractions. In addition, also constants are allowed. There is only one single rewrite rule called the μ-rule.

Definition A.1. The set of μ-terms $Ter(\mu)$ is inductively defined as:

(var) $x, y, z, ... \in Ter(\mu)$

(con) $c, d, ... \in Ter(\mu)$

(app) $M, N \in Ter(\mu) \implies M\ N \in Ter(\mu)$

(abs) $M \in Ter(\mu)$ and x a variable $\implies \mu x.M \in Ter(\mu)$

where $c, d, ...$ are constants. μ is interpreted as type constructor [4].

Example A.2. $\mu x.x$, $\mu x.(\mu z.y)$, $\mu x.x\ c$ are valid μ-terms, $\mu c.(x\ c)$ is not a valid μ-term.

Definition A.3. (μ-rule) The only rewrite rule is the so called μ-rule.

$$\mu x.M \xrightarrow{\mu} M[\![x := \mu x.M]\!]$$

In Definition A.3 we see that the abstracted variable x is replaced in M by the whole term again. This is what happens in a recursion step in the ordinary λ-calculus reducing a fixed point combinator [3, Section 6.1]. The α-paths we would have in such a simulation, therefore, correspond to the self-capturing chains presented in this section.

Example A.4. $\mu x.x \xrightarrow{\mu} \mu x.x \xrightarrow{\mu} \mu x.x \xrightarrow{\mu} \dots$

Example A.5. $\mu x.(y\ \mu y.x) \xrightarrow{\alpha} \mu x.(y\ \mu z.x) \xrightarrow{\mu} y\ \mu z.\mu x.(y\ \mu z.x) \xrightarrow{\mu} \dots.$

In this Example A.5 the last case of α-converting substitution applies in the first reduction step, where the μy is renamed to μz to prevent the free variable y to be captured. The need for this α-renaming can be predicted by the self-capturing chains introduced by Vincent van Oostrom. These chains were also inspiration for the self-capturing chains presented in this thesis for the λ-calculus. Next we will prove that $M[\![x/N]\!] = M[x/N]$ if there is no self-capturing chain in M.

Definition A.6. (Binding link) For a given term M a binder μx at position p binds a variable y at position $p1$ if the occurrence of the variable y is free at position q in $M|_{p1}$ and $x = y$. These positions are connected by a binding link (blue) starting at p.

Definition A.7. (Converse-capturing link) For a given term M a binder μx at position p captures a variable y at position p if the occurrence of the variable y is free at position q in $M|_{p1}$ and $x \neq y$. These positions are connected by a converse-capturing link (red) starting at p.

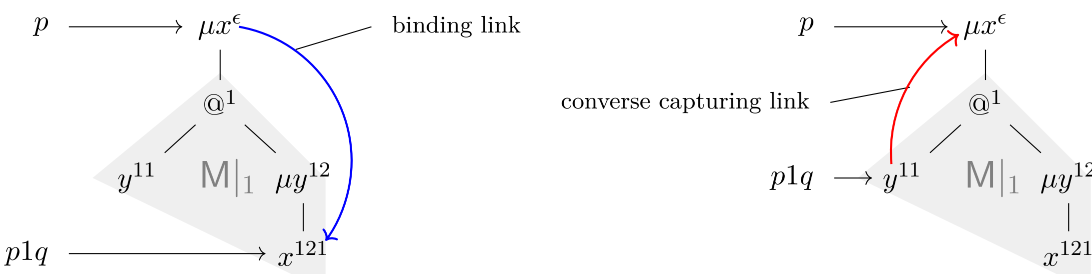
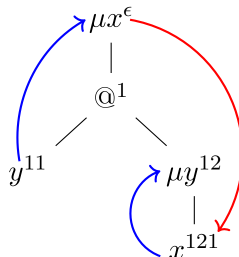


Figure A.1: Example of a binding link and a converse capturing link

Definition A.8. (Self-capturing chain) A chain is a series of alternating binding and converse capturing links. A chain is said to be *self-capturing* if it starts with x and ends with μx for some x.

Example A.9. The term $\mu x.(y\ \mu y.x)$ contains a self-capturing chain $y, \mu x, x, \mu y$ starting at y and ending at μy.



Lemma A.10. *If a term M contains no self-capturing chain then α-renaming can be avoided in a μ-reduction step.*

Proof. Let $\mu x.M$ contain no self-capturing chain (scc). If there is a chain from a free occurrence of x in M to a μy, then y cannot occur free in M, otherwise there would be a *scc* $y, \mu x, x, \ldots, \mu y$. Since y is not free in M the last case in the definition of α-converting substitution cannot apply and consequently $M[\![x/N]\!] = M[x/N]$. □

Having proved Lemma A.10 it remains to show that self-capture-freeness is preserved by μ-reduction.

Lemma A.11. *Self-capture-freeness is preserved by μ-reduction.*

Proof. This statement is proved by relating positions in the source term and the target term of a reduction step contracting a redex $\mu z.M$ at position o. This mapping is visualized by the colored areas in Figure A.2.

(context)	$p \triangleright p$	if o is not prefix of p
(body)	$o1p \triangleright op$	if $o1p$ not bound by o
(copy)	$op \triangleright oqp$	if $o1q$ bound by o

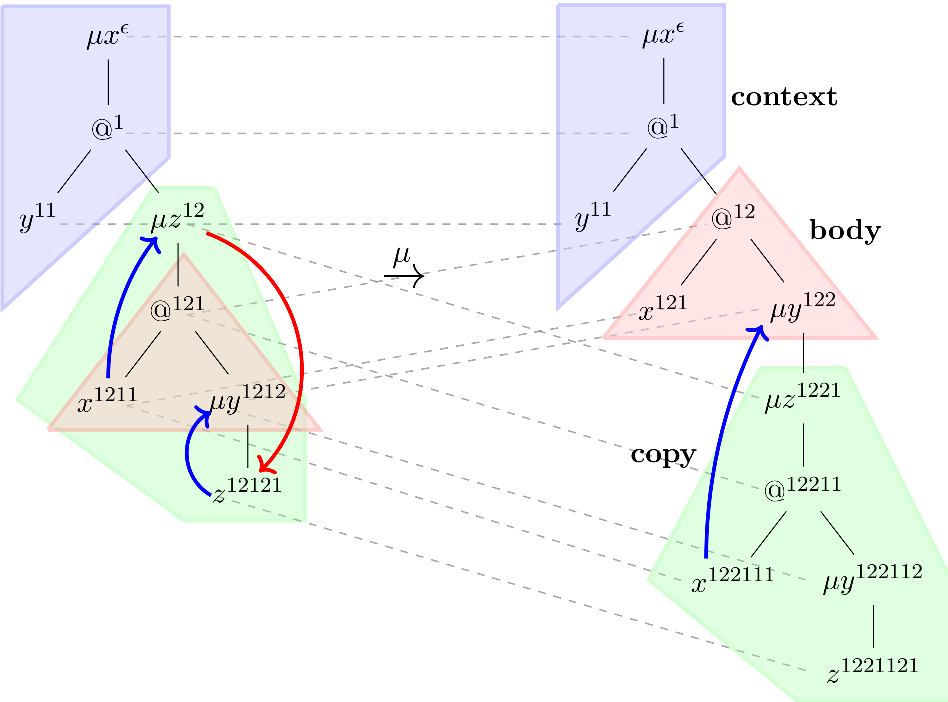


Figure A.2: Mapping back links to paths (Case 4).

The claim is that every chain in the target term maps back to a chain in the source term. Let p' denote the position of a binder μx and q' denote the position of a variable y in the target, where p' is prefix of q'. Four cases have to be distinguished:

1. both p' and q' in same component: the path between them is in the same component as well and the origin of this path is a path between their origins p, q in the source.

2. p' in the context and q' in the body: the origin of the path between them is a path between their origins p and $o1q$ in the source with side condition that $y \neq z$.

3. p' in the context and q' in a copy at o': q' has to be free in $o'1$ and the origin of the path is a path from p to oq.

4. p' in the body and q' in a copy at o': q' has to be free in $o'1$ which implies that p is connected to q via the first binder corresponding to the copy (this case is visualized in Figure A.2).

□

Lemma A.12. *For every μ-term M there exists a μ-term N such that $M \equiv_\alpha N$ and any μ-reduction from N is α-free*

Proof. By naming the binders on each chain in M distinctly and distinct from the free variables, we can conclude that the term is self-capture-free and therefore by Lemma A.10 and Lemma A.11 also α-free [17]. It would be enough to rename only the binders on self-capturing chains, but this would require more effort since the start end endpoints have to be determined first. On a chain of length 11, there can be maximal 6 binders, but there could be 22 self-capturing chains (1 of length 11, 2 of length 9, 5 of length 7, 6 of length 5 and 8 of length 3). Therefore it is cheaper to rename the binders on each chain. □

A.2 Naive replacement and α-equivalence

The α-conversion of a term M into a term N with $M \equiv_\alpha N$ can be done using at most one extra variable, as shown in [18]. The procedure described there requires replacing across a whole term M a variable x by another variable v fresh for M. This transformation has the following properties: $M \rightarrow_{x\backslash v} M'$ where $v \notin \mathcal{V}ar(M)$, $x \notin \mathcal{V}ar(M')$ and $M \equiv_\alpha M'$. Since it is not clear whether $M \equiv_\alpha M'$ holds if we naïvely replace every occurrence of x by v, we will show that this can be simulated by α-conversions, where α-equivalence holds by definition.

Definition A.13. $\mathcal{V}ar$(M), the set of all variables appearing in a term M (including the ones in λ's), can be defined inductively:

$$\mathcal{V}ar(t) = \begin{cases} \{x\} & \text{if } t = x \\ \mathcal{V}ar(M) \cup \mathcal{V}ar(N) & \text{if } t = M\ N \\ \{x\} \cup \mathcal{V}ar(M) & \text{if } t = \lambda x.M \end{cases}$$

Definition A.14. The function `alpha_step` performs one α-step (Definition 2.52):

$$\texttt{alpha_step}\ t\ x\ y = \begin{cases} \lambda y.M[\![x\backslash y]\!] & \text{if } t = \lambda x.M \\ t & \textit{else} \end{cases}$$

Definition A.15. The following function replaces in a λ-term M all appearances of a

variable x by a variable y fresh for M via α-conversions.

$$M\langle x\backslash y\rangle = \begin{cases} x & \text{if } M = x \\ z & \text{if } M = z \\ M_1\langle x\backslash y\rangle \; M_2\langle x\backslash y\rangle & \text{if } M = M_1 \; M_2 \\ (\texttt{alpha_step} \; M \; x \; y)\langle x\backslash y\rangle & \text{if } M = \lambda x.M_1 \\ \lambda z.M_1\langle x\backslash y\rangle & \text{if } M = \lambda z.M_1 \end{cases}$$

Definition A.16. The following function replaces in a λ-term M all appearances of a variable x by a variable y fresh for M via naïve replacement.

$$M(\!| x\backslash y |\!) = \begin{cases} y & \text{if } M = x \\ z & \text{if } M = z \\ M_1(\!| x\backslash y |\!) \; M_2(\!| x\backslash y |\!) & \text{if } M = M_1 \; M_2 \\ \lambda y.M_1(\!| x\backslash y |\!) & \text{if } M = \lambda x.M_1 \\ \lambda z.M_1(\!| x\backslash y |\!) & \text{if } M = \lambda z.M_1 \end{cases}$$

Theorem A.17. *Let M be the term we want to convert and y fresh for M. We can prove that $M(\!| x\backslash y |\!) \equiv_\alpha M[\![x\backslash y]\!]\langle x\backslash y\rangle$ i.e. the replacement of all appearances of x in M by y, which is fresh for M, can be simulated by α-conversions. $M[\![x\backslash N]\!]$ denotes the capture-avoiding substitution (Definition 2.56).*

Proof. We proceed by induction on M.

- For $M = x$

$$\begin{aligned} x[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha x(\!| x\backslash y |\!) \\ y\langle x\backslash y\rangle &\equiv_\alpha x(\!| x\backslash y |\!) \\ y &\equiv_\alpha x(\!| x\backslash y |\!) \\ y &\equiv_\alpha y \end{aligned}$$

- For $M = z$ where $z \neq x$ we get

$$\begin{aligned} z[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha z(\!| x\backslash y |\!) \\ z\langle x\backslash y\rangle &\equiv_\alpha z(\!| x\backslash y |\!) \\ z &\equiv_\alpha z(\!| x\backslash y |\!) \\ z &\equiv_\alpha z \end{aligned}$$

- For $M = M_1 \; M_2$ we get

$$\begin{aligned} (M_1 \; M_2)[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha (M_1 \; M_2)(\!| x\backslash y |\!) \\ (M_1[\![x\backslash y]\!]\langle x\backslash y\rangle)\,(M_2[\![x\backslash y]\!]\langle x\backslash y\rangle) &\equiv_\alpha (M_1 \; M_2)(\!| x\backslash y |\!) \\ (M_1[\![x\backslash y]\!]\langle x\backslash y\rangle)\,(M_2[\![x\backslash y]\!]\langle x\backslash y\rangle) &\equiv_\alpha M_1 \; (\!| x\backslash y |\!) \; M_2 \; (\!| x\backslash y |\!) \\ M_1[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha M_1 \; (\!| x\backslash y |\!) \; \wedge \\ M_2[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha M_2 \; (\!| x\backslash y |\!) \end{aligned}$$

here we use the induction hypothesis.

- For $M = \lambda x.N_1$ we get

$$\begin{aligned}
(\lambda x.N_1)[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha (\lambda x.N_1)(\!|x\backslash y|\!) \\
(\lambda x.N_1)\langle x\backslash y\rangle &\equiv_\alpha (\lambda x.N_1)(\!|x\backslash y|\!) \\
(\texttt{alpha_step}\ (\lambda x.N_1)\ x\ y)\langle x\backslash y\rangle &\equiv_\alpha (\lambda x.N_1)(\!|x\backslash y|\!) \\
\lambda y.N_1[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha (\lambda x.N_1)(\!|x\backslash y|\!) \\
\lambda y.N_1[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha \lambda y.N_1(\!|x\backslash y|\!)
\end{aligned}$$

  here we use the induction hypothesis.

- $M = \lambda y.N_1$ is excluded, since by assumption we have that y is fresh.
- For $M = \lambda z.N_1$ we get

$$\begin{aligned}
(\lambda z.N_1)[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha (\lambda z.N_1)(\!|x\backslash y|\!) \\
\lambda z.N_1[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha (\lambda z.N_1)(\!|x\backslash y|\!) \\
\lambda z.N_1[\![x\backslash y]\!]\langle x\backslash y\rangle &\equiv_\alpha \lambda z.N_1(\!|x\backslash y|\!)
\end{aligned}$$

  here we use the induction hypothesis.

□

A.3 De Bruijn indices and α-avoidance

When talking about α-avoidance, one could ask: ”Why not simply use De Bruijn indices?”. The name free calculus introduced by de Bruijn in 1972 [14] allows to represent λ-terms without any variables, but rather indices that indicate the ”distance” to the binding λ-node. In this representation, we cannot have name collisions. It seems, at first sight, that this representation solves the problem of α-avoidance. In this section, we will take a closer look at nameless terms to understand why they are not a proper solution to the problem we aim to solve.

Nameless terms Bound variables are implicitly connected to the λ-node that binds them. From a computational aspect, establishing and preserving this connection is the only purpose variables have. These implicit pointers do not necessarily have to be specified by variables; the de Bruijn indices offer an alternative way to do that. In Figure A.3 the connections are made explicit by a binding edge from the variables/indices to their binder.

The two examples of Figure A.3 semantically represent the same term. The connections established by variables in the usual representation of terms (lhs) can also be described by so-called de Bruijn indices (rhs), and in this case, they are identical. An expression in de Bruijn notation is generated by following grammar:

Figure A.3: Variables vs. de Bruijn indices

$$t = n \mid \lambda.t \mid t\ t$$

with n a natural number > 0 that specifies which λ-occurrence is the binder occurrence (replaces variables). It indicates the number of λ's we encounter in the abstract syntax tree on the way up to the root[1] [14]. As we can see from the definition, there are no variables involved.

This alternative representation has two advantages: i) no variable capture can occur when applying β-reduction ii) α-equivalence can be determined by checking syntactic equivalence, which allows simplifying many proofs. At first sight, it seems that this representation even solves the problem of α-avoidance, but this is not the case as we will see next.

Substitution and Shifting Obviously, we need a different notion of β-reduction for nameless terms. Indices are not absolute references and have therefore to be adapted continually. Even if we could apply naïve substitution when reducing terms in ordinary notation (see above), we could break the semantics when applying it to the same term in de Bruijn notation, as the following example shows.

$$\begin{array}{llll} & \lambda x.\underline{(\lambda y.x\ (\lambda z.y))\ (\lambda z.x\ z)} & & \lambda.\underline{(\lambda.2\ \lambda.2)\ (\lambda.2\ 1)} \\ \rightarrow_\beta & \lambda x.x\ (\lambda z.(\lambda z.x\ z)) & \not\rightarrow_\beta & \lambda.\mathbf{2}\ \lambda.(\lambda.\mathbf{2}\ 1) \\ & & \rightarrow_\beta & \lambda.\mathbf{1}\ \lambda.(\lambda.\mathbf{3}\ 1) \end{array}$$

Figure A.4: de Bruijn notation does not allow naive substitution.

A more complex substitution method is needed to avoid dangling pointers or an "index capture". These adaptations can thus be seen as a kind of α-renaming. This shows that the problem of α-avoidance is not solved but simply manifests itself differently.

Same but different de Bruijn already stated in [14] that the nameless notation is not easy to write and read for the human reader. Being easy to read is an important reason why we want to avoid α-renaming at all. This, for example, includes the possibility to relate input to output terms. Its importance was already motivated in Chapter 1. The different notations are therefore used out of different motivations. From the perspective of α-avoidance, we cannot claim that one is better than the other.

[1] free variables are mapped to indices that refer to some λ-node in the context (not part of the actual term)

A.4 PCP encoded in the λ-calculus

```
{-# LANGUAGE RankNTypes #-}
module Pcp where

-- BOOLEANS

type LCBoolT = forall a . a -> a -> a
newtype LCBool = LCBool { unBool :: LCBoolT }

true :: LCBool
true = LCBool $ \x y -> x

false :: LCBool
false = LCBool $ \x y -> y

-- if then else
ite :: LCBool -> a -> a -> a
ite = \c a b -> unBool c a b

-- returns constant true
ct :: forall b. b -> LCBool
ct = \x -> true

-- returns constant false
cf :: forall b. b -> LCBool
cf = \x -> false

-- logical and
land :: LCBool -> LCBool -> LCBool
land = \x y -> unBool x y false

-- logical or
lor :: LCBool -> LCBool -> LCBool
lor = \x y -> unBool x true y

-- PAIRS

type LCPair a = (a -> a -> a) -> a

pair :: a -> a -> LCPair a
pair = \a -> \b -> \c -> c a b

-- select first element in pair
first :: LCPair a -> a
first = \p -> p (\x y -> x)

-- select second element in pair
second :: LCPair a -> a
second = \p -> p (\x y -> y)

-- STRINGS
```

```
type LCStrT = forall a . (a -> a) -> (a -> a) -> a -> a
newtype LCStr = LCStr { unString :: LCStrT }

empty :: LCStr
empty = LCStr $ \a b x -> x

-- check if a string is the empty string
isempty :: LCStr -> LCBool
isempty = \s -> unString s cf cf true

-- concateation of two strings y, z
conc :: LCStr -> LCStr -> LCStr
conc = \y z -> LCStr $ \a b x -> unString y a b (unString z a b x)

-- prepend an a to string s
prepa :: LCStr -> LCStr
prepa = \s -> LCStr $ \a b x -> a (unString s a b x)

-- prepend a b to string s
prepb :: LCStr -> LCStr
prepb = \s -> LCStr $ \a b x -> b (unString s a b x)

-- check if the string s starts with an a
hd_a :: LCStr -> LCBool
hd_a = \s -> unString s ct cf false

-- check if the string s starts with a b
hd_b :: LCStr -> LCBool
hd_b = \s -> unString s cf ct false

-- check if the inital character of the strings x, y are equal
hd_eq :: LCStr -> LCStr -> LCBool
hd_eq = \x y -> lor (land (hd_a x) (hd_a y)) (land (hd_b x) (hd_b y))

-- get a pair of strings (s1, s2) and return (a(s1), s1)
nexta :: LCPair LCStr -> LCPair LCStr
nexta = \x -> pair (prepa (first x)) (first x)

-- get a pair of strings (s1, s2) and return (b(s1), s1)
nextb :: LCPair LCStr -> LCPair LCStr
nextb = \x -> pair (prepb (first x)) (first x)

-- get the tail of a string
tl :: LCStr -> LCStr
tl = \s -> second (unString s nexta nextb (pair empty empty))

-- check if two strings are equal
eq :: LCStr -> LCStr -> LCBool
eq = ycomb (\f x y ->
  ite (lor (isempty x) (isempty y))
  (land (isempty x) (isempty y))
  (land (hd_eq x y) (f (tl x) (tl y))))

-- check if a string x is prefix of another string y
```

```
prefix :: LCStr -> LCStr -> LCBool
prefix = ycomb (\f x y ->
  ite (isempty x) true
  (land (hd_eq x y) (f (tl x) (tl y))))

-- SOME STRINGS

a :: LCStr
a = LCStr $ \a b x -> a x

b :: LCStr
b = LCStr $ \a b x -> b x

ab :: LCStr
ab = LCStr $ \a b x -> a (b x)

ba :: LCStr
ba = LCStr $ \a b x -> b (a x)

bb :: LCStr
bb = LCStr $ \a b x -> b (b x)

baa :: LCStr
baa = LCStr $ \a b x -> b (a (a x))

bba :: LCStr
bba = LCStr $ \a b x -> b (b (a x))

abbb :: LCStr
abbb = LCStr $ \a b x -> a (b (b (b x)))

-- RECURSION

newtype Mu a = Mu (Mu a -> a)

-- y combinator
ycomb :: (b -> b) -> b
ycomb f = (\h -> h $ Mu h) (\x -> f . (\(Mu g) -> g) x $ x)


-- LISTS

type LCListPairStrT = forall a . ((LCPair LCStr) -> a -> a) -> a -> a
newtype LCListPairStr = LCList { unList :: LCListPairStrT }

-- empty list
nil :: LCListPairStr
nil = LCList $ \c -> \n -> n

-- cons
cons :: LCPair LCStr -> LCListPairStr -> LCListPairStr
cons = \h t -> LCList $ \c n -> c h (unList t c n)

-- check if list is empty
```

```
is_nil :: LCListPairStr -> LCBool
is_nil = \l -> unList l (\h t -> false) true

-- get head of a list
-- in case of empty list just return a constant pair
hd_l :: LCListPairStr -> LCPair LCStr
hd_l = \l -> unList l (\h t -> h) (pair empty empty)

-- get a list element x and a pair of lists (l1, l2) and return (x :: l1, l1)
next_l :: LCPair LCStr-> LCPair LCListPairStr -> LCPair LCListPairStr
next_l = \x p -> pair (cons x (first p)) (first p)

-- get the tail of a list
tl_l :: LCListPairStr -> LCListPairStr
tl_l = \l -> second (unList l next_l (pair nil nil))

-- append list y to list x
append :: LCListPairStr -> LCListPairStr -> LCListPairStr
append = \x y -> LCList $ \c n -> unList x c (unList y c n)

-- PCP Algorithm

-- get a pair of strings (a s1, a s2) and return (s1, s2)
simp :: LCPair LCStr -> LCPair LCStr
simp = \p -> ycomb (\f x y ->
  ite (lor (isempty x) (isempty y)) (pair x y)
  (f (tl x) (tl y))) (first p) (second p)

-- check if pair of strings is valid (at least one string prefix of the other)
pvalid :: LCPair LCStr -> LCBool
pvalid = \p -> lor (prefix (first p) (second p)) (prefix (second p) (first p))

-- check if there is a pair of two equal strings in a list of pair of strings
find_eq :: LCListPairStr -> LCBool
find_eq = ycomb (\f x ->
  ite (is_nil x) false (lor (eq (first (hd_l x)) (second (hd_l x))) (f (tl_l x))))

-- combine two pairs of strings (a1, a2), (b1, b2) to (a1 b1, a2 b2)
cmb :: LCPair LCStr -> LCPair LCStr -> LCPair LCStr
cmb = \p s -> pair (conc (first p) (first s)) (conc (second p) (second s))

-- combine a pair x with every pair in a list of pairs y
map_cmb :: LCPair LCStr -> LCListPairStr -> LCListPairStr
map_cmb = ycomb (\f x y ->
  ite (is_nil y) nil
    (ite (pvalid (cmb x (hd_l y))) (cons (simp (cmb x (hd_l y))) (f x (tl_l y)))
    (f x (tl_l y))))

-- combine two lists of pairs x,y with each other
cross_cmb :: LCListPairStr -> LCListPairStr -> LCListPairStr
cross_cmb = ycomb (\f x y ->
  ite (is_nil x) nil (append (map_cmb (hd_l x) y) (f (tl_l x) y)))

-- combine lists of pairs of strings
```

```
-- and check if any pair with equal strings is created
-- if yes return true otherwise reitercte
pcp :: LCListPairStr -> LCBool
pcp = \x -> ycomb (\f x y ->
  ite (is_nil x) false (ite (find_eq x) true (f (cross_cmb x y) y))) x x

-- PCP Problems

-- [(a, ab), (bb, b)]
problem_1 :: LCListPairStr
problem_1 = cons (pair a ab) (cons (pair bb b) nil)

-- [(a, abbb), (bb, b)]
problem_2 :: LCListPairStr
problem_2 = cons (pair a abbb) (cons (pair bb b) nil)

-- [(bba, b), (b, ab), (a, bba)]
problem_3 :: LCListPairStr -- undecidable
problem_3 = cons (pair bba b) (cons (pair b ab) (cons (pair a bba) nil))

-- [(ab, a), (ab, bba), (a, baa), (baa, ba)]
problem_4 :: LCListPairStr -- has a solution of length 76
problem_4 = cons (pair ab a) (cons (pair ab bba) (cons (pair a baa) (cons (pair baa ba)
    nil)))

-- PARSE LC ENCODINGS TO STRINGS

-- string encoding to String
lcstr_tostr :: LCStr -> String
lcstr_tostr = \s -> unString s (\a -> "a" ++ a) (\b -> "b" ++ b) ("")

-- bool encoding to String
lcbool_tostr :: LCBool -> String
lcbool_tostr = \b -> ite b "True" "False"
```