Type-Based Termination of Recursive Definitions and Constructor Subtyping in Typed Lambda Calculi

Maria João Gomes Frade

Departamento de Informática

Escola de Engenharia

Universidade do Minho

2003

Dissertação submetida à Universidade do Minho para obtenção do grau de Doutor em Informática, ramo de Fundamentos da Computação

Ao Zé.

Abstract

In type systems, a combination of subtyping and overloading is a way to achieve more precise typings. This thesis explores how to use these mechanisms in two directions: (i) as a way to ensure termination of recursive functions; (ii) as a way to capture in a type-theoretic context the use of subtyping as inclusion between inductively defined sets.

The first part of the thesis presents a mechanism that ensures termination through types and defines a system that incorporates it. More precisely, we formalize the notion of type-based termination using a restricted form of type dependency (also known as indexed types). Every datatype is replaced by a family of approximations indexed over a set of stages; then being in a certain approximation means that a term can be seen as having a certain bound on constructor usage. We introduce $\widehat{\lambda}$, a simply typed λ-calculus *à la* Curry, supporting parametric inductive datatypes, case-expressions and letrec-expressions with termination ensured by types. We show that $\widehat{\lambda}$ enjoys important meta-theoretical properties, including confluence, subject reduction and strong normalization. We also show that the calculus is powerful enough to encode many recursive definitions rejected by existing type systems, and give some examples. We prove that this system encompasses in a strict way Giménez' $\lambda_{\mathcal{G}}$, a system in which termination of typable expressions is ensured by a syntactical condition constraining the uses of recursive calls in the body of definitions.

The second part of the thesis studies properties of a type system featuring constructor subtyping. Constructor subtyping is a form of subtyping in which an inductive type σ is viewed as a subtype of another inductive type τ if each constructor c of σ is also a constructor of τ (but τ may have more constructors), and whenever $c : \theta \rightarrow \sigma$ is a declaration for σ, then $c : \theta' \rightarrow \tau$ is a declaration for τ with $\theta \leq \theta'$. In this thesis we allow for this form of subtyping in the system λ_{CS}, which is a simply typed λ-calculus *à la* Curry, supporting mutually recursive parametric datatypes, case-expressions and letrec-expressions. We establish the properties of confluence, subject reduction and decidability of type checking for this calculus. As the system features general recursion, the reduction calculus is obviously non-terminating. However, we sketch two ways of achieving strong normalization. One way is to constrain the system to guard-by-destructors recursion, following what is done for $\lambda_{\mathcal{G}}$. The other way is to enrich the type system with stages (following the ideas presented for $\widehat{\lambda}$) and enforcing termination through typing. Potential uses of constructor subtyping include proof assistants and functional programming languages. In particular, constructor subtyping provides a suitable foundation for extensible datatypes, and is specially adequate to re-usability. The combination of subtyping between datatypes and overloading of constructors allows the definition of new datatypes by restricting or by expanding the set of constructors of an already defined datatype. This flexibility in the definition of datatypes induces a convenient form of code reuse for recursive functions, allowing the definition of new functions by restricting or by expanding already defined ones. We enrich a calculus featuring constructor subtyping with a mechanism to define extensible overloaded recursive functions by pattern-matching, obtaining the system $\lambda_{\mathsf{CS+fun}}$. We formalize the concept of well-formed environment of function declarations and establish that under such environments the properties of confluence, subject reduction and decidability of type-checking hold. Moreover, we prove that the requirements imposed for the well-formed environments are decidable and show how standard techniques can still be used for compiling pattern-matching into case-expressions.

Resumo

Em sistemas de tipos, a combinação de mecanismos de subtipagem e de sobrecarga de construtores permite alcançar tipagens mais precisas para os termos. Esta tese investiga a utilização destes mecanismos, quer como forma de assegurar a terminação de funções recursivas, quer como forma de captar subtipagem através de inclusão de conjuntos num sistema com tipos indutivos.

A primeira parte da tese apresenta um sistema de tipos capaz de assegurar a terminação de funções recursivas, unicamente por tipagem. Mais concretamente, a noção de terminação baseada em tipos é formalizada utilizando uma forma restrita de dependência de tipos, também conhecida por tipos indexados. Cada tipo de dados é visto como uma família de aproximações, indexada por um conjunto de níveis, fornecendo tais níveis indicações sobre o uso de construtores na formação de termos. Esta forma de garantir terminação por tipos encontra-se formalizada no sistema $\lambda^{\frown}$ que é um cálculo lambda simplesmente tipado *à la* Curry, com tipos indutivos paramétricos, com expressões de ponto fixo e de análise de casos. Demonstra-se que $\lambda^{\frown}$ é um cálculo bem comportado, satisfazendo as propriedades de confluência, preservação de tipos ao longo da cadeia de redução, e normalização forte. O sistema $\lambda^{\frown}$ permite codificar muitas definições recursivas que são rejeitadas por outros sistemas com preocupações semelhantes de garantia de terminação. Em particular, prova-se que este cálculo engloba de modo estrito o sistema $\lambda_{\mathcal{G}}$ de Giménez, um sistema em que a terminação das expressões tipáveis é assegurada por uma condição sintáctica que restringe as chamadas recursivas de funções.

Na segunda parte da tese, apresenta-se um sistema de tipos com subtipagem por construtores e estudam-se as suas propriadades. A subtipagem por construtores é uma forma de subtipagem na qual um tipo indutivo σ é visto como um subtipo de um outro tipo indutivo τ, se τ tiver mais construtores do que σ. Neste trabalho, a subtipagem por construtores está presente no sistema λ_{CS}, um cálculo lambda simplesmente tipado, *à la* Curry, com tipos inductivos paramétricos e mutuamente recursivos, com expressões de ponto fixo e de análise de casos. Demonstra-se que este cálculo é confluente, a tipagem é decidível e a redução preserva tipos. Para garantir a normalização forte, são propostas duas abordagens: satisfação de uma condição sintáctica nas definições recursivas (à semelhança de $\lambda_{\mathcal{G}}$), ou enriquecimento do sistema de tipos com níveis (à semelhança de $\lambda^{\frown}$) de forma a garantir terminação por tipagem. Esta forma de subtipagem encontra aplicações nos sistemas de prova assistida e nas linguagens funcionais de programação. Em particular, a subtipagem por construtores revela-se adequada para o tratamento de tipos de dados extensíveis. A combinação da subtipagem com a sobrecarga de construtores permite que a definição de novos tipos de dados possa ser feita por restrição ou expansão do conjunto de construtores de um tipo de dados já definido. Esta flexibilidade na definição de tipos de dados induz uma forma de re-utilização de código adequada às funções recursivas, permitindo que a definição de novas funções se possa fazer também por restrição ou expansão de funções já definidas. Estes macanismos são estudados no âmbito do sistema $\lambda_{\mathsf{CS+fun}}$, um cálculo lambda com subtipagem por construtores e com definições recursivas sobrecarregadas e extensíveis, definidas por concordância de padrões num ambiente global. Define-se, para este cálculo, o conceito de ambiente bem formado de funções, e demonstra-se que, para estes ambientes, as propriedades de confluência, decidibilidade de tipagem e preservação de tipos são válidas. Também se demonstra que os vários requisitos impostos para garantir a boa formação do ambiente global de funções correspondem a propriedades decidíveis. Finalmente, descreve-se um algoritmo de compilação das funções definidas por concordância de padrões para expressões com análise de casos.

Acknowledgments

First of all, I would like to thank José Manuel Valença and Gilles Barthe for having supervised my research in the last years, sharing their knowledge with me.

Gilles Barthe taught me a lot about type theory and about research in general. I have been very fortunate to have him as supervisor and as a friend. I thank Gilles Barthe for being a galvanizing supervisor, always full of new ideas, enthusiasm and energy. Gilles provided me with many useful contacts which have been very important not only for me but also for my university.

This work has benefited from fruitful collaborations with Tarmo Uustalu, Eduardo Giménez and Luis Pinto. The results of this cooperation are reflected in the first part of this thesis. I am very grateful to all of them. I am also grateful to Gustavo Betarte and Jan Zwanenburg for useful discussions on extensible overloaded functions.

I would like to express my immense gratitude to Luis Pinto, for his constant support, encouragement and friendship, and for the time he spent on reading many of my manuscripts, correcting my English, suggesting better explanations, and indicating technical problems.

I wish to specially thank Peter Dybjer who carefully and promptly read an earlier version of this thesis, providing many helpful comments and remarks.

I would like to take this opportunity to thank everybody at the Informatics Department at the University of Minho for the pleasant working environment. In particular, I would like to thank the people of the Logic and Formal Methods group for their constant support, encouragement and friendship.

Last but not least I want to thank my family for their support during the ups and downs of the work on this thesis, and for all the rest that I cannot put into words.

Contents

1 Introduction **1**

1.1 Types in Programming Languages and Proof Assistants . . . 1
1.2 Inductive Definitions . . . 4
1.3 Type-Based Termination . . . 6
1.4 Subtyping . . . 8
1.5 Constructor Subtyping . . . 9
1.6 Summary of Contributions . . . 11

I Type-Based Termination of Recursive Definitions **15**

2 An Informal Account of Type-Based Termination of Recursive Definitions **17**

2.1 Background . . . 17
2.2 Outline of Type-Based Termination . . . 20
2.3 Overview of This Part . . . 20

3 The System $\widehat{\lambda}$ **23**

3.1 Datatypes . . . 23
3.2 Terms and Reduction . . . 23
3.2.1 Terms . . . 23
3.2.2 Reduction calculus . . . 24
3.3 Types, Subtyping and Typing . . . 27
3.3.1 Types and Subtyping . . . 27
3.3.2 The Typing System . . . 30
3.4 Some Examples . . . 33

4 Meta-Theoretical Results for $\widehat{\lambda}$ **39**

4.1 Confluence . . . 39
4.2 Subject Reduction . . . 41
4.3 Strong Normalization . . . 45
4.3.1 Saturated sets and interpretation domains . . . 45
4.3.2 Soundness w.r.t. the Semantics . . . 55

5 The System $\lambda_{\mathcal{G}}$ **59**

5.1 Definition of $\lambda_{\mathcal{G}}$ . . . 59
5.2 From $\lambda_{\mathcal{G}}$ to $\widehat{\lambda}$ . . . 66

6 Related Work and Conclusion **71**
6.1 Related Work . . . 71
6.2 Conclusion . . . 74

II Constructor Subtyping **77**

7 An Informal Account of Constructor Subtyping **79**
7.1 Motivations and Difficulties . . . 79
7.1.1 Problematic Examples . . . 80
7.1.2 Strict Overloading . . . 81
7.2 Further Examples . . . 82
7.3 Adding Extensible Overloaded Functions . . . 87
7.3.1 An Example of Overloading . . . 87
7.3.2 An Example of Extensibility . . . 88
7.4 Overview of This Part . . . 89

8 The Core Calculus λ_{CS} **91**
8.1 The System λ_{CS} . . . 91
8.1.1 Terms and Reductions . . . 91
8.1.2 Types and Subtyping . . . 94
8.1.3 The Typing System . . . 97
8.2 Confluence . . . 99
8.3 Subject Reduction . . . 101
8.4 Strong Normalization . . . 106
8.4.1 Guarded-by-Destructors Recursion . . . 106
8.4.2 Type-Based Termination . . . 109
8.5 Type Checking . . . 110
8.5.1 Motivation and Difficulties . . . 112
8.5.2 The System $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ . . . 114
8.5.3 The System $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ . . . 116
8.5.4 Decidability of Type Checking . . . 135

9 Extensible Overloaded Functions **139**
9.1 The System $\lambda_{\mathsf{CS+fun}}$ . . . 139
9.1.1 Types and Terms . . . 139
9.1.2 Subtyping and Typing . . . 140
9.1.3 Definition of Functions . . . 141
9.1.4 Reduction Calculus . . . 143
9.1.5 Well-Formed Environments . . . 144
9.2 Confluence . . . 148
9.3 Subject Reduction . . . 150
9.4 Strong Normalization . . . 155
9.5 Type Checking . . . 156
9.5.1 The System $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$ . . . 156
9.5.2 The System $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$ . . . 157
9.5.3 Decidability of Type Checking . . . 158
9.6 Decidability of Well-Formedness for Environments . . . 159

9.6.1 Decidability of Well-Typing 159
9.6.2 Decidability of Non-Overlapping 159
9.6.3 Decidability of Exhaustiveness 160
9.7 The System $\lambda_{\mathsf{CS+def}}$ 163
9.7.1 Types and Terms 164
9.7.2 Subtyping and Typing 164
9.7.3 Environments 164
9.7.4 Reduction Calculus 165
9.7.5 Meta-Theoretical Properties of $\lambda_{\mathsf{CS+def}}$ 166
9.8 Compiling $\lambda_{\mathsf{CS+fun}}$ into $\lambda_{\mathsf{CS+def}}$ 167

10 Related Work and Conclusion **173**
10.1 Related Work 173
10.2 Conclusion 176

List of Figures

3.1 Positive-negative occurrences of type variables or datatypes . . . 28
3.2 Stage comparison rules $\lambda^{\frown}$ . . . 30
3.3 Subtyping rules for $\lambda^{\frown}$ . . . 30
3.4 Typing rules for $\lambda^{\frown}$ . . . 31
3.5 Positive-negative occurrences of a stage variable . . . 32

5.1 Strictly positive rules . . . 60
5.2 Guarded-by-destructors rules for $\lambda_{\mathcal{G}}$ . . . 62
5.3 Typing rules for $\lambda_{\mathcal{G}}$ . . . 65

8.1 Positive-Negative rules . . . 95
8.2 Strictly positive rules . . . 95
8.3 Subtyping rules for λ_{CS} . . . 97
8.4 Typing rules for λ_{CS} . . . 98
8.5 Guarded-by-destructors rules for λ_{CS} . . . 108
8.6 Typing rule for letrec-expressions in $\lambda_{\mathsf{CS}}^{\mathcal{G}}$ . . . 108
8.7 Typing rules for $\lambda_{\widehat{\mathsf{CS}}}$ . . . 111
8.8 Rules for $\sqsubseteq$ . . . 117
8.9 Subtyping rules for $\lambda_{\mathsf{CS}}^{\mathsf{ac}}$ . . . 117
8.10 Typing rules for $\lambda_{\mathsf{CS}}^{\mathsf{ac}}$ . . . 118
8.11 The algorithm unify . . . 125
8.12 The algorithm unifyData . . . 125
8.13 The algorithm match . . . 126
8.14 The algorithm typeJudg . . . 128
8.15 The algorithm derivable . . . 136
8.16 The algorithm satisfiable . . . 137

9.1 Subtyping rules for $\lambda_{\mathsf{CS+fun}}$ . . . 140
9.2 Typing rules for $\lambda_{\mathsf{CS+fun}}$ . . . 141
9.3 Structurally smaller relation . . . 156
9.4 The algorithm typeJudg (for functions) . . . 157
9.5 Rules for overlap . . . 160
9.6 Rules for exh . . . 161
9.7 The algorithm translate . . . 168
9.8 The algorithm compile . . . 168

Chapter 1

Introduction

1.1 Types in Programming Languages and Proof Assistants

The central feature of type systems is their emphasis on classification of objects: in a type system, objects do not exist in isolation, but always in relation to their types. This relationship between objects and types is captured by judgments of the form $a : \sigma$ denoting that a is an object of type σ (or, a is an inhabitant of σ).

Type systems were first introduced by Russell [139], in the early 1900s, as a mean of avoiding the logical paradoxes that threatened the foundations of mathematics.

In 1932/33 Church [36, 37] attempted to found mathematics on a logical system based on the notions of function and function application. Church's foundational attempt failed but originated the pure theory of functions—the *lambda calculus*—where terms are built up from variables by λ-abstraction ($\lambda x.a$ is the function of argument x and body a) and application ($a\,b$ is the application of the function a to b), and manipulated by the β reduction rule $(\lambda x.a)\,b \to_\beta a[x := b]$ that indicates how to compute the value of a function for an argument. Not all computations lead to results, and reduction sequences may not terminate: e.g. $(\lambda x.x\,x)\,(\lambda x.x\,x)$ reduces to itself.

In 1940, Church introduced the *typed lambda calculus* [38, 39], in which functions are classified with simple types that determine the type of their arguments and the type of the values they produce, and can be applied only to arguments of the appropriate type. Terms are now of the form $(\lambda x^\tau.a^\sigma)^{\tau\to\sigma}$ and $(a^{\tau\to\sigma}\,b^\tau)^\sigma$, writing types as superscripts. Typed λ-calculus is a theory of total functions, as all reduction sequences starting from a typable term terminate. An important consequence of termination is that the equational theory of typed λ-calculus is decidable: to know whether two terms are equal w.r.t. β equality (the least equivalence relation to contain β reduction) one just has to compare their normal forms (it follows from confluence that every term has a unique normal form).

A different typed version of lambda calculus had been already introduced in 1934, by Curry, for the combinatory logic [49]. In Curry's approach to typed lambda calculus, terms are those of the type-free theory (i.e. without any type annotation), whereas in Church's approach terms are annotated versions of the type-free terms.

Very important in the understanding of the language of type theory is the distinction between *canonical* objects from the others. The canonical objects are the ones that represent the values of the type. The canonical inhabitants of a type are defined as its closed objects in normal form and provide a fundamental tool in the semantical understanding of a type. The notion of *computation* is another basic concept of type theory which generates an equivalence relation—the *computational equality* between the objects in the language of type theory. An important property of computation is that every object has a unique value under computation and the objects which are computationally equal have the same value. A type may be understood as consisting of its canonical objects. In this sense, a judgment $a : \sigma$ means that a computes into a canonical object of type σ.

The meaning of type theory is explained in terms of computations. It is essential that the computation relation enjoys some fundamental properties such as:

- *confluence*, which ensures that the way of computing objects is irrelevant, as all possible ways lead to the same canonical expression (whenever there exists one);
- *subject reduction*, which ensures that the type of an object is preserved under reduction;
- *strong normalization*, which ensures that all reduction sequences starting from a typable expression (object) terminate.

These properties are important as they ensure that the use of the objects of a type are in accordance with the characterization of the types in terms of canonical inhabitants. Moreover, it implies the decidability of the equational theory and consistency of the calculus as a logic.

The simply typed lambda calculus is simple and elegant but it has a weak expressive power. Subsequent research has focused on extending simple typed lambda calculus to systems with the same meta-theoretical properties, but with greater expressive power. Some of the major landmarks are *constructive type theory* [97, 98] and *pure type systems* [16, 15], just to name two. These extensions have contributed to the fact that during the twentieth century, types permeated programming languages and have become standard tools in logic [86]. Below we briefly describe the application of type systems to programming languages, and also their connection to logic.

Types in programming languages

Type systems are useful in programming because they provide partial specifications of the programs: a typed program specifies the set of admissible inputs and guarantees that the output will be of a certain kind. In this sense, the checking that the type of a program matches the type declared for it can be seen as a partial correctness verification. Type systems provided the basis for typed programming languages. Types were first used in FORTRAN, in the 1950s, to make very simple distinctions between integer and floating point representations of numbers. In the 1960s and early 1970s, type systems were used to classify also structured data such as arrays and records, in programming languages like ALGOL-68 or PASCAL. Later, more sophisticated typed languages appeared, such as ML, Miranda, Haskell, Modula, Java or C#.

The most obvious benefit of type checking is that it allows early detection of some programming errors. Typing errors can be caught at compile time and can be fixed immediately. Besides, type systems enforce disciplined programming. Type declarations constitute a form of documentation, giving useful hints about the behavior of the functions. Moreover, this form of documentation is always effective, since it is checked during every run of the compiler. In this sense, a type-checker can also be an invaluable maintenance tool that (after some change in a program) helps to find the places in a program where code needs to be fixed. Type checking can expose a surprisingly broad range of errors. The ability to detect errors early has led to considerable improvements in the productivity and quality of programming. The strength of this effect depends on the expressiveness of the type system. But a type is by no means a complete specification of programs. To describe and reason about programs in type theory one also needs logic.

Types in logic

The connection of type theory to logic is via the *propositions-as-types principle* that establishes a precise relation between intuitionistic logic and computation. Intuitionistic logic is based on the notion of proof—a proposition is true when we can provide a constructive proof of it. On this basis, a proposition P can be seen as a type—the type of its proofs— and a proof of P as an object of type P. Hence, a false proposition is interpreted as an empty type (i.e. a type that has no inhabitants) and a true proposition as a non-empty type. The introduction and elimination rules for implication correspond quite naturally to the λ-abstraction and application rules of term formation in typed lambda calculus.

The correspondence between intuitionistic logic and lambda calculus is known as the *Curry-Howard isomorphism*, and is at the core of constructive type theory. This isomorphism has been expanded to new systems of types corresponding to various intuitionistic theories (second-order, higher-order, many sorted). For instance: a proof of $\forall x \in A.B(x)$ is a function that transforms every object a of type A into a proof of $B(a)$; and a proof of $\exists x \in A.B(x)$ is a pair with an object a of type A and a proof of $B(a)$.

Types for programming and reasoning

The unifying view of a type system as a programming language and as a logic has important applications in programming, because a type can be used to specify a programming problem. When a type is seen as a specification, the inhabitants of the type are the programs that satisfy the specification. For example, consider the specification given by the formula $\forall x \in A.\exists y \in B.R(x, y)$. According to the intuitionistic interpretation of the logical connectives, given above, the proof of this statement is a function that takes an object a of type A and yields an object b_a of type B together with a proof that $R(a, b_a)$ holds. Thus, a proof term of $\forall x \in A.\exists y \in B.R(x, y)$ contains an algorithm to construct an element $b_a \in B$ for every $a \in A$ such that $R(a, b_a)$ holds. This opens the possibility of extracting programs from proofs. These programs are certified with their proof of correctness.

The formal language of type theory is used as a (functional) programming language, a specification language (a specification is expressed as a type, the type of all correct programs satisfying the specification) and a programming logic (which has rules for deducting the correctness of programs). So, type theory provides a unified framework for specifying, developing and reasoning about functional programs. In type theory the problem of program correctness boils down to the problem of type inhabitance. Furthermore, because all programs terminate, correctness, means total correctness.

A practical application of the Curry-Howard isomorphism is the possibility of doing mathematics on a computer by implementing the formal system of a typed lambda calculus. The first systems of proof checking (type checking) based on the propositions-as-types principle were the systems of the AUTOMATH project [52]. Modern systems provide computer assistance for the interactive construction of proofs, aggregating to the proof checker a proof-development system for helping the user to develop the proofs (i.e. to construct proof-terms). These systems are called proof assistants. We can mention as examples of proof assistants, the systems Coq [125] and Lego [92] which are based on versions of the calculus of constructions extended with inductive types, the systems Alf [5] and Agda [42] which are based on Martin-Löf's type theory, and the system Nuprl [41] which is an implementation of extensional Martin-Löf's type theory.

1.2 Inductive Definitions

Typed functional programming languages and proof assistants support mechanisms to declare inductive datatypes and to define recursive functions, so that one can program with and reason about elements of these datatypes, as functions defined over inductive types are usually recursive. Directly related to recursion we have the problem of termination (normalization), which plays a crucial role in the semantical understanding of the datatypes, as recursive definitions introduce non-canonical elements.

We can define a new datatype d inductively by giving its constructors together with their types. The constructors (which are the *introduction rules* of the datatype) give the more basic or canonical ways of constructing one element of the new type d. The type of constructors must be of the form $\tau_1 \to \ldots \to \tau_n \to d$ (with $n \geq 0$), where the types τ_i can be function types (we are considering higher-order datatypes). The datatype defined is the smallest set (of objects) closed under its introduction rules. Informally speaking, the elements of a datatype are the objects that can be obtained by a finite number of applications of the datatype constructors. A well-known example of an inductive datatype, is the type Nat of natural numbers defined by:

```
data Nat = O : Nat
         | S : Nat -> Nat
```

To program with and reason about a datatype we must have means to analyze its objects. The *elimination rules* for the datatype express ways to use the objects of the datatype in order to define objects of other types, and are associated to new computational rules. From the explanations given before, it is natural to consider case analysis as a natural elimination rule for inductive types. For instance, $n : \mathsf{Nat}$ means that n was introduced using either $\mathtt{O}$ or $\mathtt{S}$, so we may define an object $\mathsf{case}\ n\ \mathsf{of}\ \{\mathtt{O} \Rightarrow b_1 \mid \mathtt{S} \Rightarrow b_2\}$ in another type σ depending on which constructor was used to introduce n. A typing rule for this construction is

$$\frac{n : \mathsf{Nat} \qquad b_1 : \sigma \qquad b_2 : \mathsf{Nat} \to \sigma}{\mathsf{case}\ n\ \mathsf{of}\ \{\mathtt{O} \Rightarrow b_1 \mid \mathtt{S} \Rightarrow b_2\} : \sigma}$$

and the associated computing rules are

$$\begin{array}{lll} \mathsf{case}\ \mathtt{O}\ \mathsf{of}\ \{\mathtt{O} \Rightarrow b_1 \mid \mathtt{S} \Rightarrow b_2\} & \to_\iota & b_1 \\ \mathsf{case}\ (\mathtt{S}\,x)\ \mathsf{of}\ \{\mathtt{O} \Rightarrow b_1 \mid \mathtt{S} \Rightarrow b_2\} & \to_\iota & b_2\,x \end{array}$$

The case analysis rule is very useful but it does not give a mechanism to define recursive functions.

Recursors

A safe way to express recursion without introducing non-normalizable objects is to associate an elimination rule, with appropriate computational rules, to each inductive datatype. The elimination rule associated with a datatype is a constant (recursor) that represents the structural induction principle for the elements of the datatype, and the computation rule associated to it defines a safe recursive scheme for programming, known as primitive recursion. For example, the recursor for Nat, $\mathsf{nat\_elim}$, has the following typing rule:

$$\frac{a : \theta \qquad f : \mathsf{Nat} \to \theta \to \theta \qquad n : \mathsf{Nat}}{\mathsf{nat\_elim}\,a\,f\,n : \theta}$$

To make sure that the functions compute in a correct way, the reduction rules associated to $\mathsf{nat\_elim}$ are

$$\begin{array}{lll} \mathsf{nat\_elim}\,a\,f\,\mathtt{O} & \to & a \\ \mathsf{nat\_elim}\,a\,f\,(\mathtt{S}\,x) & \to & f\,x\,(\mathsf{nat\_elim}\,a\,f\,x) \end{array}$$

An example of the use of $\mathsf{nat\_elim}$ is the definition of the function that doubles a natural number

```
let double n = nat_elim O (fun x y -> S (S y)) n
```

This approach is adopted by the traditional presentations of type-based proof development systems [46, 54, 90, 112], and guarantees the logical consistency of the system. However, codifying recursive functions in terms of elimination constants can be rather difficult, and is quite far from the way we are used to program.

General recursion

Functional programming languages feature *general* recursion, allowing recursive functions to be defined by means of pattern-matching and a general fixpoint operator to encode recursive calls. The typing rule for Nat fixpoint expressions is

$$\frac{f : \mathsf{Nat} \to \theta \;\vdash\; e : \mathsf{Nat} \to \theta}{\vdash\; (\mathsf{letrec}\ f = e) : \mathsf{Nat} \to \theta}$$

and the associated computation rule is

$$(\mathsf{letrec}\ f = e)\,(c\,\vec{a}) \;\to_{\mu}\; e[f := (\mathsf{letrec}\ f = e)]\,(c\,\vec{a})$$

With pattern-matching and fixpoint operators the function `double` can be written as follows

```
let rec double n = case n of
                     O      -> O
                   | (S x)  -> S (S (double x))
                   end
```

This is a much more pleasant (and flexible) way of programming, but leaves to the programmer the responsibility of writing terminating programs (if that is what he wants). Of course, this approach opens the door to the introduction of non-normalizable objects, but it raises the level of expressiveness of the language.

On positivity

While functional programming languages allow non-terminating functions, proof development systems based on the Curry-Howard analogy only allow terminating functions, both to guarantee the decidability of type checking and to avoid problems related to the consistency of the logical system.

As illustrated above, general recursion permits the definition of non-terminating functions. So does the possibility of declaring non well-founded datatypes, as illustrated by the following example of a non-normalizing term taken from [65]:

Consider a datatype d defined by a single introduction rule $\mathtt{C} : (d \to \theta) \to d$, where θ may be any type (even the empty type). Let $\mathsf{app} \equiv \lambda x.\lambda y.\, \mathsf{case}\ x\ \mathsf{of}\ \{\mathtt{C} \Rightarrow \lambda f.\, f\, y\}$, $\mathtt{t} \equiv (\lambda z.\, \mathsf{app}\, z\, z)$ and $\Omega \equiv \mathsf{app}\,(\mathtt{C}\,\mathtt{t})\,(\mathtt{C}\,\mathtt{t})$. We have $\mathsf{app} : d \to d \to \theta$, $\mathtt{t} : d \to \theta$ and $\Omega : \theta$. However, Ω is a looping term which has no canonical form

$$\Omega \;\twoheadrightarrow_{\beta}\; \mathsf{case}\ (\mathtt{C}\,\mathtt{t})\ \mathsf{of}\ \{\mathtt{C} \Rightarrow \lambda f.\, f\,(\mathtt{C}\,\mathtt{t})\} \;\to_{\iota}\; (\lambda f.\, f\,(\mathtt{C}\,\mathtt{t}))\,\mathtt{t} \;\to_{\beta}\; \mathtt{t}\,(\mathtt{C}\,\mathtt{t}) \;\to_{\beta}\; \Omega$$

What enables to construct a non-normalizing term in θ is the negative occurrence of d in the domain of $\mathtt{C}$.

In order to banish non-well-founded elements from the language, proof assistants usually impose a *strict positivity* condition on the possible forms of the introduction rules of the inductive datatypes. This condition states that the datatype under definition can only occur in the domain of its constructors in a strict positive position, i.e. it never appear on the left side of some arrow. (It is however possible to relax the strict positivity condition, see e.g. [101, 4]) Notice that the strict positivity condition still permits functional recursive arguments in the constructors. A well-known example of a higher-order datatype is the type `Ord` of ordinal notations:

```
data Ord  = Zero: Ord
          | Succ : Ord -> Ord
          | Lim : (Nat -> Ord) -> Ord
```

Here, the constructor `Lim` has a function as argument.

1.3 Type-Based Termination

The tension between termination and expressiveness (and simplicity) is an issue in the design of proof-development systems. On the one hand, the use of recursors is theoretically sound and enables the representation of a large class of functions. On the other hand, such a codification introduces some rigidity in the language of type theory. Programming with elimination rule is rather inconvenient and is quite far from the way we are used to program.

In some recent works, recursive functions are described in terms of a pattern-matching operator (`case`) and a general fixpoint operator (`let-rec`), in the style of functional programming languages. In this approach, the function `plus` could be introduced as follows:

```
let rec plus n m = case m of
                     O       -> n
                   | (S p)   -> (S (plus n p))
                   end
```

which is more readable than the codification of `plus` using the recursor for Nat:

```
let plus n m = nat_elim n (fun p r -> (S r)) m
```

However, in general, there is no guarantee that a recursive function defined with the fixpoint operator will always terminate.

Looking for a good compromise between termination and presentation issues, [44] suggested that recursors should be replaced by case-expressions and a restricted form of fixpoint expressions, see also [67]. The restriction is imposed through a predicate $\mathcal{G}_f$ on untyped terms. This predicate enforces termination by constraining all recursive calls to be applied to terms structurally smaller than the formal argument x of f —for instance, a pattern variable issued from a case-expression on x. The restricted typing rule for fixpoint expressions hence becomes:

$$\frac{f : \mathsf{Nat} \to \theta \;\vdash\; e : \mathsf{Nat} \to \theta}{\vdash\; (\mathsf{letrec}\ f = e) : \mathsf{Nat} \to \theta} \qquad \text{if } \mathcal{G}_f(e)$$

This alternative approach, called *guarded-by-destructors* recursion in [67], has been implemented in the Coq system. Several years of experiments carried out with Coq have shown that it actually provides much more palatable representations of recursive functions.

However, the use of an external predicate $\mathcal{G}$ on untyped terms suffers from several weaknesses. In particular, the guard predicate is very sensitive to syntax, quite weak, and hard to implement[1]. In order to circumvent those weaknesses, some authors have proposed semantically motivated type systems that ensure the termination of recursive definitions through typing [66, 8, 17]. The idea, which already occurs in Mendler's work [106], consists in regarding an inductive type d as the least fixpoint of a monotonic operator $\widehat{-}^d$ on types, and to enforce termination of recursive functions by requiring that the definition of $f : \widehat{\alpha}^d \to \theta$, where α may be thought as a subtype of d, only relies on structurally smaller function calls, embodied by a function $f_{\mathrm{ih}} : \alpha \to \theta$. This approach to terminating recursion, which we call *type-based*, offers several advantages over the *guarded by destructors* approach. In particular, it addresses all the above-mentioned weaknesses.

In this thesis, we have devised a mechanism that ensures termination by typing and define a system that incorporates it: $\lambda\widehat{\ }$, a simply typed λ-calculus that supports type-based recursive definitions. Although heavily inspired from previous work by Giménez [66] and closely related to recent work by Amadio and Coupet [8], the technical machinery behind our system puts a slightly different emphasis on the interpretation of types. More precisely, we formalize the notion of type-based termination using a restricted form of type dependency (a.k.a. indexed types), as popularized by Xi and Pfenning [141, 142]. This leads to a simple and intuitive system which is robust under several extensions, such as mutually inductive datatypes and mutually recursive function definitions.

The basic idea is to proceed as follows:

- First, every datatype d is replaced by a family of approximations indexed over a set of *stages*, which are used to record a bound on the "depth" of values. Here, we adopt a simple minded approach and let stages range over the syntax

  $$s := \imath \mid \widehat{s} \mid \infty$$

  where $\imath$ ranges over stage variables, the hat operator $\widehat{-}$ is a function mapping a stage to its "successor" and ∞ is the stage at which the iterative approximation process converges to the datatype itself. On the stages, we introduce an ordering relation, and induced by this relation on stages we have a subtyping relation on types stating that a given approximation of a datatype is always included in the next one.

- Second, a recursive definition of a function, say $f : d \to \theta$ should be given by a term e constructing a function $g' : d^{\widehat{\imath}} \to \theta$ from $g : d^{\imath} \to \theta$, where $\imath$ ranges over stages (in other words, e should be stage-polymorphic).

In order to illustrate the machinery, let us consider the inductive type Nat whose constructors are $\mathtt{O} : \mathsf{Nat}$ and $\mathtt{S} : \mathsf{Nat} \to \mathsf{Nat}$. Constructors are overloaded, in the sense that we also have $\mathtt{O} : \mathsf{Nat}^{\widehat{s}}$ and $\mathtt{S} : \mathsf{Nat}^{s} \to \mathsf{Nat}^{\widehat{s}}$. The typing rules are

$$\frac{}{\vdash\ \mathtt{O} : \mathsf{Nat}^{\widehat{s}}} \qquad\qquad \frac{\vdash\ n : \mathsf{Nat}^{s}}{\vdash\ \mathtt{S}\ n : \mathsf{Nat}^{\widehat{s}}}$$

and, as an approximation of a datatype is always included in the next one, we also have

$$\frac{\vdash\ n : \mathsf{Nat}^{s}}{\vdash\ n : \mathsf{Nat}^{\widehat{s}}}$$

[1] In fact, some versions of Coq accept non-terminating recursive definitions.

Finally recursive functions from Nat to θ are constructed with the following typing rule:

$$\frac{f : \mathsf{Nat}^{\imath} \to \theta \;\vdash\; e : \mathsf{Nat}^{\widehat{\imath}} \to \theta}{\vdash\; (\mathsf{letrec}\ f = e) : \mathsf{Nat} \to \theta}$$

where $\imath$ is fresh w.r.t. θ. As shown in Part I, such recursive functions are terminating and, despite its simplicity, this mechanism is powerful enough to capture course-of-value primitive recursion.

1.4 Subtyping

While strong typing has been recognized as one of the big successes in computer science, it also imposes stringent limits to the expressiveness of typed languages, as each object has a single type. Consequently, several authors have repeatedly advocated the use of subtyping in typed functional programming languages, see e.g. [57, 73], and proof-development systems, see e.g. [9, 91, 117].

A basic mechanism to enhance the flexibility of type systems is to endorse the set of types with a subtyping relation $\leq$ and to enforce a *subsumption* rule

$$\frac{a : \tau \qquad \tau \leq \sigma}{a : \sigma}$$

which makes the bridge between the typing relation and the subtyping relation. This rule tells us that, if $\tau \leq \sigma$ (pronounced "τ is a subtype of σ" or "σ is a supertype of τ"), then every element a of τ is also an element of σ.

The subtyping relation $\tau \leq \sigma$ captures different concepts, accordingly to the interpretation we give to types: when τ and σ represent sets, then an element belonging to τ also belongs to σ; when τ and σ represent object descriptions, then an object with interface τ can be safely used where an object with interface σ is expected; when τ and σ represent propositions, then a proof of proposition τ is also a proof of proposition σ; when τ and σ represent specifications, then a program satisfying specification τ also satisfies specification σ. As a result, type systems with subtyping provide flexible and expressive frameworks in which to develop programs, proofs or specifications.

Subtyping is a powerful abstraction that permeates computer science. To mention only two examples, subtyping plays a crucial role in object-oriented languages [1, 72, 114] and specification languages [50, 74]. The basic idea of subtyping, as embodied in the subtyping relation $\tau \leq \sigma$ between types, is that every inhabitant a of τ is also an inhabitant of σ. Consequently, the main importance of subtyping is that it allows substitutivity: if τ is a subtype of σ, then elements of type τ can be safely used where an element of type σ is required.

The subtype relation is formalized as a collection of inference rules for deriving the subtyping statements $\tau \leq \sigma$, and it should be guided by the principle that in any context where an expression of type σ is required, an expression of type τ should be allowed too. It follows naturally from this principle that the subtyping relation should be reflexive and transitive:

$$\frac{}{\tau \leq \tau} \qquad\qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

The subtyping between base types should be introduced by axioms; and for each form of type (function types, datatypes, record types,...) adequate subtyping rules should formalize situations where it is safe to allow elements of one type of this form to be used where another is expected. For instance, the subtype relationship between function types is described by the rule

$$\frac{\tau' \leq \tau \qquad \sigma \leq \sigma'}{\tau \to \sigma \leq \tau' \to \sigma'}$$

The intuition is that, if we have a function f of type $\tau \to \sigma$, then we know that f accepts elements of type τ and returns elements of type σ; thus, f will also accept elements of a subtype τ' of τ, and we can view the elements returned by f belonging to any supertype σ' of σ. That is, any function of type $\tau \to \sigma$ can also be viewed as having type $\tau' \to \sigma'$. So, function subtyping is covariant in the result type and contravariant in the argument type.

There are several approaches to subtyping which serve different purposes. Well-known forms of subtyping include: coercive subtyping, declarative subtyping, structural subtyping, and record subtyping.

1.5 Constructor Subtyping

In this thesis, we study another form of subtyping, called constructor subtyping, which formalizes the view that an inductively defined set τ is a subtype of an inductively defined set σ if σ has more constructors than τ. As such, constructor subtyping captures in a type-theoretic context the ubiquitous use of subtyping as inclusion between inductively defined sets.

In its simplest instances, constructor subtyping enforces subtyping from natural numbers to integers or from non-empty lists over σ to lists over σ. Moreover, it is adequate to work with mutually recursive datatypes such as odd and even numbers. In this context, constructor subtyping provides a solution to the problem of datatypes in typed λ-calculi with subtyping. An important application of constructor subtyping is thus typed functional programming languages, where the theoretical issues underpinning the interactions between subtyping and datatypes have prevented a long-recognized need for subtyping.

In more elaborate instances, the usefulness of constructor subtyping is largely demonstrated by its ability to express concisely and accurately a variety of formal languages. In this context, constructor subtyping provides an appropriate framework for formal verification of programs and protocols and is directly relevant to the design of type theory-based proof-development systems.

In fact, the idea of constructor subtyping can already be found in T. Coquand's paper on pattern-matching [44], where it is suggested that constructor subtyping is useful to represent rather faithfully Kahn's natural semantics [84]. The idea is also explored, from a somewhat different perspective, in the context of ABEL, a specification language developed at Oslo University [50]. This work emphasizes the expressibility of the framework and suggests a paradigm, called terminating generator induction (TGI), which provides a pattern-matching-like facility for recursive definitions.

Constructor subtyping combines subtyping between datatypes and overloading of constructors. In order to integrate constructor subtyping safely to typed λ-calculus, maintaining properties such as subject reduction and confluence of reductions in the resulting system, one needs to constrain the overloading of constructors to strict overloading [24], guaranteeing coherence between domain and codomain of overloaded constructors. Let us illustrate this with an example enforcing odd and even numbers to naturals.

```
data Even = O : Even
          | S : Odd -> Even

data Odd = S : Even -> Odd

sub Even <= Nat
sub Odd <= Nat
```

The constructor `S` is strictly overloaded, as we have `S:Nat->Nat`, `S:Even->Odd`, `S:Odd->Even`, but also `Even` $\leq$ `Nat` and `Odd` $\leq$ `Nat`. Note `Even` and `Odd` are mutually recursive datatypes.

Constructor subtyping is specially adequate for re-usability, allowing the definition of new types by restricting or by expanding the set of constructors of an already defined datatype. Moreover, it can help to avoid partial functions, as it provides some facilities in defining more accurate types.

In order for constructor subtyping to be usable in practice, one must allow users to define overloaded and extensible recursive functions. More precisely, recursive definitions must be: overloaded (i.e. to have several types) and extensible (i.e. to be scalable from a datatype to another datatype). In addition, we would prefer that, unlike in many object-oriented programming languages, the computational behavior of recursive functions does not depend on typing. One reason is that we are eventually interested in extending the mechanism to dependent types and that letting reduction depend on typing would create a circularity—in dependent type systems, typing depends on reduction through the conversion rule.

Let us now show how constructor subtyping and overloaded definitions may be used to enhance expressiveness and re-usability of a functional language. For instance, we can define the type of positive naturals `NatP` by restricting `Nat`

```
data NatP = S : Nat -> NatP

sub NatP <= Nat
```

With this declaration we have overloaded the constructor `S` and forced `NatP` to be a subtype of `Nat`. We have that `O` is of type `Nat` and `(S O)` is, simultaneously, of type `NatP` and `Nat`, by subsumption. This form of more accurate typing is very useful to avoid introducing partial functions. For example, the division operator over natural numbers can be precisely typed with type `Nat -> NatP -> Nat` as the division by zero is not defined.

```
div : Nat -> NatP -> Nat
div x y = if (less x y) then 0
                        else S (div (minus x y) y)

less : Nat -> Nat -> Bool
less x 0 = False
less (S x) (S y) = less x y
less 0 (S y) = True

minus : Nat -> Nat -> Nat
minus x 0 = x
minus (S x) (S y) = minus x y
```

An example of a datatype defined by expansion of the set of constructors is the type of integers `Int` which extends the datatype `Nat` with a constructor `Neg` that builds the negative integers from the positive naturals.

```
data Int extends Nat = Neg : NatP -> Int
```

With this declaration we force `Nat` to be a subtype of `Int`. The constructors of `Int` are `O` , `S`, and `Neg`. With the capacity of overloading constructors the datatype `Ord` of ordinal notations could also be presented as an extension of `Nat` as follows:

```
data Ord extends Nat = S : Ord -> Ord
                     | Lim : (Nat -> Ord) -> Ord
```

The coherent combination of subtyping and the mechanism of expanding and contracting datatypes has an associated form of code reuse for functions on datatypes, allowing also the definition of new functions by restricting or by expanding already defined ones. To illustrate this, consider we have the predecessor function defined over `NatP`.

```
pred : NatP -> Nat
pred (S x) = x
```

We can extend the definition of `pred` to work on integers just by adding the appropriate computational rules for the new constructors.

```
pred : Int -> Int
pred O = Neg (S O)
pred (Neg x) = Neg (S x)
```

Another typical example of an overloaded recursive function is addition. Assume we have the following declaration for the addition of natural numbers

```
add : Nat -> Nat -> Nat
add O  y = y
add (S x) y = S (add x y)
```

We can completely reuse this program to define addition for even and odd numbers by just declaring:

```
add : Even -> Even -> Even
add : Odd -> Even -> Odd
```

Note how overloading is essential here, for the second equation to be well-typed. We can also reuse the definition of addition for natural numbers by restricting the set of computational rules that can be used, giving a more specific type to the function.

```
add : NatP -> Nat -> NatP
```

Note that this type could not be achieved by subsumption. Finally, we can extend the definition of `add` to work on integers just by adding the computational rule for the negative integers.

```
add : Int -> Int -> Int
add (Neg (S x)) y = if x == O then pred y
                              else add (Neg x) (pred y)
```

The Part II of this thesis, devoted to the study of constructor subtyping systems, proposes a way of adding overloaded extensible definitions in the style illustrated above.

1.6 Summary of Contributions

This thesis is concerned with the meta-theoretical study of two typed λ-calculi with inductive types:

- the first calculus deals with a simply-typed λ-calculus where termination of recursive definitions is ensured by types. The results concerning this calculus are presented in the first part of this thesis and are based on:

  G. Barthe, M. J. Frade, E. Giménez, L. Pinto and T. Uustalu. Type-based termination of recursive definitions. In Mathematical Structures in Computer Science, volume 14, number 1, pages 97–141, 2004. Cambridge University Press.

- the second calculus deals with a simply typed λ-calculus that supports constructor subtyping and overloaded definitions. The results concerning this calculus are presented in the second part of this thesis and are based on:

  G. Barthe and M. J. Frade. Constructor subtyping. In D. Swiestra, editor, Proceedings of ESOP'99, volume 1576 of Lecture Notes in Computer Science, pages 109–127. Springer-Verlag 1999.

  However, the calculus presented in this thesis is different from the one in that paper in the following respect: in the calculus studied in this work

  – we do not have records;

  – we do not have any kind of type annotations on terms: the calculus is à la Curry;

  – the calculus features general recursion and we consider the problem of achieving termination;

  – we aggregate a theory of definitions which is flexible enough to support overloaded definitions and that it is extensional.

Below we briefly summarize our main contributions with respect to these two calculi.

Part I In this part, we have devised a mechanism that ensures termination by typing and define a system that incorporates it. We introduce $\lambda^{\frown}$, a simply typed λ-calculus à la Curry, supporting parametric inductive datatypes, case-expressions and letrec-expressions with termination ensured by types. We show that $\lambda^{\frown}$ enjoys important meta-theoretical properties, including confluence, subject reduction and strong normalization. We also show that the calculus is powerful enough to encode many recursive definitions rejected by existing type systems, and give some examples. We prove that this system encompasses in a strict way Giménez' $\lambda_{\mathcal{G}}$ [67], a system in which termination of typable expressions is ensured by a syntactical condition constraining the uses of recursive calls in the body of definitions.

Part II In this part, we introduce and study the properties of a type system featuring constructor subtyping. We define the system λ_{CS}, a simply typed λ-calculus, à la Curry, supporting mutually recursive parametric datatypes, constructor subtyping, case-expressions and letrec-expressions. We establish the properties of confluence, subject reduction and decidability of type-checking for this calculus. As the system features general recursion, the reduction calculus is obviously non-terminating. However, we sketch two ways of achieving strong normalization. One way is constraining the system to guard-by-destructors recursion, following what is done for $\lambda_{\mathcal{G}}$. The other way is enriching the type system with stages (following the ideas presented for $\lambda^{\frown}$) and enforcing termination through typing.

We enrich a calculus featuring constructor subtyping with a mechanism to define extensible overloaded functions. We define the system $\lambda_{\mathsf{CS+fun}}$ a simply typed λ-calculus with mutually recursive parametric datatypes, constructor subtyping and extensible overloaded recursive functions defined by pattern-matching. We formalize the concept of well-formed environment of function declarations. We establish the properties of confluence, subject reduction and decidability of type-checking for this calculus. Moreover, we prove that the requirements imposed for the well-formed environments are decidable properties. In what concerns termination, we just provide a simple criterion inspired from [44]. Furthermore, we conjecture that the standard compilation of pattern-matching into case-expressions extends to our setting. We define $\lambda_{\mathsf{CS+def}}$ as a mild variation of $\lambda_{\mathsf{CS+fun}}$: recursive functions defined by pattern-matching are replaced by case-expressions and recursive function definitions. We establish the properties of confluence, subject reduction and decidability of type-checking for $\lambda_{\mathsf{CS+def}}$ and we describe the translation from $\lambda_{\mathsf{CS+fun}}$ to $\lambda_{\mathsf{CS+def}}$.

Part I

Type-Based Termination of Recursive Definitions

Chapter 2

An Informal Account of Type-Based Termination of Recursive Definitions

2.1 Background

Most functional programming languages (ML, Haskell, etc) and proof development systems based on the proofs-as-programs paradigm of logic (Coq, HOL, PVS, etc) rely on powerful type theories featuring inductive types such as natural numbers or lists. Those languages come equipped with a mechanism for recursive definition of functions. However, there are significant differences between the mechanisms used in functional programming languages and in proof development systems.

The first difference concerns the termination of recursive functions. While in functional programming languages recursive functions are allowed to diverge, in proof development systems non-terminating functions must be banished from the language, as they almost always lead to logical paradoxes.

The second difference concerns how recursive definitions are introduced. In functional programming languages, recursive functions are described in terms of a pattern-matching operator (`case`) and a general fixpoint operator (`let-rec`). For example, the addition of two natural numbers could be introduced as follows:

```
let rec plus n m = case m of
                     O        -> n
                   | (S p)    -> (S (plus n p))
                   end
```

On the other hand, in the traditional presentations of type-based proof development systems [46, 54, 90, 112], a recursive function $f : d \to \theta$ on an inductive type d is defined by means of the elimination rule of d, where both pattern matching and recursion are built into a single scheme which ensures termination. In this approach, the function `plus` can be encoded using the elimination rule of natural numbers $\mathsf{nat\_elim} : \theta \to (\mathsf{Nat} \to \theta \to \theta) \to \mathsf{Nat} \to \theta$, which corresponds to the primitive recursion scheme:

```
let plus n m = nat_elim n (fun p r -> (S r)) m
```

This approach is theoretically sound. However practice has shown that eliminators are rather cumbersome to use, whereas case-expressions and fixpoint expressions lead to more concise and readable definitions. Looking for a good compromise between termination and presentation issues, [44] suggested that recursors should be replaced by case-expressions and a restricted form of fixpoint expressions, see also [67]. The restriction is imposed through a predicate $\mathcal{G}_f$ on untyped terms. This predicate enforces termination by constraining all recursive calls to be applied to terms smaller than the formal argument x of f—for instance, a pattern variable issued from a case expression on x. The restricted typing rule for fixpoint expressions hence becomes:

$$\frac{f : \mathsf{Nat} \to \theta \ \vdash \ e : \mathsf{Nat} \to \theta}{\vdash \ (\mathsf{letrec}\ f = e) : \mathsf{Nat} \to \theta} \qquad \text{if } \mathcal{G}_f(e) \qquad\qquad (*)$$

This alternative approach, called *guarded-by-destructors* recursion in [67], has been implemented in the Coq system. Several years of experiments carried out with Coq have shown that it actually provides much more palatable representations of recursive functions.

However, the use of an external predicate $\mathcal{G}$ on untyped terms suffers from several weaknesses:

1. *The guard predicate is too syntax-sensitive and too weak.*
   The acceptance of a recursive definition becomes too sensitive to the syntactical shape of its body. Sometimes, a small change in the definition could make it to no longer satisfy the guardedness condition. As an example, consider the following modification of the `plus` function, where the condition is no longer satisfied because of the introduction of a redex in the definition:

   ```
   let comp f g x   = (f (g x))
   let rec plus n m = case m of
                        O       -> n
                      | (S p)   -> (comp S (plus n) p)
                      end
   ```

   In addition, the guard predicate rejects many terminating recursive definitions such as the Euclidean division, Ackermann's function, or functions that swap arguments, such as subtyping algorithms for higher-order languages

   ```
   let rec sub a a' =
            case a a' of
              (base b) (base b')         -> sub_base b b'
            | (fun b1 b2) (fun b'1 b'2)  -> (sub b'1 b1) && (sub b2 b'2)
            | ...                        -> ...
            end
   ```

2. *The guard predicate is hard to implement and hard to extend.*
   The guardedness condition is among the main sources of bugs in the implementation of the proof system. In order to improve the number of definitions accepted by the system, the guardedness condition has become more and more complicated hence prone to errors.

   Besides, it is easier to extend the type system than to extend the guardedness condition: type conditions are expressed as local constraints associated to each construction of the language whereas the guard predicate yields global constraints.

3. *The guard predicate is often defined on normal forms.*
   Often the guard predicate is defined on normal forms only, which renders the typing rule $(*)$ useless in practice. Subsequently, the typing rule $(*)$ is usually replaced by the more liberal typing rule

$$\frac{f : \mathsf{Nat} \to \theta \ \vdash \ e : \mathsf{Nat} \to \theta}{\vdash \ (\mathsf{letrec}\ f = e) : \mathsf{Nat} \to \theta} \qquad \text{if } \mathcal{G}_f(\mathsf{nf}\ e)$$

   where `nf` is the partial function associating to an expression its normal form. Now the modified rule introduces two further complications:

   (a) *The new guard condition leads to inefficient type-checking.*
   Verifying the guardedness condition makes type-checking less efficient as the body of a recursive definition has to be reduced to be checked—expanding previously defined constants like the constant `comp` in the example above.

   (b) *The new guard condition destroys strong normalization.*
   For example, the normal form of the following definition satisfies the guardedness condition, but not the definition itself:

```
let K x y = x
let rec diverging_id n = case n of
                           O       -> K n (diverging_id n)
                         | (S p)   -> S (diverging_id p)
                         end
```

   There is an infinite reduction sequence for the term `diverging_id O`:[1]

   `diverging_id O` $\to$ `(K O (diverging_id O))` $\to$ `(K O (K O (diverging_id O)))` $\to \ldots$

   One solution to this problem (the solution has been considered for Coq) is to store recursive definitions with their bodies in normal forms, as enforced by the rule

$$\frac{f : \mathsf{Nat} \to \theta \ \vdash \ e : \mathsf{Nat} \to \theta}{\vdash \ (\mathsf{letrec}\ f = (\mathsf{nf}\ e)) : \mathsf{Nat} \to \theta} \qquad \text{if } \mathcal{G}_f(\mathsf{nf}\ e)$$

   but the rule has severe drawbacks: (1) proof terms become huge (because terms are usually much smaller than their normal forms, in particular since all definitions have been unfolded in the latter); (2) the expressions being stored are not those constructed interactively by the user; (3) the modified typing rule for fixpoint expressions is not syntax-directed, i.e. one cannot guess the expression e appearing in the premise from the conclusion of the rule.

In order to circumvent those weaknesses, some authors have proposed semantically motivated type systems that ensure the termination of recursive definitions through typing [66, 8, 17]. The idea, which already occurs in Mendler's work [106], consists in regarding an inductive type d as the least fixpoint of a monotonic operator $\widehat{-}^d$ on types, and to enforce termination of recursive functions by requiring that the definition of $f : \widehat{\alpha}^d \to \theta$, where α may be thought as a subtype of d, only relies on structurally smaller function calls, embodied by a function $f_{\mathrm{ih}} : \alpha \to \theta$. This approach to terminating recursion, which we call *type-based*, offers several advantages over the *guarded-by-destructors* approach. In particular, it addresses all the above-mentioned weaknesses.

[1] In fact, Coq 7.2 accepts this definition of `diverging_id`!

2.2 Outline of Type-Based Termination

In this work we introduce $\lambda^{\widehat{}}$, a simply typed λ-calculus that supports type-based recursive definitions. Although heavily inspired by previous work by Giménez [66] and closely related to recent work by Amadio and Coupet [8], the technical machinery behind our system puts a slightly different emphasis on the interpretation of types. More precisely, we formalize the notion of type-based termination using a restricted form of type dependency (a.k.a. indexed types), as popularized by [141, 142]. This leads to a simple and intuitive system which is robust under several extensions, such as mutually inductive datatypes and mutually recursive function definitions; however, such extensions are not treated here.

The basic idea is to proceed as follows:

- First, every datatype d is replaced by a family of approximations indexed over a set of *stages*, which are used to record a bound on the "depth" of values. Here, we adopt a simple minded approach and let stages range over the syntax

  $$s ::= \imath \mid \widehat{s} \mid \infty$$

  where $\imath$ ranges over stage variables, the hat operator $\widehat{\_}$ is a function mapping a stage to its "successor" and ∞ is the stage at which the iterative approximation process converges to the datatype itself.

- Second, a recursive definition of a function, say $f : d \to \theta$ should be given by a term e constructing a function $g' : d^{\widehat{\imath}} \to \theta$ from $g : d^{\imath} \to \theta$, where $\imath$ ranges over stages (in other words, e should be stage-polymorphic).

In order to illustrate the machinery involved, let us consider the inductive type Nat whose constructors are $\mathsf{o} : \mathsf{Nat}$ and $\mathsf{s} : \mathsf{Nat} \to \mathsf{Nat}$. The typing rules are

$$\frac{}{\vdash\ \mathsf{o} : \mathsf{Nat}^{\widehat{s}}} \qquad\qquad \frac{\vdash\ n : \mathsf{Nat}^{s}}{\vdash\ \mathsf{s}\ n : \mathsf{Nat}^{\widehat{s}}}$$

and, as an instance of the subsumption rule,

$$\frac{\vdash\ n : \mathsf{Nat}^{s}}{\vdash\ n : \mathsf{Nat}^{\widehat{s}}}$$

Finally recursive functions from Nat to θ are constructed with the following typing rule:

$$\frac{f : \mathsf{Nat}^{\imath} \to \theta\ \vdash\ e : \mathsf{Nat}^{\widehat{\imath}} \to \theta}{\vdash\ (\mathsf{letrec}\ f = e) : \mathsf{Nat} \to \theta}$$

where $\imath$ is fresh wrt. θ. As shall be shown later, such recursive functions are terminating and, despite its simplicity, this mechanism is powerful enough to capture course-of-value primitive recursion.

2.3 Overview of This Part

The remainder of this part is organized as follows.

In Chapter 3 we formally present the system $\lambda^{\widehat{}}$, a simply typed λ-calculus supporting inductive types and recursive function definitions with termination ensured by types. We also show that the calculus is powerful enough to encode many recursive definitions rejected by existing type systems, and give some examples.

In Chapter 4 we show that $\lambda^{\frown}$ is well-behaved enjoying important meta-theoretic properties. In particular the reduction calculus is confluent, and enjoys subject reduction and strong normalization.

In Chapter 5 we introduce $\lambda_{\mathcal{G}}$, a system in which termination of typable recursively defined functions is ensured by a syntactical condition $\mathcal{G}$ constraining the uses of recursive calls in the body of definitions, and prove that $\lambda^{\frown}$ strictly extends the system $\lambda_{\mathcal{G}}$.

In Chapter 6 we review related work and conclude.

Chapter 3

The System $\lambda^{\frown}$

In this Chapter, we introduce $\lambda^{\frown}$, a simply typed lambda calculus featuring strongly positive, finitely iterated parametric inductive types (in the sense of, e.g., [96]) and type-based termination of recursive definitions. The calculus is *à la* Curry: terms come without any type annotations.

3.1 Datatypes

Datatypes and constructors are named: we assume given two finite sets $\mathcal{D}$ of *datatype identifiers* and $\mathcal{C}$ of *constructor identifiers*. On datatypes, we assume a stratification that ensures that the dependency relation between datatypes is well-founded. Hence each datatype d is assigned a stratum $\mathsf{str}(d) \in \mathbb{N}$. Datatypes and constructors may only accept a fixed number of arguments, so we stipulate that every datatype identifier d (resp. constructor c) has a fixed *arity* $\mathsf{ar}(d) \in \mathbb{N}$ (resp. $\mathsf{ar}(c) \in \mathbb{N}$) that indicates the number of parameters taken by d (resp. c). Finally, we require that every datatype $d \in \mathcal{D}$ comes equipped with a set $\mathsf{C}(d) \subseteq \mathcal{C}$ of constructors, and if $d \neq d'$ then $\mathsf{C}(d) \cap \mathsf{C}(d') = \emptyset$.

For the sake of clarity, we adopt the following naming conventions: $d, d', d_i, \ldots$ range over $\mathcal{D}$ and $c, c', c_i, \ldots$ range over $\mathcal{C}$.

3.2 Terms and Reduction

3.2.1 Terms

Terms are built from variables, abstractions, applications, constructors, case-expressions and recursive definitions. Assume we have a denumerable set $\mathcal{V}_{\mathcal{E}}$ of *(object) variables*, and let $x, x', x_i, y, \ldots$ range over $\mathcal{V}_{\mathcal{E}}$.

Notation 3.2.1 *For every set A, we let A^* denote the set of lists over A, and $[]$ denote the empty list. $\vec{a}$ ranges over A^* if a ranges over A. $\#\vec{a}$ denotes the length of $\vec{a}$, and $\vec{a}[i]$ denotes, when it exists, the ith element of $\vec{a}$. For convenience, we will sometimes write lists in the form $[a_1, \ldots, a_n]$ instead of $a_1 \ldots a_n$.*

Definition 3.2.2 (Terms) *The set $\mathcal{E}$ of* terms *is given by the abstract syntax*

$$\mathcal{E} \ni a, b ::= x \mid \lambda x.\, a \mid a\, b \mid c \mid \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots\ \mid c_n \Rightarrow b_n\} \mid \mathsf{letrec}\ x = e$$

where in the clause for case-expressions it is assumed that $\{c_1, \ldots, c_n\} = \mathsf{C}(d)$ for some $d \in \mathcal{D}$.

Notation 3.2.3 *Sometimes we write* $\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ *as an abbreviation of* $\mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_n \Rightarrow b_n\}$ *and* $c\,\vec{a}$ *as an abbreviation of* $c\,a_1 \ldots a_{\mathsf{ar}(c)}$

Definition 3.2.4 *The set of* free variables *of an expression* e*, denoted by* $\mathsf{FV}(e)$*, is defined by induction on the structure of* e *as follows:*

$$\begin{array}{rcl}
\mathsf{FV}(x) & = & \{x\} \\
\mathsf{FV}(c) & = & \emptyset \\
\mathsf{FV}(\lambda x.a) & = & \mathsf{FV}(a)\backslash\{x\} \\
\mathsf{FV}(a\ b) & = & \mathsf{FV}(a) \cup \mathsf{FV}(b) \\
\mathsf{FV}(\mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1\ |\ \ldots\ |\ c_n \Rightarrow b_n\}) & = & \mathsf{FV}(a) \cup \mathsf{FV}(b_1) \cup \ldots \cup \mathsf{FV}(b_n) \\
\mathsf{FV}(\mathsf{letrec}\ x = e) & = & \mathsf{FV}(e)\backslash\{x\}
\end{array}$$

A variable x *is said to* occur free *or to* be free in e *if* $x \in \mathsf{FV}(e)$*. A variable in* e *that is not free in* e *is said to* be bound *or to* occur bound in e*. An expression with no free variables is said to be* closed.

The usual conventions for omitting parentheses are adopted: application is left associative and the scope of λ extends to the right as far as possible. We identify terms that are equal up to a renaming of bound variables (or α-conversion). Moreover we assume standard variable convention [14], so, all bound variables are chosen to be different from free variables.

Definition 3.2.5 (Term substitution) *A* term substitution *is a function from* $\mathcal{V}_{\mathcal{E}}$ *to* $\mathcal{E}$*. We write* $[x_1 := e_1, \ldots, x_n := e_n]$ *(or briefly* $[\vec{x} := \vec{e}]$*) for the substitution mapping* x_i *to* e_i *for* $1 \leq i \leq n$*, and mapping every other variable to itself.*

Given a term $a \in \mathcal{E}$ *and a type substitution* $S = [\vec{x} := \vec{e}]$*, we write* $S(a)$ *or* $a[\vec{x} := \vec{e}]$ *to denote the term obtained by the simultaneous substitution of terms* e_i *for the free occurrences of variables* x_i *in* a*.*

Remark 3.2.6 *In the application of a substitution to a term, we rely on a variable convention. The action of a substitution over a term is defined, as usual, with possible changes of bound variables.*

We now present a result which allows us to reorder substitutions.

Lemma 3.2.7 *If* $x \neq y$ *and* $x \notin \mathsf{FV}(b)$ *then*

$$e[x := a][y := b] = e[y := b][x := a[y := b]]$$

Proof. By induction on the structure of e. □

3.2.2 Reduction calculus

The reduction calculus is given by β-reduction for function application, ι-reduction for case analysis and μ-reduction for unfolding recursive definitions—only allowed in the context of application to a constructor application.

Definition 3.2.8 (Reductions)

1. *β-reduction $\rightarrow_\beta$ is defined as the compatible closure of the rule*

$$(\lambda x.\ e)\ e' \ \rightarrow_\beta\ e[x := e']$$

2. *ι-reduction $\rightarrow_\iota$ is defined as the compatible closure of the rule*

$$\mathsf{case}\ (c_i\ \vec{a})\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}\ \rightarrow_\iota\ e_i\ \vec{a}$$

where $\#\vec{a} = \mathsf{ar}(c_i)$.

3. *μ-reduction $\rightarrow_\mu$ is defined as the compatible closure of the rule*

$$(\mathsf{letrec}\ f = e)\ (c\ \vec{a})\ \rightarrow_\mu\ e[f := (\mathsf{letrec}\ f = e)]\ (c\,\vec{a})$$

where $\#\vec{a} = \mathsf{ar}(c)$.

4. *The terms of the forms* $(\lambda x.a)\,b$, $\mathsf{case}\ (c_i\,\vec{a})\ \mathsf{of}\,\{\vec{c} \Rightarrow \vec{b}\}$ *and* $(\mathsf{letrec}\ f = e)\ (c\,\vec{a})$ *are called* β-redexes, ι-redexes *and* μ-redexes, *with* $a[x := b]$, $b_i\,\vec{a}$ *and* $e[f := (\mathsf{letrec}\ f = e)]\ (c\,\vec{a})$ *being their* contracta, *respectively. As expected we call* $\beta\iota\mu$-redex *to a term that is either a β-redex, an ι-redex or a μ-redex .*

5. *$\beta\iota\mu$-reduction $\rightarrow_{\beta\iota\mu}$ is defined as $\rightarrow_\beta \cup \rightarrow_\iota \cup \rightarrow_\mu$. $\twoheadrightarrow_{\beta\iota\mu}$ and $=_{\beta\iota\mu}$ are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of $\rightarrow_{\beta\iota\mu}$. We say that e* reduces to *e' (or e* computes into *e') whenever $e \twoheadrightarrow_{\beta\iota\mu} e'$. One defines similarly the relations $\twoheadrightarrow_\beta$, $\twoheadrightarrow_\iota$, $\twoheadrightarrow_{\beta\iota}$, $=_\beta$, $=_\iota$ and $=_{\beta\iota}$.*

Remark 3.2.9 *In the formulation of the β- and μ-reduction rules, we rely on a variable convention: in the β-rule, the bound variables of e are assumed to be different from the free variables of e'; in the μ-rule, the bound and the free variables of e are assumed to be different.*

The mechanics of the reduction calculus is illustrated by the following example.

Example 3.2.10 *Consider the inductive type of natural numbers* Nat *with* $\mathsf{C}(\mathsf{Nat}) = \{\mathsf{o}, \mathsf{s}\}$, $\mathsf{a}(\mathsf{o}) = 0$ *and* $\mathsf{a}(\mathsf{s}) = 1$. *Let* $\mathrm{plus} \equiv (\mathsf{letrec}\ \mathit{plus} = \lambda x.\ \lambda y.\ \mathsf{case}\ x\ \mathsf{of}\,\{\mathsf{o}\ \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda x'.\ \mathsf{s}\ (\mathit{plus}\ x'\ y)\})$. *The following is a reduction sequence that computes one plus two.*

$$\begin{array}{ll}
& \mathrm{plus}\ (\mathsf{s}\ \mathsf{o})\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})) \\
\rightarrow_\mu & (\lambda x.\ \lambda y.\ \mathsf{case}\ x\ \mathsf{of}\,\{\mathsf{o}\ \Rightarrow\ y \mid \mathsf{s}\ \Rightarrow\ \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ y)\})\ (\mathsf{s}\ \mathsf{o})\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})) \\
\twoheadrightarrow_\beta & \mathsf{case}\ \mathsf{s}\ \mathsf{o}\ \mathsf{of}\,\{\mathsf{o}\ \Rightarrow\ \mathsf{s}\ (\mathsf{s}\ \mathsf{o}) \mid \mathsf{s}\ \Rightarrow\ \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})))\} \\
\rightarrow_\iota & (\lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))))\ \mathsf{o} \\
\rightarrow_\beta & \mathsf{s}\ (\mathrm{plus}\ \mathsf{o}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))) \\
\rightarrow_\mu & \mathsf{s}\ ((\lambda x.\ \lambda y.\ \mathsf{case}\ x\ \mathsf{of}\,\{\mathsf{o}\ \Rightarrow\ y \mid \mathsf{s}\ \Rightarrow\ \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ y)\})\ \mathsf{o}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))) \\
\twoheadrightarrow_\beta & \mathsf{s}\ (\mathsf{case}\ \mathsf{o}\ \mathsf{of}\,\{\mathsf{o}\ \Rightarrow\ \mathsf{s}\ (\mathsf{s}\ \mathsf{o}) \mid \mathsf{s}\ \Rightarrow\ \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})))\}) \\
\rightarrow_\iota & \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))
\end{array}$$

Term substitution has some useful properties with respect to reducibility.

Lemma 3.2.11 (Substitution lemma for reductions)

1. $e \rightarrow_{\beta\iota\mu} e' \quad\Rightarrow\quad e[x := a] \rightarrow_{\beta\iota\mu} e'[x := a]$

2. $a \rightarrow_{\beta\iota\mu} a' \quad\Rightarrow\quad e[x := a] \twoheadrightarrow_{\beta\iota\mu} e[x := a']$

3. $e \rightarrow_{\beta\iota\mu} e' \ \wedge\ a \rightarrow_{\beta\iota\mu} a' \quad\Rightarrow\quad e[x := a] \twoheadrightarrow_{\beta\iota\mu} e'[x := a']$

Proof.

1. By induction on the structure of e. Assume $e \rightarrow_{\beta\iota\mu} e'$. e can neither be a variable nor a constructor.

   - Case $e \equiv e_1\, e_2$. There are four possibilities:

     – $e_1 \rightarrow_{\beta\iota\mu} e_1'$ and $e' \equiv e_1'\, e_2$. In this case we have $e_1[x := a] \rightarrow_{\beta\iota\mu} e_1'[x := a]$ by induction hypothesis. So, $(e_1\, e_2)[x := a] = e_1[x := a]\, e_2[x := a] \rightarrow_{\beta\iota\mu} e_1'[x := a]e_2[x := a] = (e_1'\, e_2)[x := a]$.

     – $e_2 \rightarrow_{\beta\iota\mu} e_2'$ and $e' \equiv e_1\, e_2'$. Similar.

     – $e_1 \equiv (\lambda y.b)$ with $x \not\equiv y$ and $y \notin \mathsf{FV}(a)$, and $e' \equiv b[y := e_2]$. In this case we have $e[x := a] = (\lambda y.b[x := a])\,(e_2[x := a]) \rightarrow_{\beta\iota\mu} b[x := a][y := e_2[x := a]]$. Finally by Lemma 3.2.7 we can conclude that $b[x := a][y := e_2[x := a]] = b[y := e_2][x := a] = e'[x := a]$.

     – $e_1 \equiv (\mathsf{letrec}\ y = b)$, $e_2 \equiv (c\,\vec{a'})$ with $x \not\equiv y$ and $y \notin \mathsf{FV}(a)$, and $e' \equiv b[y := (\mathsf{letrec}\ y = b)]\,(c\,\vec{a'})$. In this case we have $e[x := a] = (\mathsf{letrec}\ y = b[x := a])\,(c\,\overrightarrow{a'[x := a]}) \rightarrow_{\beta\iota\mu} b[x := a][y := (\mathsf{letrec}\ y = b[x := a])]\,(c\,\overrightarrow{a'[x := a]})$. Finally by Lemma 3.2.7 we can conclude that $b[x := a][y := (\mathsf{letrec}\ y = b[x := a])]\,(c\,\overrightarrow{a'[x := a]}) = b[y := (\mathsf{letrec}\ y = b)][x := a]\,(c\,\overrightarrow{a'[x := a]}) = e'[x := a]$.

   - Case $e \equiv (\lambda y.b)$ with $x \not\equiv y$ and $y \notin \mathsf{FV}(a)$. The only case here is $b \rightarrow_{\beta\iota\mu} b'$ so, the result follows by induction hypothesis.

   - Case $e \equiv \mathsf{case}\ b\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{e}\}$. The proof of this case is very similar to the case where e is an application.

2. By induction on the structure of e.

3. Directly from properties 1 and 2.

□

Definition 3.2.12 (Strongly normalizing terms) *Let $a \in \mathcal{E}$.*

1. *The term a is in* normal form *if it does not contain any $\beta\iota\mu$-redex, i.e., if there is no term b such that $a \rightarrow_{\beta\iota\mu} b$.*

2. *The term a* strongly normalizes *if there is no infinite $\beta\iota\mu$-reduction sequence starting with a.*

3. *The set SN of strongly normalizing terms is inductively defined by the following clause:*

   If $b \in \mathsf{SN}$ for all term b such that $a \rightarrow_{\beta\iota\mu} b$, then $a \in \mathsf{SN}$.

It follows from the above definition that the set SN is not empty, since $\mathcal{V}_{\mathcal{E}} \subseteq \mathsf{SN}$; and that if e is strongly normalizing and $e \twoheadrightarrow_{\beta\iota\mu} e'$, then e' is also strongly normalizing. Moreover, observe that any subterm of a strongly normalizing term is also strongly normalizing, since the $\beta\iota\mu$-reduction relation is compatible with respect to the formation of terms.

3.3 Types, Subtyping and Typing

3.3.1 Types and Subtyping

Assume now given two denumerable sets $\mathcal{V}_\mathcal{T}$ of *type variables* and $\mathcal{V}_\mathcal{S}$ of *stage variables*. Adopt the naming conventions that $\alpha, \alpha', \alpha_i, \beta, \delta, \ldots$ range over $\mathcal{V}_\mathcal{T}$ and $\imath, \jmath, \ldots$ range over $\mathcal{V}_\mathcal{S}$. Proceeding from these, we define stage and type expressions. Stage expressions are built of stage variables, a symbol for the successor function on stages, and a symbol for the limit stage. A type expression is either a type variable, a function type expression or a datatype approximation expression.

Definition 3.3.1 (Stages and types)

1. *The set $\mathcal{S}$ of* stage expressions *is given by the abstract syntax:*

$$\mathcal{S} \ni s, r ::= \imath \mid \widehat{s} \mid \infty$$

2. *The set $\mathcal{T}$ of* type expressions *is given by the abstract syntax:*

$$\mathcal{T} \ni \sigma, \tau ::= \alpha \mid \tau \rightarrow \sigma \mid d^s\, \vec{\tau}$$

where in the last clause, the length of $\vec{\tau}$ is exactly $\mathsf{ar}(d)$.

The usual conventions of omitting parentheses are adopted: the type constructor $\rightarrow$ is right associative.

Notation 3.3.2 *Very often we write $\vec{\tau} \rightarrow \sigma$ as an abbreviation for $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \sigma$, and $d\,\vec{\sigma}$ as an abbreviation for $d^\infty\, \vec{\sigma}$.*

Every datatype is seen as a family of approximations indexed over a set of stages. The hat operator maps stage to its successor and ∞ is the stage at which the iterative approximation process converges to the datatype itself. So, datatypes are equipped with size information given by the stages which are used to record a bound on the "depth" of values. Stages play a central role in the definiton of restrictive typing rules in which only terminating functions are typable. The rougth idea is that a recursive function is accepted as terminating if the sizes of arguments to recursive calls are bounded by the size of the function input.

Definition 3.3.3 (Type and stage substitutions)

1. *A* type substitution *is a function from $\mathcal{V}_\mathcal{T}$ to $\mathcal{T}$. We write $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$ (or briefly $[\vec{\alpha} := \vec{\sigma}]$) for the substitution mapping α_i to σ_i for $1 \leq i \leq n$, and mapping every other type variable to itself. We write $\tau[\vec{\alpha} := \vec{\sigma}]$ to denote the type obtained by replacing simultaneously each variable α_i in τ with σ_i.*

2. *A* stage substitution *is a function from $\mathcal{V}_\mathcal{S}$ to $\mathcal{S}$. We write $[\imath_1 := s_1, \ldots, \imath_n := s_n]$ (or briefly $[\vec{\imath} := \vec{s}]$) for the substitution mapping $\imath_i$ to s_i for $1 \leq i \leq n$, and mapping every other stage variable to itself. We write $\tau[\vec{\imath} := \vec{s}]$ to denote the type obtained by replacing simultaneously each stage variable $\imath_i$ in τ with s_i.*

In order to present the typing rules for constructors and case-expressions, we have to have a means to fixing the intended typings of the constructors. To this end, we introduce the notions of constructor scheme, constructor declaration and constructor scheme instantiation.

Definition 3.3.4 (Constructor scheme) *The set $\mathcal{CS}$ of* constructor schemes *is given by the abstract syntax:*

$$\varsigma ::= \forall\,\vec{\alpha}.\forall\,\vec{\imath}.\ \sigma$$

where $\vec{\alpha}$ are the free type variables of σ and $\vec{\imath}$ are the free stage variables of σ.

Definition 3.3.5 (Constructor declaration) *There is a map $\mathsf{D} : \mathcal{C} \rightarrow \mathcal{CS}$ such that, for every $d \in \mathcal{D}$ and $c \in \mathsf{C}(d)$,*

$$\mathsf{D}(c) = \forall\,\vec{\alpha}.\forall\,\imath.\ \vec{\sigma} \rightarrow d^{\widehat{\imath}}\vec{\alpha}$$

where:

1. $\#\vec{\alpha} = \mathsf{ar}(d)$ *and* $\#\vec{\sigma} = \mathsf{ar}(c)$;
2. *every occurrence of d in σ_i is of the form $d^{\imath}\vec{\alpha}$;*
3. *every occurrence of $\imath$ in σ_i is of the form $d^{\imath}\vec{\alpha}$;*
4. *each σ_i is positive w.r.t. $d^{\imath}\vec{\alpha}$, i.e. $d^{\imath}\vec{\alpha}$ pos σ_i , see Figure 3.1;*
5. *each σ_i is positive w.r.t. α_j, i.e. α_j pos σ_i see Figure 3.1;*
6. *any $d' \neq d \in \mathcal{D}$ appearing in $\vec{\sigma}$ satisfies* $\mathsf{str}(d') < \mathsf{str}(d)$.

(pos0) $$\frac{}{\theta\ \mathsf{pos}\ \theta}$$

(pos1) $$\frac{\theta \neq \alpha}{\theta\ \mathsf{pos}\ \alpha}$$ (neg1) $$\frac{\theta \neq \alpha}{\theta\ \mathsf{neg}\ \alpha}$$

(pos2) $$\frac{\theta\ \mathsf{neg}\ \tau \quad \theta\ \mathsf{pos}\ \sigma}{\theta\ \mathsf{pos}\ \tau \rightarrow \sigma}$$ (neg2) $$\frac{\theta\ \mathsf{pos}\ \tau \quad \theta\ \mathsf{neg}\ \sigma}{\theta\ \mathsf{neg}\ \tau \rightarrow \sigma}$$

(pos3) $$\frac{\theta\ \mathsf{pos}\ \tau_i \quad (1 \leq i \leq \mathsf{ar}(d))}{\theta\ \mathsf{pos}\ d^s\vec{\tau}}$$ (neg3) $$\frac{\theta\ \mathsf{neg}\ \tau_i \quad (1 \leq i \leq \mathsf{ar}(d))}{\theta\ \mathsf{neg}\ d^s\vec{\tau}}$$

where θ ranges over type variables and datatypes.

Figure 3.1: Positive-negative occurrences of type variables or datatypes

The last condition ensures that only finitely iterated inductive definitions are permitted (excluding mutual induction) and is made use of in the model construction (Definition 4.3.16) in the proof of strong normalization.

A constructor declaration $\mathsf{D}(c) = \forall\,\vec{\alpha}.\forall\,\imath.\ \vec{\sigma} \rightarrow d^{\widehat{\imath}}\vec{\alpha}$ specifies the possible typings for constructor $c \in \mathsf{C}(d)$: $\vec{\alpha}$ are the parameters of the datatype, $\imath$ the stage variable for the datatype d and $\vec{\sigma}$ is a possible typing for the arguments of the constructor. Note that, because of conditions 2 and 3, $\mathsf{D}(c)$ defines c as a way of transforming values in a given approximation of a datatype, $d^{\imath}\vec{\alpha}$, in values of the next approximation, $d^{\widehat{\imath}}\vec{\alpha}$.

Observe that type parameters may appear only positively in the argument types of the constructors. This makes it possible to parameterize the type of lists with respect to the type of elements, binary trees with respect to the type of node labels, arbitrarily branching trees with respect to the type of node labels, but not with respect to the branching type.

Example 3.3.6 *Consider* $\mathsf{Bool}, \mathsf{Nat}, \mathsf{List}, \mathsf{Tree}, \mathsf{Ord}, \mathsf{Maybe}, \mathsf{BTree}, \mathsf{DTree} \in \mathcal{D}$, *with* $\mathsf{ar}(\mathsf{Bool}) = \mathsf{ar}(\mathsf{Nat}) = \mathsf{ar}(\mathsf{Ord}) = 0$ *and* $\mathsf{ar}(\mathsf{List}) = \mathsf{ar}(\mathsf{Tree}) = \mathsf{ar}(\mathsf{Maybe}) = \mathsf{ar}(\mathsf{BTree}) = \mathsf{ar}(\mathsf{DTree}) = 1$. *We have*

$$\mathsf{C}(\mathsf{Bool}) = \{\mathsf{true}, \mathsf{false}\} \qquad \begin{array}{lcl} \mathsf{D}(\mathsf{true}) & = & \forall\,\imath.\ \mathsf{Bool}^{\widehat{\imath}} \\ \mathsf{D}(\mathsf{false}) & = & \forall\,\imath.\ \mathsf{Bool}^{\widehat{\imath}} \end{array}$$

for the datatype of booleans;

$$\mathsf{C}(\mathsf{Nat}) = \{\mathsf{o}, \mathsf{s}\} \qquad \begin{array}{lcl} \mathsf{D}(\mathsf{o}) & = & \forall\,\imath.\ \mathsf{Nat}^{\widehat{\imath}} \\ \mathsf{D}(\mathsf{s}) & = & \forall\,\imath.\ \mathsf{Nat}^{\imath} \rightarrow \mathsf{Nat}^{\widehat{\imath}} \end{array}$$

for the datatype of natural numbers;

$$\mathsf{C}(\mathsf{List}) = \{\mathsf{nil}, \mathsf{cons}\} \qquad \begin{array}{lcl} \mathsf{D}(\mathsf{nil}) & = & \forall\,\alpha.\forall\,\imath.\ \mathsf{List}^{\widehat{\imath}}\alpha \\ \mathsf{D}(\mathsf{cons}) & = & \forall\,\alpha.\forall\,\imath.\ \alpha \rightarrow \mathsf{List}^{\imath}\alpha \rightarrow \mathsf{List}^{\widehat{\imath}}\alpha \end{array}$$

for lists;

$$\mathsf{C}(\mathsf{Tree}) = \{\mathsf{branch}\} \qquad \mathsf{D}(\mathsf{branch}) \ = \ \forall\,\alpha.\forall\,\imath.\ \alpha \rightarrow \mathsf{List}(\mathsf{Tree}^{\imath}\alpha) \rightarrow \mathsf{Tree}^{\widehat{\imath}}\alpha$$

for finitely branching trees;

$$\mathsf{C}(\mathsf{Ord}) = \{\mathsf{zero}, \mathsf{succ}, \mathsf{lim}\} \qquad \begin{array}{lcl} \mathsf{D}(\mathsf{zero}) & = & \forall\,\imath.\ \mathsf{Ord}^{\widehat{\imath}} \\ \mathsf{D}(\mathsf{succ}) & = & \forall\,\imath.\ \mathsf{Ord}^{\imath} \rightarrow \mathsf{Ord}^{\widehat{\imath}} \\ \mathsf{D}(\mathsf{lim}) & = & \forall\,\imath.\ (\mathsf{Nat} \rightarrow \mathsf{Ord}^{\imath}) \rightarrow \mathsf{Ord}^{\widehat{\imath}} \end{array}$$

for ordinals (or better said, for ordinal notations);

$$\mathsf{C}(\mathsf{Maybe}) = \{\mathsf{nothing}, \mathsf{just}\} \qquad \begin{array}{lcl} \mathsf{D}(\mathsf{nothing}) & = & \forall\,\alpha.\forall\,\imath.\ \mathsf{Maybe}^{\widehat{\imath}}\alpha \\ \mathsf{D}(\mathsf{just}) & = & \forall\,\alpha.\forall\,\imath.\ \alpha \rightarrow \mathsf{Maybe}^{\widehat{\imath}}\alpha \end{array}$$

for the datatype maybe;

$$\mathsf{C}(\mathsf{BTree}) = \{\mathsf{void}, \mathsf{bnode}\} \qquad \begin{array}{lcl} \mathsf{D}(\mathsf{void}) & = & \forall\,\alpha.\forall\,\imath.\ \mathsf{BTree}^{\widehat{\imath}}\alpha \\ \mathsf{D}(\mathsf{bnode}) & = & \forall\,\alpha.\forall\,\imath.\ \alpha \rightarrow \mathsf{BTree}^{\imath}\alpha \rightarrow \mathsf{BTree}^{\imath}\alpha \rightarrow \mathsf{BTree}^{\widehat{\imath}}\alpha \end{array}$$

for binary trees; and

$$\mathsf{C}(\mathsf{DTree}) = \{\mathsf{empty}, \mathsf{node}\} \qquad \begin{array}{lcl} \mathsf{D}(\mathsf{empty}) & = & \forall\,\alpha.\forall\,\imath.\ \mathsf{DTree}^{\widehat{\imath}}\alpha \\ \mathsf{D}(\mathsf{node}) & = & \forall\,\alpha.\forall\,\imath.\ \alpha \rightarrow (\mathsf{Bool} \rightarrow \mathsf{DTree}^{\imath}\alpha) \rightarrow \mathsf{DTree}^{\widehat{\imath}}\alpha \end{array}$$

for decision trees. That is, binary trees where at each node one has a value and a boolean function. In traversing a decision tree the boolean function determines which subtree to choose.

Each particular legal typing for the arguments of a constructor is obtained by instantiating the associated constructor declaration. The concept of instance of a constructor declaration is formally defined as follows.

Definition 3.3.7 (Instance and domain) *Let* $d \in \mathcal{D}$, $c \in \mathsf{C}(d)$, $s \in \mathcal{S}$ *and* $\vec{\tau} \in \mathcal{T}$ *such that* $\#\vec{\tau} = \mathsf{ar}(d)$. *Assume* $\mathsf{D}(c) = \forall\,\vec{\alpha}.\forall\,\imath.\ \vec{\sigma} \rightarrow d^{\widehat{\imath}}\vec{\alpha}$. *An* instance of c w.r.t. s and $\vec{\tau}$ *is defined as follows*

$$\mathsf{Inst}^{s}_{\vec{\tau}}(c) = \vec{\sigma}[\imath := s][\vec{\alpha} := \vec{\tau}] \rightarrow d^{\widehat{s}}\vec{\tau}$$

A domain of c w.r.t. s and $\vec{\tau}$ *is defined as follows*

$$\mathsf{Dom}^{s}_{\vec{\tau}}(c) = \vec{\sigma}[\imath := s][\vec{\alpha} := \vec{\tau}]$$

We now turn to the typing system. On the stages, we introduce a comparison relation. Importantly, the stage comparison rules state that all stages beyond the limiting stage are equivalent. On top of the stage comparison relation, another set of rules defines a subtyping relation on types. A crucial fact stated by these rules is that a given approximation of a datatype is always included in the next one. This cumulative character is reflected by a chain of subtyping relations $d^{\imath}\vec{\tau} \leq d^{\widehat{\imath}}\vec{\tau} \leq d^{\widehat{\widehat{\imath}}}\vec{\tau} \leq \ldots \leq d^{\infty}\vec{\tau}$ that can be derived by those rules.

Definition 3.3.8 (Stage comparison and subtyping) *τ is a* subtype *of σ, written $\tau \leq \sigma$, is defined by the rules of Figure 3.3, where $s \preccurlyeq r$ is defined by the rules of Figure 3.2.*

$$\text{(refl)}\ \frac{}{s \preccurlyeq s} \qquad \text{(trans)}\ \frac{s \preccurlyeq r \quad r \preccurlyeq p}{s \preccurlyeq p} \qquad \text{(hat)}\ \frac{}{s \preccurlyeq \widehat{s}} \qquad \text{(infty)}\ \frac{}{s \preccurlyeq \infty}$$

Figure 3.2: Stage comparison rules $\lambda\widehat{\ }$

$$\text{(refl)}\ \frac{}{\sigma \leq \sigma} \qquad \text{(data)}\ \frac{s \preccurlyeq r \quad \tau_i \leq \tau_i' \quad (1 \leq i \leq \mathsf{ar}(d))}{d^s\vec{\tau} \leq d^r\vec{\tau'}} \qquad \text{(func)}\ \frac{\tau' \leq \tau \quad \sigma \leq \sigma'}{\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'}$$

Figure 3.3: Subtyping rules for $\lambda\widehat{\ }$

Notation 3.3.9 *We write $\vec{\sigma} \leq \vec{\tau}$, if $\#\vec{\sigma} = \#\vec{\tau}$ and $\sigma[i] \leq \tau[i]$ for $i = 1..\#\vec{\sigma}$.*

Lemma 3.3.10 *If $\sigma \leq \tau$ and $\tau \leq \theta$, then $\sigma \leq \theta$.*

Proof. By induction on the sum of the derivations heights of $\sigma \leq \tau$ and $\tau \leq \theta$. □

Lemma 3.3.11 *If $\widehat{r} \preccurlyeq \widehat{s}$, then $r \preccurlyeq s$.*

Proof. By induction on the proof of $p \preccurlyeq \widehat{s}$, one can show that $p \preccurlyeq \widehat{s}$ implies $p \preccurlyeq s$ or $p = \widehat{s}$ from where the claim can be inferred by instantiating $p = \widehat{r}$. □

3.3.2 The Typing System

In order to define the typing relation between terms and type expressions, we need the concepts of context and judgment.

Definition 3.3.12 (Contexts and judgments)

(var)	$\dfrac{}{\Gamma \vdash x : \sigma}$	if $(x : \sigma) \in \Gamma$
(abs)	$\dfrac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.\, e : \tau \to \sigma}$	
(app)	$\dfrac{\Gamma \vdash e : \tau \to \sigma \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}$	
(cons)	$\dfrac{}{\Gamma \vdash c : \mathsf{Inst}^{s}_{\vec{\tau}}(c)}$	if $c \in \mathsf{C}(d)$
(case)	$\dfrac{\Gamma \vdash e' : d^{\widehat{s}}\vec{\tau} \qquad \Gamma \vdash e_i : \mathsf{Dom}^{s}_{\vec{\tau}}(c_i) \to \theta \quad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{case}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta}$	if $\mathsf{C}(d) = \{c_1, \ldots, c_n\}$
(rec)	$\dfrac{\Gamma, f : d^{\imath}\vec{\tau} \to \theta \vdash e : d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}] \qquad \imath\ \mathsf{pos}\ \theta}{\Gamma \vdash (\mathsf{letrec}\ f = e) : d^{s}\vec{\tau} \to \theta[\imath := s]}$	if $\imath$ not in $\Gamma, \vec{\tau}$
(sub)	$\dfrac{\Gamma \vdash e : \sigma \qquad \sigma \leq \sigma'}{\Gamma \vdash e : \sigma'}$	

Figure 3.4: Typing rules for $\widehat{\lambda}$

1. *A* context Γ *is a finite set of assumptions* $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ *such that the* x_i*s are pairwise distinct elements of* $\mathcal{V}_{\mathcal{E}}$ *and* $\tau_i \in \mathcal{T}$. Γ *can be seen as a partial function so, we write* $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ *and* $\Gamma(x_i) = \tau_i$. *Usually we drop the curly brackets and write simply* $x_1 : \tau_1, \ldots, x_n : \tau_n$. *Moreover, whenever it is written* $\Gamma, x : \tau$ *or* Γ, Γ' *it is assumed that* $(x : \tau) \notin \Gamma$ *and* $\Gamma \cap \Gamma' = \emptyset$.

2. *A* typing judgment *is a triple of the form* $\Gamma \vdash e : \sigma$, *where* Γ *is a context,* e *is a term and* σ *is a type expression.*

The definition of the typing relation itself depends on that of subtyping.

Definition 3.3.13 (Typing)

1. *A typing judgment is* derivable *if it can be inferred from the rules of Figure 3.4 where the positivity condition* $\imath\ \mathsf{pos}\ \sigma$ *in the (rec) rule is defined in Figure 3.5.*

2. *A term* $e \in \mathcal{E}$ *is typable if* $\Gamma \vdash e : \sigma$ *is derivable for some context* Γ *and type* σ.

The rules (var), (abs), and (app) come from the standard simply typed λ-calculus. The rule (sub) is present in any λ-calculus with subtyping and provides a link between the subtyping and typing relations. The remaining rules—(cons), (case) and (rec)—deserve some comments.

The (cons) rule says that applying a constructor of a given datatype to values in an approximation of the datatype gives a value that is guaranteed to be an element in the next approximation. Observe that constructors are stage-polymorphic.

$$\text{(sp1)} \quad \frac{}{\imath \ \mathsf{pos}\ \alpha} \qquad \text{(sn1)} \quad \frac{}{\imath \ \mathsf{neg}\ \alpha}$$

$$\text{(sp2)} \quad \frac{\imath \ \mathsf{neg}\ \tau \quad \imath \ \mathsf{pos}\ \sigma}{\imath \ \mathsf{pos}\ \tau \to \sigma} \qquad \text{(sn2)} \quad \frac{\imath \ \mathsf{pos}\ \tau \quad \imath \ \mathsf{neg}\ \sigma}{\imath \ \mathsf{neg}\ \tau \to \sigma}$$

$$\text{(sp3)} \quad \frac{\imath \ \mathsf{pos}\ \tau_i \quad (1 \leq i \leq \mathsf{ar}(d))}{\imath \ \mathsf{pos}\ d^s\vec{\tau}} \qquad \text{(sn3)} \quad \frac{\imath \ \mathsf{nocc}\ s \quad \imath \ \mathsf{neg}\ \tau_i \quad (1 \leq i \leq \mathsf{ar}(d))}{\imath \ \mathsf{neg}\ d^s\vec{\tau}}$$

Figure 3.5: Positive-negative occurrences of a stage variable

The (case) rule says that the converse is also true: any value in the approximation next to some given one is a result of applying one of the constructors of the datatype to values in the given approximation and can therefore be subjected to case analysis.

The (rec) rule, finally, says that any systematic way of extending a function defined on a given approximation of a datatype to work also on the next approximation induces a function defined on the whole datatype, the limit of the approximations. The premise of this rule involves an implicit universal quantification over the set of all stages (freshness condition) and incarnates the step of induction, taking the function from stage $\imath$ to stage $\widehat{\imath}$.

Despite its simplicity, this rule is powerful enough to capture course-of-value primitive recursion. If we have computed the predecessor $x' : d^\imath\vec{\tau}$ of an input $x : d^{\widehat{\imath}}\vec{\tau}$ within the body of the recursive function f, we can use the fact that $d^\imath\vec{\tau} \leq d^{\widehat{\imath}}\vec{\tau}$ to type $x' : d^{\widehat{\imath}}\vec{\tau}$ by subsumption, and then compute its predecessor again. Going on like this, we can analyze arbitrarily deep the input x to obtain subcomponents which can then be used as arguments to recursive calls of f since they inhabit $d^\imath\vec{\tau}$. An example of a function which has a course-of-value recursion implementation is the function that tests if a natural number is even or not (see Example 3.4.2).

A feature of the (rec) rule, we have not mentioned yet, is that the stage variable $\imath$ can occur in the codomain of the function f. The fact that θ may depend on $\imath$ enables the type system to track stage dependencies between input and output of the function, providing a means to achieve more precise typings. This feature is very valuable for defining functions where the argument to the recursive call is derived from the input argument through another function. An example of such a function is the Euclidean division (see Example 3.4.4), where argument to the recursive call of *div* is the result of a subtraction. However, the occurrences of $\imath$ in θ are restricted to positive positions, by the side condition $\imath\ \mathsf{pos}\ \theta$. Indeed, if we simply drop this side condition we can type non-terminating functions. This condition is a form of guaranteeing covariance between the stage approximation of the domain and the codomain of function f. In other words, the type of the codomain of a recursive function can only monotonically depend on the type of the function domain. We make use of this positivity condition when proving the soundness of the typing system (see Theorem 4.3.30 and Lemma 4.3.29).

To better help the comprehension of $\lambda\hat{}$ we show *in extenso* a typing derivation for $\vdash$ plus : $\mathsf{Nat}^s \to \mathsf{Nat} \to \mathsf{Nat}$, where s can be any stage, in particular ∞.

Example 3.3.14 *Consider again the addition of two natural numbers:*

$$\text{plus} \equiv (\mathsf{letrec}\ plus = \lambda x.\, \lambda y.\, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda x'.\, \mathsf{s}\,(plus\, x'\, y)\})$$

The function plus *can be shown to have type* $\mathsf{Nat}^s \to \mathsf{Nat} \to \mathsf{Nat}$ *as follows*

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash x : \mathsf{Nat}^{\widehat{\imath}}}\,(\mathrm{var}) \qquad \dfrac{}{\Gamma_2 \vdash y : \mathsf{Nat}}\,(\mathrm{var}) \qquad \dfrac{\dfrac{(3.1) \qquad (3.2)}{\Gamma_3 \vdash \mathsf{s}\,(plus\,x'\,y) : \mathsf{Nat}}\,(\mathrm{app})}{\Gamma_2 \vdash \lambda x'.\,\mathsf{s}\,(plus\,x'\,y) : \mathsf{Nat}^{\imath} \to \mathsf{Nat}}\,(\mathrm{abs})}{\dfrac{\Gamma_2 \vdash \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda x'.\,\mathsf{s}\,(plus\,x'\,y)\} : \mathsf{Nat}}{\Gamma_1 \vdash \lambda x.\lambda y.\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda x'.\,\mathsf{s}\,(plus\,x'\,y)\} : \mathsf{Nat}^{\widehat{\imath}} \to \mathsf{Nat} \to \mathsf{Nat}}\,2 \times (\mathrm{abs})}\,(\mathrm{case})}{\vdash \mathrm{plus} : \mathsf{Nat}^s \to \mathsf{Nat} \to \mathsf{Nat}}\,(\mathrm{rec})$$

where

$$\begin{aligned}
\Gamma_1 &\equiv plus : \mathsf{Nat}^{\imath} \to \mathsf{Nat} \to \mathsf{Nat} \\
\Gamma_2 &\equiv plus : \mathsf{Nat}^{\imath} \to \mathsf{Nat} \to \mathsf{Nat}, x : \mathsf{Nat}^{\widehat{\imath}}, y : \mathsf{Nat} \\
\Gamma_3 &\equiv plus : \mathsf{Nat}^{\imath} \to \mathsf{Nat} \to \mathsf{Nat}, x : \mathsf{Nat}^{\widehat{\imath}}, y : \mathsf{Nat}, x' : \mathsf{Nat}^{\imath}
\end{aligned}$$

$$\dfrac{\dfrac{}{\Gamma_3 \vdash \mathsf{s} : \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\widehat{\infty}}}\,(\mathrm{cons}) \qquad \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\widehat{\infty}} \leq \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\infty}}{\Gamma_3 \vdash \mathsf{s} : \mathsf{Nat} \to \mathsf{Nat}}\,(\mathrm{sub}) \tag{3.1}$$

and

$$\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash plus : \mathsf{Nat}^{\imath} \to \mathsf{Nat} \to \mathsf{Nat}}\,(\mathrm{var}) \qquad \dfrac{}{\Gamma_3 \vdash x' : \mathsf{Nat}^{\imath}}\,(\mathrm{var})}{\Gamma_3 \vdash plus\,x' : \mathsf{Nat} \to \mathsf{Nat}}\,(\mathrm{app}) \qquad \dfrac{}{\Gamma_3 \vdash y : \mathsf{Nat}}\,(\mathrm{var})}{\Gamma_3 \vdash plus\,x'\,y : \mathsf{Nat}}\,(\mathrm{app}) \tag{3.2}$$

3.4 Some Examples

In order to illustrate the mechanics and expressive power of our calculus, we now give a few examples of programming. We start from simple recursive definitions that are definable in all useful existing systems.

Example 3.4.1 (Standard examples)

- *The concatenation of two lists and the concatenation of a list of lists.*

$$\begin{array}{ll}
\mathrm{append} \equiv & (\mathsf{letrec}\ append_{:\mathsf{List}^{\imath}\tau \to \mathsf{List}\,\tau \to \mathsf{List}\,\tau} = \lambda x_{:\mathsf{List}^{\widehat{\imath}}\tau}.\ \lambda y_{:\mathsf{List}\,\tau}. \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow y \\
& \qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z_{:\tau}.\ \lambda x'_{:\mathsf{List}^{\imath}\tau}.\ \mathsf{cons}\ z\ \underbrace{(append\ x'\ y)}_{:\mathsf{List}\,\tau} \\
& \qquad\qquad \} \\
) : & \mathsf{List}^{s}\tau \to \mathsf{List}\,\tau \to \mathsf{List}\,\tau
\end{array}$$

$$\begin{array}{ll}
\mathrm{conc} \equiv & (\mathsf{letrec}\ conc_{:\mathsf{List}^{\imath}(\mathsf{List}\,\tau) \to \mathsf{List}\,\tau} = \lambda x_{:\mathsf{List}^{\widehat{\imath}}(\mathsf{List}\,\tau)}. \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil} \\
& \qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z_{:\mathsf{List}\,\tau}.\ \lambda x'_{:\mathsf{List}^{\imath}(\mathsf{List}\,\tau)}.\ \mathrm{append}\ z\ \underbrace{(conc\ x')}_{:\mathsf{List}\,\tau} \\
& \qquad\qquad \} \\
) : & \mathsf{List}^{s}(\mathsf{List}\,\tau) \to \mathsf{List}\,\tau
\end{array}$$

- *The addition of two ordinals.*

$$
\begin{array}{ll}
\mathrm{add} \equiv & (\mathsf{letrec}\ add_{:\mathsf{Ord}^{\imath}\to\mathsf{Ord}\to\mathsf{Ord}} = \lambda x_{:\mathsf{Ord}^{\hat{\imath}}}.\ \lambda y_{:\mathsf{Ord}}. \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{zero}\ \Rightarrow\ y \\
& \qquad\qquad \mid \mathsf{succ}\ \Rightarrow\ \lambda x'_{:\mathsf{Ord}^{\imath}}.\ \mathsf{succ}\ \underbrace{(add\ x'\ y)}_{:\mathsf{Ord}} \\
& \qquad\qquad \mid \mathsf{lim}\ \Rightarrow\ \lambda x'_{:\mathsf{Nat}\to\mathsf{Ord}^{\imath}}.\ \mathsf{lim}\ \underbrace{(\lambda z_{:\mathsf{Nat}}.\ add\ \underbrace{\underbrace{(x'\ z)}_{:\mathsf{Ord}^{\imath}}\ y}_{:\mathsf{Ord}})}_{:\mathsf{Ord}^{\widehat{\infty}}\leq\mathsf{Ord}} \\
& \qquad\qquad \} \\
)\ : & \mathsf{Ord}^{s}\to\mathsf{Ord}\to\mathsf{Ord}
\end{array}
$$

Observe that in the types inferred for all these functions, the stage s can be any stage. In fact, the minimal type is achieved when s is ∞, because s does not occur in the codomain of these functions. So, we could simply have dropped the stage s in these examples.

The following examples illustrate the use of recursive calls on deep components of the formal argument of the function.

Example 3.4.2 (Examples of course-of-value recursion)

- *The predicate that decides if a natural number is even or not may be defined as follows. This program involves a recursive call on a deep recursive component of the argument value. To type it, therefore, the subsumption rule has to be used. This is an example of course-of-value primitive recursion.*

$$
\begin{array}{ll}
\mathrm{even} \equiv (\mathsf{letrec} & even_{:\mathsf{Nat}^{\imath}\to\mathsf{Bool}} = \lambda x_{:\mathsf{Nat}^{\hat{\imath}}}. \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o}\ \Rightarrow\ \mathsf{true} \\
& \qquad\qquad \mid \mathsf{s}\ \Rightarrow\ \lambda x'_{:\mathsf{Nat}^{\imath}\leq\mathsf{Nat}^{\hat{\imath}}}.\ \mathsf{case}\ x'\ \mathsf{of}\ \{\mathsf{o}\ \Rightarrow\ \mathsf{false} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s}\ \Rightarrow\ \lambda x''_{:\mathsf{Nat}^{\imath}}.\ \underbrace{even\ x''}_{:\mathsf{Bool}}\ \} \\
& \qquad\qquad \} \\
)\ : & \mathsf{Nat}^{s}\to\mathsf{Bool}
\end{array}
$$

- *The function that takes a decision tree T and a list of booleans, determining a path in T, and returns the node last visited.*

$$
\begin{array}{ll}
\mathrm{ans} \equiv (\mathsf{letrec} & ans_{:\mathsf{DTree}^{\imath}\alpha\to\mathsf{List\,Bool}\to\mathsf{Maybe}\,\alpha} = \lambda x_{:\mathsf{DTree}^{\hat{\imath}}\alpha}.\lambda l_{:\mathsf{List\,Bool}}.\ \mathsf{case}\ x\ \mathsf{of}\ \{ \\
& \quad \mathsf{empty}\ \Rightarrow\ \mathsf{nothing} \\
& \mid \mathsf{node}\ \Rightarrow\ \lambda a_{:\alpha}.\lambda f_{:\mathsf{Bool}\to\mathsf{DTree}^{\imath}\alpha}.\ \mathsf{case}\ l\ \mathsf{of}\ \{ \\
& \qquad\qquad \mathsf{nil}\ \Rightarrow\ \mathsf{just}\,a \\
& \qquad\quad \mid \mathsf{cons}\ \Rightarrow\ \lambda y_{:\mathsf{Bool}}.\lambda z_{:\mathsf{List\,Bool}}.\ \mathsf{case}\ \overbrace{(f\,y)}^{:\mathsf{DTree}^{\imath}\alpha\,\leq\,\mathsf{DTree}^{\hat{\imath}}\alpha}\ \mathsf{of}\ \{ \\
& \qquad\qquad\qquad\qquad \mathsf{empty}\ \Rightarrow\ \mathsf{just}\,a \\
& \qquad\qquad\qquad\quad \mid \mathsf{node}\ \Rightarrow\ \lambda b_{:\alpha}.\lambda g_{:\mathsf{Bool}\to\mathsf{DTree}^{\imath}\alpha}.\ \underbrace{ans\,(f\,y)\,z}_{:\mathsf{Maybe}\,\alpha} \\
& \qquad\qquad\qquad\quad \} \\
& \qquad\quad \} \\
& \} \\
)\ : & \mathsf{DTree}^{s}\alpha\to\mathsf{List\,Bool}\to\mathsf{Maybe}\,\alpha
\end{array}
$$

The following examples demonstrate the specific, novel features of $\widehat{\lambda}$. First of all, stages provide a limited means of controlling the effect of a recursively defined function in terms of the relation between the depths of argument and result values.

Example 3.4.3 (Examples of "exact" typings)

- *The length of a list. This standard program for calculating the length of a list admits an unusually "exact" (i.e., tight and therefore informative) type in $\widehat{\lambda}$.*

$$
\begin{array}{ll}
\mathrm{length} \equiv & (\mathsf{letrec}\ length_{:\mathsf{List}^{\imath}\tau \to \mathsf{Nat}^{\imath}} = \\
& \lambda x_{:\mathsf{List}^{\widehat{\imath}}\tau}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil}\ \Rightarrow\ \mathsf{o} \\
& \qquad\qquad\qquad \mid \mathsf{cons}\ \Rightarrow\ \lambda z_{:\tau}.\ \lambda x'_{:\mathsf{List}^{\imath}\tau}.\ \underbrace{\mathsf{s}\ \underbrace{(length\ x')}_{:\mathsf{Nat}^{\imath}}}_{:\mathsf{Nat}^{\widehat{\imath}}} \\
& \qquad\qquad\qquad \} \\
) : & \mathsf{List}^{s}\tau \to \mathsf{Nat}^{s}
\end{array}
$$

- *The map of a function on a list. This program is very similar to that for the length function and also admits an "exact" typing, but becomes crucial in an example below.*

$$
\begin{array}{ll}
\mathrm{map} \equiv & (\lambda f_{:\tau \to \sigma}.\quad (\mathsf{letrec}\ map_{:\mathsf{List}^{\imath}\tau \to \mathsf{List}^{\imath}\sigma} = \lambda x_{:\mathsf{List}^{\widehat{\imath}}\tau}. \\
& \qquad\quad \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil}\ \Rightarrow\ \mathsf{nil} \\
& \qquad\qquad\qquad \mid \mathsf{cons}\ \Rightarrow\ \lambda z_{:\tau}.\ \lambda x'_{:\mathsf{List}^{\imath}\tau}.\ \underbrace{\mathsf{cons}\ \underbrace{(f\ z)}_{:\sigma}\ \underbrace{(map\ x')}_{:\mathsf{List}^{\imath}\sigma}}_{:\mathsf{List}^{\widehat{\imath}}\sigma} \\
& \qquad\qquad\qquad \} \\
& \qquad\quad)_{:\mathsf{List}^{s}\tau \to \mathsf{List}^{s}\sigma} \\
) : & (\tau \to \sigma) \to \mathsf{List}^{s}\tau \to \mathsf{List}^{s}\sigma
\end{array}
$$

The precision of the types inferred for these functions is achieved by the fact that, in both examples, the codomain of the function depends on the stage variable of the domain. This enables the type system to track stage dependencies between input and output of the function, and to produce more informative types. The type inferred for length *informs that when this function is applied to a list of a certain height produces a natural that does not exceed the height of the input list. The type inferred for* map *provide the additional information that, for $f : \tau \to \sigma$ and $l : \mathsf{List}^{s}\tau$, the output list* (map f l) *is in the same approximation stage as the input list l. Therefore, the list produced cannot be bigger than the input list.*

Further, recursive calls are allowed on structurally smaller arguments that cannot be verified to be structurally smaller using a viable syntactic criterion.

Example 3.4.4 (Examples not handled by the guard condition)

- *Euclidean division $\lceil \frac{x}{y+1} \rceil$. This program for Euclidean division depends on a program for subtraction. It is not typable in systems with a syntactic guard predicate, as, syntactically,* (minus x' y) *is not properly structurally smaller than x in the program below. In $\widehat{\lambda}$, it is typable because of the "exact" type assignable to* minus, *that guarantees the term* (minus x' y)

is in the same approximation stage as x', and so must be smaller than x.

$$
\begin{array}{ll}
\mathrm{minus} \equiv & (\mathsf{letrec}\ \mathit{minus}_{:\mathsf{Nat}^{\imath}\rightarrow\mathsf{Nat}\rightarrow\mathsf{Nat}^{\imath}} = \lambda x_{:\mathsf{Nat}^{\hat{\imath}}}.\ \lambda y_{:\mathsf{Nat}}. \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \\
& \qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x'_{:\mathsf{Nat}^{\imath}}.\ \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y'_{:\mathsf{Nat}}.\ \underbrace{\mathit{minus}\ x'\ y'}_{:\mathsf{Nat}^{\imath}\leq\mathsf{Nat}^{\hat{\imath}}}\ \} \\
& \qquad\qquad \} \\
):\ & \mathsf{Nat}^{s}\rightarrow\mathsf{Nat}\rightarrow\mathsf{Nat}^{s}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{div} \equiv & (\mathsf{letrec}\ \mathit{div}_{:\mathsf{Nat}^{\imath}\rightarrow\mathsf{Nat}\rightarrow\mathsf{Nat}^{\imath}} = \\
& \lambda x_{:\mathsf{Nat}^{\hat{\imath}}}.\ \lambda y_{:\mathsf{Nat}}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \\
& \qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x'_{:\mathsf{Nat}^{\imath}}.\ \underbrace{\mathsf{s}\ \underbrace{(\mathit{div}\ \underbrace{(\mathrm{minus}\ x'\ y)}_{:\mathsf{Nat}^{\imath}}\ y)}_{:\mathsf{Nat}^{\imath}}}_{:\mathsf{Nat}^{\hat{\imath}}} \\
& \qquad\qquad\qquad\qquad \} \\
):\ & \mathsf{Nat}^{s}\rightarrow\mathsf{Nat}\rightarrow\mathsf{Nat}^{s}
\end{array}
$$

- *Flattening of finitely branching trees. This program depends on* map *and* conc*. Similarly to* div*, it is not typable in systems with a syntactic guard predicate.*

$$
\begin{array}{ll}
\mathrm{flatten} \equiv & (\mathsf{letrec}\ \mathit{flatten}_{:\mathsf{Tree}^{\imath}\tau\rightarrow\mathsf{List}\,\tau} = \\
& \lambda x_{:\mathsf{Tree}^{\hat{\imath}}\tau}.\ \mathsf{case}\ x\ \mathsf{of}\ \{ \\
& \quad \mathsf{branch} \Rightarrow \lambda z_{:\tau}.\ \lambda x'_{:\mathsf{List}\,(\mathsf{Tree}^{\imath}\tau)}.\ \mathsf{cons}\ z\ \underbrace{(\mathrm{conc}\ \underbrace{(\mathrm{map}\ \mathit{flatten}\ x')}_{:\mathsf{List}\,(\mathsf{List}\,\tau)})}_{:\mathsf{List}\,\tau} \\
& \qquad\qquad \} \\
):\ & \mathsf{Tree}^{s}\tau\rightarrow\mathsf{List}\ \tau
\end{array}
$$

Differently from the examples presented before, the recursive calls of functions div *and* flatten *are not derived from the input directly, i.e., using only case analysis: in* div*, the argument of the recursive call is connected to a direct subterm of the input via function* minus*; in* flatten *the recursive call is performed via function* map*. Both functions are problematic for syntactic methods, and they are not handled by Giménez guard condition.*

Finally, we give an example demonstrating the usefulness of having the capability of naming stages explicitly: if one recursion is nested in another, we need two distinct free stage variables.

Example 3.4.5 (Examples involving several stage variables)

- *The Ackermann function. The natural definition of the Ackermann function is not typable in our system. A definition with two recursions, one nesting the other, however, is easy to* *type.*

$$
\begin{array}{ll}
\mathrm{ack} \equiv & (\mathsf{letrec}\ ack_{:\mathsf{Nat}^{\imath}\to\mathsf{Nat}\to\mathsf{Nat}} = \lambda x_{:\mathsf{Nat}^{\hat{\imath}}}. \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o}\ \Rightarrow\ \underbrace{\lambda z.\,(\mathsf{s}\ z)}_{:\mathsf{Nat}\to\mathsf{Nat}} \\
& \qquad\qquad \mid \mathsf{s}\ \Rightarrow\ \lambda x'_{:\mathsf{Nat}^{\imath}}.\ (\mathsf{letrec}\ ack\_x_{:\mathsf{Nat}^{\jmath}\to\mathsf{Nat}} = \lambda y_{:\mathsf{Nat}^{\hat{\jmath}}}. \\
& \qquad\qquad\qquad\qquad \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{o}\ \Rightarrow\ \underbrace{ack\ x'\ (\mathsf{s}\ \mathsf{o})}_{:\mathsf{Nat}} \\
& \qquad\qquad\qquad\qquad\qquad \mid \mathsf{s}\ \Rightarrow\ \lambda y'_{:\mathsf{Nat}^{\jmath}}.\ \underbrace{ack\ x'\ \underbrace{(ack\_x\ y')}_{:\mathsf{Nat}}}_{:\mathsf{Nat}}\ \} \\
& \qquad\qquad\qquad\qquad)_{:\mathsf{Nat}\to\mathsf{Nat}} \\
& \qquad\qquad \} \\
)\ : & \mathsf{Nat}^{s} \to \mathsf{Nat} \to \mathsf{Nat}
\end{array}
$$

- *Sum of all the nodes of a finitely branching tree of naturals. Recall that datatypes* Tree *and* List *are interleaving. Naturally recursion over these nested datatypes is expressed by mutually interleaving recursive functions. Observe that* $\mathsf{List}(\mathsf{Tree}^{\imath}\mathsf{Nat}) \to \mathsf{Nat}$ *is the minimal type that the inner letrec-expression admits because of the recursive call* $(sumt\ y')$.

$$
\begin{array}{ll}
\mathrm{sumt} \equiv & (\mathsf{letrec}\ sumt_{:\mathsf{Tree}^{\imath}\mathsf{Nat}\to\mathsf{Nat}} = \lambda x_{:\mathsf{Tree}^{\hat{\imath}}\mathsf{Nat}}.\mathsf{case}\ x\ \mathsf{of}\ \{ \\
& \mathsf{branch}\ \Rightarrow\ \lambda x'_{:\mathsf{Nat}}.\lambda x''_{:\mathsf{List}(\mathsf{Tree}^{\imath}\mathsf{Nat})}. \\
& \quad \big(\mathrm{plus}\ x'\ (\mathsf{letrec}\ suml_{:\mathsf{List}^{\jmath}(\mathsf{Tree}^{\imath}\mathsf{Nat})\to\mathsf{Nat}} = \lambda y_{:\mathsf{List}^{\hat{\jmath}}(\mathsf{Tree}^{\imath}\mathsf{Nat})}.\ \mathsf{case}\ y\ \mathsf{of}\ \{ \\
& \qquad\qquad \mathsf{nil}\ \Rightarrow\ \mathsf{o} \\
& \qquad\qquad \mid \mathsf{cons}\ \Rightarrow\ \lambda y'_{:\mathsf{Tree}^{\imath}\mathsf{Nat}}.\lambda y''_{:\mathsf{List}^{\jmath}(\mathsf{Tree}^{\imath}\mathsf{Nat})}.\ \mathrm{plus}\ \underbrace{(sumt\ y')}_{:\mathsf{Nat}}\ \underbrace{(suml\ y'')}_{:\mathsf{Nat}}\ \} \\
& \qquad\qquad)_{:\mathsf{List}(\mathsf{Tree}^{\imath}\mathsf{Nat})\to\mathsf{Nat}}\ x''\,\big)_{:\mathsf{Nat}} \\
& \ \} \\
)\ : & \mathsf{Tree}^{s}\mathsf{Nat} \to \mathsf{Nat}
\end{array}
$$

Example 3.4.6 (Another example of "exact" typings) *Converting a list of naturals into a binary (search) tree. The precise typing* $\mathrm{ltobt} : \mathsf{List}^{s}\mathsf{Nat} \to \mathsf{BTree}^{s}\mathsf{Nat}$ *informs that the height of the result tree never exceeds the height of the input list. Notice that to achieve this precision it is crucial to have* $\mathrm{ins} : \mathsf{BTree}^{s}\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{BTree}^{\hat{s}}\mathsf{Nat}$.

$$
\begin{array}{ll}
\mathrm{leq} \equiv & (\mathsf{letrec}\ leq_{:\mathsf{Nat}^{\imath}\to\mathsf{Nat}\to\mathsf{Bool}^{\imath}} = \lambda x_{:\mathsf{Nat}^{\hat{\imath}}}.\ \lambda y_{:\mathsf{Nat}}.\ \mathsf{case}\ x\ \mathsf{of}\ \{ \\
& \qquad\quad \mathsf{o}\ \Rightarrow\ \mathsf{case}\ \overbrace{y}^{:\mathsf{Nat}\leq\mathsf{Nat}^{\widehat{\infty}}}\ \mathsf{of}\ \{\mathsf{o}\ \Rightarrow\ \overbrace{\mathsf{true}}^{:\mathsf{Bool}^{\hat{\imath}}} \\
& \qquad\qquad\qquad\qquad\qquad \mid \mathsf{s}\ \Rightarrow\ \lambda y'_{:\mathsf{Nat}}.\ \mathsf{false}\ \ \} \\
& \qquad\quad \mid \mathsf{s}\ \Rightarrow\ \lambda x'_{:\mathsf{Nat}^{\imath}}.\ \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{o}\ \Rightarrow\ \mathsf{false} \\
& \qquad\qquad\qquad\qquad\qquad \mid \mathsf{s}\ \Rightarrow\ \lambda y'_{:\mathsf{Nat}}.\ \underbrace{leq\ x'\ y'}_{:\mathsf{Bool}^{\imath}\leq\mathsf{Bool}^{\hat{\imath}}}\ \ \} \\
& \qquad\quad \} \\
)\ : & \mathsf{Nat}^{s} \to \mathsf{Nat} \to \mathsf{Bool}^{s}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{ins} \equiv & (\mathsf{letrec}\ ins_{:\mathsf{BTree}^{\imath}\mathsf{Nat}\rightarrow\mathsf{Nat}\rightarrow\mathsf{BTree}^{\widehat{\imath}}\mathsf{Nat}} = \lambda x_{:\mathsf{BTree}^{\widehat{\imath}}\mathsf{Nat}}.\ \lambda y_{:\mathsf{Nat}}.\ \mathsf{case}\ x\ \mathsf{of}\,\{ \\
& \qquad \mathsf{void}\ \Rightarrow\ \overbrace{\mathsf{bnode}\ y\ \mathsf{void}\ \mathsf{void}}^{:\mathsf{BTree}^{\widehat{\widehat{\imath}}}} \\
& \qquad \mid \mathsf{bnode}\ \Rightarrow\ \lambda z_{:\mathsf{Nat}}.\ \lambda x'_{:\mathsf{BTree}^{\imath}\mathsf{Nat}}.\ \lambda x''_{:\mathsf{BTree}^{\imath}\mathsf{Nat}}.\ \mathsf{case}\ \overbrace{(\mathrm{leq}\ y\ z)}^{\mathsf{Bool}\leq\mathsf{Bool}^{\widehat{\infty}}}\ \mathsf{of}\,\{ \\
& \qquad\qquad\qquad \mathsf{true}\ \Rightarrow\ \mathsf{bnode}\ (ins\ x'\ y)\ x'' \\
& \qquad\qquad\qquad \mid \mathsf{false}\ \Rightarrow\ \underbrace{\mathsf{bnode}\ \underbrace{x'}_{:\mathsf{BTree}^{\imath}\mathsf{Nat}\leq\mathsf{BTree}^{\widehat{\imath}}\mathsf{Nat}}\ \underbrace{(ins\ x''\ y)}_{:\mathsf{BTree}^{\widehat{\imath}}\mathsf{Nat}}}_{:\mathsf{BTree}^{\widehat{\widehat{\imath}}}\mathsf{Nat}}\ \} \\
& \qquad \} \\
):\ & \mathsf{BTree}^{s}\mathsf{Nat}\rightarrow\mathsf{Nat}\rightarrow\mathsf{BTree}^{\widehat{s}}\mathsf{Nat}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{ltobt} \equiv & (\mathsf{letrec}\ ltobt_{:\mathsf{List}^{\imath}\mathsf{Nat}\rightarrow\mathsf{BTree}^{\imath}\mathsf{Nat}} = \lambda x_{:\mathsf{List}^{\widehat{\imath}}\mathsf{Nat}}.\ \mathsf{case}\ x\ \mathsf{of}\,\{ \\
& \qquad \mathsf{nil}\ \Rightarrow\ \mathsf{void} \\
& \qquad \mid \mathsf{cons}\ \Rightarrow\ \lambda z_{:\mathsf{Nat}}.\ \lambda x'_{:\mathsf{List}^{\imath}\mathsf{Nat}}.\ \underbrace{\mathrm{ins}\ \underbrace{(ltobt\ x')}_{:\mathsf{BTree}^{\imath}\mathsf{Nat}}\ z}_{:\mathsf{BTree}^{\widehat{\imath}}\mathsf{Nat}} \\
& \qquad \} \\
):\ & \mathsf{List}^{s}\mathsf{Nat}\rightarrow\mathsf{BTree}^{s}\mathsf{Nat}
\end{array}
$$

Chapter 4

Meta-Theoretical Results for $\lambda^{\frown}$

This chapter establishes some fundamental meta-theoretic properties of $\lambda^{\frown}$, including confluence of the reduction calculus, subject reduction and strong normalizability of typable terms.

4.1 Confluence

In $\lambda^{\frown}$, all reduction strategies to compute an expression yield the same result. This is a consequence of the confluence property of the computation relation, which states that if an expression e can be partially computed into two different expressions e_1 and e_2, then there exists a third expression e' such that both e_1 and e_2 can be computed into e'. The confluence property, therefore, guarantees the uniqueness of the value (normal form) of any term. The proof of this property is done by the standard technique of Tait and Martin-Löf.

Definition 4.1.1 *A binary relation* $\rightarrow$ *satisfies the* diamond property *if*

$$a \rightarrow a' \;\wedge\; a \rightarrow a'' \quad\Rightarrow\quad \exists\, b \in \mathcal{E}.\; a' \rightarrow b \;\wedge\; a'' \rightarrow b$$

Lemma 4.1.2 *If a binary relation satisfies the diamond property, then its transitive closure satisfies also the diamond property.*

Proof. By a simple diagram chase suggested by a tiled diagram. □

To demonstrate that $\twoheadrightarrow_{\beta\iota\mu}$ is confluent, we must show that $\twoheadrightarrow_{\beta\iota\mu}$ satisfies the diamond property. However $\rightarrow_{\beta\iota\mu}$ does not satisfies it. In order to apply the above result to show that $\twoheadrightarrow_{\beta\iota\mu}$ satisfies the diamond property, we define a new relation $\twoheadrightarrow_1$ which satisfies the diamond property and which has $\twoheadrightarrow_{\beta\iota\mu}$ as its transitive closure. Let us introduce the relation $\twoheadrightarrow_1$ which is reflexive and allows multiple $\beta\iota\mu$-reductions in one step, and establish some of its properties. We only show the parts involving case-expressions, more details can be found, for instance, in [15].

Definition 4.1.3 *The* parallel one-step relation, $\twoheadrightarrow_1$, *is defined on* $\mathcal{E}$ *inductively as follows:*

1. $a \twoheadrightarrow_1 a$
2. $a \twoheadrightarrow_1 a' \quad\Rightarrow\quad \lambda x.a \twoheadrightarrow_1 \lambda x.a'$
3. $a \twoheadrightarrow_1 a' \;\wedge\; b \twoheadrightarrow_1 b' \quad\Rightarrow\quad a\, b \twoheadrightarrow_1 a'\, b'$
4. $a \twoheadrightarrow_1 a' \;\wedge\; b \twoheadrightarrow_1 b' \quad\Rightarrow\quad (\lambda x.a)\, b \twoheadrightarrow_1 a'[x := b']$

5. $a \twoheadrightarrow_1 a' \;\wedge\; b_i \twoheadrightarrow_1 b'_i$ *with* $i = 1..n \;\;\Rightarrow\;\; \mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 \mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\}$

6. $a_j \twoheadrightarrow_1 a'_j \;\wedge\; b_k \twoheadrightarrow_1 b'_k$ *for some* k*, with* $j = 1..\mathsf{ar}(c_k) \;\;\Rightarrow\;\; \mathsf{case}\ (c_k\,\vec{a})\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 b'_k\,\vec{a'}$

7. $e \twoheadrightarrow_1 e' \;\;\Rightarrow\;\; \mathsf{letrec}\ x = e \twoheadrightarrow_1 \mathsf{letrec}\ x = e'$

8. $e \twoheadrightarrow_1 e' \;\wedge\; a_i \twoheadrightarrow_1 a'_i$ *with* $i = 1..\mathsf{ar}(c) \;\;\Rightarrow\;\; (\mathsf{letrec}\ x = e)\,(c\,\vec{a}) \twoheadrightarrow_1 e'[x := \mathsf{letrec}\ x = e']\,(c\,\vec{a'})$

Lemma 4.1.4

$$a \twoheadrightarrow_1 a' \;\wedge\; b \twoheadrightarrow_1 b' \;\;\Rightarrow\;\; a[x := b] \twoheadrightarrow_1 a'[x := b']$$

Proof. By induction on the definition of $a \twoheadrightarrow_1 a'$. We only treat the cases for clauses 5 and 6 here.

5. Assume $a \twoheadrightarrow_1 a'$ is $\mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 \mathsf{case}\ e'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\}$ and is a direct consequence of $e \twoheadrightarrow_1 e'$, $b_i \twoheadrightarrow_1 b'_i$ with $i = 1..n$. By induction hypothesis $e[x := b] \twoheadrightarrow_1 e'[x := b']$ and $b_i[x := b] \twoheadrightarrow_1 b'_i[x := b']$. Hence $a[x := b] = \mathsf{case}\ e[x := b]\ \mathsf{of}\ \{\vec{c} \Rightarrow \overrightarrow{b[x := b]}\} \twoheadrightarrow_1 \mathsf{case}\ e'[x := b']\ \mathsf{of}\ \{\vec{c} \Rightarrow \overrightarrow{b'[x := b']}\} = a'[x := b']$.

6. Assume $a \twoheadrightarrow_1 a'$ is $\mathsf{case}\ (c_k\,\vec{a})\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 b_k\,\vec{a'}$ and is a direct consequence of $a_i \twoheadrightarrow_1 a'_i$ with $i = 1..\mathsf{ar}(c_k)$, $b_k \twoheadrightarrow_1 b'_k$. By induction hypothesis $b_k[x := b] \twoheadrightarrow_1 b'_k[x := b']$ and $a_i[x := b] \twoheadrightarrow_1 a'_i[x := b']$. Then $a[x := b] = \mathsf{case}\ (c_k\,\overrightarrow{a[x := b]}\ \mathsf{of}\ \{\vec{c} \Rightarrow \overrightarrow{b[x := b]}\} \twoheadrightarrow_1 b'_k[x := b']\,\overrightarrow{a'[x := b']} = a'[x := b']$.

$\square$

Lemma 4.1.5 (Generation lemma for $\twoheadrightarrow_1$)

1. $\lambda x.a \twoheadrightarrow_1 e$ *implies* $e \equiv \lambda x.a'$ *with* $a \twoheadrightarrow_1 a'$.
2. $a_1\,a_2 \twoheadrightarrow_1 e$ *implies either:*
   (a) $e \equiv a'_1\,a'_2$ *with* $a_1 \twoheadrightarrow_1 a'_1$ *and* $a_2 \twoheadrightarrow_1 a'_2$*;*
   (b) $a_1 \equiv \lambda x.b$, $e \equiv b'[x := a'_2]$ *with* $b \twoheadrightarrow_1 b'$ *and* $a_2 \twoheadrightarrow_1 a'_2$*;*
   (c) *or* $a_1 \equiv \mathsf{letrec}\ x = b$, $a_2 \equiv (c\,\vec{e})$, $e \equiv b'[x := \mathsf{letrec}\ x = b']\,(c\,\vec{e'})$ *with* $b \twoheadrightarrow_1 b'$ *and* $e_i \twoheadrightarrow_1 e'_i$, $i = 1..\mathsf{ar}(c)$ *.*
3. $\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 e$ *implies either:*
   (a) $e \equiv \mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\}$ *with* $a \twoheadrightarrow_1 a'$ *and* $b_i \twoheadrightarrow_1 b'_i$*;*
   (b) *or* $a \equiv c_k\,\vec{a}$, $e \equiv b'_k\,\vec{a'}$ *with* $b_k \twoheadrightarrow_1 b'_k$ *and* $a_j \twoheadrightarrow_1 a'_j$, $j = 1..\mathsf{ar}(c_k)$*.*
4. $\mathsf{letrec}\ x = b \twoheadrightarrow_1 e$ *implies* $e \equiv \mathsf{letrec}\ x = b'$ *with* $b \twoheadrightarrow_1 b'$*.*

Proof. By induction on the definition of $\twoheadrightarrow_1$. $\square$

Lemma 4.1.6 $\twoheadrightarrow_1$ *satisfies the diamond property, i.e.,*

$$a \twoheadrightarrow_1 a_1 \;\wedge\; a \twoheadrightarrow_1 a_2 \;\;\Rightarrow\;\; \exists\, a_3 \in \mathcal{E}.\; a_1 \twoheadrightarrow_1 a_3 \;\wedge\; a_2 \twoheadrightarrow_1 a_3$$

Proof. By induction on the definition of $a \twoheadrightarrow_1 a_1$. We focus on the cases of clauses 5 and 6.

5. Assume $a \twoheadrightarrow_1 a_1$ is $\mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 \mathsf{case}\ e'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\}$ and is a direct consequence of $e \twoheadrightarrow_1 e'$, $b_i \twoheadrightarrow_1 b'_i$ with $i = 1..n$. By Lemma 4.1.5, there are two possibilities:

   1. $a_2 \equiv \mathsf{case}\ e''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b''}\}$ and $e \twoheadrightarrow_1 e''$, $b_i \twoheadrightarrow_1 b''_i$. By induction hypothesis, there are e''', b'''_i such that $e' \twoheadrightarrow_1 e'''$, $e'' \twoheadrightarrow_1 e'''$ and $b'_i \twoheadrightarrow_1 b'''_i$, $b''_i \twoheadrightarrow_1 b'''_i$. Hence we can take $a_3 \equiv \mathsf{case}\ e'''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'''}\}$.
   2. $e \equiv c_k\, \vec{a''}$ with $a_j \twoheadrightarrow_1 a''_j$, $j = 1..\mathsf{ar}(c_k)$, $b_k \twoheadrightarrow_1 b''_k$ and $a_2 \equiv b''_k\, \vec{a''}$. By induction hypothesis, there is b'''_k such that $b'_k \twoheadrightarrow_1 b'''_k$, $b''_k \twoheadrightarrow_1 b'''_k$. By Lemma 4.1.5 one has $e' \equiv c_k\, \vec{a'''}$. Hence we can take $a_3 \equiv b'''_k\, \vec{a'''}$.

6. Assume $a \twoheadrightarrow_1 a_1$ is $\mathsf{case}\ (c_k\, \vec{a})\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 b'_k\, \vec{a'}$ and is a direct consequence of $a_j \twoheadrightarrow_1 a'_j$, $j = 1..\mathsf{ar}(c_k)$, $b_k \twoheadrightarrow_1 b'_k$. By Lemma 4.1.5 one can distinguish two cases:

   1. $a_2 \equiv \mathsf{case}\ (c_k\, \vec{a''})\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b''}\}$, $a_j \twoheadrightarrow_1 a''_j$, $b_i \twoheadrightarrow_1 b''_i$. By induction hypothesis, there are a'''_j, b'''_i such that $a'_j \twoheadrightarrow_1 a'''_j$, $a''_j \twoheadrightarrow_1 a'''_j$, $b'_i \twoheadrightarrow_1 b'''_i$, $b''_i \twoheadrightarrow_1 b'''_i$. Hence we can take $a_3 \equiv b'''_k\, \vec{a'''}$.
   2. $a_2 \equiv b''_k\, \vec{a''}$, $b_k \twoheadrightarrow_1 b''_k$, $a_j \twoheadrightarrow_1 a''_j$, $j = 1..\mathsf{ar}(c_k)$. By induction hypothesis, there are a'''_j, b'''_k such that $a'_j \twoheadrightarrow_1 a'''_j$, $a''_j \twoheadrightarrow_1 a'''_j$, $b'_k \twoheadrightarrow_1 b'''_k$, $b''_k \twoheadrightarrow_1 b'''_k$. Hence we can take $a_3 \equiv b'''_k\, \vec{a'''}$.

□

Lemma 4.1.7 *$\twoheadrightarrow_{\beta\iota\mu}$ is the transitive closure of $\twoheadrightarrow_1$.*

Proof. $\twoheadrightarrow_1$ contains the reflexive closure of $\rightarrow_{\beta\iota\mu}$. Moreover, $\twoheadrightarrow_1 \subseteq \twoheadrightarrow_{\beta\iota\mu}$. Since $\twoheadrightarrow_{\beta\iota\mu}$ is the reflexive-transitive closure of $\rightarrow_{\beta\iota\mu}$ it is also the transitive closure of $\twoheadrightarrow_1$. □

Theorem 4.1.8 (Confluence) *$\rightarrow_{\beta\iota\mu}$ is confluent:*

$$a_1 =_{\beta\iota\mu} a_2 \quad \Rightarrow \quad \exists\, e \in \mathcal{E}.\ a_1 \twoheadrightarrow_{\beta\iota\mu} e \ \wedge\ a_2 \twoheadrightarrow_{\beta\iota\mu} e$$

Proof. Assume $a_1 =_{\beta\iota\mu} a_2$, then $\exists\, a \in \mathcal{E}.\ a \twoheadrightarrow_{\beta\iota\mu} a_1 \ \wedge\ a \twoheadrightarrow_{\beta\iota\mu} a_2$. As $\twoheadrightarrow_{\beta\iota\mu}$ is the transitive closure of $\twoheadrightarrow_1$, $\twoheadrightarrow_{\beta\iota\mu}$ satisfies also the diamond property, by Lemma 4.1.2. So, we conclude. □

Corollary 4.1.9 (Uniqueness of normal forms) *Any expression $e \in \mathcal{E}$ has at most one normal form.*

Proof. From Theorem 4.1.8, by absurdity with the assumption that a term could have two different normal forms. □

4.2 Subject Reduction

The proof of subject reduction for $\lambda^{\frown}$ is quite standard. Before going into this proof, some lemmata involving monotonicity and substitution properties for stages, as well as generation and substitution properties for typing and subtyping, are considered.

Lemma 4.2.1 (Generation lemma for subtyping)

1. $\sigma \sqsubseteq \tau_1 \to \tau_2 \quad \Rightarrow \quad \sigma \equiv \tau_1' \to \tau_2' \;\wedge\; \tau_1 \sqsubseteq \tau_1' \;\wedge\; \tau_2' \sqsubseteq \tau_2$
2. $\tau_1 \to \tau_2 \sqsubseteq \sigma \quad \Rightarrow \quad \sigma \equiv \tau_1' \to \tau_2' \;\wedge\; \tau_1' \sqsubseteq \tau_1 \;\wedge\; \tau_2 \sqsubseteq \tau_2'$
3. $\theta \sqsubseteq d^s\vec{\tau} \quad \Rightarrow \quad \theta \equiv d^r\vec{\sigma} \;\wedge\; r \preccurlyeq s \;\wedge\; \vec{\sigma} \leq \vec{\tau}$
4. $d^s\vec{\tau} \sqsubseteq \theta \quad \Rightarrow \quad \theta \equiv d^r\vec{\sigma} \;\wedge\; s \preccurlyeq r \;\wedge\; \vec{\tau} \leq \vec{\sigma}$
5. $\alpha \sqsubseteq \sigma \quad \Rightarrow \quad \sigma \equiv \alpha$
6. $\sigma \sqsubseteq \alpha \quad \Rightarrow \quad \sigma \equiv \alpha$

Proof. Immediate by analysis of the subtyping rules. □

Lemma 4.2.2 (Generation lemma for typing)

1. $\Gamma \vdash x : \sigma \quad \Rightarrow \quad (x : \tau) \in \Gamma \;\wedge\; \tau \sqsubseteq \sigma$
2. $\Gamma \vdash a\, b : \sigma \quad \Rightarrow \quad \Gamma \vdash a : \tau \to \sigma' \;\wedge\; \Gamma \vdash b : \tau \;\wedge\; \sigma' \sqsubseteq \sigma$
3. $\Gamma \vdash \lambda x.e : \sigma \quad \Rightarrow \quad \sigma \equiv \tau_1 \to \tau_2 \;\wedge\; \Gamma, x : \tau_1' \vdash e : \tau_2' \;\wedge\; \tau_1 \sqsubseteq \tau_1' \;\wedge\; \tau_2' \sqsubseteq \tau_2$
4. $\Gamma \vdash c : \sigma \quad \Rightarrow \quad \sigma \equiv \vec{\gamma} \to \theta \;\wedge\; \vec{\gamma} \sqsubseteq \mathsf{Inst}^s_{\vec{\tau}}(c) \;\wedge\; d^{\widehat{s}}\vec{\tau} \sqsubseteq \theta \;\wedge\; c \in \mathsf{C}(d)$
5. $\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} : \sigma \quad \Rightarrow \quad \Gamma \vdash a : d^{\widehat{s}}\vec{\tau} \;\wedge\; \Gamma \vdash b_i : \mathsf{Inst}^s_{\vec{\tau}}(c_i) \to \theta \;\wedge\; \theta \sqsubseteq \sigma$
6. $\Gamma \vdash \mathsf{letrec}\ f = e : \sigma \quad \Rightarrow \quad \Gamma, f : d^\imath\vec{\tau} \to \theta \vdash e : (d^\imath\vec{\tau} \to \theta)[\imath := \widehat{\imath}] \;\wedge\; (d^\imath\vec{\tau} \to \theta)[\imath := s] \sqsubseteq \sigma$ *with* $\imath \in \mathcal{V}_\mathcal{S}$, $\imath\ \mathsf{pos}\ \theta$ *and* $\imath$ *fresh in* $\Gamma, \vec{\tau}$

Proof. By inspection on the derivation of the antecedent judgments. □

Lemma 4.2.3

1. *If* $\imath\ \mathsf{pos}\ \theta$ *and* $r \preccurlyeq s$, *then* $\theta[\imath := r] \leq \theta[\imath := s]$.
2. *If* $\imath\ \mathsf{neg}\ \theta$ *and* $r \preccurlyeq s$, *then* $\theta[\imath := s] \leq \theta[\imath := r]$.

Proof. By simultaneous induction on the structure of θ. □

Lemma 4.2.4

1. *If* $\sigma \leq \sigma'$, $\tau_i \leq \tau_i'$ *and* $\alpha_i\ \mathsf{pos}\ \sigma$ *for* $i = 1..n$, *then* $\sigma[\vec{\alpha} := \vec{\tau}] \leq \sigma'[\vec{\alpha} := \vec{\tau'}]$.
2. *If* $\sigma \leq \sigma'$, $\tau_i \leq \tau_i'$ *and* $\alpha_i\ \mathsf{neg}\ \sigma$ *for* $i = 1..n$, *then* $\sigma[\vec{\alpha} := \vec{\tau'}] \leq \sigma'[\vec{\alpha} := \vec{\tau}]$.

Proof. By simultaneous induction on the structure of σ. □

Lemma 4.2.5 *If* $r \preccurlyeq s$ *and* $\vec{\tau} \leq \vec{\sigma}$, *then* $\mathsf{Dom}^r_{\vec{\tau}}(c) \leq \mathsf{Dom}^s_{\vec{\sigma}}(c)$.

Proof. Follows from the conditions imposed on declarations of constructors (see Definition 3.3.5) and from lemmas 4.2.3 and 4.2.4. □

Lemma 4.2.6 (Substitution lemma)
If $\Gamma, x : \tau \vdash a : \sigma$ *and* $\Gamma \vdash b : \tau$*, then* $\Gamma \vdash a[x := b] : \sigma$*.*

Proof. By induction on the derivation of $\Gamma, x : \tau \vdash a : \sigma$. □

The following lemma shows the polymorphic nature of stage variables. In fact, in a derivable judgment a stage variable can be replaced throughout by a stage without affecting derivability.

Lemma 4.2.7 *If* $\Gamma \vdash a : \sigma$ *then* $\Gamma[\imath := s] \vdash a : \sigma[\imath := s]$.

Proof. Without loss of generality, one can assume $\imath$ nocc s, otherwise one could firstly apply this weaker version of the lemma with $\imath$ being replaced by a new stage variable κ (for the set of stage variables is infinite) and use again the weaker version of the lemma with κ replaced by s.

By induction on the derivation of $\Gamma \vdash a : \sigma$. The only interesting case is when the last rule applied is (rec). (The other cases can be easily proved using the induction hypothesis.) Assume the last step is

$$\frac{\Gamma, f : d^{\jmath}\vec{\tau} \to \theta \vdash e : d^{\widehat{\jmath}}\vec{\tau} \to \theta[\jmath := \widehat{\jmath}] \qquad \jmath \text{ pos } \theta}{\Gamma \vdash (\mathsf{letrec}\ f = e) : d^{r}\vec{\tau} \to \theta[\jmath := r]} \qquad \jmath \text{ fresh in } \Gamma, \vec{\tau}$$

A stage variable κ can be chosen such that κ is fresh in $\Gamma, \vec{\tau}, \theta$, $\kappa \neq \imath$ and κ nocc s. Then, from the induction hypothesis, $\Gamma, f : d^{\kappa}\vec{\tau} \to \theta[\jmath := \kappa] \vdash e : d^{\widehat{\kappa}}\vec{\tau} \to \theta[\jmath := \widehat{\jmath}][\jmath := \kappa]$. Therefore $\Gamma, f : d^{\kappa}\vec{\tau} \to \theta[\jmath := \kappa] \vdash e : d^{\widehat{\kappa}}\vec{\tau} \to \theta[\jmath := \kappa][\kappa := \widehat{\kappa}]$ and therefore, using again the induction hypothesis,

$$\begin{aligned}\Gamma[\imath := s],\quad & f : d^{\kappa}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s] \vdash \\ & e : d^{\widehat{\kappa}}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\kappa := \widehat{\kappa}][\imath := s]\end{aligned}$$

Since κ nocc s and $\kappa \neq \imath$, the substitutions $[\kappa := \widehat{\kappa}]$ and $[\imath := s]$ can be exchanged, obtaining

$$\begin{aligned}\Gamma[\imath := s],\quad & f : d^{\kappa}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s] \vdash \\ & e : d^{\widehat{\kappa}}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s][\kappa := \widehat{\kappa}]\end{aligned} \tag{4.1}$$

and, as κ is fresh in $\Gamma[\imath := s]$ and in $\vec{\tau}[\imath := s]$, one can apply the rule (rec).

Let $u \equiv r[\imath := s]$. From (4.1) by (rec),

$$\Gamma[\imath := s] \vdash (\mathsf{letrec}\ f = e) : d^{u}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s][\kappa := u]$$

So, as κ nocc s, $\imath$ nocc s and κ is θ-fresh, $\Gamma[\imath := s] \vdash (\mathsf{letrec}\ f = e) : (d^{r}\vec{\tau})[\imath := s] \to \theta[\jmath := u][\imath := s]$. Hence, $\Gamma[\imath := s] \vdash (\mathsf{letrec}\ f = e) : (d^{r}\vec{\tau} \to \theta[\jmath := r])[\imath := s]$. □

We are now ready to prove that $\lambda^{\widehat{}}$ enjoys the property of subject reduction.

Theorem 4.2.8 (Subject reduction)

$$\Gamma \vdash e_1 : \sigma \quad \wedge \quad e_1 \to_{\beta\iota\mu} e_2 \quad \Rightarrow \quad \Gamma \vdash e_2 : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash e_1 : \sigma$. The interesting cases are when the last rule applied is (app) or (case):

(app) Assume $e_1 \equiv a\, b$ and the last step is

$$\frac{\Gamma \vdash a : \tau_1 \to \tau_2 \qquad \Gamma \vdash b : \tau_1}{\Gamma \vdash a\, b : \tau_2}$$

Then one may have the following cases:

$e_2 \equiv e[x := b]$, with $a \equiv \lambda x.e$. From the typing derivation for a, using Lemma 4.2.2, follows that $\Gamma, x : \tau_1' \vdash e : \tau_2'$, $\tau_1 \sqsubseteq \tau_1'$ and $\tau_2' \sqsubseteq \tau_2$, and from the typing derivation for b, by (sub) one derives $\Gamma \vdash b : \tau_1'$. Thus, by Lemma 4.2.6, $\Gamma \vdash e[x := b] : \tau_2'$ and finally by the rule (sub), $\Gamma \vdash e[x := b] : \tau_2$.

$e_2 \equiv (e[f := (\mathsf{letrec}\ f = e)])\,(c\,\vec{a})$, with $a \equiv (\mathsf{letrec}\ f = e)$ and $b \equiv (c\,\vec{a})$. Applying Lemma 4.2.2 to the typing derivation for a one has:

$$\Gamma, f : d^{\imath}\vec{\tau} \to \theta \vdash e : (d^{\imath}\vec{\tau} \to \theta)[\imath := \widehat{\imath}] \tag{4.2}$$

$$(d^{\imath}\vec{\tau} \to \theta)[\imath := s] \sqsubseteq \tau_1 \to \tau_2 \tag{4.3}$$

$$\imath \in \mathcal{V}_{\mathcal{S}} \ \wedge\ \imath \ \mathsf{pos}\ \theta \ \wedge\ \imath \text{ fresh in } \Gamma, \vec{\tau} \tag{4.4}$$

From (4.3) by Lemma 4.2.1, $\tau_1 \leq d^{s}\vec{\tau} \ \wedge\ \theta[\imath := s] \leq \tau_2$ and thus, using again Lemma 4.2.1,

$$\tau_1 \equiv d^{p}\vec{\tau'} \quad \wedge \quad p \preccurlyeq s \quad \wedge \quad \vec{\tau'} \leq \vec{\tau}$$

From (4.2) and (4.4), by (rec), $\Gamma \vdash (\mathsf{letrec}\ f = e) : (d^{\imath}\vec{\tau} \to \theta)[\imath := q]$ holds for an arbitrary stage q. Therefore, choosing $q \equiv \imath$ and taking into account (4.2), by Lemma 4.2.6 one can derive

$$\Gamma \vdash e[f := (\mathsf{letrec}\ f = e)] : (d^{\imath}\vec{\tau} \to \theta)[\imath := \widehat{\imath}] \tag{4.5}$$

Thus we have $\Gamma \vdash (c\,\vec{a}) : d^{p}\vec{\tau'}$ and so, by lemmas 4.2.1 and 4.2.2, one of two possibilities for p must arise: $p \equiv \jmath^{n}$ with $n \geq 1$ or $p \equiv \infty^{m}$ with $m \geq 0$, where for a stage s and for $k \in \mathbb{N}$, s^{k} means s hatted k times.

- Case $p \equiv \jmath^{n}$ with $n \geq 1$ then, using Lemma 4.2.7 on (4.5) with substitution $[\imath := \jmath^{(n-1)}]$, and since $\imath$ is fresh w.r.t. Γ and $\vec{\tau}$,

$$\Gamma \vdash e[f := (\mathsf{letrec}\ f = e)] : d^{\jmath^{n}}\vec{\tau} \to \theta[\imath := \jmath^{n}]$$

  Thus, since $\Gamma \vdash (c\,\vec{a}) : \tau_1$ and $\tau_1 \equiv d^{p}\vec{\tau'}$, by (sub) and (app), follows $\Gamma \vdash e[f := (\mathsf{letrec}\ f = e)]\,(c\,\vec{a}) : \theta[\imath := \jmath^{n}]$. One has $\jmath^{n} \preccurlyeq s$ and $\imath\ \mathsf{pos}\ \theta$ so, by Lemma 4.2.3, $\theta[\imath := \jmath^{n}] \leq \theta[\imath := s]$ and the proof of this case is concluded using the rule (sub).
- Case $p \equiv \infty^{m}$ with $m \geq 0$ then, observe that by (sub) $\Gamma \vdash (c\,\vec{a}) : d^{\infty^{(m+1)}}\vec{\tau'}$, and the proof could now be completed arguing as in the previous case.

The remaining cases, where $e_2 \equiv a'\,b$ with $a \to_{\beta\iota\mu} a'$, or $e_2 \equiv a\,b'$ with $b \to_{\beta\iota\mu} b'$, follow by routine induction.

(case) Assume $e_1 \equiv \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ and the last step is

$$\frac{\Gamma \vdash a : d^{\widehat{s}}\vec{\tau} \qquad \Gamma \vdash b_i : \mathsf{Dom}^{s}_{\vec{\tau}}(c_i) \to \theta \qquad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \theta}$$

Then one may have:

$e_2 \equiv b_i\,a_1 \ldots a_{\mathsf{ar}(c_i)}$, with $a \equiv c_i\,a_1 \ldots a_{\mathsf{ar}(c_i)}$. From $\Gamma \vdash c_i\,a_1 \ldots a_{\mathsf{ar}(c_i)} : d^{\widehat{s}}\vec{\tau}$, by Lemma 4.2.2, it follows that

$$\Gamma \vdash c_i : \vec{\gamma} \to \sigma \quad \wedge \quad \sigma \leq d^{\widehat{s}}\vec{\tau}$$

and also for $1 \leq j \leq \mathsf{ar}(c_i)$

$$\Gamma \vdash a_j : \gamma_j \quad \wedge \quad \gamma_j \leq \mathsf{Dom}^{r}_{\vec{\psi}}(c_i)[j] \quad \wedge \quad d^{\widehat{r}}\vec{\psi} \leq \sigma$$

So, $d^{\widehat{r}}\vec{\psi} \leq d^{\widehat{s}}\vec{\tau}$ and therefore, by lemmas 4.2.1 and 3.3.11, $r \preccurlyeq s$ and $\vec{\psi} \leq \vec{\tau}$. Using Lemma 4.2.5 and the (sub) rule, one has $\Gamma \vdash a_j : \mathsf{Dom}^{s}_{\vec{\tau}}(c_i)[j]$ for $1 \leq j \leq \mathsf{ar}(c_i)$, which can be combined with the typing derivation of b_i, by means of the rule (app), to conclude the proof of this case.

The remaining cases, $e_2 \equiv \mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ with $a \rightarrow_{\beta\iota\mu} a'$, and $e_2 \equiv \mathsf{case}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b'_i \mid \ldots \mid c_n \Rightarrow b_n\}$ with $b_i \rightarrow_{\beta\iota\mu} b'_i$, follow by routine induction.

□

4.3 Strong Normalization

In $\lambda^{\frown}$ every computation sequence starting from a well-typed term terminates. That is, any strategy for computing a well-typed term leads to a term that cannot be further computed. This property is known as *strong normalization*, as we say that a term e is *strongly normalizing* with respect to $\rightarrow_{\beta\iota\mu}$, if all $\beta\iota\mu$-reduction sequences starting with e terminate. Recall that SN denotes the set of terms that are strongly normalizing with respect to $\rightarrow_{\beta\iota\mu}$.

This subsection is devoted to prove that strong normalization holds for $\lambda^{\frown}$. To prove that every typable term is in SN we use the standard technique [90] of constructing a model based on saturated sets. The method consists in developing a semantics for the typing calculus in which terms are interpreted as terms and types as sets of terms known by construction only to contain strongly normalizing terms and always be non-empty (saturated sets). Under this interpretation a typing assertion $e : \sigma$ is viewed as the set membership $e \in [\![\sigma]\!]$, where $[\![\sigma]\!]$ is the saturated set associated with the type σ. Since saturated sets only contain strong normalizing terms, the strong normalizability of all typable terms follows from the soundness of the interpretation.

4.3.1 Saturated sets and interpretation domains

The formal definition of saturated set makes use of two auxiliary notions: base term and key reduction. This way one makes explicit the ideas behind the concept of saturated set, and also gets a generic definition (to consider new terms and types, one only needs to extend the definition of base term and key reduction). We start by defining the notions of base terms and key reduction, and state some of their properties.

Definition 4.3.1 (Base terms) *The set* Base *of* base terms *is defined inductively as follows:*

1. $\mathcal{V}_{\mathcal{E}} \subseteq \mathsf{Base}$.
2. *If* $b \in \mathsf{Base}$ *and* $e \in \mathsf{SN}$ *then* $b\,e \in \mathsf{Base}$.
3. *If* $b \in \mathsf{Base}$ *and* $e_1, \ldots, e_n \in \mathsf{SN}$*, then* $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \in \mathsf{Base}$.
4. *If* $b \in \mathsf{Base}$ *and* $e \in \mathsf{SN}$ *then* $(\mathsf{letrec}\ f = e)\,b \in \mathsf{Base}$.

Informally, a base term is a term whose reduction is stopped by the occurrence of a variable and whose subterms are strongly normalizing. Every base term is strongly normalizing.

Lemma 4.3.2 $\mathsf{Base} \subseteq \mathsf{SN}$

Proof. By induction on the structure of base terms. □

Definition 4.3.3 (Key reduction) *The relation of* key reduction *between terms is defined inductively as follows:*

1. *If e is a $\beta\iota\mu$-redex and e' is the contractum, then $e \rightarrow_k e'$.*
2. *If $a \rightarrow_k a'$, then $a\, e \rightarrow_k a'\, e$,*
3. *If $a \rightarrow_k a'$, then* $\mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \rightarrow_k \mathsf{case}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}$.
4. *If $a \rightarrow_k a'$, then* $(\mathsf{letrec}\ f = e)\, a \rightarrow_k (\mathsf{letrec}\ f = e)\, a'$.

The redex reduced by a key reduction is called key-redex.

Any term has at most one key-redex. The interest behind the notion of key reduction is that for computing a normal form of a term that has a key redex one has necessarily to perform a key reduction. Key reduction commutes with reduction in the following sense.

Lemma 4.3.4 *If $a \rightarrow_k b$ and $a \rightarrow_{\beta\iota\mu} a' \neq b$ by a non-key reduction, then $a' \rightarrow_k b'$ and $b \twoheadrightarrow_{\beta\iota\mu} b'$ for some b'.*

Proof. By induction on the structure of a. Assume $a \rightarrow_k b$ and $a \rightarrow_{\beta\iota\mu} a'$ by a non-key reduction. There are five different possibilities for a and b:

1. $a \equiv (\lambda x.e)\, e'$ and $b \equiv e[x := e']$. In this case one has two possible forms for a':
   - $a' \equiv (\lambda x.e'')\, e'$ and $e \rightarrow_{\beta\iota\mu} e''$. We have $a' \rightarrow_k e''[x := e']$ and, by Lemma 3.2.11, $b \equiv e[x := e'] \rightarrow_{\beta\iota\mu} e''[x := e']$. Hence $b' \equiv e''[x := e']$.
   - $a' \equiv (\lambda x.e)\, e''$ and $e' \rightarrow_{\beta\iota\mu} e''$. We have $a' \rightarrow_k e[x := e'']$ and, by Lemma 3.2.11, $b \equiv e[x := e'] \rightarrow_{\beta\iota\mu} e[x := e'']$. So, $b' \equiv e'[x := e'']$.
2. $a \equiv a_1\, a_2$, $a_1 \rightarrow_k a_1'$ and $b \equiv a_1'\, a_2$. In this case one has two possible forms for a':
   - $a' \equiv a_1\, a_2'$ and $a_2 \rightarrow_{\beta\iota\mu} a_2'$. We have $a' \rightarrow_k a_1'\, a_2'$ and $b \equiv a_1'\, a_2 \rightarrow_{\beta\iota\mu} a_1'\, a_2'$. So, $b' \equiv a_1'\, a_2'$.
   - $a' \equiv a_1''\, a_2$ and $a_1 \rightarrow_{\beta\iota\mu} a_1''$ by a non-key reduction. We have $a_1 \rightarrow_k a_1'$ and $a_1 \rightarrow_{\beta\iota\mu} a_1''$ by a non-key reduction, then, by induction hypothesis, there is a term e such that $a_1'' \rightarrow_k e$ and $a_1' \twoheadrightarrow_{\beta\iota\mu} e$. Therefore, $a' \equiv a_1''\, a_2 \rightarrow_k e\, a_2$ and $b \equiv a_1'\, a_2 \twoheadrightarrow_{\beta\iota\mu} e\, a_2$. Hence $b' \equiv e\, a_2$.
3. $a \equiv \mathsf{case}\ (c_i\, \vec{e})\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ and $b \equiv b_i\, \vec{e}$. In this case one has two possible forms for a':
   - $a' \equiv \mathsf{case}\ (c_i\, e_1 \ldots e_j' \ldots e_m)\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ and $e_j \rightarrow_{\beta\iota\mu} e_j'$. We have $a' \rightarrow_k b_i\, e_1 \ldots e_j' \ldots e_m$ and $b \equiv b_i\, e_1 \ldots e_j \ldots e_m \rightarrow_{\beta\iota\mu} b_i\, e_1 \ldots e_j' \ldots e_m$. So, $b' \equiv b_i\, e_1 \ldots e_j' \ldots e_m$.
   - $a' \equiv \mathsf{case}\ (c_i\, \vec{e})\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_j \Rightarrow b_j' \mid \ldots c_n \Rightarrow b_n\}$ and $b_j \rightarrow_{\beta\iota\mu} b_j'$. We have:
     - if $i \neq j$ then $a' \rightarrow_k b_i\, \vec{e}$. So, trivially, $b' \equiv b_i\, \vec{e}$.
     - if $i = j$ then $a' \rightarrow_k b_j'\, \vec{e}$ and $b \equiv b_j\, \vec{e} \rightarrow_{\beta\iota\mu} b_j'$. So, $b' \equiv b_j'\, \vec{e}$.
4. $a \equiv \mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$, $e \rightarrow_k e'$ and $b \equiv \mathsf{case}\ e'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$. In this case one has two possible forms for a':
   - $a' \equiv \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_j \Rightarrow b_j' \mid \ldots c_n \Rightarrow b_n\}$ and $b_j \rightarrow_{\beta\iota\mu} b_j'$. We have $a' \rightarrow_k \mathsf{case}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_j \Rightarrow b_j' \mid \ldots c_n \Rightarrow b_n\}$ and $b \rightarrow_{\beta\iota\mu} \mathsf{case}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_j \Rightarrow b_j' \mid \ldots c_n \Rightarrow b_n\}$. So, $b' \equiv \mathsf{case}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_j \Rightarrow b_j' \mid \ldots c_n \Rightarrow b_n\}$.

- $a' \equiv \mathsf{case}\ e''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ and $e \rightarrow_{\beta\iota\mu} e''$ by a non-key reduction. By induction hypothesis, there is a term e''' such that $e'' \rightarrow_k e'''$ and $e' \twoheadrightarrow_{\beta\iota\mu} e'''$. Therefore, $a' \rightarrow_k \mathsf{case}\ e'''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ and $b \twoheadrightarrow_{\beta\iota\mu} \mathsf{case}\ e'''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$. So, $b' \equiv \mathsf{case}\ e'''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$.

5. $a \equiv (\mathsf{letrec}\ f = e)\,(c\,\vec{a})$ and $b \equiv e[f := (\mathsf{letrec}\ f = e)]\,(c\,\vec{a})$. In this case one has two possible forms for a':

   - $a' \equiv (\mathsf{letrec}\ f = e')\,(c\,\vec{a})$ and $e \rightarrow_{\beta\iota\mu} e'$. We have $a' \rightarrow_k e'[f := (\mathsf{letrec}\ f = e')]\,(c\,\vec{a})$ and, by Lemma 3.2.11, $b \equiv e[f := (\mathsf{letrec}\ f = e)]\,(c\,\vec{a}) \rightarrow_{\beta\iota\mu} e'[f := (\mathsf{letrec}\ f = e')]\,(c\,\vec{a})$. Hence $b' \equiv e'[f := (\mathsf{letrec}\ f = e')]\,(c\,\vec{a})$.
   - $a' \equiv (\mathsf{letrec}\ f = e)\,(c\,a_1 ... a_i ... a_{\mathsf{ar}(c)})$ and $a_i \rightarrow_{\beta\iota\mu} a_i'$. We have $a' \rightarrow_k e[f := (\mathsf{letrec}\ f = e)]\,(c\,a_1 ... a_i' ... a_{\mathsf{ar}(c)})$ and, by Lemma 3.2.11, $b \equiv e[f := (\mathsf{letrec}\ f = e)]\,(c\,\vec{a}) \rightarrow_{\beta\iota\mu} e[f := (\mathsf{letrec}\ f = e)]\,(c\,a_1 ... a_i' ... a_{\mathsf{ar}(c)})$. So, $b' \equiv e[f := (\mathsf{letrec}\ f = e)]\,(c\,a_1 ... a_i' ... a_{\mathsf{ar}(c)})$.

$\square$

The following two lemmas provide sufficient conditions for an expression to be strongly normalizing.

Lemma 4.3.5

1. *If* $a \in \mathsf{SN}$, $a \rightarrow_k a'$ *and* $a'\,b \in \mathsf{SN}$, *then* $a\,b \in \mathsf{SN}$.
2. *If* $a \in \mathsf{SN}$, $a \rightarrow_k a'$ *and* $\mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$, *then* $\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$.
3. *If* $a \in \mathsf{SN}$, $a \rightarrow_k a'$ *and* $(\mathsf{letrec}\ f = e)\,a' \in \mathsf{SN}$, *then* $(\mathsf{letrec}\ f = e)\,a \in \mathsf{SN}$.

Proof.

1. By induction on the sum of the lengths of the $\beta\iota\mu$-reduction chains of a and b. We have $a \in \mathsf{SN}$, $a \rightarrow_k a'$ and $a'\,b \in \mathsf{SN}$. So, $b \in \mathsf{SN}$ and $a' \in \mathsf{SN}$. We prove that $a\,b \in \mathsf{SN}$ showing that every term e such that $a\,b \rightarrow_{\beta\iota\mu} e$ is strong normalizing. Since a has a key-redex, a cannot be a λ-abstraction. Therefore there are the following cases to consider:

   - $e \equiv a'\,b \in \mathsf{SN}$ by hypothesis.
   - $e \equiv a''\,b$ and $a \rightarrow_{\beta\iota\mu} a''$ by a non-key reduction. By Lemma 4.3.4, there is a term a''' such that $a' \twoheadrightarrow_{\beta\iota\mu} a'''$ and $a'' \rightarrow_k a'''$. We have $a'' \in \mathsf{SN}$ (since $a \in \mathsf{SN}$ and $a \rightarrow_{\beta\iota\mu} a''$), $a'' \rightarrow_k a'''$ and $a'''\,b \in \mathsf{SN}$ (since $a'\,b \in \mathsf{SN}$ and $a' \twoheadrightarrow_{\beta\iota\mu} a'''$). So, by induction hypothesis, $e \equiv a''\,b \in \mathsf{SN}$.
   - $e \equiv a\,b'$ and $b \rightarrow_{\beta\iota\mu} b'$. We have $a \in \mathsf{SN}$, $a \rightarrow_k a'$ and $a'\,b' \in \mathsf{SN}$ (since $a'\,b \in \mathsf{SN}$ and $b \rightarrow_{\beta\iota\mu} b'$). So, by induction hypothesis, $e \equiv a\,b' \in \mathsf{SN}$.

2. By induction on the sum of the lengths of the $\beta\iota\mu$-reduction chains of a and b_i with $1 \leq i \leq \#\vec{b}$. We have $a \in \mathsf{SN}$, $a \rightarrow_k a'$ and $\mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$. So, $a' \in \mathsf{SN}$ and $b_i \in \mathsf{SN}$ for $1 \leq i \leq \#\vec{b}$. We prove that $\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$ showing that every term e such that $\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \rightarrow_{\beta\iota\mu} e$ is strong normalizing. Since a has a key-redex, a cannot be of the form $(c_i\,\vec{e})$. In consequence, there are the following cases to consider:

   - $e \equiv \mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$ by hypothesis.

- $e \equiv \mathsf{case}\ a''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ and $a \rightarrow_{\beta\iota\mu} a''$ by a non-key reduction. By Lemma 4.3.4, there is a term a''' such that $a' \twoheadrightarrow_{\beta\iota\mu} a'''$ and $a'' \rightarrow_k a'''$. We have $a'' \in \mathsf{SN}$ (since $a \rightarrow_{\beta\iota\mu} a''$ and $a \in \mathsf{SN}$), $a'' \rightarrow_k a'''$ and $\mathsf{case}\ a'''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$ (since $\mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$ and $a' \twoheadrightarrow_{\beta\iota\mu} a'''$). Hence, by induction hypothesis, $e \equiv \mathsf{case}\ a''\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$.
- $e \equiv \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b'_i \mid \ldots \mid c_n \Rightarrow b_n\}$ and $b_i \rightarrow_{\beta\iota\mu} b'_i$. We have $a \in \mathsf{SN}$, $a \rightarrow_k a'$ and $\mathsf{case}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b'_i \mid \ldots \mid c_n \Rightarrow b_n\} \in \mathsf{SN}$ (since $\mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \in \mathsf{SN}$ and $b_i \rightarrow_{\beta\iota\mu} b'_i$). So, by induction hypothesis, $e \equiv \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b'_i \mid \ldots \mid c_n \Rightarrow b_n\} \in \mathsf{SN}$.

3. By induction on the sum of the lengths of the $\beta\iota\mu$-reduction chains of a and e.

□

Lemma 4.3.6

1. *If $a, e, a[x := e] \in \mathsf{SN}$, then $(\lambda x.\ a)\ e \in \mathsf{SN}$.*
2. *If $\vec{a}, e_1, \ldots, e_n, e_i\ \vec{a} \in \mathsf{SN}$, then $\mathsf{case}\ (c_i\ \vec{a})\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \in \mathsf{SN}$.*
3. *If $\vec{a}, e, e[f := (\mathsf{letrec}\ f = e)]\ (c\ \vec{a}) \in \mathsf{SN}$, then $(\mathsf{letrec}\ f = e)\ (c\ \vec{a}) \in \mathsf{SN}$.*

Proof.

1. By induction on the sum of the lengths of the $\beta\iota\mu$-reduction chains of a and e. Assume $a, e, a[x := e] \in \mathsf{SN}$. We have to prove that $b \in \mathsf{SN}$ for any one b such that $(\lambda x.\, a)\, e \rightarrow_{\beta\iota\mu} b$. There are three cases to consider:
   - $b \equiv a[x := e] \in \mathsf{SN}$ by hypothesis.
   - $b \equiv (\lambda x.\, a')\, e$ with $a \twoheadrightarrow_{\beta\iota\mu} a'$. As $a \in \mathsf{SN}$ we have $a' \in \mathsf{SN}$. Moreover $a'[x := e] \in \mathsf{SN}$, because $a[x := e] \twoheadrightarrow_{\beta\iota\mu} a'[x := e]$ by Lemma 3.2.11 and $a[x := e] \in \mathsf{SN}$. Hence one has $(\lambda x.\, a')\, e \in \mathsf{SN}$ by induction hypothesis.
   - $b \equiv (\lambda x.\, a)\, e'$ with $e \twoheadrightarrow_{\beta\iota\mu} e'$. Similar.

Properties 2 and 3 are proved similarly. □

Next we define saturated sets and state some of their closure properties. Saturated sets are sets of strongly normalizing terms containing the base terms and closed with respect to key expansion.

Definition 4.3.7 (Saturated sets)

1. *A set $X \subseteq \mathcal{E}$ is said to be a* saturated set, *if*
   (a) $X \subseteq \mathsf{SN}$,
   (b) $\mathsf{Base} \subseteq X$,
   (c) *if $a \in \mathsf{SN}$ and $a \rightarrow_k a'$ for some $a' \in X$, then $a \in X$.*

   The set of all saturated sets is denoted by SAT.
2. *For any $X \subseteq \mathcal{E}$, let $\ulcorner X \urcorner = \{a \in \mathsf{SN} \mid \exists\, b \in \mathsf{Base} \cup X.\, a \twoheadrightarrow_k b\}$.*

Lemma 4.3.8 $a' \in \mathsf{SN}\ \wedge\ a' \twoheadrightarrow_k a\ \wedge\ a \in X\ \wedge\ X \in \mathsf{SAT}\ \Rightarrow\ a' \in X$

Proof. By induction on the length of the key-reduction sequence $a' \twoheadrightarrow_k a$. □

The following lemma establishes some basic properties of the closure operator $\ulcorner \cdot \urcorner$.

Lemma 4.3.9

1. *If $X \subseteq Y$, then $\ulcorner X \urcorner \subseteq \ulcorner Y \urcorner$.*
2. *If $X \subseteq \mathsf{SN}$, then $\ulcorner X \urcorner$ is a saturated set, in fact, the smallest saturated set containing X.*
3. *$\ulcorner X_1 \cup \ldots \cup X_n \urcorner = \ulcorner X_1 \urcorner \cup \ldots \cup \ulcorner X_n \urcorner$.*
4. *If X_i is a saturated set for any $i \in I$, then $\bigcup_{i \in I} X_i$ is a saturated set. (We say that $\bigcup_{i \in \emptyset} X_i = \ulcorner \emptyset \urcorner$.)*

Proof.

1. $a \in \ulcorner X \urcorner \;\Rightarrow\; \exists b \in \mathsf{Base} \cup X.\, a \twoheadrightarrow_k b \;\Rightarrow\; \exists b \in \mathsf{Base} \cup Y.\, a \twoheadrightarrow_k b \;\Rightarrow\; a \in \ulcorner Y \urcorner$
2. First, let us prove that $\ulcorner X \urcorner$ is a saturated set. Clearly, $\mathsf{Base} \subseteq \ulcorner X \urcorner \subseteq \mathsf{SN}$. Moreover, if $a \in \mathsf{SN}$ and $a \to_k a'$, for some $a' \in \ulcorner X \urcorner$, then $\exists b \in \mathsf{Base} \cup X.\, a' \twoheadrightarrow_k b$ and $a' \in \mathsf{SN}$. So, $a \twoheadrightarrow_k b$ and $a \in \mathsf{SN}$. Hence, $a \in \ulcorner X \urcorner$.

   Now, let us show that $\ulcorner X \urcorner$ the smallest saturated set containing X. Assume $X \subseteq Y \in \mathsf{SAT}$, to prove that $\ulcorner X \urcorner \subseteq Y$. If $a' \in \ulcorner X \urcorner$, then $\exists b \in \mathsf{Base} \cup X.\, a' \twoheadrightarrow_k b$ and $a' \in \mathsf{SN}$. Moreover, $b \in Y \in \mathsf{SAT}$. Hence, by Lemma 4.3.8, we have $a' \in Y$.
3. $\ulcorner X_1 \cup \ldots \cup X_n \urcorner = \{a \in \mathsf{SN} \mid \exists b \in \mathsf{Base} \cup X_1 \cup \ldots \cup X_n.\, a \twoheadrightarrow_k b\} = \bigcup_{i=1..n} \{a \in \mathsf{SN} \mid \exists b \in \mathsf{Base} \cup X_i.\, a \twoheadrightarrow_k b\} = \ulcorner X_1 \urcorner \cup \ldots \cup \ulcorner X_n \urcorner$
4. Assume $X_i \in \mathsf{SAT}$ for every $i \in I$. Let us prove the $\bigcup_{i \in I} X_i$ is a saturated set.
   - By hypothesis $X_i \subseteq \mathsf{SN}$ for every $i \in I$. So, $\bigcup_{i \in I} X_i \subseteq \mathsf{SN}$.
   - $\mathsf{Base} \subseteq X_i$ for every $i \in I$. Hence, $\mathsf{Base} \subseteq \bigcup_{i \in I} X_i$.
   - Let $a \in \mathsf{SN}$ and $a \to_k a'$, for some $a' \in \bigcup_{i \in I} X_i$. Thus, $a' \in X_i$ for some $i \in I$. As X_i is saturated, $a \in X_i$. Hence, $a \in \bigcup_{i \in I} X_i$.

□

On saturated sets, we can define a function-space forming operation. This is needed for the interpretation of function-space types.

Definition 4.3.10 *For any $X, Y \subseteq \mathcal{E}$, let $X \to Y = \{a \in \mathcal{E} \mid \forall e \in X.\, a\, e \in Y\}$. Moreover, $\vec{X} \to Y$ stands for $X_1 \to (X_2 \to (\ldots (X_n \to Y) \ldots))$.*

Lemma 4.3.11 *If $X' \subseteq X \subseteq \mathcal{E}$ and $Y \subseteq Y' \subseteq \mathcal{E}$, then $X \to Y \subseteq X' \to Y'$.*

Proof. $a \in X \to Y \;\Rightarrow\; \forall e \in X.\, a\, e \in Y \;\Rightarrow\; \forall e \in X'.\, a\, e \in Y \subseteq Y' \;\Rightarrow\; a \in X' \to Y'$ □

Lemma 4.3.12 *If $X, Y \in \mathsf{SAT}$, then $X \to Y \in \mathsf{SAT}$.*

Proof. Suppose X and Y are saturated. Let us check that $X \to Y$ satisfies the conditions of saturatedness:

- Clearly any $a \in X \to Y$ is strongly normalizing: as X is non-empty, we can pick some $e \in X$ and then $a \in \mathsf{SN}$ because $a\,e \in Y \subseteq \mathsf{SN}$.
- Suppose $b \in \mathsf{Base}$ and consider any $e \in X$. As $e \in \mathsf{SN}$, we have that $b\,e \in \mathsf{Base} \subseteq Y$. Hence $b \in X \to Y$.
- Suppose $a \in \mathsf{SN}$, $a \to_k a'$ and $a' \in X \to Y$. We have to show that $a \in X \to Y$, i.e., that, for any $e \in X$, $a\,e \in Y$. Consider any $e \in X$. We have $a\,e \to_k a'\,e$ and $a'\,e \in Y \subseteq \mathsf{SN}$, hence Lemma 4.3.5 applies and $a\,e \in \mathsf{SN}$. Since Y is saturated, we get $a\,e \in Y$.

□

Type and term interpretation

In what now follows, we define a semantics of the language of stages, types, and terms and show that the rules of stage comparison, subtyping and typing are sound with respect to that semantics. Types will be interpreted as saturated sets of terms, terms will be interpreted as terms.

We start with the definitions of valuations and interpretation for stages and types. Stages will be interpreted as ordinals below Ω, the first uncountable ordinal, types as saturated sets of terms. Inductive types are interpreted as limits of a monotone approximation process from below. As the universe, SN, is countable, the approximation process is guaranteed to converge before Ω.

Definition 4.3.13 (Stage valuation)

1. *A* stage valuation *is a map* $\pi : \mathcal{V}_\mathcal{S} \to \Omega + 1$.
2. *For every stage valuation* π, $\imath \in \mathcal{V}_\mathcal{S}$, *and* $x \in \Omega + 1$, *the stage valuation* $\pi(\imath := x)$ *is defined as follows:*

$$\pi(\imath := x)(\imath') = \begin{cases} x & \text{if } \imath' \equiv \imath \\ \pi(\imath') & \text{if } \imath' \not\equiv \imath \end{cases}$$

Definition 4.3.14 (Interpretation of stages) *Let* π *be a stage valuation. The corresponding stage interpretation function* $[\![.]\!]_\pi : \mathcal{S} \to \Omega + 1$ *is defined as follows:*

$$\begin{array}{rcl} [\![\imath]\!]_\pi & = & \pi(\imath) \ \text{ if } \imath \in \mathcal{V}_\mathcal{S} \\ [\![\infty]\!]_\pi & = & \Omega \\ [\![\widehat{s}]\!]_\pi & = & \begin{cases} [\![s]\!]_\pi + 1 & \text{if } [\![s]\!]_\pi < \Omega \\ [\![s]\!]_\pi & \text{if } [\![s]\!]_\pi = \Omega \end{cases} \end{array}$$

Definition 4.3.15 (Type valuation)

1. *A* type valuation *is a map* $\xi : \mathcal{V}_\mathcal{T} \to \mathsf{SAT}$.
2. *For every type valuation* ξ, $\alpha \in \mathcal{V}_\mathcal{T}$, *and* $X \in \mathsf{SAT}$, *the type valuation* $\xi(\alpha := X)$ *is defined as follows:*

$$\xi(\alpha := X)(\alpha') = \begin{cases} X & \text{if } \alpha' \equiv \alpha \\ \xi(\alpha') & \text{if } \alpha' \not\equiv \alpha \end{cases}$$

Definition 4.3.16 (Interpretation of types) *Let π be a stage valuation and ξ a type valuation. The corresponding type interpretation function $[\![.]\!]_{\pi,\xi} : \mathcal{T} \rightarrow \mathsf{SAT}$ is defined by induction on heights (because of the stratification on datatype identifiers, every type has finite height):*

$$\begin{array}{rcl}
[\![\alpha]\!]_{\pi,\xi} & = & \xi(\alpha) \ \text{ if } \alpha \in \mathcal{V}_{\mathcal{T}} \\
[\![\tau \rightarrow \sigma]\!]_{\pi,\xi} & = & [\![\tau]\!]_{\pi,\xi} \rightarrow [\![\sigma]\!]_{\pi,\xi} \\
[\![d^s \vec{\tau}]\!]_{\pi,\xi} & = & D^d_{\pi,\xi}([\![\vec{\tau}]\!]_{\pi,\xi}, [\![s]\!]_{\pi})
\end{array}$$

where $D^d_{\pi,\xi}(\vec{X}, x)$ is defined by induction on x by

$$\begin{array}{rcl}
D^d_{\pi,\xi}(\vec{X}, 0) & = & \ulcorner \emptyset \urcorner \\
D^d_{\pi,\xi}(\vec{X}, y+1) & = & \displaystyle\bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi,\xi(\delta := D^d_{\pi,\xi}(\vec{X},y), \vec{\alpha} := \vec{X})} \urcorner \\
& & \text{where, assuming } \mathsf{D}(c) = \forall\, \vec{\alpha}.\forall\, \imath.\ \vec{\sigma} \rightarrow d^{\widehat{\imath}} \vec{\alpha} \text{ and } \delta \text{ fresh,} \\
& & \vec{\theta} \text{ is obtained from } \vec{\sigma} \text{ replacing the occurrences of } d^{\imath} \vec{\alpha} \text{ by } \delta. \\
D^d_{\pi,\xi}(\vec{X}, x) & = & \displaystyle\bigcup_{y < x} D^d_{\pi,\xi}(\vec{X}, y) \quad \text{if } x \text{ is a limit ordinal}
\end{array}$$

Note that we write $c\, [\![\vec{\sigma}]\!]_{\pi,\xi}$ as an abbreviation for $\{c\, a_1 \ldots a_n \mid a_i \in [\![\sigma_i]\!]_{\pi,\xi} \text{ for } i = 1..\#\vec{\sigma}\}$.

Remark 4.3.17 *In the definition of $[\![d^s \vec{\tau}]\!]_{\pi,\xi}$, a form of variable convention for type variables is relied upon: for each $c \in \mathsf{C}(d)$ such that $\mathsf{D}(c) = \forall\, \vec{\alpha}.\forall\, \imath.\ \vec{\sigma} \rightarrow d^{\widehat{\imath}} \vec{\alpha}$, $\vec{\alpha}$ are assumed not to appear in $\vec{\tau}$. Alternatively, without the convention, some variable renaming of $\vec{\alpha}$ may be necessary.*

Lemma 4.3.18

1. $D^d_{\pi,\xi}(\vec{X}, x) = D^d_{\pi(\imath := s),\xi}(\vec{X}, x)$
2. $D^d_{\pi,\xi}(\vec{X}, x) = D^d_{\pi,\xi(\alpha := \tau)}(\vec{X}, x)$

Proof. Both proofs proceed by induction on x.

1. For $x = 0$ it is trivial. For $x = y + 1$, we have:

$$\begin{array}{rcll}
D^d_{\pi(\imath := s),\xi}(\vec{X}, y+1) & = & \displaystyle\bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi(\imath := s),\xi(\delta := D^d_{\pi(\imath := s),\xi}(\vec{X},y), \vec{\alpha} := \vec{X})} \urcorner & (4.6) \\
& = & \displaystyle\bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi(\imath := s),\xi(\delta := D^d_{\pi,\xi}(\vec{X},y), \vec{\alpha} := \vec{X})} \urcorner & (4.7) \\
& = & \displaystyle\bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi,\xi(\delta := D^d_{\pi,\xi}(\vec{X},y), \vec{\alpha} := \vec{X})} \urcorner & (4.8) \\
& = & D^d_{\pi,\xi}(\vec{X}, y+1) & (4.9)
\end{array}$$

   The equation (4.7) is justified using induction hypothesis and (4.8) is because $\vec{\theta}$ has no occurrences of stage variables.

   If x is a limit ordinal, the proof follows from induction hypothesis.

2. For $x = 0$ it is trivial. For $x = y + 1$, we have:

$$D^d_{\pi,\xi(\alpha := \tau)}(\vec{X}, y+1) \quad = \quad \bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi,\xi(\alpha := \tau, \delta := D^d_{\pi,\xi(\alpha := \tau)}(\vec{X},y), \vec{\alpha} := \vec{X})} \urcorner \qquad (4.10)$$

$$= \bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi,\xi(\alpha:=\tau,\delta:=D^d_{\pi,\xi}(\vec{X},y),\vec{\alpha}:=\vec{X})} \urcorner \tag{4.11}$$

$$= \bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi,\xi(\delta:=D^d_{\pi,\xi}(\vec{X},y),\vec{\alpha}:=\vec{X})} \urcorner \tag{4.12}$$

$$= D^d_{\pi,\xi}(\vec{X}, y+1) \tag{4.13}$$

The equation (4.11) is justified using induction hypothesis and (4.12) is due to the fact that the only type variables that can appear in $\vec{\theta}$ are $\vec{\alpha}$ and δ.

If x is a limit ordinal, then the proof follows from induction hypothesis.

□

Lemma 4.3.19 (Substitution lemma for the interpretation of types)

1. $[\![\sigma[\imath := s]]\!]_{\pi,\xi} = [\![\sigma]\!]_{\pi(\imath:=[\![s]\!]_\pi),\xi}$
2. $[\![\sigma[\alpha := \tau]]\!]_{\pi,\xi} = [\![\sigma]\!]_{\pi,\xi(\alpha:=[\![\tau]\!]_{\pi,\xi})}$

Proof. Both proofs proceed by induction on the structure of σ, using Lemma 4.3.18. □

The following lemma states that the sequence of approximations of any datatype is non-decreasing with respect to set inclusion and converges before Ω.

Lemma 4.3.20

1. *If* $X \subseteq X'$ *and* α pos σ, *then* $[\![\sigma]\!]_{\pi,\xi(\alpha:=X)} \subseteq [\![\sigma]\!]_{\pi,\xi(\alpha:=X')}$.
   If $X \subseteq X'$ *and* α neg σ, *then* $[\![\sigma]\!]_{\pi,\xi(\alpha:=X')} \subseteq [\![\sigma]\!]_{\pi,\xi(\alpha:=X)}$.
2. *If* $X_i \subseteq X'_i$ *and* $x \leq x'$, *then* $D^d_{\pi,\xi}(\vec{X}, x) \subseteq D^d_{\pi,\xi}(\vec{X'}, x')$.
3. $D^d_{\pi,\xi}(\vec{X}, \Omega + 1) = D^d_{\pi,\xi}(\vec{X}, \Omega)$.

Proof.

1. By mutual induction on the structure of σ, using Lemma 4.3.18.
2. By induction on x. Assume $X_i \subseteq X'_i$ for $i = 1..\#\vec{X}$. For $x = 0$ it is trivial. For $x = y+1$, then $x' = y' + 1$ with $y \leq y'$. So, we have:

$$D^d_{\pi,\xi}(\vec{X}, y+1) = \bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, \underbrace{[\![\vec{\theta}]\!]_{\pi,\xi(\delta:=\underbrace{D^d_{\pi,\xi}(\vec{X},y)}_{\subseteq D^d_{\pi,\xi}(\vec{X'},y')},\vec{\alpha}:=\underbrace{\vec{X}}_{\subseteq \vec{X'}})}}_{\subseteq [\![\vec{\theta}]\!]_{\pi,\xi(\delta:=D^d_{\pi,\xi}(\vec{X'},y'),\vec{\alpha}:=\vec{X'})}} \urcorner \tag{4.14}$$

$$\subseteq \bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\vec{\theta}]\!]_{\pi,\xi(\delta:=D^d_{\pi,\xi}(\vec{X'},y'),\vec{\alpha}:=\vec{X'})} \urcorner \tag{4.15}$$

$$= D^d_{\pi,\xi}(\vec{X'}, y'+1) \tag{4.16}$$

The equation (4.15) is justified using induction hypothesis, item 1 of this lemma and Lemma 4.3.9 (item 1). If x is a limit ordinal, then the proof follows from induction hypothesis.

3. From 2, using the fact that $\mathcal{E}$ is countable. The iteration process has to converge before Ω: the opposite would imply that $\mathcal{E}$ is uncountable, as Ω is uncountable.

□

Lemma 4.3.21

$$[\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi} = \bigcup_{c\in\mathsf{C}(d)} \ulcorner c\,[\![\mathsf{Dom}^s_{\vec{\tau}}(c)]\!]_{\pi,\xi}\urcorner$$

Proof.

$$\begin{aligned}
[\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi} &= D^d_{\pi,\xi}([\![\vec{\tau}]\!]_{\pi,\xi}, [\![\widehat{s}]\!]_{\pi}) && (4.17)\\
&= D^d_{\pi,\xi}([\![\vec{\tau}]\!]_{\pi,\xi}, [\![s]\!]_{\pi}+1) && (4.18)\\
&= \bigcup_{c\in\mathsf{C}(d)} \ulcorner c\,[\![\vec{\theta}]\!]_{\pi,\xi(\delta:=D^d_{\pi,\xi}([\![\vec{\tau}]\!]_{\pi,\xi},[\![s]\!]_{\pi}),\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})}\urcorner && (4.19)\\
&= \bigcup_{c\in\mathsf{C}(d)} \ulcorner c\,[\![\mathsf{Dom}^s_{\vec{\tau}}(c)]\!]_{\pi,\xi}\urcorner && (4.20)
\end{aligned}$$

The justification for equation (4.18) is: if $[\![s]\!]_{\pi} < \Omega$, then $[\![\widehat{s}]\!]_{\pi} = [\![s]\!]_{\pi}+1$; if $[\![s]\!]_{\pi} = \Omega$, then $[\![\widehat{s}]\!]_{\pi} = \Omega$, so the equation follows using Lemma 4.3.20 (item 3). The equation (4.20) is justified by

$$\begin{aligned}
[\![\mathsf{Dom}^s_{\vec{\tau}}(c)]\!]_{\pi,\xi} &= [\![\vec{\sigma}[\imath := s][\vec{\alpha} := \vec{\tau}]]\!]_{\pi,\xi} && (4.21)\\
&= [\![\vec{\theta}[\delta := d^s\vec{\tau}]]\!]_{\pi,\xi(\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})} && (4.22)\\
&= [\![\vec{\theta}]\!]_{\pi,\xi(\delta:=[\![d^s\vec{\tau}]\!]_{\pi,\xi(\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})},\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})} && (4.23)\\
&= [\![\vec{\theta}]\!]_{\pi,\xi(\delta:=D^d_{\pi,\xi(\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})}([\![\vec{\tau}]\!]_{\pi,\xi(\vec{\alpha}:=[\![\vec{\theta}]\!]_{\pi,\xi})},[\![s]\!]_{\pi}),\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})} && (4.24)\\
&= [\![\vec{\theta}]\!]_{\pi,\xi(\delta:=D^d_{\pi,\xi}([\![\vec{\tau}]\!]_{\pi,\xi(\vec{\alpha}:=[\![\vec{\theta}]\!]_{\pi,\xi})},[\![s]\!]_{\pi}),\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})} && (4.25)\\
&= [\![\vec{\theta}]\!]_{\pi,\xi(\delta:=D^d_{\pi,\xi}([\![\vec{\tau}]\!]_{\pi,\xi},[\![s]\!]_{\pi}),\vec{\alpha}:=[\![\vec{\tau}]\!]_{\pi,\xi})} && (4.26)
\end{aligned}$$

Notice that in (4.22) we are assuming $\mathsf{D}(c) = \forall\,\vec{\alpha}.\forall\,\imath.\ \vec{\sigma} \to d^{\widehat{\imath}}\vec{\alpha}$ and δ fresh, and neither δ nor $\vec{\alpha}$ appear in $\vec{\tau}$. Furthermore, $\vec{\theta}$ is obtained from $\vec{\sigma}$ replacing the occurrences of $d^{\imath}\vec{\alpha}$ by δ. This equation is justified by Lemma 4.3.19. Equation (4.23) follows from Lemma 4.3.19, equation (4.25) follows from Lemma 4.3.18, and equation (4.26) is justified by the fact of none $\vec{\alpha}$ occurs in $\vec{\tau}$.

□

Next we define valuations and interpretation for terms.

Definition 4.3.22 (Term valuation)

1. *A* term valuation *is a map* $\rho : \mathcal{V}_{\mathcal{E}} \to \mathcal{E}$.

2. *For every term valuation* ρ*,* $e \in \mathcal{E}$ *and* $x \in \mathcal{V}_{\mathcal{E}}$*, the term valuation* $\rho(x := e)$ *is defined as follows:*

$$\rho(x := e)(z) = \begin{cases} e & \text{if } z \equiv x \\ \rho(z) & \text{if } z \not\equiv x \end{cases}$$

Definition 4.3.23 (Interpretation of terms) *For any term valuation ρ, the map $(\![.]\!)_\rho : \mathcal{E} \to \mathcal{E}$ is defined inductively as follows:*

$$\begin{array}{rcl}
(\![x]\!)_\rho &=& \rho(x) \\
(\![\lambda x.\, e]\!)_\rho &=& \lambda x. (\![e]\!)_{\rho(x:=x)} \\
(\![e\, e']\!)_\rho &=& (\![e]\!)_\rho\, (\![e']\!)_\rho \\
(\![c]\!)_\rho &=& c \\
(\![\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}]\!)_\rho &=& \mathsf{case}\ (\![e]\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \\
(\![\mathsf{letrec}\ f = e]\!)_\rho &=& \mathsf{letrec}\ f = (\![e]\!)_{\rho(f:=f)}
\end{array}$$

Remark 4.3.24 *In the clauses for lambda-abstraction and letrec, a form of variable convention is relied upon: namely, x resp. f is assumed not to appear as a free variable in any of the terms $\rho(y)$ where y is free in e. Alternatively, without the convention, some variable renaming may be necessary: in the case of lambda-abstraction, one would set*

$$(\![\lambda x.\, e]\!)_\rho = \lambda x'. (\![e]\!)_{\rho(x:=x')}$$

where x' is some variable free in no $\rho(y)$ where y is free in e.

Lemma 4.3.25 (Substitution lemma for the interpretation of terms)

1. $(\![e[x := e']]\!)_\rho = (\![e]\!)_{\rho(x:=(\![e']\!)_\rho)}$.

2. $(\![e]\!)_\rho = e[\vec{y} := \rho(\vec{y})]$ *where $\vec{y}$ are the free variables of e.*

Proof. By induction on the structure of e. □

The notion of satisfaction and validity are defined as usual: satisfaction of subtyping is set inclusion, and satisfaction of typing is set membership.

Definition 4.3.26 (Satisfaction, validity)

1. *A stage valuation π* satisfies *a stage comparison judgment $s \preccurlyeq s'$, if $[\![s]\!]_\pi \leq [\![s']\!]_\pi$. A stage comparison judgment $s \preccurlyeq s'$ is* valid*, if every stage valuation satisfies it.*

2. *A stage valuation π and a type valuation ξ* satisfy *a subtyping judgment $\sigma \leq \sigma'$, if $[\![\sigma]\!]_{\pi,\xi} \subseteq [\![\sigma']\!]_{\pi,\xi}$. A subtyping judgment $\sigma \leq \sigma'$ is* valid*, if every pair of stage and type valuations satisfies it.*

3. *A* valuation *is a triple (π, ξ, ρ), where π is a stage valuation, ξ is a type valuation and ρ is a term valuation.*

4. *Let (π, ξ, ρ) be a valuation.*

   (a) (π, ξ, ρ) satisfies a context Γ*, written $(\pi, \xi, \rho) \models \Gamma$, if $\rho(x) \in [\![\tau]\!]_{\pi,\xi}$ for each $(x : \tau) \in \Gamma$.*

   (b) (π, ξ, ρ) satisfies a typing judgment $\Gamma \vdash e : \sigma$*, if*

   $$(\pi, \xi, \rho) \models \Gamma \quad \Rightarrow \quad (\![e]\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$$

5. *A typing judgment $\Gamma \vdash e : \sigma$ is* valid*, written $\Gamma \models e : \sigma$, if every valuation satisfies it.*

4.3.2 Soundness w.r.t. the Semantics

Next we prove that the rules of $\lambda^{\frown}$ for stage comparison, subtyping, and typing are sound with respect to the semantics just defined. The strong normalization theorem follows as a corollary from the typing soundness.

Proposition 4.3.27 (Stage comparison soundness)

$$s \preccurlyeq s' \text{ derivable} \quad \Rightarrow \quad s \preccurlyeq s' \text{ valid}$$

Proof. By induction on the derivation of $s \preccurlyeq s'$. ☐

Proposition 4.3.28 (Subtyping soundness)

$$\sigma \leq \sigma' \text{ derivable} \quad \Rightarrow \quad \sigma \leq \sigma' \text{ valid}$$

Proof. By induction on the derivation of $\sigma \leq \sigma'$, using lemmas 4.3.11 and 4.3.20. ☐

Lemma 4.3.29 *Let ξ be a type valuation. Then*

1. *If $\imath$ pos σ and $x \leq x'$, then $[\![\sigma]\!]_{\pi(\imath:=x),\xi} \subseteq [\![\sigma]\!]_{\pi(\imath:=x'),\xi}$.*
2. *If $\imath$ neg σ and $x \leq x'$, then $[\![\sigma]\!]_{\pi(\imath:=x),\xi} \supseteq [\![\sigma]\!]_{\pi(\imath:=x'),\xi}$.*

Proof. By simultaneous induction on the structure of σ. ☐

Theorem 4.3.30 (Typing soundness)

$$\Gamma \vdash e : \sigma \text{ derivable} \quad \Rightarrow \quad \Gamma \models e : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash e : \sigma$.

(var) Assume the last (and the only) step is

$$\frac{}{\Gamma \vdash x : \tau} \quad \text{and} \quad (x : \tau) \in \Gamma$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![x]\!)_\rho \in [\![\tau]\!]_{\pi,\xi}$. This is true, as $(x : \tau) \in \Gamma$.

(abs) Assume the last step is

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.\, e : \tau \to \sigma}$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![\lambda x.\, e]\!)_\rho \in [\![\tau \to \sigma]\!]_{\pi,\xi}$. Since $[\![\tau \to \sigma]\!]_{\pi,\xi} = [\![\tau]\!]_{\pi,\xi} \to [\![\sigma]\!]_{\pi,\xi}$ and $(\![\lambda x.\, e]\!)_\rho = \lambda x.(\![e]\!)_{\rho_0}$, where $\rho_0 = \rho(x := x)$, this amounts to showing that $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \in [\![\sigma]\!]_{\pi,\xi}$ for any $a \in [\![\tau]\!]_{\pi,\xi}$.

Observe first that, since $(\pi, \xi, \rho_0) \models \Gamma$ and $\rho_0(x) = x \in \mathcal{V}_\mathcal{E} \subseteq [\![\tau]\!]_{\pi,\xi}$, the induction hypothesis tells us that $(\![e]\!)_{\rho_0} \in [\![\sigma]\!]_{\pi,\xi} \subseteq \mathsf{SN}$.

Suppose now $a \in [\![\tau]\!]_{\pi,\xi} \subseteq \mathsf{SN}$ and let $\rho' = \rho(x := a)$. Since $(\pi, \xi, \rho') \models \Gamma$ and $\rho'(x) = a \in [\![\tau]\!]_{\pi,\xi}$, by the induction hypothesis, we get that $(\![e]\!)_{\rho'} \in [\![\sigma]\!]_{\pi,\xi} \subseteq \mathsf{SN}$. Write $\vec{y}$ for the free variables of e, then $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \to_k (\![e]\!)_{\rho_0}[x := a] = e[\vec{y} := \rho_0(\vec{y})][x := a] = e[\vec{y} := \rho'(\vec{y})] = (\![e]\!)_{\rho'}$ (by the variable convention, Remark 4.3.24, x does not occur free in $\rho_0(\vec{y})$). By Lemma 4.3.6, $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \in \mathsf{SN}$. As $[\![\sigma]\!]_{\pi,\xi}$ is a saturated set, we get that $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \in [\![\sigma]\!]_{\pi,\xi}$.

(app) Assume the last step is

$$\frac{\Gamma \vdash e : \tau \to \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\!|e\, e'|\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$. As $(\!|e\, e'|\!)_\rho = (\!|e|\!)_\rho\, (\!|e'|\!)_\rho$, this amounts to showing that $(\!|e|\!)_\rho\, (\!|e'|\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$.

As $(\pi, \xi, \rho) \models \Gamma$, the induction hypothesis gives that $(\!|e|\!)_\rho \in [\![\tau \to \sigma]\!]_{\pi,\xi} = [\![\tau]\!]_{\pi,\xi} \to [\![\sigma]\!]_{\pi,\xi}$ and $(\!|e'|\!)_\rho \in [\![\tau]\!]_{\pi,\xi}$. Thus $(\!|e|\!)_\rho\, (\!|e'|\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$.

(cons) Assume the last (and the only) step is

$$\frac{}{\Gamma \vdash c : \mathsf{Inst}^s_{\vec{\tau}}(c)} \quad \text{with} \quad c \in \mathsf{C}(d)$$

Observe first that $\mathsf{Inst}^s_{\vec{\tau}}(c) = \mathsf{Dom}^s_{\vec{\tau}}(c) \to d^{\widehat{s}}\vec{\tau}$. Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\!|c|\!)_\rho \in [\![\mathsf{Dom}^s_{\vec{\tau}}(c) \to d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi} = [\![\mathsf{Dom}^s_{\vec{\tau}}(c)]\!]_{\pi,\xi} \to [\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi}$. As $(\!|c|\!)_\rho = c$, this amounts to showing that $c\,\vec{a} \in [\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi}$ for any $\vec{a} \in [\![\mathsf{Dom}^s_{\vec{\tau}}(c)]\!]_{\pi,\xi}$. But that holds trivially, since $[\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi} = \bigcup_{c \in \mathsf{C}(d)} \ulcorner c\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c)]\!]_{\pi,\xi} \urcorner$ as stated in Lemma 4.3.21.

(case) Assume the last step is

$$\frac{\Gamma \vdash e : d^{\widehat{s}}\vec{\tau} \quad \Gamma \vdash e_1 : \mathsf{Dom}^s_{\vec{\tau}}(c_1) \to \theta \quad \ldots \quad \Gamma \vdash e_n : \mathsf{Dom}^s_{\vec{\tau}}(c_n) \to \theta}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta} \text{ with } \mathsf{C}(d) = \{c_1, ..., c_n\}$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\!|\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}|\!)_\rho \in [\![\theta]\!]_{\pi,\xi}$. As $(\!|\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}|\!)_\rho = \mathsf{case}\ (\!|e|\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\}$, this amounts to showing that $\mathsf{case}\ (\!|e|\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi}$.

As $(\pi, \xi, \rho) \models \Gamma$, from the induction hypothesis we get that $(\!|e|\!)_\rho \in [\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi} \subseteq \mathsf{SN}$ and $(\!|e_i|\!)_\rho \in [\![\mathsf{Dom}^s_{\vec{\tau}}(c_i) \to \theta]\!]_{\pi,\xi} = [\![\mathsf{Dom}^s_{\vec{\tau}}(c_i)]\!]_{\pi,\xi} \to [\![\theta]\!]_{\pi,\xi} \subseteq \mathsf{SN}$ for each $i \in 1..n$.

Since $[\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi} = \ulcorner c_1\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c_1)]\!]_{\pi,\xi} \cup \ldots \cup c_n\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c_n)]\!]_{\pi,\xi} \urcorner$, it must be the case that $(\!|e|\!)_\rho \twoheadrightarrow_k b$ for some $b \in \mathsf{Base} \cup c_1\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c_1)]\!]_{\pi,\xi} \cup \ldots \cup c_n\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c_n)]\!]_{\pi,\xi}$.

From $b \in \mathsf{Base} \cup c_1\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c_1)]\!]_{\pi,\xi} \cup \ldots \cup c_n\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c_n)]\!]_{\pi,\xi}$, it follows that $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi} \subseteq \mathsf{SN}$. Indeed, if $b \in \mathsf{Base}$, then $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \in \mathsf{Base} \subseteq [\![\theta]\!]_{\pi,\xi}$, as $(\!|e_i|\!)_\rho \in \mathsf{SN}$ for each $i \in 1..n$; if $b \in c_i\, [\![\mathsf{Dom}^s_{\vec{\tau}}(c_i)]\!]_{\pi,\xi}$ for some $i \in 1..n$, then $b = c_i\,\vec{a}$ for some $\vec{a} \in [\![\mathsf{Dom}^s_{\vec{\tau}}(c_i)]\!]_{\pi,\xi}$ and therefore $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \to_k (\!|e_i|\!)_\rho \vec{a} \in [\![\theta]\!]_{\pi,\xi}$ and, by Lemma 4.3.6, $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \in \mathsf{SN}$, hence $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi}$ as $[\![\theta]\!]_{\pi,\xi}$ is saturated.

From $(\!|e|\!)_\rho \twoheadrightarrow_k b$ it follows that $\mathsf{case}\ (\!|e|\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \twoheadrightarrow_k \mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\}$; further, by Lemma 4.3.5, $\mathsf{case}\ (\!|e|\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \in \mathsf{SN}$. Since $[\![\theta]\!]_{\pi,\xi}$ is saturated, we get that $\mathsf{case}\ (\!|e|\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\!|e_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|e_n|\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi}$.

(rec) Assume the last step is

$$\frac{\Gamma, f : d^{\imath}\vec{\tau} \to \theta \vdash e : d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}] \quad \imath\ \mathsf{pos}\ \theta}{\Gamma \vdash (\mathsf{letrec}\ f = e) : d^{s}\vec{\tau} \to \theta[\imath := s]} \quad \text{and } \imath \text{ fresh in } \Gamma, \vec{\tau}$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![(\mathsf{letrec}\ f = e)]\!)_\rho \in [\![d^s\vec{\tau} \to \theta[\imath := s]]\!]_{\pi,\xi}$. As $[\![d^s\vec{\tau} \to \theta[\imath := s]]\!]_{\pi,\xi} = [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi_0,\xi} = [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi} \to [\![\theta]\!]_{\pi_0,\xi}$ and $(\![(\mathsf{letrec}\ f = e)]\!)_\rho = (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})$ where $\pi_0 = \pi(\imath := [\![s]\!]_\pi)$ and $\rho_0 = \rho(f := f)$, this amounts to showing that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in [\![\theta]\!]_{\pi_0,\xi}$ for any $a \in [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi}$.

As $(\pi_0, \xi, \rho_0) \models \Gamma$ and $\rho_0(f) = f \in \mathcal{V}_\mathcal{E} \subseteq [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi_0,\xi}$, by the induction hypothesis we get that $(\![e]\!)_{\rho_0} \in [\![d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}]]\!]_{\pi_0,\xi} \subseteq \mathsf{SN}$.

We prove our goal by induction on $\pi_0(\imath)$.

$(\pi_0(\imath) = 0)$ Suppose $a \in [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi} = \ulcorner\emptyset\urcorner \subseteq \mathsf{SN}$. Then $a \twoheadrightarrow_k b$ for some $b \in \mathsf{Base}$.

Since $(\![e]\!)_{\rho_0} \in \mathsf{SN}$, from $b \in \mathsf{Base}$ it follows that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b \in \mathsf{Base} \subseteq [\![\theta]\!]_{\pi_0,\xi} \subseteq \mathsf{SN}$.

From $a \twoheadrightarrow_k b$ it follows that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \twoheadrightarrow_k (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b$ and, by Lemma 4.3.5, $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in \mathsf{SN}$.

Since $[\![\theta]\!]_{\pi_0,\xi}$ is a saturated set, we can conclude that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in [\![\theta]\!]_{\pi_0,\xi}$.

$(\pi_0(\imath) = y + 1)$ Let $\pi' = \pi(\imath := y)$ and $\rho' = \rho(f := (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0}))$. As $(\pi', \xi, \rho') \models \Gamma$ and as by the inner induction hypothesis $\rho'(f) = (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0}) \in [\![d^\imath\vec{\tau}]\!]_{\pi',\xi} \to [\![\theta]\!]_{\pi',\xi} = [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi',\xi}$, by the outer induction hypothesis we get that $(\![e]\!)_{\rho'} \in [\![d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}]]\!]_{\pi',\xi} = [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi_0,\xi}$.

Suppose $a \in [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi} = [\![d^{\widehat{\imath}}\vec{\tau}]\!]_{\pi',\xi} = \ulcorner c_1\, [\![\mathsf{Dom}^\imath_{\vec{\tau}}(c_1)]\!]_{\pi',\xi} \cup \ldots \cup c_n\, [\![\mathsf{Dom}^\imath_{\vec{\tau}}(c_n)]\!]_{\pi',\xi}\urcorner \subseteq \mathsf{SN}$ using Lemma 4.3.21.

Then $a \twoheadrightarrow_k b$ for some $b \in \mathsf{Base} \cup c_1\, [\![\mathsf{Dom}^\imath_{\vec{\tau}}(c_1)]\!]_{\pi',\xi} \cup \ldots \cup c_n\, [\![\mathsf{Dom}^\imath_{\vec{\tau}}(c_n)]\!]_{\pi',\xi}$.

From $b \in \mathsf{Base} \cup c_1\, [\![\mathsf{Dom}^\imath_{\vec{\tau}}(c_1)]\!]_{\pi',\xi} \cup \ldots \cup c_n\, [\![\mathsf{Dom}^\imath_{\vec{\tau}}(c_n)]\!]_{\pi',\xi}$ we get that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b \in [\![\theta]\!]_{\pi_0,\xi} \subseteq \mathsf{SN}$. Indeed, if $b \in \mathsf{Base}$, then $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b \in \mathsf{Base} \subseteq [\![\theta]\!]_{\pi_0,\xi}$, since $(\![e]\!)_{\rho_0} \in \mathsf{SN}$; if $b \in c_i[\![\mathsf{Dom}^\imath_{\vec{\tau}}(c_i)]\!]_{\pi',\xi} \subseteq [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi}$ for some $i \in 1..n$, then $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b \to_k (\![e]\!)_{\rho_0}[f := (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})]\, b = (\![e]\!)_{\rho'}\, b \in [\![\theta]\!]_{\pi_0,\xi}$ and, by Lemma 4.3.6, $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b \in \mathsf{SN}$, hence $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b \in [\![\theta]\!]_{\pi_0,\xi}$ as $[\![\theta]\!]_{\pi_0,\xi}$ is saturated.

From $a \twoheadrightarrow_k b$ it follows that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \twoheadrightarrow_k (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b$ and, by Lemma 4.3.5, $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in \mathsf{SN}$.

Since $[\![\theta]\!]_{\pi_0,\xi}$ is a saturated set, we can conclude that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in [\![\theta]\!]_{\pi_0,\xi}$.

($\pi_0(\imath) = x$ where x is a limit ordinal) Suppose $a \in [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi} = \bigcup_{y<x}[\![d^\imath\vec{\tau}]\!]_{\pi(\imath:=y),\xi}$. Then $a \in [\![d^\imath\vec{\tau}]\!]_{\pi(\imath:=y),\xi}$ for some $y < x$. By the inner induction hypothesis and by the positivity of $\imath$ in θ, we therefore get that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in [\![\theta]\!]_{\pi(\imath:=y),\xi} \subseteq [\![\theta]\!]_{\pi_0,\xi}$ using Lemma 4.3.29.

(sub) Assume the last step is

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \leq \sigma'}{\Gamma \vdash e : \sigma'}$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![e]\!)_\rho \in [\![\sigma']\!]_{\pi,\xi}$. As $(\pi, \xi, \rho) \models \Gamma$, the induction hypothesis gives $(\![e]\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$ and the subtyping soundness gives $[\![\sigma]\!]_{\pi,\xi} \subseteq [\![\sigma']\!]_{\pi,\xi}$. Together, these give $(\![e]\!)_\rho \in [\![\sigma']\!]_{\pi,\xi}$.

□

The main result of this subsection follows as an immediate corollary of the soundness of the typing system.

Theorem 4.3.31 (Strong normalization) $\rightarrow_{\beta\iota\mu}$ *is strongly normalizing on typable expressions:*

$$\Gamma \vdash e : \sigma \textit{ derivable} \quad \Rightarrow \quad e \in \mathsf{SN}$$

Proof. Assume $\Gamma \vdash e : \sigma$. Then, by Theorem 4.3.30, $\Gamma \models e : \sigma$. Consider a valuation (π, ξ, ρ) where, for every $x \in \mathcal{V}_\mathcal{E}$, $\rho(x) = x$. For every $(x : \tau) \in \Gamma$, $(\!|x|\!)_\rho = x \in [\![\tau]\!]_{\pi,\xi}$, since $[\![\tau]\!]_{\pi,\xi}$ is saturated, hence $(\pi, \xi, \rho) \models \Gamma$. Therefore $(\!|e|\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$. As $(\!|e|\!)_\rho = e$, we have

$$e \in [\![\sigma]\!]_{\pi,\xi} \subseteq \mathsf{SN}$$

$\square$

Chapter 5

The System $\lambda_{\mathcal{G}}$

In this chapter we present the system $\lambda_{\mathcal{G}}$, a simply typed λ-calculus with inductive types. The terms allowed in $\lambda_{\mathcal{G}}$ are the same as those allowed in $\widehat{\lambda}$. In particular, we continue to have the letrec constructor for defining functions recursively, but in $\lambda_{\mathcal{G}}$ (following what is done in [67]) termination of typable recursively defined functions is ensured by a syntactical condition $\mathcal{G}$ constraining uses of recursive calls in the body of definitions. The condition $\mathcal{G}$ is checked directly on the body of the function and not on its normal form because of the problem this would raise (as discussed in the introduction).

5.1 Definition of $\lambda_{\mathcal{G}}$

The systems $\lambda_{\mathcal{G}}$ and $\widehat{\lambda}$ allow the same set of terms; they differ at the level of types in the following aspects:

1. stages are not present in $\lambda_{\mathcal{G}}$ and so datatypes are not annotated by stages;

2. in $\lambda_{\mathcal{G}}$ there is no subtyping relation;

3. the set of typing rules is different, and $\lambda_{\mathcal{G}}$'s typing rule

$$\frac{\Gamma, f : d\,\vec{\tau} \to \sigma \;\vdash\; e : d\,\vec{\tau} \to \sigma \qquad \mathcal{G}^{x}_{f}(\emptyset, a)}{\Gamma \;\vdash\; (\mathsf{letrec}\ f = e) : d\,\vec{\tau} \to \sigma} \quad \text{if } \; e \equiv \lambda x.a$$

   for letrec-expressions is complemented by the syntactical condition $\mathcal{G}$;

4. following Giménez [67], the datatypes allowed in $\lambda_{\mathcal{G}}$ are slightly more restricted than those of $\widehat{\lambda}$ for, in the argument types of the constructors of a datatype, this datatype can only have *strictly positive* occurrences. The justification for this restriction is that syntactic methods fail for non-strictly positive datatypes.

Definition 5.1.1 (Constructor scheme) *The set $\mathcal{CS}_{\mathcal{G}}$ of* constructor schemes *is given by the abstract syntax:*

$$\varsigma \;::=\; \forall\,\vec{\alpha}.\ \sigma$$

where $\vec{\alpha}$ are the free type variables of σ.

Definition 5.1.2 (Constructor declaration) *There is a map* $\mathsf{D} : \mathcal{C} \to \mathcal{CS}_{\mathcal{G}}$ *such that, for every* $d \in \mathcal{D}$ *and* $c \in \mathsf{C}(d)$,

$$\mathsf{D}(c) = \forall\, \vec{\alpha}.\ \vec{\sigma} \to d\,\vec{\alpha}$$

where:

1. $\#\vec{\alpha} = \mathsf{ar}(d)$ *and* $\#\vec{\sigma} = \mathsf{ar}(c)$*;*
2. *each* σ_i *is strictly positive w.r.t.* d*, see Figure 5.1;*
3. *every occurrence of* d *in* σ_i *is of the form* $d\,\vec{\alpha}$*;*
4. *any* $d' \neq d \in \mathcal{D}$ *appearing in* $\vec{\sigma}$ *satisfies* $\mathsf{str}(d') < \mathsf{str}(d)$.

$$\text{(spos1)} \qquad \frac{d \ \mathsf{nocc}\ \tau}{\tau\ \mathsf{spos}\ d}$$

$$\text{(spos2)} \qquad \frac{d\ \mathsf{nocc}\ \theta_i \quad (1 \leq i \leq n)}{\theta_1 \to \ldots \to \theta_n \to d\,\vec{\alpha}\ \mathsf{spos}\ d}$$

Figure 5.1: Strictly positive rules

Let us focus on the letrec operator and on the syntactical condition $\mathcal{G}$ it satisfies. This condition complements the reduction rule $\to_\mu$, ensuring that each expansion of the letrec operator consumes (at least) the constructor in the head of its argument. Informally, for a term ($\mathsf{letrec}\ f = e$) we should have the following:

1. f may occur in e only as the head of an application;
2. any application of f must be protected by a case analysis of the formal argument of e, say x (for this reason f is said to be *guarded-by-destructors*); therefore f must occur inside e_i's in the following context:

$$\begin{array}{lll} \mathsf{case}\ x\ \mathsf{of}\ \{ & c_1 \Rightarrow \lambda x_{1,1}.\ldots.\lambda x_{1,\mathsf{ar}(c_1)}.\, e_1 \\ & \vdots \\ \quad\quad\quad\quad\ \mid & c_n \Rightarrow \lambda x_{n,1}.\ldots.\lambda x_{n,\mathsf{ar}(c_n)}.\, e_n \\ \quad\quad\quad\quad\ \} \end{array}$$

3. considering that the *components* of x are the x_{ij} (*direct components*) together with the components of each x_{ij} (*inner components*), f must be applied to a term of the form $z\,\vec{a}$ where z is a *recursive component* of x (i.e., z is a component of x whose type has occurrences of the type of x).

To illustrate the observations above, let us consider the examples already given in Section 3.4, *plus* and *even*, now transposed to $\lambda_{\mathcal{G}}$.

Example 5.1.3

- *The addition of two natural numbers.*

$$\begin{array}{lll}
(\mathsf{letrec} & \mathit{plus} = \lambda x.\, \lambda y.\, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y & \\
 & \qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathsf{s} \Rightarrow \lambda n.\mathsf{s}\,(\mathit{plus}\ n\ y) & \\
 & \qquad\qquad\qquad\qquad\qquad\quad\ \} & \\
):\ & \mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat} &
\end{array}$$

Here the only application of plus is protected by a case analysis on x, the formal argument of plus. The argument of this application is the pattern variable n, a direct component of x.

- *A function that indicates whether or not a natural number is even.*

$$\begin{array}{ll}
(\mathsf{letrec} & \mathit{even} = \lambda x.\, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{true} \\
 & \qquad\qquad\qquad\qquad\ \mid \mathsf{s} \Rightarrow \lambda y.\, \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{false} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{s} \Rightarrow \lambda z.\, \mathit{even}\ z\,\} \\
 & \qquad\qquad\qquad\qquad\ \} \\
):\ & \mathsf{Nat} \rightarrow \mathsf{Bool}
\end{array}$$

In this example the application of even is guarded by a case analysis on the argument x. The argument of this application is the pattern variable z, an inner component of x which becomes available in the case analysis on the pattern variable y, a direct component of x.

The formal description of the guarded-by-destructors condition is provided by the predicate $\mathcal{G}^x_f(V, a)$ defined below. The V argument is a set of variables used to collect the pattern variables in a representing the recursive components of x. In order to identify the recursive components of a variable, we start by characterizing the recursive positions of a constructor scheme as follows:

Definition 5.1.4 *Let c be a $\lambda_{\mathcal{G}}$ constructor such that $\mathsf{D}(c) = \forall\vec{\alpha}.\vec{\sigma} \rightarrow d\,\vec{\alpha}$. We say that the number j corresponds to a* recursive position *of $\mathsf{D}(c)$, written $\mathsf{RP}(j, \mathsf{D}(c))$, if σ_j is of the form $\vec{\gamma} \rightarrow d\,\vec{\alpha}$.*

The predicate $\mathcal{G}$ is now defined as follows:

Definition 5.1.5 ($\mathcal{G}$ predicate) *Let $U \subseteq \mathcal{V}$, let x and f be distinct variables not in U and let $a \in \mathcal{E}$. The predicate $\mathcal{G}^x_f(U, a)$ is derivable using the rules in Figure 5.2.*

Lemma 5.1.6 *If f nocc a then $\mathcal{G}^x_f(U, a)$.*

Proof. By induction on the structure of a. $\square$

One can check that the guard predicate holds on addition.

Example 5.1.7 *The function plus of Example 5.1.3 can be shown guarded as follows*

$$\dfrac{\dfrac{\mathit{plus} \neq y}{\mathcal{G}^x_{\mathit{plus}}(\emptyset, y)}\,(1) \qquad \dfrac{\dfrac{}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathsf{s})}\,(4) \quad \dfrac{\dfrac{\dfrac{\mathit{plus} \neq n}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, n)}\,(1)}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathit{plus}\, n)}\,(6) \quad \dfrac{\mathit{plus} \neq y}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, y)}\,(1)}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathit{plus}\, n\, y)}\,(5)}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathsf{s}\,(\mathit{plus}\, n\, y))}\,(5)}{\dfrac{\mathcal{G}^x_{\mathit{plus}}(\emptyset, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda n.\mathsf{s}\,(\mathit{plus}\, n\, y)\})}{\mathcal{G}^x_{\mathit{plus}}(\emptyset, \lambda y.\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda n.\mathsf{s}\,(\mathit{plus}\, n\, y)\})}\,(2)}\,(8)$$

1. $$\frac{f \neq y}{\mathcal{G}_f^x(U, y)} \qquad \text{if } y \text{ is a variable}$$

2. $$\frac{\mathcal{G}_f^x(U, a)}{\mathcal{G}_f^x(U, \lambda z.a)}$$

3. $$\frac{\mathcal{G}_f^x(U, e)}{\mathcal{G}_f^x(U, \mathsf{letrec}\ g = e)}$$

4. $$\frac{}{\mathcal{G}_f^x(U, c)}$$

5. $$\frac{\mathcal{G}_f^x(U, a) \quad \mathcal{G}_f^x(U, b)}{\mathcal{G}_f^x(U, a\, b)}$$

6. $$\frac{\mathcal{G}_f^x(U, z\, \vec{a})}{\mathcal{G}_f^x(U, f\ (z\, \vec{a}))} \qquad \text{if } z \in U$$

7. $$\frac{\mathcal{G}_f^x(U, e) \quad \mathcal{G}_f^x(U, b_i) \quad (1 \leq i \leq n)}{\mathcal{G}_f^x(U, \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})} \qquad \text{if } \begin{cases} e \not\equiv z\, \vec{a} \\ \quad \vee \\ (e \equiv z\, \vec{a}\ \wedge\ z \notin U \cup \{x\}) \end{cases}$$

8. $$\frac{\mathcal{G}_f^x(U, a_j) \quad (1 \leq j \leq m) \quad \mathcal{G}_f^x(V_i, e_i) \quad (1 \leq i \leq n)}{\mathcal{G}_f^x(U, \mathsf{case}\ (z\, a_1 \ldots a_m)\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})}$$

$$\text{if } \begin{cases} z \in U \cup \{x\} \\ b_i \equiv \lambda y_1 \ldots . \lambda y_{\mathsf{ar}(c_i)} . e_i \\ V_i \equiv U \cup \{y_j \mid \mathsf{RP}(j, \mathsf{D}(c_i)) \text{ for } 1 \leq j \leq \mathsf{ar}(c_i)\} \end{cases}$$

Figure 5.2: Guarded-by-destructors rules for $\lambda_{\mathcal{G}}$

As suggested in the introduction, the predicate $\mathcal{G}$ is very sensitive to syntax. This is illustrated by the example below.

Example 5.1.8 *Consider the following expression.*

$$\begin{array}{lll} \mathsf{letrec} & plus = \lambda x.\,\lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{ & \mathsf{o} \Rightarrow y \\ & & \mid \mathsf{s} \Rightarrow \lambda n.\mathsf{s}\,((\lambda g.\,g\,n\,y)\ plus) \\ & & \} \end{array}$$

This expression also defines the addition of two natural numbers: it is obtained from the plus function defined in Example 5.1.3 by a β-expansion. However, this definition of plus does not satisfy condition $\mathcal{G}$ because the occurrence of plus in the letrec *body is not the head of an application. So, when trying to prove the condition $\mathcal{G}$ we would have to derive $\mathcal{G}^x_{plus}(\{n\}, plus)$ which, by looking at the rules defining $\mathcal{G}$, we can immediately say to be underivable.*

Another example of an expression not satisfying the guard condition is the Euclidean division already considered.

Example 5.1.9 *The Euclidean division defined in Example 3.4.4 does not satisfy $\mathcal{G}$ for in the recursive call of the div function (div (minus x' y) y) its argument (minus x' y) is not a recursive component of its formal argument (x), but instead the result of applying the previously defined* minus *to a recursive component (x') of x.*

To better demonstrate the usage of rule 8 of Figure 5.2 we give another example of a derivation of a guard predicate.

Example 5.1.10 *Recall the function ans of Example 3.4.2:*

$$\begin{array}{l} \mathsf{letrec}\ ans = \lambda x.\lambda l.\ \mathsf{case}\ x\ \mathsf{of}\ \{ \\ \qquad \mathsf{empty} \Rightarrow \mathsf{nothing} \\ \qquad \mid \mathsf{node} \Rightarrow \lambda a.\lambda f.\ \mathsf{case}\ l\ \mathsf{of}\ \{ \\ \qquad\qquad \left.\begin{array}{l} \mathsf{nil} \Rightarrow \mathsf{just}\,a \\ \mid \mathsf{cons} \Rightarrow \lambda y.\lambda z.\ \mathsf{case}\ (f\ y)\ \mathsf{of}\ \{ \\ \qquad \mathsf{empty} \Rightarrow \mathsf{just}\,a \\ \qquad \mid \mathsf{node} \Rightarrow \lambda b.\lambda g.\ ans\,(f\ y)\,z \end{array}\right\} \equiv \mathbf{A} \\ \qquad\qquad\qquad \} \\ \qquad\qquad \} \\ \qquad \} \end{array}$$

This function can be shown guarded as follows

$$\dfrac{\dfrac{\dfrac{}{\mathcal{G}^x_{ans}(\emptyset, \mathsf{nothing})}\,(4) \qquad \dfrac{\dfrac{ans \neq l}{\mathcal{G}^x_{ans}(\{f\}, l)}\,(1) \quad (\mathbf{5.1}) \quad (\mathbf{5.2})}{\mathcal{G}^x_{ans}(\{f\}, \mathsf{case}\ l\ \mathsf{of}\ \{\mathbf{A}\})}\,(7)}{\mathcal{G}^x_{ans}(\emptyset, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{empty} \Rightarrow \mathsf{nothing} \mid \mathsf{node} \Rightarrow \lambda a.\lambda f.\ \mathsf{case}\ l\ \mathsf{of}\ \{\mathbf{A}\}\})}\,(8)}{\mathcal{G}^x_{ans}(\emptyset, \lambda l.\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{empty} \Rightarrow \mathsf{nothing} \mid \mathsf{node} \Rightarrow \lambda a.\lambda f.\ \mathsf{case}\ l\ \mathsf{of}\ \{\mathbf{A}\}\})}\,(2)$$

$$\dfrac{\dfrac{}{\mathcal{G}^x_{ans}(\{f\}, \mathsf{just})}\,(4) \qquad \dfrac{ans \neq a}{\mathcal{G}^x_{ans}(\{f\}, a)}\,(1)}{\mathcal{G}^x_{ans}(\{f\}, \mathsf{just}\,a)}\,(5) \tag{5.1}$$

$$\dfrac{\dfrac{ans \neq y}{\mathcal{G}^x_{ans}(\{f\}, y)}\,(1) \qquad (\mathbf{5.3}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{ans \neq f}{\mathcal{G}^x_{ans}(\{f,g\}, f)}\,(1) \quad \dfrac{ans \neq y}{\mathcal{G}^x_{ans}(\{f,g\}, y)}\,(1)}{\mathcal{G}^x_{ans}(\{f,g\}, f\,y)}\,(5)}{\mathcal{G}^x_{ans}(\{f,g\}, ans\,(f\,y))}\,(6) \qquad \dfrac{ans \neq z}{\mathcal{G}^x_{ans}(\{f,g\}, z)}\,(1)}{\mathcal{G}^x_{ans}(\{f,g\}, ans\,(f\,y)\,z)}\,(5)}{\dfrac{\mathcal{G}^x_{ans}(\{f\}, \mathsf{case}\ (f\,y)\ \mathsf{of}\ \{\mathsf{empty} \Rightarrow \mathsf{just}\,a \mid \mathsf{node} \Rightarrow \lambda b.\lambda g.\ ans\,(f\,y)\,z\})}{\mathcal{G}^x_{ans}(\{f\}, \lambda y.\lambda z.\mathsf{case}\ (f\,y)\ \mathsf{of}\ \{\mathsf{empty} \Rightarrow \mathsf{just}\,a \mid \mathsf{node} \Rightarrow \lambda b.\lambda g.\ ans\,(f\,y)\,z\})}\,2 \times (2)}\,(8) \tag{5.2}$$

$$\dfrac{\dfrac{}{\mathcal{G}^x_{ans}(\{f,g\}, \mathsf{just})}\,(4) \quad \dfrac{ans \neq a}{\mathcal{G}^x_{ans}(\{f,g\}, a)}\,(1)}{\mathcal{G}^x_{ans}(\{f,g\}, \mathsf{just}\,a)}\,(5) \tag{5.3}$$

We now turn to the typing system. First, one needs to define instances of constructors. The definition is almost identical to the one for $\lambda^{\frown}$, the only difference being the absence of stages.

Definition 5.1.11 (Instance and domain) *Let $d \in \mathcal{D}$, $c \in \mathsf{C}(d)$, and $\vec{\tau} \in \mathcal{T}$ such that $\#\vec{\tau} = \mathsf{ar}(d)$. Assume $\mathsf{D}(c) = \forall\, \vec{\alpha}.\ \vec{\sigma} \to d\,\vec{\alpha}$. An* instance of c w.r.t. $\vec{\tau}$ *is defined as follows*

$$\mathsf{Inst}_{\vec{\tau}}(c) = \vec{\sigma}[\vec{\alpha} := \vec{\tau}] \to d\,\vec{\tau}$$

A domain of c w.r.t. $\vec{\tau}$ *is defined as follows*

$$\mathsf{Dom}_{\vec{\tau}}(c) = \vec{\sigma}[\vec{\alpha} := \vec{\tau}]$$

Typing of terms is defined in the usual way.

Definition 5.1.12 (Typing) *The typing judgment $\Gamma \vdash e : \sigma$ is derivable if it can be inferred by the rules of Figure 5.3, where $\mathcal{G}^x_f(\emptyset, a)$ is the* guarded-by-destructors *condition defined in Figure 5.2.*

Below are presented some properties of $\lambda_{\mathcal{G}}$ used in the interpretation of $\lambda_{\mathcal{G}}$ into $\lambda^{\frown}$ exhibited in the following section.

Lemma 5.1.13 (Generation lemma for $\mathcal{G}$) *If $\mathcal{G}^x_f(U, a)$ has a derivation D, then only one rule can be applied as the last step of D.*

Proof. By case analysis on a. Note that only the conclusions of the rules 5 and 6 can be matched. Furthermore, in order to match the conclusions of such rules a must be of the form $f\,(z\,\vec{b})$, in which case rule 5 cannot be applied as last rule because its left premise would be underivable. □

Lemma 5.1.14 *If $\mathcal{G}^x_f(U, a)$ and $U \subseteq V$, then $\mathcal{G}^x_f(V, a)$.*

Proof. By induction on the derivation of $\mathcal{G}^x_f(U, a)$. The interesting case is when the last rule applied is rule 7.

Assume $a \equiv \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ and the last step is

$$\dfrac{\mathcal{G}^x_f(U, e) \quad \mathcal{G}^x_f(U, b_i) \quad (1 \leq i \leq n)}{\mathcal{G}^x_f(U, \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})}$$

$$
\begin{array}{lcl}
\text{(var)} & \dfrac{}{\Gamma \vdash x : \sigma} & \text{if } (x:\sigma) \in \Gamma \\[2ex]
\text{(abs)} & \dfrac{\Gamma, x:\tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \to \sigma} & \\[2ex]
\text{(app)} & \dfrac{\Gamma \vdash e : \tau \to \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\,e' : \sigma} & \\[2ex]
\text{(cons)} & \dfrac{}{\Gamma \vdash c : \mathsf{Dom}_{\vec{\tau}}(c) \to d\,\vec{\tau}} & \text{if } c \in \mathsf{C}(d) \\[2ex]
\text{(case)} & \dfrac{\Gamma \vdash e : d\,\vec{\tau} \quad \Gamma \vdash e_i : \mathsf{Dom}_{\vec{\tau}}(c_i) \to \theta \quad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta} & \text{if } \mathsf{C}(d) = \{c_1, \ldots, c_n\} \\[2ex]
\text{(rec)} & \dfrac{\Gamma, f : d\,\vec{\tau} \to \sigma \vdash e : d\,\vec{\tau} \to \sigma \qquad \mathcal{G}^x_f(\emptyset, a)}{\Gamma \vdash (\mathsf{letrec}\ f = e) : d\,\vec{\tau} \to \sigma} & \text{if } e \equiv \lambda x.a
\end{array}
$$

Figure 5.3: Typing rules for $\lambda_{\mathcal{G}}$

- If $e \not\equiv z\,\vec{a}$ or if $e \equiv z\,a_1 \ldots a_m$, $z \notin U \cup \{x\}$ and $z \notin V$, then by induction hypothesis $\mathcal{G}^x_f(V, e)$ and $\mathcal{G}^x_f(V, b_i)$ for $1 \leq i \leq n$, thus $\mathcal{G}^x_f(V, a)$ can be derived using rule 7.
- Consider now $e \equiv z\,a_1 \ldots a_m$, $z \notin U \cup \{x\}$ and $z \in V$. Each b_i must be of the form $\lambda y_1. \ldots \lambda y_{\mathsf{ar}(c_i)}. e_i$. Let $Q_i \equiv V \cup \{y_j \mid \mathsf{RP}(j, \mathsf{D}(c_i))$ for $1 \leq j \leq \mathsf{ar}(c_i)\}$. For $1 \leq i \leq n$, since $V \subseteq Q_i$, using the induction hypothesis $\mathcal{G}^x_f(Q_i, b_i)$ and then, by Lemma 5.1.13, $\mathcal{G}^x_f(Q_i, e_i)$. Also, from the induction hypothesis we have $\mathcal{G}^x_f(V, a_j)$ for $1 \leq j \leq m$ and therefore, applying rule 8, $\mathcal{G}^x_f(V, a)$.

The remaining cases can easily be proved using the induction hypothesis. $\square$

Lemma 5.1.15 (Generation lemma for $\lambda_{\mathcal{G}}$)

1. $\Gamma \vdash x : \sigma \;\Rightarrow\; (x : \sigma) \in \Gamma$
2. $\Gamma \vdash e\,e' : \sigma \;\Rightarrow\; \exists \tau \in \mathcal{T}.\, \Gamma \vdash e : \tau \to \sigma \;\wedge\; \Gamma \vdash e' : \tau$
3. $\Gamma \vdash \lambda x.e : \theta \;\Rightarrow\; \theta \equiv \tau \to \sigma \;\wedge\; \Gamma, x : \tau \vdash e : \sigma$
4. $\Gamma \vdash c : \theta \;\Rightarrow\; \theta \equiv \mathsf{Dom}_{\vec{\tau}}(c) \to d\,\vec{\tau}$ *with* $c \in \mathsf{C}(d)$
5. $\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 | \ldots | c_n \Rightarrow e_n\} : \theta \;\Rightarrow\; \exists d \in \mathcal{D}\, \exists \vec{\tau} \in \mathcal{T}.\ \Gamma \vdash e : d\,\vec{\tau} \;\wedge\; \Gamma \vdash e_i : \mathsf{Dom}_{\vec{\tau}}(c_i) \to \theta$ *for* $1 \leq i \leq n$ *with* $c_i \in \mathsf{C}(d)$
6. $\Gamma \vdash \mathsf{letrec}\ f = e : \theta \;\Rightarrow\; \theta \equiv d\,\vec{\tau} \to \sigma \wedge \Gamma, f : d\,\vec{\tau} \to \sigma \vdash e : d\,\vec{\tau} \to \sigma \wedge e \equiv \lambda x.a \wedge \mathcal{G}^x_f(\emptyset, a)$

Proof. By inspection on the derivation of the antecedent judgments. $\square$

5.2 From $\lambda_{\mathcal{G}}$ to $\lambda^{\frown}$

In this section we show that $\lambda^{\frown}$ is a more general system than $\lambda_{\mathcal{G}}$. The Examples 5.1.8 and 5.1.9 already illustrated are terms typable in $\lambda^{\frown}$ that cannot be typed in $\lambda_{\mathcal{G}}$. In this section we show that: *if* $\Gamma \vdash_{\lambda_{\mathcal{G}}} a : \sigma$ *then* $\Gamma \vdash_{\lambda^{\frown}} a : \sigma$ (the subscript at the turnstyle sign indicating the type system considered, and implicitly using the notation $d\,\vec{\tau}$ to abbreviate $d^{\infty}\vec{\tau}$). Naturally, the main difficulty in moving from $\lambda_{\mathcal{G}}$ to $\lambda^{\frown}$ is posed by letrec-expressions because the two systems have different kinds of typing rules for these expressions.

Given $\Gamma \vdash_{\lambda_{\mathcal{G}}} (\mathsf{letrec}\ f = \lambda x.a) : d\,\vec{\tau} \to \sigma$, by the generation lemma for $\lambda_{\mathcal{G}}$, we have

$$\Gamma, f : d\,\vec{\tau} \to \sigma, x : d\,\vec{\tau} \vdash_{\lambda_{\mathcal{G}}} a : \sigma \quad \wedge \quad \mathcal{G}_f^x(\emptyset, a) \tag{5.4}$$

However, we would want to have

$$\Gamma, f : d^{\imath}\vec{\tau} \to \sigma, x : d^{\widehat{\imath}}\vec{\tau} \vdash_{\lambda^{\frown}} a : \sigma \qquad (\imath \text{ fresh in } \Gamma, \vec{\tau}) \tag{5.5}$$

in order to use the $\lambda^{\frown}$ rec-rule and so derive $\Gamma \vdash_{\lambda^{\frown}} (\mathsf{letrec}\ f = \lambda x.a) : d\,\vec{\tau} \to \sigma$. Intuitively (5.4) is sufficient to guarantee (5.5) because, as we have $\mathcal{G}_f^x(\emptyset, a)$, all the possible occurrences of f in a are of the form $f\,(z\,\vec{a})$, with z being a recursive component of x. In (5.5) we have $x : d^{\widehat{\imath}}\vec{\tau}$ so, if z is a recursive component of x we should have $z : \vec{\gamma} \to d^{\imath}\vec{\tau}$. Hence $f\,(z\,\vec{a})$ is also typable in $\lambda^{\frown}$.

The remainder of this subsection is devoted to the embedding of $\lambda_{\mathcal{G}}$ into $\lambda^{\frown}$. In this embedding the Main Lemma below plays a central role. There we present the full construction underlying the lemma because it explains the details of the relation between the systems $\lambda_{\mathcal{G}}$ and $\lambda^{\frown}$.

In the following we assume that each variable x_i is uniquely associated to a stage variable $\jmath_i$. Recall also that, in $\lambda^{\frown}$, the notation $d\,\vec{\tau}$ abbreviates the datatype $d^{\infty}\vec{\tau}$.

Lemma 5.2.1 (Main Lemma) *Let*

$$\begin{array}{lcll}
\Gamma_0 & = & \Gamma & \\
\Gamma_i & = & \Gamma_{i-1}, f_i : d_i\,\vec{\tau_i} \to \sigma_i, x_i : d_i\,\vec{\tau_i} & \text{for } 1 \le i \le n \\
 & & & \\
\widehat{\Gamma_0} & = & \Gamma_0 & \\
\widehat{\Gamma_i} & = & \widehat{\Gamma_{i-1}}, f_i : d_i^{\jmath_i}\vec{\tau_i} \to \sigma_i, x_i : d_i^{\widehat{\jmath_i}}\vec{\tau_i} & \text{for } 1 \le i \le n \\
 & & & \text{where } \jmath_i \text{ is a fresh stage variable} \\
 & & & \text{associated to } x_i
\end{array}$$

and, for $1 \le i \le n$, let U_i be a set of variables such that for each $z \in U_i$, $z : \vec{\gamma} \to d_i\,\vec{\tau_i} \in \Gamma$ and so that all the U_i's are disjoint. Then,

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} a : \sigma \quad \wedge \quad (\forall i \in \{1, \ldots, n\}.\ \mathcal{G}_{f_i}^{x_i}(U_i, a)) \quad \Rightarrow \quad [\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} a : \sigma$$

where $U = \bigcup_{1 \le i \le n} U_i$ and $[\widehat{\Gamma_n}]_U$ is obtained from $\widehat{\Gamma_n}$ by replacing each declaration $z : \vec{\gamma} \to d_i\,\vec{\tau_i}$ (with $z \in U_i$) by $z : \vec{\gamma} \to d_i^{\jmath_i}\,\vec{\tau_i}$. Note that in order to make Γ_n a context, in particular, all the f_i's and x_i's must be distinct and cannot be declared in Γ.

Proof. By induction on the structure of a.

1. Case $a \equiv x$, the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} x : \sigma \quad \wedge \quad \forall i \in \{1, \ldots, n\}.\ \mathcal{G}_{f_i}^{x_i}(U_i, x)$$

- If $x \equiv x_i$ for some $i \in \{1, \ldots, n\}$, $\sigma \equiv d_i\, \vec{\tau_i}$ and so, $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} x : d_i^{\widehat{\jmath_i}}\, \vec{\tau_i}$. As $d_i^{\widehat{\jmath_i}}\, \vec{\tau_i} \sqsubseteq d_i^{\infty} \vec{\tau_i}$, using the rule (sub),

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} x : d_i \vec{\tau_i}$$

- If $x \not\equiv x_i$ for every $i \in \{1, \ldots, n\}$ then, since $\forall i \in \{1, \ldots, n\}.\ \mathcal{G}_{f_i}^{x_i}(U_i, x)$, $x \not\equiv f_i$ for every $i \in \{1, \ldots, n\}$. Therefore, using Lemma 5.1.15, $(x : \sigma) \in \Gamma$. Hence
  (a) If $x \notin U$, then $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} x : \sigma$.
  (b) If $x \in U$ then, $\sigma \equiv \vec{\gamma} \to d_i \vec{\tau_i}$ for some $i \in \{1, \ldots, n\}$. So, $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} x : \vec{\gamma} \to d_i^{\jmath_i} \vec{\tau_i}$ and, since $\vec{\gamma} \to d_i^{\jmath_i}\, \vec{\tau_i} \sqsubseteq \vec{\gamma} \to d_i^{\infty} \vec{\tau_i}$, by (sub)

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} x : \sigma$$

2. Case $a \equiv e\, e'$ the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} e\, e' : \sigma \quad \wedge \quad \forall i \in \{1, \ldots, n\}.\ \mathcal{G}_{f_i}^{x_i}(U_i, e\, e')$$

- If $e \equiv f_i$ for some $i \in \{1, \ldots, n\}$ then, by Lemma 5.1.13, $e' \equiv z\, \vec{b}$, $\mathcal{G}_{f_i}^{x_i}(U_i, e')$ and $z \in U_i$. Moreover:
  (a) $\Gamma_n \vdash_{\lambda_{\mathcal{G}}} f_i : d_i \vec{\tau_i} \to \sigma_i$ and $\sigma \equiv \sigma_i$. So, $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} f_i : d_i^{\jmath_i}\, \vec{\tau_i} \to \sigma$.
  (b) $\Gamma_n \vdash_{\lambda_{\mathcal{G}}} z : \vec{\gamma} \to d_i \vec{\tau_i}$. So, $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} z : \vec{\gamma} \to d_i^{\jmath_i}\, \vec{\tau_i}$ because $z \in U_i$.
  (c) $\Gamma_n \vdash_{\lambda_{\mathcal{G}}} \vec{b} : \vec{\gamma}$ (using this notation to abbreviate the list of judgments $\Gamma_n \vdash_{\lambda_{\mathcal{G}}} b_k : \gamma_k$ for each $b_k \in \vec{b}$) and for every $b_k \in \vec{b}$, $\mathcal{G}_{f_i}^{x_i}(U_i, b_k)$ because $z \neq f_i$. For $j \in \{1, \ldots, n\} - \{i\}$, $e \neq f_j$ and, by Lemma 5.1.13, $\mathcal{G}_{f_j}^{x_j}(U_j, b_k)$ for every $b_k \in \vec{b}$. Therefore, by induction hypothesis,

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} \vec{b} : \vec{\gamma}$$

  From (a), (b) and (c), using (app), one can then obtain

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} f_i\, (z\, \vec{b}) : \sigma$$

- If $e \not\equiv f_i$ for every $i \in \{1, \ldots, n\}$ then, using Lemmas 5.1.13 and 5.1.15,

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} e : \gamma \to \sigma \quad \wedge \quad \Gamma_n \vdash_{\lambda_{\mathcal{G}}} e' : \gamma$$

  and

$$\forall i \in \{1, \ldots, n\}.\ \ \mathcal{G}_{f_i}^{x_i}(U_i, e) \quad \wedge \quad \mathcal{G}_{f_i}^{x_i}(U_i, e')$$

  Hence, by induction hypothesis,

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} e : \gamma \to \sigma \quad \wedge \quad [\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} e' : \gamma$$

  Using the rule (app) one obtains $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} e\, e' : \sigma$.

3. Case $a \equiv \lambda y.e$, the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} \lambda y.e : \sigma \quad \wedge \quad \forall i \in \{1, \ldots, n\}.\ \mathcal{G}_{f_i}^{x_i}(U_i, \lambda y.e)$$

Using Lemma 5.1.15, $\sigma \equiv \gamma \to \sigma'$ for some $\gamma, \sigma' \in \mathcal{T}$ and also

$$\Gamma, y : \gamma, f_1 : d_1 \vec{\tau_1} \to \sigma_2, x_1 : d_1 \vec{\tau_1}, \ldots, f_n : d_n \vec{\tau_n} \to \sigma, x_n : d_n \vec{\tau_n} \vdash_{\lambda_{\mathcal{G}}} e : \sigma'$$

By Lemma 5.1.13, $\forall i \in \{1, \ldots, n\}.\ \mathcal{G}_{f_i}^{x_i}(U_i, e)$. Hence, by induction hypothesis,

$$[\Gamma, y : \gamma, f_1 : d_1^{\jmath_1}\, \vec{\tau_1} \to \sigma_1, x_1 : d_1^{\widehat{\jmath_1}}\, \vec{\tau_1}, \ldots, f_n : d_n^{\jmath_n}\, \vec{\tau_n} \to \sigma_n, x_n : \widehat{d_n^{\jmath_n} \vec{\tau_n}}]_U \vdash_{\lambda^{\frown}} e : \sigma'$$

We know that $y \notin \Gamma$ and so, $y \notin U$. Therefore, $[\widehat{\Gamma_n}]_U, y : \gamma \vdash_{\lambda^{\frown}} e : \sigma'$ and the proof of this case is concluded applying rule (abs).

4. Case $a \equiv c$ and $c \in \mathsf{C}(d)$, one assumes $\Gamma_n \vdash_{\lambda_{\mathcal{G}}} c : \mathsf{Dom}_{\vec{\tau}}(c) \to d\,\vec{\tau}$. Thus in $\lambda^{\frown}$ one can apply (cons) to obtain $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} c : \mathsf{Dom}^{\infty}_{\vec{\tau}}(c) \to \widehat{d^{\infty}}\vec{\tau}$, and since, $\mathsf{Dom}_{\vec{\tau}}(c)$ is being used as an abbreviation for $\mathsf{Dom}^{\infty}_{\vec{\tau}}(c)$ and $\widehat{d^{\infty}}\vec{\tau} \sqsubseteq d^{\infty}\vec{\tau}$, one also has

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} c : \mathsf{Dom}_{\vec{\tau}}(c) \to d\,\vec{\tau}$$

5. Case $a \equiv \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_m \Rightarrow b_m\}$, the hypotheses are

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} \mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} : \sigma \tag{5.6}$$

$$\forall i \in \{1, \ldots, n\}.\ \mathcal{G}^{x_i}_{f_i}(U_i, \mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}) \tag{5.7}$$

and from (5.6), applying Lemma 5.1.15, there exists d, $\vec{\tau}$ such that

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} e : d\,\vec{\tau} \tag{5.8}$$

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} b_k : \mathsf{Dom}_{\vec{\tau}}(c_k) \to \sigma \tag{5.9}$$

for each $1 \leq k \leq m$. Two cases can now occur.

- If $e \not\equiv z\,\vec{a}$ or $e \equiv z\,\vec{a}$ and $z \notin U_i \cup \{x_i\}$ for every $i \in \{1, \ldots, n\}$, then from (5.7) by Lemma 5.1.13

$$\forall i \in \{1, \ldots, n\}. \forall k \in \{1, \ldots, m\}.\ \mathcal{G}^{x_i}_{f_i}(U_i, e) \quad \wedge \quad \mathcal{G}^{x_i}_{f_i}(U_i, b_k)$$

  Thus applying the induction hypothesis to (5.8), followed by rule (sub), one has $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} e : \widehat{d^{\infty}}\vec{\tau}$ and applying the induction hypothesis to (5.9) one obtains $[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} b_k : \mathsf{Dom}^{\infty}_{\vec{\tau}}(c_k) \to \sigma$. Derivations of these judgments can now be put together by means of the rule (case), proving

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^{\frown}} \mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} : \sigma$$

- Consider now that $e \equiv z\,\vec{a}$ and $z \in U_i \cup \{x_i\}$ for some $i \in \{1, \ldots, n\}$ (recall that such i must be unique since: the U_j's are disjoint and contain none of the x_j's; and the x_j's are distinct). Let, for each $1 \leq k \leq m$,

$$\begin{cases} b_k \equiv \lambda \vec{y_k}.e_k \\ V_{k,i} \equiv U_i \cup \{y_{k,r} \mid \mathsf{RP}(r, \mathsf{D}(c_k)) \text{ for } 1 \leq r \leq \mathsf{ar}(c_k)\} \\ V_{k,j} \equiv U_j \text{ for } j \in \{1, \ldots, n\} - \{i\} \\ V_k \equiv \bigcup_{1 \leq j \leq n} V_{k,j} \end{cases}$$

  where $y_{k,r}$ denotes the r-th component of vector $\vec{y_k}$. Applying Lemma 5.1.13 to (5.7), one can now assume that for each $1 \leq j \leq n$

$$\forall\, a_s \in \vec{a}\ .\ \mathcal{G}^{x_j}_{f_j}(U_j, a_s)\ \wedge\ \forall\, k \in \{1, \ldots, m\}\ .\ \mathcal{G}^{x_j}_{f_j}(V_{k,j}, e_k) \tag{5.10}$$

  From (5.9) by Lemma 5.1.15, one has

$$\Gamma, \vec{y_k} : \mathsf{Dom}_{\vec{\tau}}(c_k), \Gamma_n \setminus \Gamma \vdash_{\lambda_{\mathcal{G}}} e_k : \sigma$$

  where $\Gamma_n \setminus \Gamma$ is the context Γ_n without the declarations in Γ. Moreover, $y_{k,r} : \gamma_{\vec{y_{k,r}}} \to d_i\vec{\tau_i} \in (\vec{y_k} : \mathsf{Dom}_{\vec{\tau}}(c_k))$ for each $1 \leq r \leq \mathsf{ar}(c_k)$ such that $\mathsf{RP}(r, \mathsf{D}(c_k))$ and thus, for each $1 \leq k \leq m$ and $1 \leq j \leq n$, and for each $z \in V_{k,j}$ we have $z : \vec{\gamma_z} \to d_i\,\vec{\tau_i} \in (\Gamma, \vec{y_k} : \mathsf{Dom}_{\vec{\tau}}(c_k))$. Hence, by the induction hypothesis

$$[\Gamma, \vec{y_k} : \widehat{\mathsf{Dom}_{\vec{\tau}}(c_k)}, \Gamma_n \setminus \Gamma]_{V_k} \vdash_{\lambda^\frown} e_k : \sigma$$

from which one can show $[\widehat{\Gamma_n}]_U, \vec{y_k} : \mathsf{Dom}^{\jmath_i}_{\vec{\tau}}(c_k) \vdash_{\lambda^\frown} e_k : \sigma$ (observe that $V_k = U \cup \{y_{k,r} \mid \mathsf{RP}(r, \mathsf{D}(c_k))$ for $1 \leq r \leq \mathsf{ar}(c_k)\})$ and therefore, by the rule (abs), $[\widehat{\Gamma_n}]_U \vdash_{\lambda^\frown} b_k : \mathsf{Dom}^{\jmath_i}_{\vec{\tau}}(c_k) \to \sigma$ holds.

To conclude the proof of this case, it suffices now to show that

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^\frown} e : d_i^{\widehat{\jmath_i}}\,\vec{\tau_i} \tag{5.11}$$

and to use then the rule (case). In order to prove (5.11) one proceeds as follows.

(a) Case $e \equiv x_i$, $[\widehat{\Gamma_n}]_U \vdash_{\lambda^\frown} x_i : d_i^{\widehat{\jmath_i}}\,\vec{\tau_i}$ is derivable.

(b) Case $e \equiv z\,\vec{a}$ with $z \in U_i$, from (5.8) by Lemma 5.1.15, $\Gamma_n \vdash_{\lambda_{\mathcal{G}}} z : \vec{\gamma} \to d\,\vec{\tau}$ (thus, $d\,\vec{\tau} \equiv d_i\,\vec{\tau_i}$) and

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} \vec{a} : \vec{\gamma} \tag{5.12}$$

Now, since (5.10) holds, one can apply the induction hypothesis to (5.12) obtaining $[\widehat{\Gamma_n}]_U \vdash_{\lambda^\frown} \vec{a} : \vec{\gamma}$. It is also true that $z : \vec{\gamma} \to d_i^{\jmath_i}\,\vec{\tau_i} \in [\widehat{\Gamma_n}]_U$, for $z \in U_i$, and since $\vec{\gamma} \to d_i^{\jmath_i}\,\vec{\tau_i} \sqsubseteq \vec{\gamma} \to d_i^{\widehat{\jmath_i}}\,\vec{\tau_i}$, by (sub) and (app), $[\widehat{\Gamma_n}]_U \vdash_{\lambda^\frown} z\,\vec{a} : d_i^{\widehat{\jmath_i}}\,\vec{\tau_i}$ holds.

6. Case $a \equiv (\mathsf{letrec}\ f = \lambda x.a')$, we must have $\sigma \equiv d\,\vec{\tau} \to \sigma'$ for some $d\,\vec{\tau}, \sigma' \in \mathcal{T}$, and the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathcal{G}}} (\mathsf{letrec}\ f = \lambda x.a') : d\,\vec{\tau} \to \sigma' \quad \wedge \quad \forall i \in \{1, \ldots, n\}.\ \mathcal{G}^{x_i}_{f_i}(U_i, \mathsf{letrec}\ f = \lambda x.a')$$

By Lemma 5.1.15 we get

$$\Gamma_n, f : d\,\vec{\tau} \to \sigma', x : d\,\vec{\tau} \vdash_{\lambda_{\mathcal{G}}} a' : \sigma'$$

and $\mathcal{G}^x_f(\emptyset, a')$. Again by the hypothesis, by Lemma 5.1.13, $\forall i \in \{1, \ldots, n\}.\ \mathcal{G}^{x_i}_{f_i}(U_i, a')$ hence, assuming $U_{n+1} = \emptyset$, $x_{n+1} = x$ and $f_{n+1} = f$, we have

$$\forall i \in \{1, \ldots, n+1\}.\ \ \mathcal{G}^{x_i}_{f_i}(U_i, a')$$

So, by induction hypothesis $[\widehat{\Gamma_{n+1}}]_U \vdash_{\lambda^\frown} a' : \sigma'$. Applying (abs) and (rec) we get $[\widehat{\Gamma_n}]_U \vdash_{\lambda^\frown} (\mathsf{letrec}\ f = \lambda x.a') : (d^\imath\vec{\tau} \to \sigma')[\imath := \infty]$. Hence, for $\imath$ has no occurrences in $\vec{\tau}$ nor in σ',

$$[\widehat{\Gamma_n}]_U \vdash_{\lambda^\frown} (\mathsf{letrec}\ f = \lambda x.a') : d\,\vec{\tau} \to \sigma'$$

□

We are now ready to prove the main result of this section.

Theorem 5.2.2

$$\Gamma \vdash_{\lambda_{\mathcal{G}}} a : \sigma \quad \Rightarrow \quad \Gamma \vdash_{\lambda^\frown} a : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash_{\lambda_{\mathcal{G}}} a : \sigma$.

(rec) Assume the last step is

$$\frac{\Gamma, f : d\,\vec{\tau} \to \sigma \;\vdash_{\lambda_{\mathcal{G}}}\; e : d\,\vec{\tau} \to \sigma \qquad \mathcal{G}^x_f(\emptyset, a)}{\Gamma \;\vdash_{\lambda_{\mathcal{G}}}\; (\textsf{letrec}\ f = e) : d\,\vec{\tau} \to \sigma} \quad \text{with} \;\; e \equiv \lambda x.a$$

By Lemma 5.1.15, $\Gamma, f : d\,\vec{\tau} \to \sigma, x : d\,\vec{\tau} \;\vdash_{\lambda_{\mathcal{G}}}\; a : \sigma$ and since $\mathcal{G}^x_f(\emptyset, a)$ we are in conditions of applying the Main Lemma and conclude $\Gamma, f : d^{\imath}\vec{\tau} \to \sigma, x : d^{\widehat{\imath}}\vec{\tau} \;\vdash_{\widehat{\lambda}}\; a : \sigma$. Hence, applying the rules (abs) and (rec) one derives $\Gamma \;\vdash_{\widehat{\lambda}}\; (\textsf{letrec}\ f = e) : (d^{\imath}\vec{\tau} \to \sigma)[\imath := \infty]$ which is the same as

$$\Gamma \;\vdash_{\widehat{\lambda}}\; (\textsf{letrec}\ f = e) : d\,\vec{\tau} \to \sigma$$

for $\imath$ does not occur in σ or $\vec{\tau}$.

All the remaining cases can be easily proved using the induction hypothesis. $\square$

Chapter 6

Related Work and Conclusion

6.1 Related Work

For the sake of clarity, we split existing systems into five categories: (1) based on traditional-style terminating recursors, (2) based on a fixpoint operator controlled by a syntactic guard predicate, (3) exploiting pattern matching, (4) based on a fixpoint operator controlled by an unusual typing ensuring that the recursion actually terminates, (5) relying on other type-based techniques for ensuring termination.

Comparison with works on traditional-style terminating recursors

Most formalizations of inductive types in type theory support recursive definitions only indirectly via eliminators behaving as iterators or primitive recursors [88, 120, 118, 116, 46, 54, 62, 102, 6, 131]. Such systems are well-understood meta-theoretically and enjoy good properties, but are hard to use in practical programming: this requires the programmer to translate all recursive definitions into explicit definitions involving primitive recursion.

It is possible to devise similar eliminators capturing more sophisticated schemes of terminating recursion such as course-of-value iteration or course-of-value primitive recursion [133, 104], but the resulting systems are even clumsier to use practically.

Comparison with works relying on a fixpoint operator controlled by an external guard predicate

Coquand [45] introduces a simple guard predicate to ensure termination of fixpoint expressions in a calculus of infinite objects. Building on Coquand's work, Giménez [67] defines a more liberal guarded-by-constructors predicate for productive corecursion and also a guarded-by-destructors predicate for terminating recursion. Giménez shows that primitive recursor expressions can be rendered as fixpoint expressions guarded by one destructor. In the opposite direction, a fixpoint expression guarded-by-destructors can be coded as an expression involving primitive recursors, but the translation is not uniform. The predicates defined by Giménez form the basis of the mechanism for (co)inductive types in Coq. More recently, Blanqui [29], building on Jouannaud and Okada [82], propose another definition of the guard predicate for inductive types, that allows for yet more expressions to be typed. In a similar line of research, Abel and Altenkirch [3] propose a basic framework for studying and comparing the different termination conditions that have been proposed so far, focusing their attention on what conditions should be fulfilled for a checking to be sound. An application of such framework to a particular condition can be found in [2].

One possible objection against this line of work is that the system becomes more unpredictable to the user as the complexity of the guard predicate builds up. Besides, the guard predicate remains purely syntactic, which is not appropriate for a number of applications, including separate compilation or interactive proof construction.

Comparison with works on pattern-matching

Coquand [44] investigates the use of pattern-matching in dependent type theory. While pattern-matching yields leaner definitions, its proof-theoretical status in the context of dependent types remains unclear. Differently from guarded-by-destructors recursion, general pattern-matching is not a conservative improvement over primitive recursors: Hofmann and Streicher [76] prove the derivability of uniqueness of equality proofs in a type theory with pattern-matching, while equality proofs cannot be shown to be unique in the usual Calculus of Inductive Constructions. To our knowledge, there is no complete account of the meta-theoretical properties of pattern-matching in dependent type theory. McBride [104] has shown that, under the uniqueness of equality proofs as an extra axiom, pattern matching is admissible. Ongoing work on checking the termination of recursive function definitions in functional languages, see e.g. [132, 3, 64, 94], bears relevance for this direction of type-theoretic developments. Of particular interest for the future type-theoretic formalizations might be the recent work of Lee, Jones and Ben-Amram [87] on the size-change principle for program termination.

As to implementations, restricted forms of pattern-matching have been implemented in Coq by Cornes [47] and Lego by Elbers [55]. Both implementations take advantage of translations to recursors. Pattern-matching has also been consistently supported in Alf and its subsequent versions, although no mechanism for termination checking was ever implemented. In order to simplify the proof engine, Agda, which is the latest incarnation of Alf, only supports a limited form of pattern-matching in which variables are only allowed to occur once in the type of a constructor. This restriction rules out, for example, inductive definitions such as equality.

Comparison with works on guarded types

This line of work is really about non-traditional-style terminating recursors that look like fixpoint operators, but where the computation is guaranteed to terminate by an unusual (stronger) typing system. Such a system involves introducing some kind of annotations on recursive types, a notion of subtyping enabling the transformation of such annotations, and a typing rule for the term $\mathsf{letrec}\ f = e$ where the type of f and the type of e are marked differently. In this sense, some of the systems mentioned in this section are not far from the so called *abstract interpretation* techniques [48], even though they are formulated from a type-theoretical point of view. The exact relation of such typing systems with respect to abstract interpretation techniques has not been studied in detail yet, and could be a subject for further research.

Mendler [105] was, to our knowledge, the first author to propose a formalization of inductive and coinductive types in a simply typed lambda calculus where primitive recursion and primitive corecursion were formulated in a fixpoint-like style. In Mendler's system, type annotations on the fixpoint rule correspond to type variables. [106] considered a system supporting only iteration and coiteration. Works that comment on these two papers include [89, 62, 134, 133, 100, 103, 131]. Among these, [89, 62] were the first papers to contrast and compare traditional-style and Mendler-style terminating recursors. Uustalu [134, 135, 133] showed that Mendler's approach is readily generalizable for course-of-value (co)recursion (in other words, full structural (co)recursion).

Giménez [65] introduced an extension of the Calculus of Constructions with inductive and coinductive types, called CC^{∞}. The fixpoint rules in CC^{∞} make use of three kinds of marks, corresponding to the stages ∞, $\imath$ and $\widehat{\imath}$ using the notation of $\lambda\widehat{\ }$. This means that in CC^{∞} the hat operator cannot be applied to another stage, but only to stage variables. In [65], marks also have a second component, specifying whether the recursive type is inductive or coinductive. There is no stage polymorphism, and hence the function *div* of Example 3.4.4 cannot be typed.

One of the main disadvantages of [65] is that it tried to tackle too many problems at once, rendering the typing calculus less clear. Among the extra features introduced in CC^{∞} which are not considered in this work are the following:

– Inductive lists are considered a subtype of coinductive ones, so that a function defined on the type of coinductive lists can also be used on inductive lists.

– Annotations are placed on typing judgments, writing $x :^{s} \mathsf{List}$ instead of $x : \mathsf{List}^{s}$. One of the original motivations for this notation was to enable the description of an abstract recursion schema, where the type of the decreasing argument of the function is abstracted away using a term of the form $\lambda A : Set \cdot \mathsf{letrec}\ f = \lambda x :^{s} A \cdot e$. Also, the choice of having two different universal quantifiers renders unnecessary the introduction of two types of lists (one for inductive and the other for coinductive ones) with the same constructors. On the other hand, it is less clear how an ordinal based semantics like the one proposed in this work could be used to make sense of a term of the form $\lambda A : Set \cdot \lambda x :^{s} A \cdot e$. This is why, even though annotated quantifications were kept, the calculus in [65] forces A in a term of the form $\lambda x :^{s} A \cdot e$ to have a recursive type at its rightmost position.

– CC^{∞} is built on the top of the Calculus of Constructions, so it uses Church's style for variable binding, where the type of the abstracted variable is explicitly mentioned. Thus, types—and hence marks—may appear in the terms. As a consequence, the reduction rule for fixpoints has to replace all mark variables by the ∞ mark, in order to avoid having residual unbound mark identifiers in the definiens. Note that this problem does not appear in $\lambda\widehat{\ }$, where variable binding is *à la* Curry.

Giménez [66] introduced CCR, a different extension of the Calculus of Constructions with inductive and coinductive types, based on (not fully general) sub- and supertyping and bounded universal quantification over types. In CCR, marks are represented as type variables, like in Mendler's works, the hat operator is a type constructor, and stages are just types. Since stage variables are type variables, stage replacement corresponds just to the ordinary substitution operation of the calculus. The calculus in [66] was the first calculus to introduce stage polymorphism, enabling to type definitions like the function *div* of Example 3.4.4. However, the stage polymorphism allowed by types involving bounded quantification is constrained to the ascending chain of approximations of the inductive type. The calculus of $\lambda\widehat{\ }$ is very much inspired by [66], but replaces subtypes with approximating types, and bounded type quantification with stage quantification—the change allows the structure of stages to be uniform over all datatypes and simplifies the introduction of recursive definitions on mutually dependent inductive types. The meta-theory of CCR has not been studied yet, nor its connection with implemented extensions of a calculus of (co)inductive constructions like the system Coq. The detailed study of the main meta-theoretical properties of $\lambda\widehat{\ }$ presented in this work can be seen as a basic stage for developing the meta-theory of an extension of the Calculus of Construction where the termination of functions is ensured by typing constraints.

Amadio and Coupet [8] define a simply typed λ-calculus *à la* Curry featuring guarded coinductive types. Starting from Coquand's guardedness condition, they propose a semantics for such extension of lambda calculus based on partial equivalent relations and ordinal iteration to interpret coinductive types. From that semantics, they derive a typing rule for corecursive definitions using a mark system with three kinds of marks, that correspond in our notation to ∞, $\imath$ and $\widehat{\imath}$. The semantic interpretation used to study the meta-theory of $\lambda\widehat{\ }$ in this work is actually an extension of the one introduced in [8] for coinductive types. Also, the need for the constraint $\imath \ \mathsf{pos}\ \sigma$ in the typing rule for recursive definitions have been already noticed by Amadio and Coupet. Their calculus introduces an extra rule enabling to treat nested fixpoint definitions of the form ($\mathsf{letrec}\ f = (\mathsf{letrec}\ g = e)$) by reusing the mark introduced in the definition of f as the mark for the variable g. However, the calculus described in [8] lacks of full stage polymorphism, and does not consider inductive types. On the positive side, their calculus is shown to have decidable type inference in [11].

Barras [17] formalizes in Coq a variant of Giménez' calculus CC^{∞}, with the purpose of proving the decidability of its typing judgment and extracting a type-checker from the proof. In Barras' calculus, inductive types are annotated with lists of marks, each one corresponding to the stages ∞, $\imath$ and $\widehat{\imath}$ of our system. The use of lists of marks enables to type nested recursive function definitions like the ones considered in [8], but for inductive types. He does not consider coinductive types nor stage polymorphism. As the underlying lambda calculus is *à la* Church, Barras introduces a distinguished primitive type $\mathcal{M}$ for marks, and marks are just variables of that type. Mark variables are bound in fixpoint terms, so mark erasure in fixpoint reductions corresponds just to ordinary variable substitution. The complete meta-theory of Barras' system has not been studied yet, but his system is the only mark based one for which a type-checking algorithm has been developed.

Other type-based approaches to termination analysis

Xi [143] proposes a system of restricted dependent types, built upon DML [142], to ensure program termination. In essence, his system is closely related to ours since it uses stage information to ensure termination. However, Xi's system differs from ours in its expressiveness and complexity: while we focus on the weakest calculus that uses type-based termination and extends other calculi based on a simple syntactic guard predicate, Xi presents a very rich system with stage arithmetic, and a notion of metric that is very useful to handle functions in several arguments. Of course, expressiveness is achieved at the cost of simplicity and Xi's system is much more complex than ours. Xi's system is qualified for practical termination checking of realistic functional programs. However, his approach cannot handle higher-order datatypes which limits its applicability in proof assistants. Grobauer [71] uses DML to find cost recurrences for first-order recursive definitions: a cost recurrence is an upper bound to the running time of the program w.r.t. the size of its input, and hence a witness that the recursive definition is terminating. In his work, Grobauer exploits complex features of DML, including stage arithmetic, so his techniques do not seem directly applicable to $\lambda\widehat{\ }$. Closely related is the recent work on sized types for termination and productivity checking of functional programs [77, 115, 35].

6.2 Conclusion

We have introduced $\lambda\widehat{\ }$, a novel type system for terminating recursive functions. The salient features of $\lambda\widehat{\ }$ are its type-based approach to ensure termination through the notion of stage, and its support for stage polymorphism. The system is conceptually simple, it overcomes the problems of guard-based solutions and keeps the technical overhead to a minimum.

The calculus of $\widehat{\lambda}$ is closely related to the works of Giménez [66] and of Amadio and Coupet [8]. While Giménez [66] guaranteed termination by types involving bounded quantification in the framework of the Calculus of Constructions, Amadio and Coupet [8] (only coinductive types are treated) used a simply-typed framework with a kind of stages. $\widehat{\lambda}$ combines the best of both worlds: iterated successor stages are possible, the inductive type is simply seen as stage infinity, and the stages may as well enter the result type of the recursively defined functions. This allows for more precise typings, which is a very valuable feature for defining functions (like the Euclidean division of Example 3.4.4) where the argument to the recursive call is derived from the input argument via another function. In [66], the same was achieved by the technically more demanding bounded quantification while in [8], this was only partly available, and required a second typing rule for letrec. Here, both are unified by allowing to instantiate the inferred type of letrec by an arbitrary stage expression. Moreover, the stage polymorphism of $\widehat{\lambda}$ allows to give unusually exact (and therefore informative) types to functions like length : $\mathsf{List}^s\tau \to \mathsf{Nat}^s$ (of Example 3.4.3) or ltobt : $\mathsf{List}^s\mathsf{Nat} \to \mathsf{BTree}^s\mathsf{Nat}$ (of Example 3.4.6). This kind of exact typings are impossible to achieve with bounded quantification, since the approximation of the inductive type is represented by a type variable bounded by the inductive type. Instead subtypes and bounded type quantification, $\widehat{\lambda}$ supports approximating types and stage quantification. This change allows the structure of stages to be uniform over all datatypes and simplifies the introduction of recursive definitions on mutually dependent inductive types.

We have proved $\widehat{\lambda}$ is well-behaved enjoying confluence, subject reduction and strong normalization. We have also proved that this system encompasses in a strict way (typability and even types are preserved) the system $\lambda_{\mathcal{G}}$ where termination ensured by a syntactical condition (following what is done by Giménez in [67]). $\widehat{\lambda}$ is powerful enough to encode many recursive definitions rejected by $\lambda_{\mathcal{G}}$, extends easily to mutually inductive types and supports separate compilation. In comparison to $\lambda_{\mathcal{G}}$, it has a much clearer syntax and admits a clean semantics; the strong normalization can be proved by means of a standard method. For practice, this means that $\widehat{\lambda}$ is less difficult to implement (implementing the guard condition of $\lambda_{\mathcal{G}}$ is error-prone) and the code written in it is more easily maintainable. This makes $\widehat{\lambda}$ a good candidate base system for type theory based proof-assistants such as Coq.

In order to validate this claim, the following steps need to be taken:

- develop type checking and type inference algorithms for $\widehat{\lambda}$. For the purpose of proof assistants, it may be of interest to study a calculus where type annotations are given and stage annotations are inferred. Actually, we have developed a type inference algorithm for $\widehat{\lambda}$ (and we have implemented it in Haskell) that correctly infers the types of all the functions presented in Section 3.4. We have not included this work in this document because more work has to be done to prove the soundness and completeness of our algorithm. However we hope to finish this task as soon as possible.

- extend $\widehat{\lambda}$ to dependent types and polymorphism *à la* system F, as in [17, 66], and to wider classes of recursive definitions such as mutually recursive definitions and recursive definitions in several parameters. In the case of dependent types, the interest is to support dependent eliminations (induction schemes) and inductive families. For more advanced forms of recursive definitions some form of stage arithmetic might be needed.

It should also be checked that the extension of $\widehat{\lambda}$ with coinductive types, sketched in [22], is well-behaved.

Part II

Constructor Subtyping

Chapter 7

An Informal Account of Constructor Subtyping

This chapter is an informal introduction (with examples) to the concepts of constructor subtyping and of extensible overloaded functions. A formal presentation of these features is done in the next chapters.

7.1 Motivations and Difficulties

Type theory with inductive types form the basis of most functional programming languages and proof-assistants. Type systems are pervasive in modern functional programming languages, such as Haskell [81] and ML [107]. These systems support inductive datatypes and the definition of functions by pattern matching. However, there are cases where the standard type systems are not flexible enough and some interesting programs are rejected (despite their semantical soundness) or are assigned some uninformative types. Proof-development systems, such as Coq [125] and Lego [92], rely on powerful type systems featuring inductive types (which capture the notion of algebraic datatype in a type-theoretical framework) and support the definition of functions by recursion and also a mechanism to prove properties by induction. Inductive definitions play an essential role in the expressibility of these systems and are extensively used in the formalization of programming languages, communication and cryptographic protocols, ... However, despite the success of such works, user efforts are often hindered by the rigidity of the existing tools. In order to improve the usability of these languages, it is important to devise flexible (and safe) type systems, in which programs and proofs may be written easily. In particular, it is important to have type systems that allow for more informative typings and the reuse of code in the development of proofs and programs.

Subtyping and *overloading* are mechanisms that enhance the flexibility of type systems. Their relevance in programming languages has long been recognized. Subtyping is a relation on types that expresses that one type is at least as general as another one and is embedded in the type system enforcing a *subsumption* rule stating that a term of a type σ is also of type τ whenever σ is a subtype of τ. This basic mechanism of subtyping is powerful enough to capture a variety of concepts in computer science, see e.g. [32], and its use is spreading both in functional programming languages, see e.g. [95, 119, 123], and in proof assistants, see e.g. [27, 91, 130]. Overloading of constants is the ability of constants to have several types. The combination of subtyping and overloading yields a concise and readable framework for describing datatypes in terms of their constructors. A typical example is the one of odd/even/natural numbers:

Types:	$\mathsf{Odd}, \mathsf{Even}, \mathsf{Nat}$
Subtype relation:	$\mathsf{Even} \leq \mathsf{Nat}$, $\mathsf{Odd} \leq \mathsf{Nat}$
Declarations:	$\mathsf{o} : \mathsf{Even}$
	$\mathsf{s} : \mathsf{Even} \rightarrow \mathsf{Odd}$
	$\mathsf{s} : \mathsf{Odd} \rightarrow \mathsf{Even}$
	$\mathsf{s} : \mathsf{Nat} \rightarrow \mathsf{Nat}$

Observe that constructor s is assigned more than one domain and codomain, so it is overloaded. Constructor o is only declared once as being of type Even, however, through subsumption we also have o as being of type Nat since Even is a subtype of Nat. Besides the declared types for s, the subsumption rule and subtyping relation extend structurally to types, so that we can infer two more types for s: $\mathsf{Even} \rightarrow \mathsf{Nat}$ and $\mathsf{Odd} \rightarrow \mathsf{Nat}$.

Constructor subtyping is a basic form of subtyping, in which an inductive type A is viewed as a subtype of another type B if B has more inhabitants than A. As we can see from the example of even, odd and natural numbers, the relative generality of constructor subtyping relies on the possibility for constructors to be overloaded and, to a lesser extent, on the possibility for datatypes to be defined in terms of previously introduced datatypes.

Constructor subtyping combines subtyping between datatypes and the overloading of constructors. However the integration of these two features in a too liberal way threatens the maintenance of essential properties, as the confluence of the reduction calculus and subject reduction. Let us illustrate these problems with two small examples.

7.1.1 Problematic Examples

Consider the example of three hypothetic datatypes X, Y, and Z, with the following declarations:

Types:	$\mathsf{X}, \mathsf{Y}, \mathsf{Z}$
Subtype relation:	$\mathsf{X} \leq \mathsf{Y}$, $\mathsf{X} \leq \mathsf{Z}$
Declarations:	$\mathsf{c}_1 : \mathsf{X}$
	$\mathsf{c}_2 : \mathsf{Y}$
	$\mathsf{c}_3 : \mathsf{Y} \rightarrow \mathsf{Z}$
	$\mathsf{c}_3 : \mathsf{Z} \rightarrow \mathsf{Z}$

We have here a problem of non-determinism caused by the overloading of constructors. This problem induces a conflict when trying to evaluate case-expressions. Consider the following case-expression over Z:

$$\mathsf{case}\ a\ \mathsf{of}\ \{\mathsf{c}_1 \Rightarrow b_1 \mid \mathsf{c}_3 \Rightarrow b_2 \mid \mathsf{c}_3 \Rightarrow b_3\}$$

The intended meaning of such an expression is that it should evaluate to b_1 if $a = \mathsf{c}_1$, to $b_2\, e$ if $a = \mathsf{c}_3\, e$ and $e : \mathsf{Y}$, or to $b_3\, e$ if $a = \mathsf{c}_3\, e$ and $e : \mathsf{Z}$. So, it is impossible to define a type-independent evaluation rule for case-expressions for arbitrary datatypes. Besides, these computation rules are ambiguous if $a = \mathsf{c}_3\, \mathsf{c}_1$ since $\mathsf{c}_1 : \mathsf{Y}$ and $\mathsf{c}_1 : \mathsf{Z}$, by subsumption. We have

$$\begin{array}{lcl} \mathsf{case}\ (\mathsf{c}_3\, \mathsf{c}_1)\ \mathsf{of}\ \{\mathsf{c}_1 \Rightarrow b_1 \mid \mathsf{c}_3 \Rightarrow b_2 \mid \mathsf{c}_3 \Rightarrow b_3\} & \rightarrow & b_2\, \mathsf{c}_1 \\ \mathsf{case}\ (\mathsf{c}_3\, \mathsf{c}_1)\ \mathsf{of}\ \{\mathsf{c}_1 \Rightarrow b_1 \mid \mathsf{c}_3 \Rightarrow b_2 \mid \mathsf{c}_3 \Rightarrow b_3\} & \rightarrow & b_3\, \mathsf{c}_1 \end{array}$$

and, as b_2 and b_3 are arbitrary, the calculus is obviously non-confluent.

To illustrate the problems with subject reduction, consider the example of two hypothetic datatypes A and B, with the following declarations:

Types:	A, B
Subtype relation:	$\mathsf{A} \leq \mathsf{B}$
Declarations:	$\mathsf{c} : \mathsf{B}$
	$\mathsf{c}' : \mathsf{A} \rightarrow \mathsf{B}$
	$\mathsf{c}' : \mathsf{B} \rightarrow \mathsf{A}$

Further, some type σ is assumed, and two terms $t_1 : \sigma$ and $t_2 : \mathsf{A} \rightarrow \sigma$. We have

$$\mathsf{case}\ (\mathsf{c}'\,\mathsf{c})\ \mathsf{of}\ \{\mathsf{c} \Rightarrow t_1 \mid \mathsf{c}' \Rightarrow t_2\} : \sigma$$

but, this term reduces to $(t_2\,\mathsf{c})$ which is not typable. So, subject reduction fails.

7.1.2 Strict Overloading

The above examples show that overloading must be constrained in some way. The solution advocated in [21, 24] is to require constructors to be declared "essentially" at most once in a given datatype. Here "essentially" means that we allow a constructor c to be multiply defined in a datatype A, but requiring that for every $\mathsf{c} : \sigma \rightarrow \mathsf{A}$, we have $\sigma \leq \tau$ where $\mathsf{c} : \tau \rightarrow \mathsf{A}$ is the principal declaration of c in A. In other words, the only purpose of repeated declarations is to enforce the desired subtyping constraints but (once subtyping is defined) only the principal declaration needs to be used for typing expressions. This notion, which we call *strict overloading*, guarantees coherence between domain and codomain of overloaded constructors.

So, we define a partial order $\sqsubseteq$ over datatypes and require that, if datatypes A and B are related by $\mathsf{A} \sqsubseteq \mathsf{B}$, then every constructor for A is also a constructor for B. Moreover, for each datatype A each constructor c of A is declared only once, and constructors are required to be strictly overloaded. That is, constructor declarations must be monotonic, i.e., if $\mathsf{c} : \sigma \rightarrow \mathsf{A}$ and $\mathsf{c} : \tau \rightarrow \mathsf{B}$ are constructor declarations with $\mathsf{A} \sqsubseteq \mathsf{B}$, then one must have $\sigma \leq \tau$.

This notion of strict overloading is mild enough to be satisfied by most datatypes of interest. For instance, the previous example of odd/even/natural numbers satisfies these conditions. Note that Odd and Even are mutually recursive datatypes. Bellow we consider some examples of datatypes presented in functional language-like syntax.

Example 7.1.1 (Odd, even and natural numbers)

```
data Even = 0 : Even
          | S : Odd -> Even

data Odd = S : Even -> Odd

data Nat extends Even, Odd = S : Nat -> Nat
```

The notation `extends` is used to declare the subtyping axioms `Even` $\sqsubseteq$ `Nat` and `Even` $\sqsubseteq$ `Nat`. Instead of enriching datatype declarations with subtype annotations, we could adopt a slightly more general notation, introducing a separate declaration form for subtype relations between datatypes:

```
sub Even <= Nat
sub Odd <= Nat
```

One may also formalize the parametric datatypes of lists and non-empty lists under the constructor subtyping setting:

Example 7.1.2 (Lists and non-empty lists)

```
data List a = Nil : List a
            | Cons : a -> List a -> List a

data NeList a = Cons : a -> List a -> NeList a

sub NeList <= List
```

Or alternatively,

```
data NeList a = Cons : a -> List a -> NeList a

data List a extends NeList a = Nil : List a
```

More examples are presented in the next section.

7.2 Further Examples

We fill this section with further examples of datatypes and examples illustrating the adequacy of constructor subtyping to the inductive approach to formalization. We begin with the definition of a datatype of ordinals (or better of ordinal notations). Note that this datatype is higher-order, because of constructor `Lim` takes a function as input.

Example 7.2.1 (Ordinals)

```
data Ord extends Nat = S : Ord -> Ord
                     | Lim : (Nat -> Ord) -> Ord
```

Now we present two different ways of formalizing integers with the inductive-based approach.

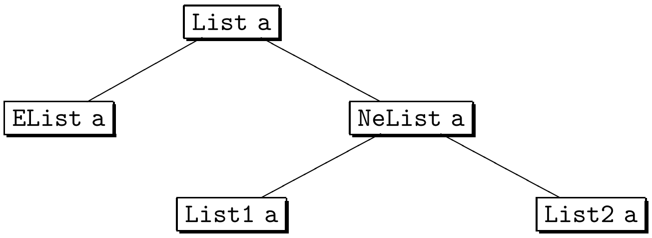
Example 7.2.2 (NatP/Int)

```
data NatP = S : Nat -> NatP

sub NatP <= Nat

data Int extends Nat = Neg : NatP -> Int
```

Example 7.2.3 (Positive/Negative/Integer)

```
data Positive = Zero : Positive
              | Succ : Positive -> Positive

data Negative = Zero : Negative
              | Pred : Negative -> Negative

data Integer extends Positive, Negative
```

Example 7.2.4 (Lists) *Here we return to the list datatype, enriching the hierarchy of list/non-empty list datatypes with types for empty lists, lists of one sole element and lists of two elements.*




```
data EList a = Nil : EList a
sub EList <= List

data List1 a = Cons : a -> EList a -> List1 a
sub List1 <= NeList

data List2 a = Cons : a -> List1 a -> List2 a
sub List2 <= NeList
```

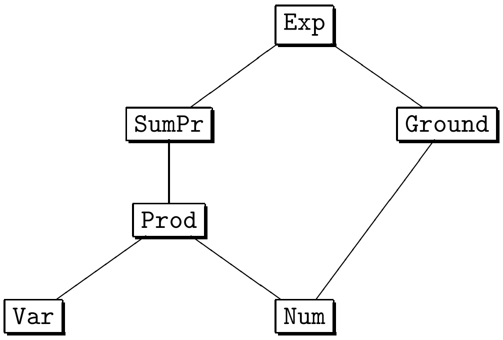
Example 7.2.5 (Arithmetic expressions) *In the following we declare datatypes for arithmetic expressions. We want to distinguish the expressions that are sums of products, i.e., that do not contain additions as subterms of multiplications. Moreover, we want to distinguish ground expressions, i.e., expressions which contain no variables. This example is adapted from [51],*

```
data Ground = Num : Nat -> Ground
            | Plus : Ground -> Ground -> Ground
            | Times : Ground -> Ground -> Ground

data Prod = Num : Nat -> Prod
          | Var : String -> Prod
          | Times : Prod -> Prod -> Prod

data SumPr extends Prod = Plus : SumPr -> SumPr -> SumPr

data Exp extends Ground, Prod = Plus : Exp -> Exp -> Exp
                              | Times : Exp -> Exp -> Exp
```

We can have a more refined hierarchy of types if we take one type for numbers and another for variables. In this case we must declare:

```
data Num = Num : Nat -> Num
data Var = Var : String -> Var
sub Num <= Ground
sub Num <= Prod
sub Var <= Prod
```

Now we have the following hierarchy:



Lastly, we give examples showing how constructor subtyping is fully compatible with the inductive-based approach to formalization. Let us see the benefits of having constructor subtyping with the formalization of the expressions of the call-by-value λ-calculus.

Example 7.2.6 (CBV λ-calculus) *The language of the call-by-value λ-calculus is described by the abstract syntax:*

$$\begin{array}{lll} \text{Expressions} & e & ::= \; x \mid v \mid e_1\, e_2 \\ \text{Values} & v & ::= \; n \mid \lambda v.e \end{array}$$

where v denote a variable and n represent a natural number. We have two sorts of entities: expressions and values. Besides, a value can be seen as an expression. In a strong typing discipline this could be formalized by the following datatypes:

```
data Expression = Var : Variable -> Expression
                | Val : Value -> Expression
                | App : Expression -> Expression -> Expression

data Value = Num : Nat -> Value
           | Abs : Value -> Expression -> Value
```

Note that the constructor `Val` *is a coercion between values and expressions. So, a simple expression like $(\lambda x.x)\, n$ is encoded as* `App (Val (Abs x (Var x))) (Val (Num n))`*. Using constructor subtyping, expressions can be declared as an extension of values:*

```
data Expression extends Value = Var : Variable -> Expression
                              | App : Expression -> Expression -> Expression
```

and $(\lambda x.x)\, n$ is encoded as `App (Abs x (Var x)) (Num n)`*, which is simpler.*

Another interesting example is the description of Harrop formulas. This example is adapted from [117].

Example 7.2.7 (Harrop formulas) *Propositional Harrop formulas, legal programs and goal formulas are described by the following abstract syntax:*

$$\begin{array}{lll} \text{Formulas} & F & ::= \; A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \supset F_2 \\ \text{Programs} & P & ::= \; A \mid P_1 \wedge P_2 \mid G \supset P \\ \text{Goals} & G & ::= \; A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid P \supset G \end{array}$$

where A ranges over atomic formulas. With constructor subtyping, the codification of programs, goals and formulas is straightforward.

```
data Prog a = Emb : a -> Prog a
            | Conj : Prog a -> Prog a -> Prog a
            | Imp : Goal a -> Prog a -> Prog a

data Goal a = Emb : a -> Goal a
            | Conj : Goal a -> Goal a -> Goal a
            | Disj : Goal a -> Goal a -> Goal a
            | Imp : Prog a -> Goal a -> Goal a

data Form a extends Goal a = Conj : Form a -> Form a -> Form a
                           | Disj : Form a -> Form a -> Form a
                           | Imp : Form a -> Form a -> Form a
```

Constructor subtyping allows an incremental type-theoretical formalization of programming languages, where the syntax of a programming language is defined in terms of inductive types. It may be used to specify most of the examples arising in natural semantics [84], and a variety of other languages, such as the BOPL [114] and the CTL* formulae [56], among others.

Example 7.2.8 (Mini-ML [84]) *Here we consider four datatypes identifiers:* `EXP` *of* expressions*,* `IDENT` *for* identifiers*,* `PAT` *of* patterns *and* `NULLPAT` *for the* nullpattern*, all with arity* 0*.*

```
data IDENT = Ident : IDENT
data NULLPAT = Nullpat : NULLPAT
data PAP extends IDENT, NULLPAT = Pairpat : PAT -> PAT -> PAT
data EXP extends IDENT, NULLPAT = Num : EXP
                                | False : EXP
                                | True : EXP
                                | Lamb : PAT -> EXP -> EXP
                                | If : EXP -> EXP -> EXP -> EXP
                                | Mlpair : EXP -> EXP -> EXP
                                | Apply : EXP -> EXP -> EXP
                                | Let : PAT -> EXP -> EXP -> EXP
                                | Letrec : PAT -> EXP -> EXP -> EXP
```

Example 7.2.9 (CAM - Categorical Abstract Machine [84])

```
data VALUE  = Int, Bool : VALUE

data COM extends COMS = Branch : COMS -> COMS -> COM
                      | Cur, Rec : COMS -> COM
                      | Push, Swap, App, Op, Cons, Cdr, Car : COM
                      | Quote : VALUE -> COM

data COMS = Coms : List COM -> COMS

data PROGRAM = Program : COMS -> PROGRAM
```

Example 7.2.10 (BOPL - Basic Object Programming Language [114]) *This example describes the syntax of the* Basic Object Programming Language*. We use as datatype identifiers the non-terminal symbols of the grammar.*

```
data LETTER = a, b, ..., z : LETTER
            | A, B, ..., Z : LETTER

data DIGIT = 0, 1, ..., 9 : DIGIT

data ID exstends LETTER = Idl : ID -> LETTER -> ID
                        | Idd : ID -> DIGIT -> ID

data INT extends DIGIT = Int : DIGIT -> INT -> INT

data BINOP = Plus, Minus, Times, Equal, And, Or, Less : BINOP

data EXP extends INT, ID = Op : EXP -> BINOP -> EXP -> EXP
                         | False, True, NIL, Self : EXP
                         | Not, ClassNew, Par : EXP -> EXP
                         | Assign : ID -> EXP -> EXP
                         | Seq, While : EXP -> EXP -> EXP
                         | If : EXP -> EXP -> EXP -> EXP
                         | New : ID -> EXP
                         | InstOf : EXP -> NeList ID -> EXP
                         | SendMes : EXP -> ID -> List EXP -> EXP

data DEC extends ID = C : DEC -> ID -> DEC

data FORMALS = P : List DEC -> FORMALS

data VAR = Var : DEC -> VAR

data METHOD = Method : ID -> FORMALS -> EXP -> METHOD

data CLASS = Class : ID -> List VAR -> List METHOD -> CLASS
           | ClassIs : ID -> ID -> ID

data PROGRAM = Prog : List CLASS -> EXP -> PROGRAM
```

Example 7.2.11 (CTL* formulas [56]) *In this example, we consider two datatypes identifiers* `SF` *of* state formulas *and* `PF` *of* path formulas, *both with arity* 1.

```
data SF a = I : a -> SF a -> SF a
          | Conj : SF a -> SF a -> SF a
          | Not : SF a -> SF a
          | Forsomefuture : PF a -> SF a
          | Forallfuture : PF a -> SF a

data PF a extends SF a = Conj : PF a -> PF a -> PF a
                       | Not : PF a -> PF a
                       | Nexttime : PF a -> PF a
                       | Until : PF a -> PF a
```

CTL and related temporal logics provide suitable frameworks in which to verify the correctness of programs and protocols, and hence are interesting calculi to formalize in proof assistants.*

7.3 Adding Extensible Overloaded Functions

By coherently combining the subtyping between datatypes and the overloading of constructors, constructor subtyping improves the flexibility and the accuracy of typing systems and seems to be tailored for extensible datatypes. We now consider the problem of defining recursive functions on (extensible) datatypes. The challenge is to define a mechanism that allows recursive definitions to be:

- overloaded, i.e. to have several types. A typical example of an overloaded recursive function is addition, which takes two even numbers and returns an even number, two odd numbers and returns an even number, two natural numbers and returns a natural number, etc;
- extensible, i.e. to extend from a datatype to another datatype with more constructors. Typically, it should be possible to extend a function simply by adding the appropriate computation rules for the new constructors. A typical example of an extensible function is an evaluation function for a given language that needs to be lifted to a richer language.

In addition, we would prefer that, unlike in many object-oriented programming languages, the computational behavior of recursive functions does not depend on typing. One reason is that we are eventually interested in extending the mechanism to dependent types and that letting reduction depend on typing would create a circularity—in dependent type systems, typing depends on reduction through the conversion rule.

Below we briefly outline the issues involved by giving two examples involving overloading and extensibility respectively. We only consider total, unambiguous functions, so recursive functions defined by pattern-matching must be exhaustive and non-overlapping. In other words, exactly one rule of the function should apply for a given pattern. An alternative approach would have been to opt for priority rewriting [12] and drop the requirement that function should be non-overlapping.

7.3.1 An Example of Overloading

In Example 7.1.1 of even/odd/natural numbers, the constructor `S` is overloaded with three incomparable types. Now assume we want to define addition on these datatypes. We would like to define, among others, the addition of two even numbers:

```
add : Even -> Even -> Even
add O y  = y
add (S x) y = S (add x y)
```

However, the second equation does not type-check since `x` is of type `Odd`. For the definition to be valid, one would need to allow `add` to be overloaded:

```
add : Even -> Even -> Even
add : Odd -> Even -> Odd
add O y  = y
add (S x) y = S (add x y)
```

Note that, by overloading addition, we also introduce some additional constraints on the set of rewrite rules. Namely, the rewrite rules should also be well-typed and exhaustive, i.e. yield total functions, for the new type. It is routine to check that the above definition complies with the additional constraints.

Now one would like to overload the function `add` even further, and give it the types

```
add : Even -> Odd -> Odd
add : Odd -> Odd -> Even
add : Nat -> Nat -> Nat
```

Again, this shall be possible, since the recursive equations type-check and are exhaustive for these additional types. Note that for typing reasons, the first and second typings need to be declared together whereas the third typing could be declared on its own.

7.3.2 An Example of Extensibility

We start from the datatype `Nat`, and we assume defined addition `add`, multiplication `mult` and division `div0` (computing $x \div (y+1)$), all of type `Nat -> Nat -> Nat`.

Consider a simple calculator, featuring only addition:

```
data Expr = Num : Nat -> Expr
          | Plus : Expr -> Expr -> Expr
```

extended with a new functionality, namely multiplication:

```
data Expr2 extends Expr = Mult : Expr2 -> Expr2 -> Expr2
                        | Plus : Expr2 -> Expr2 -> Expr2
```

Note how the example involves both subtyping and constructor overloading. Now assume we have defined an interpretation function `interp` as follows:

```
interp : Expr -> Nat
interp (Num n) = n
interp (Plus x y) = add (interp x) (interp y)
```

One would like to extend the definition of `interp` to `Expr2` by declaring

```
interp : Expr2 -> Nat
interp (Mult x y) = mult (interp x) (interp y)
```

In order to determine the validity of the extended definition, we combine all equations of `interp`, and see whether they form a total and unambiguous (exactly only one equation applies to a given pattern) recursive function for the new type.

One can take the example further by adding division as a new functionality. In the case of division by zero, we want to return a value `Undef`. So we first extend the datatype `Nat`. We also extend the function `add` so that it works on the extended datatype.

```
data MaybeNat extends Nat = Undef : MaybeNat

add : MaybeNat -> MaybeNat -> MaybeNat
add Undef x = Undef
add (x:Nat) Undef = Undef
```

Note how, in the above example, we use a type constraint in the second equation. This type constraint acts as a shorthand for the set of rules:

```
add O Undef = Undef
add (S n) Undef = Undef
```

Let us now turn to the new datatype of expressions:

```
data Expr3 extends Expr = Div : Expr3 -> Expr3 -> Expr3
                        | Plus : Expr3 -> Expr3 -> Expr3
```

Again, the example involves both subtyping and constructor overloading. Now, assume division is defined by the function `div` below:

```
div : MaybeNat -> MaybeNat -> MaybeNat
div (x:Nat) (S y) = div0 x y
div (x:Nat) O = Undef
div (x:Nat) Undef = Undef
div Undef y  = Undef
```

One can extend the interpretation function to `Expr3` by declaring:

```
interp : Expr3 -> MaybeNat
interp (Div x y) = div (interp x) (interp y)
```

Note that it is essential that the definition of `add` is extended for this definition of `interp` to be well-typed.

Finally, assume that we want to form a calculator featuring multiplication and division:

```
data Expr4 extends Expr2, Expr3 = Div : Expr4 -> Expr4 -> Expr4
                                | Mult: Expr4 -> Expr4 -> Expr4
                                | Plus: Expr4 -> Expr4 -> Expr4
```

Now we can define `interp` on `Expr4` simply by declaring:

```
interp : Expr4 -> MaybeNat
```

As before, it is essential that the definition of `mult` is extended (to a binary operation on `MaybeNat`) for this definition of `interp` to be well-typed. However, we do not need to introduce new equations as all cases have been previously treated.

7.4 Overview of This Part

This part is devoted to the presentation and study of the concepts of constructor subtyping and extensible overloaded functions. After the informal account in this chapter the formal presentation of type systems featuring constructor subtyping and extensible overloaded functions, and the study of their meta-theoretic properties are the subjects of Chapter 8 and Chapter 9.

In Chapter 8 we introduce and study the properties of a type system with constructor subtyping. We define the system λ_{CS}, a simply typed λ-calculus with mutually recursive parametric datatypes, constructor subtyping, case-expressions and letrec-expressions. We show that λ_{CS} enjoys important meta-theoretic properties, including confluence and subject reduction. As the system features general recursion, the reduction calculus is obviously non-terminating. However, we sketch two ways of achieving strong normalization. One way is constraining the system to guard-by-destructors recursion, following what is done for $\lambda_{\mathcal{G}}$. The other way is enriching the type system with stages (following the ideas presented for $\widehat{\lambda}$) and enforcing termination through typing. The subtyping relation together with the overloading of constructors becomes an issue at type inference for λ_{CS}, but we show that type-checking is decidable.

In Chapter 9 we enrich constructor subtyping with a mechanism for defining extensible overloaded functions. We define the system $\lambda_{\mathsf{CS+fun}}$ a simply typed λ-calculus with mutually recursive parametric datatypes, constructor subtyping and extensible overloaded recursive functions defined by pattern-matching. We formalize the concept of well-formed environment of function declarations. We establish the properties of confluence, subject reduction and decidability of type-checking for this calculus. Moreover, we prove that the requirements imposed for the well-formed environments are decidable properties. With respect to termination, we just provide a simple criterion inspired from [44]. Furthermore, we conjecture how the standard compilation of pattern-matching into case-expressions extends to our setting. We define $\lambda_{\mathsf{CS+def}}$ as a mild variation of $\lambda_{\mathsf{CS+fun}}$: recursive functions defined by pattern-matching are replaced by case-expressions and recursive function definitions. We establish the properties of confluence, subject reduction and decidability of type-checking for $\lambda_{\mathsf{CS+def}}$ and we describe the translation from $\lambda_{\mathsf{CS+fun}}$ to $\lambda_{\mathsf{CS+def}}$.

In Chapter 10 we review related work and conclude.

Chapter 8

The Core Calculus λ_{CS}

In this chapter we introduce λ_{CS}, a simply typed λ-calculus à la Curry with mutually recursive parametric datatypes featuring constructor subtyping, and we establish the fundamental meta-theoretic properties of the calculus.

8.1 The System λ_{CS}

We begin this section by defining the terms of λ_{CS} and providing them with a reduction calculus. Next we introduce types and subtyping. Finally, we present the typing rules for λ_{CS}.

8.1.1 Terms and Reductions

Terms

We assume given a denumerable set $\mathcal{V}_{\mathcal{E}}$ of *(object) variables*, and let $x, x', x_i, y, \ldots$ range over $\mathcal{V}_{\mathcal{E}}$. We assume further that there is a finite set $\mathcal{C}$ of *constants*, which are usually called *constructors*. $\mathcal{V}_{\mathcal{E}}$ and $\mathcal{C}$ are pairwise disjoint and we let $c, c', c_i, \ldots$ range over $\mathcal{C}$. Constructors may only accept a fixed number of arguments, so we stipulate that every constructor c has a fixed *arity* $\mathsf{ar}(c) \in \mathbb{N}$ that indicates the number of arguments taken by c. Terms are built up from standard constructions of λ-calculus: variables, abstractions, applications, constructors, case-expressions and mutually recursive definitions.

Definition 8.1.1 (Terms) *The set $\mathcal{E}_{\mathsf{cs}}$ of* terms *is given by the abstract syntax:*

$$\mathcal{E}_{\mathsf{cs}} \ni a, b \ ::= \ x \mid \lambda x.a \mid a\ b \mid c \mid \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\} \mid \mathsf{letrec}_i(x_1 = a_1, \ldots, x_n = a_n)$$

where in the clause for letrec-expressions, it is assumed that $1 \leq i \leq n$, and in the clause for case-expressions it is assumed that $\mathsf{C}(d) = \{c_1, \ldots, c_n\}$ for some $d \in \mathcal{D}$ and that all c_i's are pairwise distinct.

Notation 8.1.2 *Sometimes we write* $\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ *as an abbreviation of* $\mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_n \Rightarrow b_n\}$*,* $c\,\vec{a}$ *as an abbreviation of* $c\,a_1 \ldots a_{\mathsf{ar}(c)}$*, and* $\mathsf{letrec}_i(\vec{x} = \vec{a})$ *as an abbreviation of* $\mathsf{letrec}_i(x_1 = a_1, \ldots, x_n = a_n)$*.*

Definition 8.1.3 *The set of* free variables *of an expression e, denoted by $\mathsf{FV}(e)$, is defined by induction on the structure of e as follows:*

$$\begin{array}{rcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(c) & = & \emptyset \\ \mathsf{FV}(\lambda x.a) & = & \mathsf{FV}(a)\backslash\{x\} \\ \mathsf{FV}(a\ b) & = & \mathsf{FV}(a) \cup \mathsf{FV}(b) \\ \mathsf{FV}(\mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \dots \ \mid c_n \Rightarrow b_n\}) & = & \mathsf{FV}(a) \cup \mathsf{FV}(b_1) \cup \dots \cup \mathsf{FV}(b_n) \\ \mathsf{FV}(\mathsf{letrec}_i(x_1 = a_1, \dots, x_n = a_n)) & = & (\mathsf{FV}(a_1) \cup \dots \cup \mathsf{FV}(a_n))\backslash\{x_1, \dots, x_n\} \end{array}$$

A variable x is said to occur free *or to* be free in e *if $x \in \mathsf{FV}(e)$. A variable in e that is not free in e is said to* be bound *or to* occur bound in e. *An expression with no free variables is said to be* closed.

The usual conventions of omitting parentheses are adopted: application is left associative and the scope of λ extends to the right as far as possible. We identify terms that are equal up to a renaming of bound variables (or α-conversion). Moreover we assume standard variable convention [14], so, all bound variables are chosen to be different from free variables.

Definition 8.1.4 (Term substitution) *A* term substitution *is a function from $\mathcal{V}_{\mathcal{E}}$ to $\mathcal{E}_{\mathsf{cs}}$. We write $[x_1 := e_1, \dots, x_n := e_n]$ (or briefly $[\vec{x} := \vec{e}]$) for the substitution mapping x_i to e_i for $1 \leq i \leq n$, and mapping every other variable to itself.*

Given a term $a \in \mathcal{E}_{\mathsf{cs}}$ and a term substitution $S = [\vec{x} := \vec{e}]$, we write $S(a)$ or $a[\vec{x} := \vec{e}]$ to denote the term obtained by the simultaneous substitution of terms e_i for the free occurrences of variables x_i in a.

Remark 8.1.5 *In the application of a substitution to a term, we rely on a variable convention. The action of a substitution over a term is defined, as usual, with possible changes of bound variables.*

We now present a result which allows us to reorder substitutions.

Lemma 8.1.6 *If $x_i \neq y_j$ and $x_i \notin \mathsf{FV}(b_j)$ for $i = 1..n$, $j = 1..m$, then*

$$e[x_1 := a_1, \dots, x_n := a_n][y_1 := b_1, \dots, y_m := b_m] = e[\vec{y} := \vec{b}][x_1 := a_1[\vec{y} := \vec{b}], \dots, x_n := a_n[\vec{y} := \vec{b}]]$$

Proof. By induction on the structure of e. □

Reduction calculus

The computational behavior of λ_{CS} is drawn from the notion of β-reduction, ι-reduction and μ-reduction.

Definition 8.1.7 (Reductions)

1. *β-reduction $\rightarrow_\beta$ is defined as the compatible closure of the rule*

$$(\lambda x.a)\ b \ \rightarrow_\beta \ a[x := b]$$

2. *ι-reduction $\rightarrow_\iota$ is defined as the compatible closure of the rule*

$$\mathsf{case}\ (c_i\,\vec{a})\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \dots \ \mid c_n \Rightarrow b_n\} \ \rightarrow_\iota \ b_i\,\vec{a}$$

where $\vec{a}$ represents a vector of terms whose length is exactly $\mathsf{ar}(c_i)$ and $1 \leq i \leq n$.

3. *The μ-reduction, $\rightarrow_\mu$, is defined as the compatible closure of the rule*

$$\mathsf{letrec}_i(\vec{x} = \vec{e})\ (c\,\vec{a})\ \rightarrow_\mu\ e_i[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{e})]\ (c\,\vec{a})$$

where $\vec{a}$ represents a vector of terms whose length is exactly $\mathsf{ar}(c)$.

4. *The terms of the forms $(\lambda x.a)\,b$,* $\mathsf{case}\ (c_i\,\vec{a})\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}$ *and* $\mathsf{letrec}_i(\vec{x} = \vec{e})\ (c\,\vec{a})$ *are called* β-redexes, ι-redexes *and* μ-redexes, *with $a[x := b]$, $b_i\,\vec{a}$ and $e_i[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e}), \ldots, x_n :=$* $\mathsf{letrec}_n(\vec{x} = \vec{e})]\ (c\,\vec{a})$ *being their* contracta, *respectively. As expected we call* $\beta\iota\mu$-redex *to a term that is either a β-redex, an ι-redex or a μ-redex .*

5. *$\rightarrow_{\beta\iota\mu}$ is defined as $\rightarrow_\beta \cup \rightarrow_\iota \cup \rightarrow_\mu$. $\twoheadrightarrow_{\beta\iota\mu}$ and $=_{\beta\iota\mu}$ are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of $\rightarrow_{\beta\iota\mu}$. We say that e* reduces to *e' (or e* computes into *e') whenever $e \twoheadrightarrow_{\beta\iota\mu} e'$. One defines similarly the relations $\twoheadrightarrow_\beta$, $\twoheadrightarrow_\iota$, $\twoheadrightarrow_{\beta\iota}$, $=_\beta$, $=_\iota$ and $=_{\beta\iota}$.*

Remark 8.1.8 *In the formulation of the β- and μ-reduction rules, we rely on a variable convention: in the β-rule, the bound variables of a are assumed to be different from the free variables of b; in the μ-rule, the bound variables of e_i are assumed to be different from the free variables of $\vec{e}$.*

Term substitution has some useful properties with respect to reducibility.

Lemma 8.1.9 (Substitution lemma for reductions)

1. $e \rightarrow_{\beta\iota\mu} e' \quad \Rightarrow \quad e[x := a] \rightarrow_{\beta\iota\mu} e'[x := a]$
2. $a \rightarrow_{\beta\iota\mu} a' \quad \Rightarrow \quad e[x := a] \twoheadrightarrow_{\beta\iota\mu} e[x := a']$
3. $e \rightarrow_{\beta\iota\mu} e' \quad \wedge \quad a \rightarrow_{\beta\iota\mu} a' \quad \Rightarrow \quad e[x := a] \twoheadrightarrow_{\beta\iota\mu} e'[x := a']$

Proof.

1. By induction on the structure of e.
2. By induction on the structure of e.
3. Directly from properties 1 and 2.

$\square$

Definition 8.1.10 (Strongly normalizing terms) *Let $a \in \mathcal{E}_{\mathsf{cs}}$.*

1. *The term a is in* normal form *if it does not contain any $\beta\iota\mu$-redex, i.e., if there is no term b such that $a \rightarrow_{\beta\iota\mu} b$.*
2. *The term a* strongly normalizes *if there is no infinite $\beta\iota\mu$-reduction sequence starting with a.*
3. *The set SN of strongly normalizing terms is inductively defined by the following clause:*

   If $b \in \mathsf{SN}$ for all term b such that $a \rightarrow_{\beta\iota\mu} b$, then $a \in \mathsf{SN}$.

It follows from the above definition that the set SN is not empty, since $\mathcal{V}_{\mathcal{E}} \subseteq \mathsf{SN}$; and that if e is strongly normalizing and $e \twoheadrightarrow_{\beta\iota\mu} e'$, then e' is also strongly normalizing. Moreover, observe that any subterm of a strongly normalizing term is also strongly normalizing, since the $\beta\iota\mu$-reduction relation is compatible with respect to the formation of terms.

8.1.2 Types and Subtyping

In the sequel, we assume given a denumerable set $\mathcal{V}_{\mathcal{T}}$ of *type variables*. Datatypes are named: we assume given a finite set $\mathcal{D}$ of *datatypes*. We adopt the following naming conventions: $\alpha, \alpha', \alpha_i, \beta, \ldots$ range over $\mathcal{V}_{\mathcal{T}}$ and $d, d', d_i, \ldots$ range over $\mathcal{D}$. On datatypes we assume a stratification that ensure that the dependency relation between datatypes is well-founded. Hence each datatype d is assigned a *stratum* $\mathsf{str}(d) \in \mathbb{N}$. Every datatype $d \in \mathcal{D}$ comes equipped with a fixed *arity*, $\mathsf{ar}(d) \in \mathbb{N}$ which indicates the number of parameters it is supposed to have. In addition, we require that every datatype $d \in \mathcal{D}$ comes equipped with a set of constructors denoted by $\mathsf{C}(d)$, and the following condition holds:

$$\mathsf{C}(d) \cap \mathsf{C}(d') \neq \emptyset \quad \Rightarrow \quad \mathsf{ar}(d) = \mathsf{ar}(d') \tag{8.1}$$

We also assume given a binary subtyping relation $\sqsubseteq_{\mathcal{D}}$ over $\mathcal{D}$ that is a partial order that satisfies the following requirement for every $d, d' \in \mathcal{D}$:

$$d \sqsubseteq_{\mathcal{D}} d' \quad \Rightarrow \quad \mathsf{C}(d) \subseteq \mathsf{C}(d') \tag{8.2}$$

Finally, we assume given D a *valid* family of constructors declarations (see Definition 8.1.25).

Definition 8.1.11 (Types) *The set $\mathcal{T}_{\mathsf{cs}}$ of* types *is given by the abstract syntax:*

$$\mathcal{T}_{\mathsf{cs}} \ni \sigma, \tau \ ::= \ \alpha \mid \sigma \rightarrow \tau \mid d\,\vec{\tau}$$

where in the last clause, $\vec{\tau}$ represents a vector of types whose length is exactly $\mathsf{ar}(d)$.

The usual conventions of parenthesis omitting are adopted: the type constructor $\rightarrow$ is right associative.

Definition 8.1.12 (Type substitution) *A* type substitution *is a function from $\mathcal{V}_{\mathcal{T}}$ to $\mathcal{T}_{\mathsf{cs}}$. We write $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$ (or briefly $[\vec{\alpha} := \vec{\sigma}]$) for the substitution mapping α_i to σ_i for $1 \leq i \leq n$, and mapping every other type variable to itself.*

Given a type $\tau \in \mathcal{T}_{\mathsf{cs}}$ and a type substitution $S = [\vec{\alpha} := \vec{\sigma}]$, we write $S(\tau)$ or $\tau[\vec{\alpha} := \vec{\sigma}]$ to denote the type obtained by simultaneously replacing each variable α in τ with $S(\alpha)$.

Notation 8.1.13 *We write $X \ \mathsf{occ}\ \tau$, where X may be a type or a datatype identifier and τ is a type, to state that X occurs in τ; and we write $X \ \mathsf{nocc}\ \tau$ to state that X does not occur in τ. Very often we write $\vec{\tau} \rightarrow \sigma$ as an abbreviation for $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \sigma$.*

In order to fix the intended typings of the constructors, we introduce the notions of constructor scheme, constructor declaration and constructor scheme instantiation.

Definition 8.1.14 (Constructor scheme) *The set $\mathcal{CS}_{\mathsf{cs}}$ of* constructor schemes *is given by the abstract syntax:*

$$\varsigma \ ::= \ \forall\,\vec{\alpha}.\ \sigma$$

where $\vec{\alpha}$ are the free type variables of σ.

Definition 8.1.15 (Positive-Negative)

1. *α* occurs positively *in τ (or τ* is positive *w.r.t. α), written $\alpha \ \mathsf{pos}\ \tau$, is defined by the rules of Figure 8.1.*

2. *α* occurs negatively *in τ (or τ* is negative *w.r.t. α), written $\alpha \ \mathsf{neg}\ \tau$, is defined by the rules of Figure 8.1.*

$$\text{(pos1)} \quad \frac{}{\alpha \ \mathsf{pos} \ \alpha'} \qquad \text{(neg1)} \quad \frac{\alpha \neq \alpha'}{\alpha \ \mathsf{neg} \ \alpha'}$$

$$\text{(pos2)} \quad \frac{\alpha \ \mathsf{pos} \ \sigma \quad \alpha \ \mathsf{neg} \ \tau}{\alpha \ \mathsf{pos} \ (\tau \rightarrow \sigma)} \qquad \text{(neg2)} \quad \frac{\alpha \ \mathsf{neg} \ \sigma \quad \alpha \ \mathsf{pos} \ \tau}{\alpha \ \mathsf{neg} \ (\tau \rightarrow \sigma)}$$

$$\text{(pos3)} \quad \frac{\alpha \ \mathsf{pos} \ \sigma_i \quad (1 \leq i \leq n)}{\alpha \ \mathsf{pos} \ d\,\vec{\sigma}} \qquad \text{(neg3)} \quad \frac{\alpha \ \mathsf{neg} \ \sigma_i \quad (1 \leq i \leq n)}{\alpha \ \mathsf{neg} \ d\,\vec{\sigma}}$$

Figure 8.1: Positive-Negative rules

$$\text{(spos1)} \quad \frac{d \ \mathsf{nocc} \ \tau}{\tau \ \mathsf{spos} \ d}$$

$$\text{(spos2)} \quad \frac{d \ \mathsf{nocc} \ \theta_i \quad (1 \leq i \leq n)}{\theta_1 \rightarrow \ldots \rightarrow \theta_n \rightarrow d\,\vec{\alpha} \ \mathsf{spos} \ d}$$

Figure 8.2: Strictly positive rules

Definition 8.1.16 (Strictly positive) *θ is* strictly positive *w.r.t. d (or d* occurs strict positively *in θ), written θ spos d, is defined by the rules of Figure 8.2.*

Constructors can be overloaded. That is, a constructor c can be constructor of more that one datatype. If $c \in \mathsf{C}(d) \cap \mathsf{C}(d')$ and $d \neq d'$, then the type of c when viewed as a constructor of d is obviously different from the type of c when viewed as a constructor of d'. So, each constructor can be assigned more than one type and the type of a constructor depends on the datatype. A constructor declaration specifies the possible typings of the constructors of a datatype.

Definition 8.1.17 (Constructor declarations) *For every $d \in \mathcal{D}$, there is a map $\mathsf{D}_d : \mathsf{C}(d) \rightarrow \mathcal{CS}_{\mathsf{cs}}$ such that, for every $c \in \mathsf{C}(d)$,*

$$\mathsf{D}_d(c) = \forall\,\vec{\alpha}.\ \vec{\sigma} \rightarrow d\,\vec{\alpha}$$

where:

1. *$\#\vec{\alpha} = \mathsf{ar}(d)$ and $\#\vec{\sigma} = \mathsf{ar}(c)$;*
2. *d' occ $\mathsf{D}_d(c)$ implies $\mathsf{str}(d') \leq \mathsf{str}(d)$;*
3. *each σ_i is strictly positive w.r.t. d', whenever $\mathsf{str}(d') = \mathsf{str}(d)$;*
4. *each σ_i is positive w.r.t. α_j;*
5. *every occurrence of d' in σ_i is of the form $d'\,\vec{\alpha}$, whenever $\mathsf{str}(d') = \mathsf{str}(d)$.*

Moreover, the following condition holds:

$$\mathsf{str}(d) = \mathsf{str}(d') \quad \textit{iff} \quad \exists\, c \in \mathsf{C}(d).\exists\, c' \in \mathsf{C}(d').\ \ d' \ \mathsf{occ} \ \mathsf{D}_d(c) \ \wedge \ d \ \mathsf{occ} \ \mathsf{D}_{d'}(c') \tag{8.3}$$

Conditions 2, 3 and 5 are settle to handle mutual recursive datatypes. Moreover, with condition (8.3), we can identify the mutual recursive datatypes by its stratum.

Example 8.1.18 *Consider the datatypes of even, odd and natural numbers. We have:*

$$\begin{array}{ll}
\mathsf{Even}, \mathsf{Odd}, \mathsf{Nat} \in \mathcal{D} & \mathsf{o}, \mathsf{s} \in \mathcal{C} \\
\mathsf{ar}(\mathsf{Even}) = \mathsf{ar}(\mathsf{Odd}) = \mathsf{ar}(\mathsf{Nat}) = 0 & \mathsf{ar}(\mathsf{o}) = 0 \\
 & \mathsf{ar}(\mathsf{s}) = 1 \\
\mathsf{Odd} \sqsubseteq_{\mathcal{D}} \mathsf{Nat} & \mathsf{C}(\mathsf{Odd}) = \{\mathsf{s}\} \\
\mathsf{Even} \sqsubseteq_{\mathcal{D}} \mathsf{Nat} & \mathsf{C}(\mathsf{Even}) = \mathsf{C}(\mathsf{Nat}) = \{\mathsf{o}, \mathsf{s}\}
\end{array}$$

Note that the condition (8.1) and (8.2) are satisfied. The constructor declarations for these datatypes are:

$$\begin{array}{lcllcl}
\mathsf{D}_{\mathsf{Odd}}(\mathsf{s}) & = & \mathsf{Even} \to \mathsf{Odd} & \mathsf{D}_{\mathsf{Nat}}(\mathsf{o}) & = & \mathsf{Nat} \\
 & & & \mathsf{D}_{\mathsf{Nat}}(\mathsf{s}) & = & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{D}_{\mathsf{Even}}(\mathsf{o}) & = & \mathsf{Even} & & & \\
\mathsf{D}_{\mathsf{Even}}(\mathsf{s}) & = & \mathsf{Odd} \to \mathsf{Even} & & &
\end{array}$$

Example 8.1.19 *Consider the parametric datatypes of lists and of non-empty lists. We have:*

$$\begin{array}{ll}
\mathsf{List}, \mathsf{NeList} \in \mathcal{D} & \mathsf{nil}, \mathsf{cons} \in \mathcal{C} \\
\mathsf{ar}(\mathsf{List}) = \mathsf{ar}(\mathsf{NeList}) = 1 & \mathsf{ar}(\mathsf{nil}) = 0 \\
 & \mathsf{ar}(\mathsf{cons}) = 2 \\
\mathsf{NeList} \sqsubseteq_{\mathcal{D}} \mathsf{List} & \mathsf{C}(\mathsf{List}) = \{\mathsf{nil}, \mathsf{cons}\} \\
 & \mathsf{C}(\mathsf{NeList}) = \{\mathsf{cons}\}
\end{array}$$

The constructor declarations for these datatypes are:

$$\begin{array}{ll}
\mathsf{D}_{\mathsf{List}}(\mathsf{nil}) = \forall\,\alpha.\ \mathsf{List}\,\alpha & \mathsf{D}_{\mathsf{NeList}}(\mathsf{cons}) = \forall\,\alpha.\ \alpha \to \mathsf{List}\,\alpha \to \mathsf{NeList}\,\alpha \\
\mathsf{D}_{\mathsf{List}}(\mathsf{cons}) = \forall\,\alpha.\ \alpha \to \mathsf{List}\,\alpha \to \mathsf{List}\,\alpha &
\end{array}$$

Each particular legal typing for the arguments of a constructor is obtained by instantiating the associated constructor declaration.

Definition 8.1.20 (Instance and domain) *Let $d \in \mathcal{D}$, $c \in \mathsf{C}(d)$ and $\vec{\tau} \in \mathcal{T}_{\mathsf{cs}}$ such that $\#\vec{\tau} = \mathsf{ar}(d)$. Assume $\mathsf{D}_d(c) = \forall\,\vec{\alpha}.\ \vec{\sigma} \to d\,\vec{\alpha}$.*

1. *An* instance of c w.r.t. d and $\vec{\tau}$ *is defined as follows*

$$\mathsf{Inst}_d^{\vec{\tau}}(c) = \vec{\sigma}[\vec{\alpha} := \vec{\tau}] \to d\,\vec{\tau}$$

2. *A* domain of c w.r.t. d and $\vec{\tau}$ *is defined as follows*

$$\mathsf{Dom}_d^{\vec{\tau}}(c) = \vec{\sigma}[\vec{\alpha} := \vec{\tau}]$$

Example 8.1.21 *Instances and domains of some constructors:*

$$\begin{array}{ll}
\mathsf{Inst}_{\mathsf{Even}}(\mathsf{s}) = \mathsf{Odd} \to \mathsf{Even} & \mathsf{Dom}_{\mathsf{Nat}}(\mathsf{o}) = [] \\
\mathsf{Inst}_{\mathsf{List}}^{\mathsf{Nat}}(\mathsf{cons}) = \mathsf{Nat} \to \mathsf{List}\,\mathsf{Nat} \to \mathsf{List}\,\mathsf{Nat} & \mathsf{Dom}_{\mathsf{NeList}}^{\mathsf{Odd}}(\mathsf{cons}) = [\mathsf{Odd}, \mathsf{List}\,\mathsf{Odd}] \\
\mathsf{Inst}_{\mathsf{List}}^{\mathsf{Even}}(\mathsf{nil}) = \mathsf{List}\,\mathsf{Even} & \mathsf{Dom}_{\mathsf{List}}^{\mathsf{Odd}}(\mathsf{nil}) = []
\end{array}$$

$$\text{(refl)} \quad \frac{}{\sigma \leq \sigma} \qquad\qquad \text{(trans)} \quad \frac{\sigma \leq \sigma' \quad \sigma' \leq \sigma''}{\sigma \leq \sigma''}$$

$$\text{(func)} \quad \frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'} \qquad\qquad \text{(data)} \quad \frac{d \sqsubseteq_{\mathcal{D}} d' \quad \tau_i \leq \tau'_i \quad (1 \leq i \leq \mathsf{ar}(d))}{d\,\vec{\tau} \leq d'\,\vec{\tau'}}$$

Figure 8.3: Subtyping rules for λ_{CS}

We now turn to subtyping. The subtyping relation over $\mathcal{T}_{\mathsf{cs}}$ is generated structurally from the order on $\mathcal{D}$. Observe that rule (data) require datatypes to be monotonic in their parameters.

Definition 8.1.22 (Subtyping) *The* subtyping relation $\leq$ *over the set* $\mathcal{T}_{\mathsf{cs}}$ *of types is defined inductively by the rules of Figure 8.3. We write* $\vec{\sigma} \leq \vec{\tau}$ *as an abbreviation of* $\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n$.

Example 8.1.23 *We can easily derive the following subtyping statements:*

$$\begin{array}{rcl} \mathsf{List\,Odd} & \leq & \mathsf{List\,Nat} \\ \mathsf{NeList\,(Nat \to Even)} & \leq & \mathsf{List\,(Odd \to Nat)} \\ \mathsf{List\,Nat \to List\,Odd} & \leq & \mathsf{NeList\,Nat \to List\,Odd} \end{array}$$

Constructors may be overloaded. This feature is crucial to the applicability of constructor subtyping, as most examples require constructors to be overloaded. As shown in Chapter 7 constructor overloading leads to difficulties with subject reduction, so it must be constrained in some way. The salient feature of constructor subtyping is to impose suitable coherence conditions on constructor overloading: roughly speaking, constructor declarations are supposed to be monotonic, i.e., whenever c is a constructor for d and d' with $d \sqsubseteq_{\mathcal{D}} d'$, the domain of c w.r.t. d must be a subtype of the domain of c w.r.t. d'.

Definition 8.1.24 (Strict overloading) *A constructor* $c \in \mathcal{C}$ *is* strictly overloaded *if for every* $d, d' \in \mathcal{D}$ *such that* $c \in \mathsf{C}(d) \cap \mathsf{C}(d')$*, one has*

$$d \sqsubseteq_{\mathcal{D}} d' \quad \Rightarrow \quad \mathsf{Dom}_d^{\vec{\alpha}}(c) \leq \mathsf{Dom}_{d'}^{\vec{\alpha}}(c) \qquad \textit{with} \quad \#\vec{\alpha} = \mathsf{ar}(d)$$

We can now give the formal definition of valid family of constructor declarations.

Definition 8.1.25 (Valid family of constructor declarations) *The family* $\mathsf{D} = \bigcup_{d \in \mathcal{D}} \mathsf{D}_d$ *of constructor declarations is said to be* valid *if every* $c \in \mathcal{C}$ *is strictly overloaded.*

8.1.3 The Typing System

In order to define the typing relation between terms and types, we need the concepts of context and judgment.

Definition 8.1.26 (Contexts and judgments)

$$\text{(var)} \qquad \frac{}{\Gamma \vdash x : \tau} \qquad \text{if } (x : \tau) \in \Gamma$$

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

$$\text{(app)} \qquad \frac{\Gamma \vdash e : \tau \to \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}$$

$$\text{(cons)} \qquad \frac{}{\Gamma \vdash c : \mathsf{Inst}_d^{\vec{\tau}}(c)} \qquad \text{if } c \in \mathsf{C}(d)$$

$$\text{(case)} \qquad \frac{\Gamma \vdash a : d\,\vec{\tau} \quad \Gamma \vdash b_i : \mathsf{Dom}_d^{\vec{\tau}}(c_i) \to \sigma \quad (1 \le i \le n)}{\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma} \qquad \text{if } \mathsf{C}(d) = \{c_1, \ldots, c_n\}$$

$$\text{(rec)} \qquad \frac{\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash e_i : \tau_i \qquad (1 \le i \le n)}{\Gamma \vdash \mathsf{letrec}_j(f_1 = e_1, \ldots, f_n = e_n) : \tau_j} \qquad \text{if } 1 \le j \le n$$

$$\text{(sub)} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \le \sigma}{\Gamma \vdash e : \sigma}$$

Figure 8.4: Typing rules for λ_{CS}

1. *A* context Γ *is a finite set of assumptions* $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ *such that the* x_i*s are pairwise distinct elements of* $\mathcal{V}_{\mathcal{E}}$ *and* $\tau_i \in \mathcal{T}_{\mathsf{cs}}$. Γ *can be seen as a partial function so, we write* $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ *and* $\Gamma(x_i) = \tau_i$. *Usually we drop the curly brackets and write simply* $x_1 : \tau_1, \ldots, x_n : \tau_n$. *Moreover, whenever it is written* $\Gamma, x : \tau$ *or* Γ, Γ' *it is assumed that* $(x : \tau) \notin \Gamma$ *and* $\Gamma \cap \Gamma' = \emptyset$.

2. *A* typing judgment *is a triple of the form* $\Gamma \vdash a : \tau$, *where* Γ *is a context,* $a \in \mathcal{E}_{\mathsf{cs}}$ *and* $\tau \in \mathcal{T}_{\mathsf{cs}}$.

Definition 8.1.27 (Typing)

1. *A typing judgment is* derivable *if it can be inferred from the rules of Figure 8.4.*

2. *A term* $e \in \mathcal{E}_{\mathsf{cs}}$ *is* typable *if* $\Gamma \vdash e : \sigma$ *is derivable for some context* Γ *and type* σ.

The rules (var), (abs) and (app) are standard for the simply typed λ-calculus. The connection between typing and subtyping is done by rule (sub). Note that subtyping behaves very much like the inclusion, when type membership is seen as set membership.

The (cons) rule says the legal typings for a constructor c are obtained by instantiating the declarations of c for each datatype c is associated with. It corresponds to the introduction rules of the datatypes declared.

The (case) and (rec) rules corresponds to the elimination rules for the existing datatypes. The rule (case) is the basic rule of case analysis. The rule (rec) introduces a general recursion scheme, that allows to define mutually recursive functions.

8.2 Confluence

The computation relation of λ_{CS} is confluent. That is, if an expression a can be partially computed into two different expressions a_1 and a_2, then there exists a third expression a' such that both a_1 and a_2 can be computed into a'. Therefore, the reduction strategy used to compute an expression is not relevant, and every normalizing term has an unique normal form. The proof of the confluence property is done by the Tait and Martin-Löf technique. We show only the parts involving letrec-expressions, more details can be found, in Section 4.1 and in [15]. Let us introduce the parallel one-step relation $\twoheadrightarrow_1$.

Definition 8.2.1 *Define a binary relation $\twoheadrightarrow_1$ on $\mathcal{E}_{\mathsf{cs}}$ inductively as follows:*

1. $a \twoheadrightarrow_1 a$
2. $a \twoheadrightarrow_1 a' \quad \Rightarrow \quad \lambda x.a \twoheadrightarrow_1 \lambda x.a'$
3. $a \twoheadrightarrow_1 a' \;\wedge\; b \twoheadrightarrow_1 b' \quad \Rightarrow \quad a\,b \twoheadrightarrow_1 a'\,b'$
4. $a \twoheadrightarrow_1 a' \;\wedge\; b \twoheadrightarrow_1 b' \quad \Rightarrow \quad (\lambda x.a)\,b \twoheadrightarrow_1 a'[x := b']$
5. $a \twoheadrightarrow_1 a' \;\wedge\; b_i \twoheadrightarrow_1 b'_i$ *with* $i = 1..n \quad \Rightarrow \quad \mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 \mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\}$
6. $a_j \twoheadrightarrow_1 a'_j \;\wedge\; b_k \twoheadrightarrow_1 b'_k$ *for some* k, *with* $j = 1..\mathsf{ar}(c_k) \quad \Rightarrow \quad \mathsf{case}\ (c_k\,\vec{a})\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 b'_k\,\vec{a'}$
7. $e_i \twoheadrightarrow_1 e'_i$ *with* $i = 1..n \quad \Rightarrow \quad \mathsf{letrec}_k(\vec{x} = \vec{e}) \twoheadrightarrow_1 \mathsf{letrec}_k(\vec{x} = \vec{e'})$
8. $e_i \twoheadrightarrow_1 e'_i \;\wedge\; a_j \twoheadrightarrow_1 a'_j$ *with* $i = 1..n,\ j = 1..\mathsf{ar}(c) \quad \Rightarrow \quad \mathsf{letrec}_k(\vec{x} = \vec{e})\,(c\,\vec{a}) \twoheadrightarrow_1 e'_k[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e'}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{e'})]\,(c\,\vec{a'})$

Lemma 8.2.2

$$a \twoheadrightarrow_1 a' \;\wedge\; b \twoheadrightarrow_1 b' \quad \Rightarrow \quad a[x := b] \twoheadrightarrow_1 a'[x := b']$$

Proof. By induction on the definition of $a \twoheadrightarrow_1 a'$. We only treat here the cases for clauses 7 and 8.

7. Assume $a \twoheadrightarrow_1 a'$ is $\mathsf{letrec}_k(\vec{x} = \vec{e}) \twoheadrightarrow_1 \mathsf{letrec}_k(\vec{x} = \vec{e'})$ and is a direct consequence of $e_i \twoheadrightarrow_1 e'_i$ with $i = 1..n$. By induction hypothesis $e_i[x := b] \twoheadrightarrow_1 e'_i[x := b']$. But then $\mathsf{letrec}_k(\vec{x} = \vec{e}[x := b]) \twoheadrightarrow_1 \mathsf{letrec}_k(\vec{x} = \overrightarrow{e'[x := b']})$. Hence $a[x := b] = a'[x := b']$.
8. Assume $a \twoheadrightarrow_1 a'$ is $\mathsf{letrec}_k(\vec{x} = \vec{e})\,(c\,\vec{a}) \twoheadrightarrow_1 e'_k[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e'}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{e'})]\,(c\,\vec{a'})$ and is a direct consequence of $e_i \twoheadrightarrow_1 e'_i$ and $a_j \twoheadrightarrow_1 a'_j$, with $i = 1..n$ and $j = 1..\mathsf{ar}(c)$. By induction hypothesis $e_i[x := b] \twoheadrightarrow_1 e'_i[x := b']$ and $a_j[x := b] \twoheadrightarrow_1 a'_j[x := b']$, with $i = 1..n$ and $j = 1..\mathsf{ar}(c)$. But then $a[x := b] = \mathsf{letrec}_k(\vec{x} = \vec{e}[x := b])\,(c\,\overrightarrow{a[x := b]}) \twoheadrightarrow_1 e'_k[x := b'][x_1 := \mathsf{letrec}_1(\vec{x} = \overrightarrow{e'[x := b']}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \overrightarrow{e'[x := b']})]\,(c\,\overrightarrow{a'[x := b']}) = a'[x := b']$.

$\square$

Lemma 8.2.3 (Generation lemma for $\twoheadrightarrow_1$)

1. $\lambda x.a \twoheadrightarrow_1 e$ *implies* $e \equiv \lambda x.a'$ *with* $a \twoheadrightarrow_1 a'$.
2. $a_1\,a_2 \twoheadrightarrow_1 e$ *implies either:*
   (a) $e \equiv a'_1\,a'_2$ *with* $a_1 \twoheadrightarrow_1 a'_1$ *and* $a_2 \twoheadrightarrow_1 a'_2$;

(b) $a_1 \equiv \lambda x.b$, $e \equiv b'[x := a_2']$ *with* $b \twoheadrightarrow_1 b'$ *and* $a_2 \twoheadrightarrow_1 a_2'$;

(c) *or* $a_1 \equiv \mathsf{letrec}_k(\vec{x} = \vec{b})$, $a_2 \equiv (c\,\vec{e})$, $e \equiv b_k'[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{b'}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{b'})]\,(c\,\vec{e'})$ *with* $b_i \twoheadrightarrow_1 b_i'$ *and* $e_j \twoheadrightarrow_1 e_j'$, $i = 1..n$, $j = 1..\mathsf{ar}(c)$.

3. $\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} \twoheadrightarrow_1 e$ *implies either:*

   (a) $e \equiv \mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\}$ *with* $a \twoheadrightarrow_1 a'$ *and* $b_i \twoheadrightarrow_1 b_i'$;

   (b) *or* $a \equiv c_k\,\vec{a}$, $e \equiv b_k'\,\vec{a'}$ *with* $b_k \twoheadrightarrow_1 b_k'$ *and* $a_j \twoheadrightarrow_1 a_j'$, $j = 1..\mathsf{ar}(c_k)$.

4. $\mathsf{letrec}_k(\vec{x} = \vec{b}) \twoheadrightarrow_1 e$ *implies* $e \equiv \mathsf{letrec}_k(\vec{x} = \vec{b'})$ *with* $b_i \twoheadrightarrow_1 b_i'$.

Proof. By induction on the definition of $\twoheadrightarrow_1$. □

Lemma 8.2.4 $\twoheadrightarrow_1$ *satisfies the* diamond property, *i.e.,*

$$a \twoheadrightarrow_1 a_1 \ \wedge\ a \twoheadrightarrow_1 a_2 \quad\Rightarrow\quad \exists\, a_3 \in \mathcal{E}_{\mathsf{cs}}.\ a_1 \twoheadrightarrow_1 a_3 \ \wedge\ a_2 \twoheadrightarrow_1 a_3$$

Proof. By induction on the definition of $a \twoheadrightarrow_1 a_1$. We only treat here the cases for clauses 7 and 8.

7. Assume $a \twoheadrightarrow_1 a_1$ is $\mathsf{letrec}_k(\vec{x} = \vec{e}) \twoheadrightarrow_1 \mathsf{letrec}_k(\vec{x} = \vec{e'})$ and is a direct consequence of $e_i \twoheadrightarrow_1 e_i'$ for $i = 1..n$. By Lemma 8.2.3, $a_2 \equiv \mathsf{letrec}_k(\vec{x} = \vec{e''})$ with $e_i \twoheadrightarrow_1 e_i''$ for $i = 1..n$. By induction hypothesis there are e_i''' such that $e_i' \twoheadrightarrow_1 e_i'''$ and $e_i'' \twoheadrightarrow_1 e_i'''$, for $i = 1..n$. Hence, we can take $a_3 \equiv \mathsf{letrec}_k(\vec{x} = \vec{e'''})$.

8. Assume $a \twoheadrightarrow_1 a_1$ is $\mathsf{letrec}_k(\vec{x} = \vec{e})\,(c\,\vec{b}) \twoheadrightarrow_1 e_k'[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e'}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{e'})]\,(c\,\vec{b'})$ and is a direct consequence of $e_i \twoheadrightarrow_1 e_i'$, $b_j \twoheadrightarrow_1 b_j'$ with $i = 1..n$ and $j = 1..\mathsf{ar}(c)$. By Lemma 8.2.3, one can distinguish two cases:

   1. $a_2 \equiv \mathsf{letrec}_k(\vec{x} = \vec{e''})\,(c\,\vec{b''})$ with $e_i \twoheadrightarrow_1 e_i''$, $b_j \twoheadrightarrow_1 b_j''$ for $i = 1..n$, $j = 1..\mathsf{ar}(c)$. By induction hypothesis, there are e_i''', b_j''' such that $e_i' \twoheadrightarrow_1 e_i'''$, $e_i'' \twoheadrightarrow_1 e_i'''$, $b_j' \twoheadrightarrow_1 b_j'''$, $b_j'' \twoheadrightarrow_1 b_j'''$. Hence we can take $a_3 \equiv e_k'''[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e'''}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{e'''})]\,(c\,\vec{b'''})$.
   2. $a_2 \equiv e_k''[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e''}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{e''})]\,(c\,\vec{b''})$ with $e_i \twoheadrightarrow_1 e_i''$, $b_j \twoheadrightarrow_1 b_j''$ for $i = 1..n$, $j = 1..\mathsf{ar}(c)$. By induction hypothesis, there are e_i''', b_j''' such that $e_i' \twoheadrightarrow_1 e_i'''$, $e_i'' \twoheadrightarrow_1 e_i'''$, $b_j' \twoheadrightarrow_1 b_j'''$, $b_j'' \twoheadrightarrow_1 b_j'''$. Hence we can take $a_3 \equiv e_k'''[x_1 := \mathsf{letrec}_1(\vec{x} = \vec{e'''}), \ldots, x_n := \mathsf{letrec}_n(\vec{x} = \vec{e'''})]\,(c\,\vec{b'''})$.

□

Lemma 8.2.5 $\twoheadrightarrow_{\beta\iota\mu}$ *is the transitive closure of* $\twoheadrightarrow_1$.

Proof. $\twoheadrightarrow_1$ contains the reflexive closure of $\rightarrow_{\beta\iota\mu}$. Moreover, $\twoheadrightarrow_1 \subseteq \twoheadrightarrow_{\beta\iota\mu}$. Since $\twoheadrightarrow_{\beta\iota\mu}$ is the reflexive-transitive closure of $\rightarrow_{\beta\iota\mu}$ it is also the transitive closure of $\twoheadrightarrow_1$. □

Theorem 8.2.6 (Confluence) $\rightarrow_{\beta\iota\mu}$ *is confluent:*

$$a_1 =_{\beta\iota\mu} a_2 \quad\Rightarrow\quad \exists\, e \in \mathcal{E}_{\mathsf{cs}}.\ a_1 \twoheadrightarrow_{\beta\iota\mu} e \ \wedge\ a_2 \twoheadrightarrow_{\beta\iota\mu} e$$

Proof. Assume $a_1 =_{\beta\iota\mu} a_2$, then $\exists\, a \in \mathcal{E}_{\mathsf{cs}}.\ a \twoheadrightarrow_{\beta\iota\mu} a_1 \ \wedge\ a \twoheadrightarrow_{\beta\iota\mu} a_2$. As $\twoheadrightarrow_{\beta\iota\mu}$ is the transitive closure of $\twoheadrightarrow_1$, $\twoheadrightarrow_{\beta\iota\mu}$ satisfies also the diamond property. So, we conclude. □

Corollary 8.2.7 (Uniqueness of normal forms) *Any expression $e \in \mathcal{E}_{\mathsf{cs}}$ has at most one normal form.*

Proof. From Theorem 8.2.6, by absurdity with the assumption that a term could have two different normal forms. □

8.3 Subject Reduction

In this section we show the generation of subtyping and typing and prove that the type of an expression is preserved under computation.

Although we have constructed our subtyping system in an intuitive way (i.e., with subtyping rules reflecting closely the wanted subtyping relation) and so, we have included the transitivity rule, we can eliminate the (trans) rule and obtain an equivalent system. In other words, *transitivity is admissible* or the system has the *transitivity elimination property.* This feature, besides being a key step in the decidability of subtyping, facilitates the study of subject reduction.

Lemma 8.3.1 *Any subtyping derivation containing a sole application of (trans) rule at the last step can be transformed into one ended by the same subtyping assertion free from that rule.*

Proof. By induction on the subtyping derivation. Assume the last step of a derivation is (trans) and the derivation of the premises are transitivity-free. We proceed by case analysis of the last pair of rules used to derive the premises of the last step.

Case (refl, _): A derivation of the form

$$\dfrac{\dfrac{}{\sigma \leq \sigma}\ (\text{refl}) \qquad \sigma \leq \tau}{\sigma \leq \tau}\ (\text{trans})$$

can be transformed into $\sigma \leq \tau$.

Case (_, refl): A derivation of the form

$$\dfrac{\sigma \leq \tau \qquad \dfrac{}{\tau \leq \tau}\ (\text{refl})}{\sigma \leq \tau}\ (\text{trans})$$

can be transformed into $\sigma \leq \tau$.

Case (func, func): A derivation of the form

$$\dfrac{\dfrac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}\ (\text{func}) \qquad \dfrac{\sigma'' \leq \sigma' \quad \tau' \leq \tau''}{\sigma' \to \tau' \leq \sigma'' \to \tau''}\ (\text{func})}{\sigma \to \tau \leq \sigma'' \to \tau''}\ (\text{trans})$$

can be transformed into

$$\dfrac{\dfrac{\sigma'' \leq \sigma' \quad \sigma' \leq \sigma}{\sigma'' \leq \sigma}\,(\text{trans}) \quad \dfrac{\tau \leq \tau' \quad \tau' \leq \tau}{\tau \leq \tau''}\,(\text{trans})}{\sigma \rightarrow \tau \leq \sigma'' \rightarrow \tau''}\,(\text{func})$$

The result follows by induction hypothesis.

Case (data, data): A derivation of the form

$$\dfrac{\dfrac{d \sqsubseteq_{\mathcal{D}} d' \quad \sigma_i \leq \rho_i \quad (1 \leq i \leq \mathsf{ar}(d))}{d\,\vec{\sigma} \leq d'\,\vec{\rho}}\,(\text{data}) \quad \dfrac{d' \sqsubseteq_{\mathcal{D}} d'' \quad \rho_i \leq \tau_i \quad (1 \leq i \leq \mathsf{ar}(d'))}{d'\,\vec{\rho} \leq d''\,\vec{\tau}}\,(\text{data})}{d\,\vec{\sigma} \leq d''\,\vec{\tau}}\,(\text{trans})$$

can be transformed in

$$\dfrac{d \sqsubseteq_{\mathcal{D}} d'' \quad \dfrac{\sigma_i \leq \rho_i \quad \rho_i \leq \tau_i}{\sigma_i \leq \tau_i}\,(\text{trans}) \quad (1 \leq i \leq \mathsf{ar}(d))}{d\,\vec{\sigma} \leq d''\,\vec{\tau}}\,(\text{data})$$

because $\mathsf{ar}(d) = \mathsf{ar}(d') = \mathsf{ar}(d'')$ and $d \sqsubseteq_{\mathcal{D}} d''$ since $\sqsubseteq_{\mathcal{D}}$ is a partial order. Now the result follows by induction hypothesis.

□

Proposition 8.3.2 (Transitivity elimination) *The subtyping system of λ_{CS} has the transitivity elimination property. In other words, any subtyping derivation containing applications of the (trans) rule can be transformed into a transitivity-free derivation ended by the same subtyping judgment.*

Proof. By Lemma 8.3.1 we know that a subtyping derivation containing exactly one application of transitivity at last step can be transformed into a transitivity-free derivation. We may then eliminate transitivities from arbitrary derivations one by one, beginning with uppermost applications of transitivity. □

Lemma 8.3.3 (Generation lemma for subtyping)

1. $\sigma \leq \tau_1 \rightarrow \tau_2 \quad \Rightarrow \quad \sigma \equiv \tau_1' \rightarrow \tau_2' \;\wedge\; \tau_1 \leq \tau_1' \;\wedge\; \tau_2' \leq \tau_2$
2. $\tau_1 \rightarrow \tau_2 \leq \sigma \quad \Rightarrow \quad \sigma \equiv \tau_1' \rightarrow \tau_2' \;\wedge\; \tau_1' \leq \tau_1 \;\wedge\; \tau_2 \leq \tau_2'$
3. $\theta \leq d\,\vec{\tau} \quad \Rightarrow \quad \theta \equiv d'\,\vec{\sigma} \;\wedge\; \vec{\sigma} \leq \vec{\tau} \;\wedge\; d' \sqsubseteq_{\mathcal{D}} d$
4. $d\,\vec{\tau} \leq \theta \quad \Rightarrow \quad \theta \equiv d'\,\vec{\sigma} \;\wedge\; \vec{\tau} \leq \vec{\sigma} \;\wedge\; d \sqsubseteq_{\mathcal{D}} d'$
5. $\alpha \leq \sigma \quad \Rightarrow \quad \sigma \equiv \alpha$
6. $\sigma \leq \alpha \quad \Rightarrow \quad \sigma \equiv \alpha$

Proof. By inspection on the derivation of the antecedent, using Proposition 8.3.2. □

Lemma 8.3.4

1. *If* $\sigma \leq \sigma'$, $\tau_i \leq \tau'_i$ *and* α_i pos σ *for* $i = 1..n$, *then* $\sigma[\vec{\alpha} := \vec{\tau}] \leq \sigma'[\vec{\alpha} := \vec{\tau'}]$.
2. *If* $\sigma \leq \sigma'$, $\tau_i \leq \tau'_i$ *and* α_i neg σ *for* $i = 1..n$, *then* $\sigma[\vec{\alpha} := \vec{\tau'}] \leq \sigma'[\vec{\alpha} := \vec{\tau}]$.

Proof. By simultaneous induction on the structure of σ. □

Lemma 8.3.5

1. *If* $\Gamma \subseteq \Gamma'$ *and* $\Gamma \vdash e : \sigma$ *then* $\Gamma' \vdash e : \sigma$.
2. *If* $\Gamma \vdash e : \sigma$ *then* $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma)$.

Proof.

1. By induction on the derivation of $\Gamma \vdash e : \sigma$.

   (var) If e is a variable and $e : \sigma \in \Gamma$ then also $e : \sigma \in \Gamma'$. Hence, $\Gamma' \vdash e : \sigma$.

   (app) $\Gamma \vdash a\ b : \sigma$ follows directly from $\Gamma \vdash a : \tau \rightarrow \sigma$ and $\Gamma \vdash b : \tau$. By induction hypothesis $\Gamma' \vdash a : \tau \rightarrow \sigma$ and $\Gamma' \vdash b : \tau$. Then, by the (app) rule, $\Gamma' \vdash a\ b : \sigma$.

   (abs) $\Gamma \vdash \lambda x.\, a : \tau \rightarrow \rho$ follows directly from $\Gamma, x{:}\tau \vdash a : \rho$. By the variable convention x does not occur in Γ'. Then, $\Gamma', x{:}\tau$ is also a context which extends $\Gamma, x{:}\tau$. Therefore, by the induction hypothesis we have $\Gamma', x : \tau \vdash a : \rho$ and so, by the (abs) rule, $\Gamma' \vdash \lambda x.\, e : \tau \rightarrow \rho$.

   All the remaining cases can be easily proved using the induction hypothesis.

2. By induction on the derivation of $\Gamma \vdash e : \sigma$.

   (var) If e is a variable and $e : \sigma \in \Gamma$ then $\mathsf{FV}(e) = \{e\} \subseteq \mathsf{dom}(\Gamma)$.

   (app) $\Gamma \vdash a\ b : \sigma$ follows directly from $\Gamma \vdash a : \tau \rightarrow \sigma$ and $\Gamma \vdash b : \tau$. By induction hypothesis $\mathsf{FV}(a) \subseteq \mathsf{dom}(\Gamma)$ and $\mathsf{FV}(b) \subseteq \mathsf{dom}(\Gamma)$. Hence, $\mathsf{FV}(a\ b) = \mathsf{FV}(a) \cup \mathsf{FV}(b) \subseteq \mathsf{dom}(\Gamma)$.

   (abs) $\Gamma \vdash \lambda x.\, a : \tau \rightarrow \rho$ follows directly from $\Gamma, x{:}\tau \vdash a : \rho$. By induction hypothesis, $\mathsf{FV}(a) \subseteq \mathsf{dom}(\Gamma, x : \tau)$. Let $y \in \mathsf{FV}(\lambda x.\, a)$, then $y \in \mathsf{FV}(a)$ and $y \not\equiv x$. Therefore, $y \in \mathsf{dom}(\Gamma)$. Hence $\mathsf{FV}(\lambda x.\, a) \subseteq \mathsf{dom}(\Gamma)$.

   All the remaining cases can be easily proved using the induction hypothesis.

□

The generation for typing describes the information one can infer about a type from a derivable typing judgment.

Lemma 8.3.6 (Generation lemma for typing)

1. $\Gamma \vdash x : \sigma \quad \Rightarrow \quad (x : \tau) \in \Gamma \ \wedge\ \tau \leq \sigma$
2. $\Gamma \vdash a\, b : \sigma \quad \Rightarrow \quad \Gamma \vdash a : \tau \rightarrow \sigma' \ \wedge\ \Gamma \vdash b : \tau \ \wedge\ \sigma' \leq \sigma$
3. $\Gamma \vdash \lambda x.e : \sigma \quad \Rightarrow \quad \sigma \equiv \tau_1 \rightarrow \tau_2 \ \wedge\ \Gamma, x : \tau'_1 \vdash e : \tau'_2 \ \wedge\ \tau_1 \leq \tau'_1 \ \wedge\ \tau'_2 \leq \tau_2$
4. $\Gamma \vdash c : \sigma \quad \Rightarrow \quad \sigma \equiv \vec{\gamma} \rightarrow \theta \ \wedge\ \vec{\gamma} \leq \mathsf{Dom}^{\vec{\tau}}_{d}(c) \ \wedge\ d\, \vec{\tau} \leq \theta \ \wedge\ c \in \mathsf{C}(d)$

5. $\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} : \sigma \quad \Rightarrow \quad \Gamma \vdash a : d\,\vec{\tau} \ \wedge\ \Gamma \vdash b_i : \mathsf{Dom}_d^{\vec{\tau}}(c_i) \to \theta \ \wedge\ \theta \leq \sigma$

6. $\Gamma \vdash \mathsf{letrec}_j(f_1 = e_1, \ldots, f_n = e_n) : \sigma \quad \Rightarrow \quad \Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash e_i : \tau_i$ *for all* $i = 1..n \ \wedge\ \tau_j \leq \sigma$.

Proof. By inspection on the derivation of the antecedent judgments. □

The following lemma states that a type-preserving substitution preserves the derivability of a judgment.

Lemma 8.3.7 (Substitution lemma for typing)
If $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash a : \sigma$ *and* $\Gamma \vdash b_i : \tau_i$ *for* $1 \leq i \leq n$, *then* $\Gamma \vdash a[x_1 := b_1, \ldots, x_n := b_n] : \sigma$.

Proof. By induction on the derivation of $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash a : \sigma$, assuming that $\Gamma \vdash b_i : \tau_i$ is derivable for $1 \leq i \leq n$.

(var) In this case a is a variable: if $a \not\equiv x_j$ for every $j = 1..n$, then $a[\vec{x} := \vec{b}] = a$ and $(a : \sigma) \in \Gamma$, so we conclude by (var); if $a \equiv x_j$ for some $j \in \{1, \ldots, n\}$ then $a[\vec{x} := \vec{b}] = b_j$ and $\tau_j \equiv \sigma$ so the result follows directly from the hypothesis.

(abs) In this case $a \equiv (\lambda y.e)$, $\sigma \equiv \sigma_1 \to \sigma_2$ and $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n, y : \sigma_1 \vdash e : \sigma_2$. By induction hypothesis $\Gamma, y : \sigma_1 \vdash e[x_1 := b_1, \ldots, x_n := b_n] : \sigma_2$. Applying rule (abs) we have $\Gamma \vdash \lambda y.e[\vec{x} := \vec{b}] : \sigma_1 \to \sigma_2$. By Lemma 8.3.5 $\mathsf{FV}(b_i) \subseteq \Gamma$ for every $i = 1..n$, moreover $y \notin \mathsf{dom}(\Gamma)$ so $y \notin \mathsf{FV}(b_i)$, for $i = 1..n$. Therefore $(\lambda y.e)[\vec{x} := \vec{b}] = \lambda y.e[\vec{x} := \vec{b}]$ which concludes the proof of this case.

(rec) In this case $a \equiv \mathsf{letrec}_k(f_1 = e_1, \ldots, f_m = e_m)$ and $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n, f_1 : \sigma_1, \ldots, f_m : \sigma_m \vdash e_j : \sigma_j$, for $1 \leq j \leq m$. By induction hypothesis $\Gamma, f_1 : \sigma_1, \ldots, f_m : \sigma_m \vdash e_j[\vec{x} := \vec{b}] : \sigma_j$. Applying rule (rec) we have $\Gamma \vdash \mathsf{letrec}_k(f_1 = e_1[\vec{x} := \vec{b}], \ldots, f_m = e_m[\vec{x} := \vec{b}]) : \sigma_j$. By Lemma 8.3.5 $\mathsf{FV}(b_i) \subseteq \Gamma$ for $i = 1..n$, moreover $f_1, \ldots, f_m \notin \mathsf{dom}(\Gamma)$ so $f_1, \ldots, f_m \notin \mathsf{FV}(b_i)$ for $i = 1..n$. Hence $\mathsf{letrec}_k(f_1 = e_1[\vec{x} := \vec{b}], \ldots, f_m = e_m[\vec{x} := \vec{b}]) = \mathsf{letrec}_k(f_1 = e_1, \ldots, f_m = e_m)[\vec{x} := \vec{b}]$ which concludes the proof of this case.

The remaining cases are easily proved by routine induction. □

We arrive now to the main result of this section. The following theorem shows that computation preserves the typing relation.

Theorem 8.3.8 (Subject reduction) *Typing is closed under* $\to_{\beta\iota\mu}$:

$$\Gamma \vdash a : \sigma \ \wedge\ a \to_{\beta\iota\mu} a' \ \Rightarrow\ \Gamma \vdash a' : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash a : \sigma$, considering the last rule:

(var) In this case a is a variable so, it cannot be reduced.

(app) In this case $a \equiv e\,e'$, $\Gamma \vdash e : \tau \to \sigma$ and $\Gamma \vdash e' : \tau$. The expression $e\,e'$ can be reduced if:

$e \to_{\beta\iota\mu} e''$. In this case $e\,e' \to_{\beta\iota\mu} e''\,e'$. By the induction hypothesis $\Gamma \vdash e'' : \tau \to \sigma$. The result $\Gamma \vdash e''\,e' : \sigma$ follows using (app).

$e' \to_{\beta\iota\mu} e''$. In this case $e\ e' \to_{\beta\iota\mu} e\ e''$. By the induction hypothesis $\Gamma \vdash e'' : \tau$. The result $\Gamma \vdash e\ e'' : \sigma$ follows using (app).

$e \equiv (\lambda x.b)$. In this case we have $\Gamma \vdash (\lambda x.b) : \tau \to \sigma$ and $(\lambda x.b)\ e' \to_{\beta\iota\mu} b[x := e']$. By Lemma 8.3.6 we have $\Gamma, x : \tau' \vdash b : \sigma'$, $\tau \le \tau'$ and $\sigma' \le \sigma$. From $\Gamma \vdash e' : \tau$, by (sub), one derives $\Gamma \vdash e' : \tau'$. Thus, by Lemma 8.3.7, $\Gamma \vdash b[x := e'] : \sigma'$ and finally, by the rule (sub), $\Gamma \vdash b[x := e'] : \sigma$.

$e \equiv \mathsf{letrec}_j(f_1 = b_1, \ldots, f_n = b_n)$ and $e' \equiv (c\,\vec{a})$. In this case we have $\Gamma \vdash \mathsf{letrec}_j(f_1 = b_1, \ldots, f_n = b_n) : \tau \to \sigma$ and $\mathsf{letrec}_j(f_1 = b_1, \ldots, f_n = b_n)\,(c\,\vec{a}) \to_{\beta\iota\mu} b_j[f_1 := \mathsf{letrec}_1(\vec{f} = \vec{b}), \ldots, f_n := \mathsf{letrec}_1(\vec{f} = \vec{b})]\,(c\,\vec{b})$. By Lemma 8.3.6 we have $\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash b_i : \sigma_i$, $\sigma_i \le \tau_i$ for all $i = 1..n$, and $\sigma_j \le \tau \to \sigma$. One derives, by (sub) $\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash b_j : \tau \to \sigma$ and, by (rec) followed by (sub), $\Gamma \vdash \mathsf{letrec}_i(f_1 = b_1, \ldots, f_n = b_n) : \sigma_i$ for every $i = 1..n$. Thus, using Lemma 8.3.7, $\Gamma \vdash b_j[f_1 := \mathsf{letrec}_1(\vec{f} = \vec{b}), \ldots, f_n := \mathsf{letrec}_n(\vec{f} = \vec{b})] : \tau \to \sigma$. Hence, applying (app) we have $\Gamma \vdash b_j[f_1 := \mathsf{letrec}_1(\vec{f} = \vec{b}), \ldots, f_n := \mathsf{letrec}_n(\vec{f} = \vec{b})]\,(c\,\vec{a}) : \sigma$

(abs) In this case the result follows by induction hypothesis and the rule (abs).

(cons) In this case a is a constructor so, it cannot be reduced.

(case) In this case $a \equiv \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$, $\Gamma \vdash e : d\,\vec{\tau}$ and $\Gamma \vdash b_i : \mathsf{Dom}_d^{\vec{\tau}}(c_i) \to \sigma$ with $1 \le i \le n$ and $\mathsf{C}(d) = \{c_1, \ldots, c_n\}$. The expression $\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ can be reduced if:

$e \to_{\beta\iota\mu} e'$. In this case the result follows by induction hypothesis and the rule (case).

$b_i \to_{\beta\iota\mu} b_i'$. In this case the result follows by induction hypothesis and the rule (case).

$e \equiv c_i\, a_1 \ldots a_{\mathsf{ar}(c_i)}$. In this case $\mathsf{case}\ (c_i\,\vec{a})\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} \to_{\beta\iota\mu} b_i\ \vec{a}$. From $\Gamma \vdash c_i\, a_1 \ldots a_{\mathsf{ar}(c_i)} : d\,\vec{\tau}$, by Lemma 8.3.6, it follows that

$$\Gamma \vdash c_i : \vec{\gamma} \to \theta \quad \wedge \quad \theta \le d\,\vec{\tau}$$

and also, for $1 \le j \le \mathsf{ar}(c_i)$, and for some d' and $\vec{\tau'}$,

$$\Gamma \vdash a_j : \gamma_j \quad \wedge \quad \gamma_j \le (\mathsf{Dom}_{d'}^{\vec{\tau'}}(c_i))[j] \quad \wedge \quad d'\,\vec{\tau'} \le \theta$$

So, $d'\,\vec{\tau'} \le d\,\vec{\tau}$ and therefore, by Lemma 8.3.3, $d' \sqsubseteq_{\mathcal{D}} d$ and $\vec{\tau'} \le \vec{\tau}$. Since c_i is strictly overloaded, using Lemma 4.2.5 and transitivity we have $\gamma_j \le (\mathsf{Dom}_d^{\vec{\tau}}(c_i))[j]$. Now, using the (sub) rule, one can derive $\Gamma \vdash a_j : (\mathsf{Dom}_d^{\vec{\tau}}(c_i))[j]$ for $1 \le j \le \mathsf{ar}(c_i)$, which can be combined with the typing derivation of b_i, by means of the rule (app), to conclude that $\Gamma \vdash b_i\,\vec{a} : \sigma$.

(rec) In this case the result follows by induction hypothesis and the rule (rec).

(sub) In this case the result follows by induction hypothesis and the rule (sub).

$\square$

8.4 Strong Normalization

The system as it is presented in the beginning of this chapter is obviously not strongly normalizing, since there are infinite reduction sequences starting with well-typed terms. For example:

$$\begin{array}{lll} \mathsf{letrec}_1(f = \lambda x.f\,x)\,(\mathsf{s\,o}) & \rightarrow_{\beta\iota\mu} & (\lambda x.f\,x)[f := \mathsf{letrec}_1(f = \lambda x.f\,x)]\,(\mathsf{s\,o}) \\ & \rightarrow_{\beta\iota\mu} & \mathsf{letrec}_1(f = \lambda x.f\,x)\,(\mathsf{s\,o}) \\ & \rightarrow_{\beta\iota\mu} & \cdots \end{array}$$

The failure of strong normalization is a consequence of the (rec) rule being too permissive. To recover strong normalization for typed terms, we must restrict the typing rules for letrec-expressions. Following Part I, strong normalization can be recovered in two different ways:

- restrict the system to guard-by-destructors letrec-expressions, imposing a syntactical condition on the (rec) typing rule, as was done in Chapter 5 for $\lambda_{\mathcal{G}}$;
- enrich the type system with stages and enforce termination through typing, reproducing what was done in Chapter 3 for $\lambda\widehat{\ }$.

In the next subsections we explore these two possibilities.

8.4.1 Guarded-by-Destructors Recursion

To guarantee the strong normalization of all typable terms the idea of the *guarded-by-destructors* mechanism is to complement the (rec) rule with a syntactical condition $\mathcal{G}$ constraining the occurrences of recursive calls in the body of the letrec-expressions. The predicate $\mathcal{G}$ enforces termination by constraining all recursive calls to be applied to terms smaller than the formal argument of the function.

The intuitions and terminology about the $\mathcal{G}$ predicate was already introduced in Chapter 5. However we have to adjust the definitions to the setting of λ_{CS} as this system features mutually recursive datatypes and mutually recursive definitions. Let us illustrate the problems introduced by mutually recursive datatypes with a small example.

Example 8.4.1 *Assume we have* $\mathsf{A}, \mathsf{B} \in \mathcal{D}$ *with* $\mathsf{C}(\mathsf{A}) = \{\mathsf{k}, \mathsf{a}\}$, $\mathsf{C}(\mathsf{B}) = \{\mathsf{b}\}$ *and*

$$\begin{array}{ll} \mathsf{D}_{\mathsf{A}}(\mathsf{k}) = \mathsf{A} & \qquad \mathsf{D}_{\mathsf{B}}(\mathsf{b}) = \mathsf{A} \rightarrow \mathsf{B} \\ \mathsf{D}_{\mathsf{A}}(\mathsf{a}) = \mathsf{B} \rightarrow \mathsf{A} & \end{array}$$

Further assume

$$\begin{array}{llll} \mathrm{fun}_i \equiv & \mathsf{letrec}_i\ (& f_1 = & \lambda x.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{k} \Rightarrow \mathsf{k} \mid \mathsf{a} \Rightarrow \lambda x'.f_2\,(\mathsf{b}\,(\mathsf{a}\,x'))\} \\ & & f_2 = & \lambda y.\,\mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{b} \Rightarrow \lambda y'.f_1\,(\mathsf{a}\,(\mathsf{b}\,y'))\} \\ &) & & \end{array}$$

The term fun_i *is well-typed by the typing rules of Figure 8.4.*

According to the definition of the guard predicate given in Chapter 5, as both f_1 and f_2 do not occur in the body of its definition, apparently fun_i seams to be well-defined. However, it is easy to see that a term like $\mathrm{fun}_2\,(\mathsf{b}\,z)$ has infinite rewritings

$$\begin{array}{lll} \mathrm{fun}_2\,(\mathsf{b}\,z) & \twoheadrightarrow_{\beta\iota\mu} & (\lambda y.\mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{b} \Rightarrow \lambda y'.f_1\,(\mathsf{a}\,(\mathsf{b}\,y'))\})[f_1 := \mathrm{fun}_1, f_2 := \mathrm{fun}_2]\,(\mathsf{b}\,z) \\ & \twoheadrightarrow_{\beta\iota\mu} & \mathrm{fun}_1\,(\mathsf{a}\,(\mathsf{b}\,z)) \\ & \twoheadrightarrow_{\beta\iota\mu} & (\lambda x.\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{k} \Rightarrow \mathsf{k} \mid \mathsf{a} \Rightarrow \lambda x'.f_2\,(\mathsf{b}(\mathsf{a}\,x'))\})[f_1 := \mathrm{fun}_1, f_2 := \mathrm{fun}_2]\,(\mathsf{a}\,(\mathsf{b}\,z)) \\ & \twoheadrightarrow_{\beta\iota\mu} & \mathrm{fun}_2\,(\mathsf{b}\,(\mathsf{a}\,(\mathsf{b}\,z))) \\ & \twoheadrightarrow_{\beta\iota\mu} & \cdots \end{array}$$

Obviously, this happens because of the mutually recursive datatypes and the definitions of Chapter 5 do not apply in these cases.

Let us focus on the mutually recursive definitions and on the syntactical condition $\mathcal{G}$ they must satisfy. The motivation of this syntactic condition is to ensure that each expansion of letrec-expressions consumes (at least) the constructor in the head of its argument. Informally, for a term $\mathsf{letrec}_k(f_1 = e_1, \ldots, f_n = e_n)$ one should have that each f_i may occur only as the head of an application and, in this case, f_i must be applied to a recursive component of the formal argument. Note that these constraints to the occurrence of each f_i concern every e_j, for $j = 1..n$, and not only e_i. As we have mutual recursive datatypes the notion of recursive component has to be reviewed.

Definition 8.4.2 *Let c be a constructor such that $\mathsf{D}_d(c) = \forall\,\vec{\alpha}.\vec{\sigma} \to d\,\vec{\alpha}$. We say that the number j corresponds to a* recursive position *of $\mathsf{D}_d(c)$, written $\mathsf{RP}(j, \mathsf{D}_d(c))$, if σ_j is of the form $\vec{\gamma} \to d'\vec{\alpha}$ where $\mathsf{str}(d') = \mathsf{str}(d)$.*

Therefore, z is a recursive component of x if z is a component of x whose type has an occurrence of some of the types mutually dependent to the type of x.

The predicate $\mathcal{G}$ is now defined as follows.

Definition 8.4.3 ($\mathcal{G}$ predicate) *Let $U, F \subseteq \mathcal{V}_{\mathcal{E}}$ with $U \cap F = \emptyset$, let x be variable not in $U \cup F$ and let $a \in \mathcal{E}_{\mathsf{cs}}$. The predicate $\mathcal{G}^x_F(U, a)$ is derivable using the rules in Figure 8.5.*

The typing rule for $\mathsf{letrec}_k(f_1 = e_1, \ldots, f_n = e_n)$ is now restricted with a set of syntactical conditions $\mathcal{G}^{x_i}_F(\emptyset, a_i)$ for $i = 1..n$, with $F \equiv \{f_1, \ldots, f_n\}$ and $e_i \equiv \lambda x_i.a_i$. The system with this guarded-by-destructors recursion is denoted by $\lambda^{\mathcal{G}}_{\mathsf{CS}}$. The new typing rule for letrec-expressions is defined in Figure 8.6.

It is easy to check that the term fun_i of Example 8.4.1 is not typable with this new (rec) rule.

Example 8.4.4 *For fun_i to be typable in $\lambda^{\mathcal{G}}_{\mathsf{CS}}$ one has to be able to derive*

$$G^x_{\{f_1,f_2\}}(\emptyset, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{k} \Rightarrow \mathsf{k} \mid \mathsf{a} \Rightarrow \lambda x'.f_2\,(\mathsf{b}\,(\mathsf{a}\,x'))\})\quad \textit{and}$$
$$G^y_{\{f_1,f_2\}}(\emptyset, \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{b} \Rightarrow \lambda y'.f_1\,(\mathsf{a}\,(\mathsf{b}\,y'))\})$$

and that is impossible. For instance, to derive $G^y_{\{f_1,f_2\}}(\emptyset, \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{b} \Rightarrow \lambda y'.f_1\,(\mathsf{a}\,(\mathsf{b}\,y'))\})$, as one can only use rule 8, one has to derive $G^y_{\{f_1,f_2\}}(\{y'\}, \lambda y'.f_1\,(\mathsf{a}\,(\mathsf{b}\,y')))$, and this can only follow, by rule 2, from $G^y_{\{f_1,f_2\}}(\{y'\}, f_1\,(\mathsf{a}\,(\mathsf{b}\,y')))$ which is impossible to derive because: we cannot use rule 6 since $a \notin \{y'\}$ and if we use rule 5 we have to prove $G^y_{\{f_1,f_2\}}(\{y'\}, f_1)$ which is impossible.

Let us illustrate the guarded-by-destructors mechanism with mutual recursive datatypes with the example of addition for odd/even numbers.

Example 8.4.5 *Assume the following definition of addition for even and odd numbers.*

$$\begin{array}{lll} \mathrm{add}_i \equiv \quad \mathsf{letrec}_i\ (& f_1 = & \lambda x.\lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{s} \Rightarrow \lambda n.\,\mathsf{s}\,(f_2\,n\,y)\} \\ & f_2 = & \lambda x.\lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda n.\,\mathsf{s}\,(f_1\,n\,y)\} \\ ) & & \end{array}$$

The recursive calls are guarded-by-destructors in this definition. Let us show the derivation of

1. $$\frac{y \notin F}{\mathcal{G}_F^x(U, y)} \quad \text{if } y \text{ is a variable}$$

2. $$\frac{\mathcal{G}_F^x(U, a)}{\mathcal{G}_F^x(U, \lambda z.a)}$$

3. $$\frac{\mathcal{G}_F^x(U, e_i) \quad (1 \leq i \leq n)}{\mathcal{G}_F^x(U, \mathsf{letrec}_j(g_1 = e_1, \ldots, g_n = e_n))}$$

4. $$\frac{}{\mathcal{G}_F^x(U, c)}$$

5. $$\frac{\mathcal{G}_F^x(U, a) \quad \mathcal{G}_F^x(U, b)}{\mathcal{G}_F^x(U, a\, b)}$$

6. $$\frac{\mathcal{G}_F^x(U, z\, \vec{a})}{\mathcal{G}_F^x(U, f\, (z\, \vec{a}))} \quad \text{if } f \in F \wedge z \in U$$

7. $$\frac{\mathcal{G}_F^x(U, e) \quad \mathcal{G}_F^x(U, b_i) \quad (1 \leq i \leq n)}{\mathcal{G}_F^x(U, \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \mid c_n \Rightarrow b_n\})} \quad \text{if } \begin{cases} e \not\equiv z\, \vec{a} \\ \quad \vee \\ (e \equiv z\, \vec{a} \ \wedge \ z \notin U \cup \{x\}) \end{cases}$$

8. $$\frac{\mathcal{G}_F^x(U, a_j) \quad (1 \leq j \leq m) \quad \mathcal{G}_F^x(V_i, e_i) \quad (1 \leq i \leq n)}{\mathcal{G}_F^x(U, \mathsf{case}\ (z\, a_1 \ldots a_m)\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \mid c_n \Rightarrow b_n\})}$$

$$\text{if } \begin{cases} z \in U \cup \{x\} \\ b_i \equiv \lambda y_1. \ldots \lambda y_{\mathsf{ar}(c_i)}.\, e_i \\ V_i \equiv U \cup \{\, y_j \mid \mathsf{RP}(j, \mathsf{D}_d(c_i)) \text{ for } 1 \leq j \leq \mathsf{ar}(c_i)\} \end{cases}$$

Figure 8.5: Guarded-by-destructors rules for λ_{CS}

$$\text{(rec)} \quad \frac{\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \ \vdash \ e_i : \tau_i \quad \mathcal{G}_F^{x_i}(\emptyset, a_i) \quad (1 \leq i \leq n)}{\Gamma \ \vdash \ \mathsf{letrec}_j(f_1 = e_1, \ldots, f_n = e_n) : \sigma_j} \quad \text{if } \begin{cases} F \equiv \{f_1, \ldots, f_n\} \\ e_i \equiv \lambda x_i.a_i \end{cases}$$

Figure 8.6: Typing rule for letrec-expressions in $\lambda_{\mathsf{CS}}^{\mathcal{G}}$

$G^x_{\{f_1,f_2\}}(\emptyset, \lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{s} \Rightarrow \lambda n.\,\mathsf{s}\,(f_2\,n\,y)\})$.

$$\dfrac{\dfrac{\dfrac{x \notin \{f_1,f_2\}}{G^x_{\{f_1,f_2\}}(\emptyset, x)}\ 1 \qquad \dfrac{\dfrac{\mathsf{s} \notin \{f_1,f_2\}}{G^x_{\{f_1,f_2\}}(\{n\}, \mathsf{s})}\ 1 \qquad \dfrac{\dfrac{\dfrac{n \notin \{f_1,f_2\}}{G^x_{\{f_1,f_2\}}(\{n\}, n)}\ 1}{G^x_{\{f_1,f_2\}}(\{n\}, f_2\,n)}\ 6 \qquad \dfrac{y \notin \{f_1,f_2\}}{G^x_{\{f_1,f_2\}}(\{n\}, y)}\ 1}{G^x_{\{f_1,f_2\}}(\{n\}, f_2\,n\,y)}\ 5}{G^x_{\{f_1,f_2\}}(\{n\}, \mathsf{s}\,(f_2\,n\,y))}\ 5}{G^x_{\{f_1,f_2\}}(\emptyset, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{s} \Rightarrow \lambda n.\,\mathsf{s}\,(f_2\,n\,y)\})}\ 8}{G^x_{\{f_1,f_2\}}(\emptyset, \lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{s} \Rightarrow \lambda n.\,\mathsf{s}\,(f_2\,n\,y)\})}\ 2$$

The derivation of $G^x_{\{f_1,f_2\}}(\emptyset, \lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda n.\,\mathsf{s}\,(f_1\,n\,y))\}$ *is very similar to this one.*

8.4.2 Type-Based Termination

In a system featuring constructor subtyping and mutual recursive datatypes, strong normalization can be ensured by typing, following what was done in Chapter 3 for $\widehat{\lambda}$. Of course the type system of λ_{CS} introduced in sections 8.1 is not expressive enough. One needs to enrich λ_{CS} with stages and to adapt some definitions and the typing rules to this new feature.

In this subsection we sketch the system $\lambda_{\widehat{\mathsf{CS}}}$, a simply typed λ-calculus featuring constructor subtyping, parameterized mutual inductively defined types finitely iterated and type-based termination of recursive definitions. The terms allowed in $\lambda_{\widehat{\mathsf{CS}}}$ are the same as those allowed in λ_{CS}. At level of types $\lambda_{\widehat{\mathsf{CS}}}$ and λ_{CS} differ in the following aspects:

1. Stages are now present in $\lambda_{\widehat{\mathsf{CS}}}$ and its set of types is exactly the same of $\widehat{\lambda}$ (see Definition 3.3.1).

2. $\lambda_{\widehat{\mathsf{CS}}}$ the subtyping relation on types is generated structurally from the partial order $\sqsubseteq_{\mathcal{D}}$ on datatypes and the comparison relation on stages $\preccurlyeq$ (see Figure 3.2). The subtyping rule for datatypes is now

$$(\text{data}) \qquad \frac{d_1 \sqsubseteq_{\mathcal{D}} d_2 \quad s \preccurlyeq r \quad \sigma_i \leq \tau_i \quad (1 \leq i \leq \mathsf{ar}(d_1))}{d_1^s\,\vec{\sigma} \leq d_2^r\,\vec{\tau}}$$

3. The notion of constructor scheme is the same as the one given for $\widehat{\lambda}$ (see Definition 3.3.4), but constructor declarations for $\lambda_{\widehat{\mathsf{CS}}}$ have to contemplate mutual recursive datatypes (see Definition 8.4.6).

4. The notion of instance and domain of constructor in $\lambda_{\widehat{\mathsf{CS}}}$ have to be adapted to the presence of stages (see Definition 8.4.7).

5. The set of typing rules of $\lambda_{\widehat{\mathsf{CS}}}$ is different (see Definition 8.4.9).

Let us focus on the definitions of constructor declaration and of instance and domain of a constructor in $\lambda_{\widehat{\mathsf{CS}}}$.

Definition 8.4.6 (Constructor declaration) *For every* $d \in \mathcal{D}$ *, there is a map* $\mathsf{D}_d : \mathsf{C}(d) \rightarrow \mathcal{CS}$ *such that, for every* $c \in \mathsf{C}(d)$*,*

$$\mathsf{D}_d(c) = \forall\,\vec{\alpha}.\forall\,\imath.\ \vec{\sigma} \rightarrow \widehat{d^{\imath}}\vec{\alpha}$$

where:

1. $\#\vec{\alpha} = \mathsf{ar}(d)$ *and* $\#\vec{\sigma} = \mathsf{ar}(c)$*;*

2. d_1 occ $\mathsf{D}_d(c)$ *implies* $\mathsf{str}(d_1) \leq \mathsf{str}(d)$*;*

3. *each* σ_i *is strictly positive w.r.t.* d_1*, whenever* $\mathsf{str}(d_1) = \mathsf{str}(d)$*;*

4. *each* σ_i *is positive w.r.t.* α_j*;*

5. *every occurrence of* d_1 *in* σ_i *is of the form* $d_1^{\imath}\vec{\alpha}$*, whenever* $\mathsf{str}(d_1) = \mathsf{str}(d)$*;*

6. *every occurrence of* $\imath$ *in* σ_i *is of the form* $d_1^{\imath}\vec{\alpha}$*, whenever* $\mathsf{str}(d_1) = \mathsf{str}(d)$*.*

Moreover, the following condition holds:

$$\mathsf{str}(d_1) = \mathsf{str}(d_2) \quad \textit{iff} \quad \exists\, c_1 \in \mathsf{C}(d_1).\exists\, c_2 \in \mathsf{C}(d_2).\;\; d_2 \;\mathsf{occ}\; \mathsf{D}_{d_1}(c_1) \;\wedge\; d_1 \;\mathsf{occ}\; \mathsf{D}_{d_2}(c_2)$$

Definition 8.4.7 (Instance and domain) *Let* $d \in \mathcal{D}$*,* $c \in \mathsf{C}(d)$*,* $s \in \mathcal{S}$ *and* $\vec{\tau} \in \mathcal{T}$ *such that* $\#\vec{\tau} = \mathsf{ar}(d)$*. Assume* $\mathsf{D}_d(c) = \forall\,\vec{\alpha}.\forall\,\imath.\; \vec{\sigma} \to \widehat{d^{\imath}}\vec{\alpha}$*. An* instance of c w.r.t. d, s and $\vec{\tau}$ *is defined as follows*

$$\mathsf{Inst}_d^{s,\vec{\tau}}(c) = \vec{\sigma}[\imath := s][\vec{\alpha} := \vec{\tau}] \to d^{\widehat{s}}\vec{\tau}$$

A domain of c w.r.t. d, s and $\vec{\tau}$ *is defined as follows*

$$\mathsf{Dom}_d^{s,\vec{\tau}}(c) = \vec{\sigma}[\imath := s][\vec{\alpha} := \vec{\tau}]$$

Observe that these definitions are almost identical to the ones for λ_{CS}, the only difference is the presence of stages. The notion of strict overloading in $\lambda_{\widehat{\mathsf{CS}}}$ is defined as follows.

Definition 8.4.8 (Strict overloading) *A constructor* $c \in \mathcal{C}$ *is* strictly overloaded *if for every* $d, d' \in \mathcal{D}$ *such that* $c \in \mathsf{C}(d) \cap \mathsf{C}(d')$*, one has*

$$d \sqsubseteq_{\mathcal{D}} d' \quad \Rightarrow \quad \mathsf{Dom}_d^{\imath,\vec{\alpha}}(c) \leq \mathsf{Dom}_{d'}^{\imath,\vec{\alpha}}(c) \qquad \textit{with} \quad \#\vec{\alpha} = \mathsf{ar}(d)$$

As expected we assume we have a valid family of constructor declarations, i.e., where every constructor is strictly overloaded. We now present the typing system.

Definition 8.4.9 (Typing) *A typing judgment* $\Gamma \vdash e : \sigma$ *is* derivable *if it can be inferred from the rules of Figure 8.7 where the positivity condition* $\imath$ pos σ *in the (rec) rule is defined in Figure 3.5.*

8.5 Type Checking

Decidability of type checking is the property to decide whether or not a typing judgment is derivable according to a type system. It is a fundamental property of a type system considering that program correctness in a typed programming language and proof-checking in a proof-development system are often reduced to type checking itself. Type inference is the problem of inferring a most general type (if existing) to a term in a given context. In the presence of subtyping, most general means minimal with respect to the subtyping relation.

$$\text{(var)} \qquad \frac{}{\Gamma \vdash x : \sigma} \qquad \text{if } (x : \sigma) \in \Gamma$$

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

$$\text{(app)} \qquad \frac{\Gamma \vdash e : \tau \to \sigma \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}$$

$$\text{(cons)} \qquad \frac{}{\Gamma \vdash c : \mathsf{Inst}_d^{s,\vec{\tau}}(c)} \qquad \text{if } c \in \mathsf{C}(d)$$

$$\text{(case)} \qquad \frac{\Gamma \vdash e' : d^{\widehat{s}}\vec{\tau} \qquad \Gamma \vdash e_i : \mathsf{Dom}_d^{s,\vec{\tau}}(c_i) \to \theta \quad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{case}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta} \qquad \text{if } \mathsf{C}(d) = \{c_1, \ldots, c_n\}$$

$$\text{(rec)} \qquad \frac{\Gamma, f_1 : d_1^{\imath}\vec{\tau_1} \to \theta_1, \ldots, f_n : d_n^{\imath}\vec{\tau_n} \to \theta_n \vdash e_i : d_j^{\widehat{\imath}}\vec{\tau_j} \to \theta_j[\imath := \widehat{\imath}] \quad \imath\ \mathsf{pos}\ \theta_j \quad (1 \leq j \leq n)}{\Gamma_i \vdash (\mathsf{letrec}_k(f_1 = e_1, \ldots, f_n = e_n)) : d_k^{s}\vec{\tau_k} \to \theta_k[\imath := s]}$$

$$\text{if } \imath \text{ not in } \Gamma, \vec{\tau_1}, \ldots, \vec{\tau_n}, \text{ and } 1 \leq k \leq n$$

$$\text{(sub)} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma \leq \sigma'}{\Gamma \vdash e : \sigma'}$$

Figure 8.7: Typing rules for $\lambda_{\widehat{\mathsf{CS}}}$

8.5.1 Motivation and Difficulties

Type inference and type checking are related problems. In the presence of minimal types, the type checking algorithm can be decomposed into: a type-inference algorithm to compute, if it exists, the minimal type of a term in a given context; and a subtype-checking algorithm to decide whether the minimal type is a subtype of the type given. Both type inference and type checking are complicated problems for λ_{CS} because this system does not satisfy the minimal type property.

The non-existence of minimal types in λ_{CS} is essentially due to two reasons: the overloading of constructors and the subtyping relation. We illustrate the problems caused by the overloading of constructors with the following examples.

Example 8.5.1 *Remember the definition of even, odd and natural numbers given in Example 8.1.18. In λ_{CS} a constructor is a term by itself. How do we give a minimal type to* s *?*

$$\vdash_{\lambda_{\mathsf{CS}}} \mathsf{s} : \ ? \begin{cases} \mathsf{Even} \rightarrow \mathsf{Odd} \\ \mathsf{Odd} \rightarrow \mathsf{Even} \\ \mathsf{Nat} \rightarrow \mathsf{Nat} \end{cases}$$

There are three minimal types for s[1]*, but there is no common lower type for these three types. Forcing constructors to be fully applied does not solve this problem*

$$\vdash_{\lambda_{\mathsf{CS}}} \lambda x.\, \mathsf{s}\, x : \ ?$$

Apparently one should not have this problem if the variables had type annotations, but even in a λ_{CS} *à la* Church, one has to impose some restrictions in order to have minimal types.

Example 8.5.2 *Assume one has another datatype* $\mathsf{A} \in \mathcal{D}$*,* $\mathsf{C}(\mathsf{A}) = \{\mathsf{s}\}$ *and* $\mathsf{D}_{\mathsf{A}}(\mathsf{s}) = \mathsf{Nat} \rightarrow \mathsf{A}$*. But* A *and* Nat *are not related by* $\sqsubseteq_{\mathcal{D}}$*.*

$$\vdash_{\lambda_{\mathsf{CS}}} \lambda x{:}\mathsf{Nat}.\, \mathsf{s}\, x : \ ? \begin{cases} \mathsf{Nat} \rightarrow \mathsf{A} \\ \mathsf{Nat} \rightarrow \mathsf{Nat} \end{cases}$$

There is no common lower type for $\{\mathsf{Nat} \rightarrow \mathsf{A}, \mathsf{Nat} \rightarrow \mathsf{Nat}\}$*.*

A solution to this problem is to require constructors to be *regular*. This notion already appears in [68, 21]. A constructor c is *regular* if for every d such that $c \in \mathsf{C}(d)$

$$\{\ d' \in \mathcal{D} \ \mid\ \mathsf{Dom}_d^{\vec{\alpha}}(c) \leq \mathsf{Dom}_{d'}^{\vec{\alpha}}(c)\ \} \qquad \text{has a minimum.}$$

For instance, when the constructor s is just associated to Even/Odd/Nat datatypes, s is a regular datatype since the sets $\{\mathsf{Nat}\}$, $\{\mathsf{Even}, \mathsf{Nat}\}$ and $\{\mathsf{Odd}, \mathsf{Nat}\}$ all have minimums. However, with the declaration of datatype A in Example 8.5.2 the constructor s is not regular anymore because the set $\{\mathsf{Nat}, \mathsf{A}\}$ has no minimum.

To achieve the goal of having minimal types, we still have a problem and this problem is related to the subtyping relation of λ_{CS}.

Example 8.5.3 *Assume one has* $\mathsf{B}_1, \mathsf{B}_2, \mathsf{Z}, \mathsf{W} \in \mathcal{D}$ *which implement the following enumerated sets:* $\mathsf{B}_1 = \{\mathsf{b}_1\}, \mathsf{B}_2 = \{\mathsf{b}_2\}, \mathsf{Z} = \{\mathsf{z}, \mathsf{b}_1, \mathsf{b}_2\}$ *and* $\mathsf{W} = \{\mathsf{w}, \mathsf{b}_1, \mathsf{b}_2\}$*. The subtyping relation between*

[1]Note that the other possible types for s are $\mathsf{Odd} \rightarrow \mathsf{Nat}$ and $\mathsf{Even} \rightarrow \mathsf{Nat}$, which are bigger.

these datatypes is illustrated in the following diagram:

$$\begin{tikzcd}
\mathsf{Z} \arrow[d, dash] \arrow[dr, dash] & \mathsf{W} \arrow[d, dash] \arrow[dl, dash] \\
\mathsf{B}_1 & \mathsf{B}_2
\end{tikzcd}$$

What is the minimal type for the following case-expression ?

$$a : \mathsf{Nat} \vdash_{\lambda_{\mathsf{CS}}} \mathsf{case}_{\mathsf{Nat}}\ a \ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{b}_1 \mid \mathsf{s} \Rightarrow \lambda x{:}\mathsf{Nat}.\mathsf{b}_2\} :\ ?\left\{\begin{array}{l}\mathsf{Z}\\ \mathsf{W}\end{array}\right. \tag{8.4}$$

Types B_1 *and* B_2 *have two upper bounds, but no least upper bound.*

A possible way to solve this problem is to force $\sqsubseteq_{\mathcal{D}}$ to be a lattice, but this is too limited. For instance, it would exclude the $\mathsf{Odd}/\mathsf{Even}$ datatypes. Another way to fix this problem is to tag case-expressions with their types

$$a : \mathsf{Nat} \vdash_{\lambda_{\mathsf{CS}}} \mathsf{case}^{\mathsf{Z}}_{\mathsf{Nat}}\ a \ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{b}_1 \mid \mathsf{s} \Rightarrow \lambda x{:}\mathsf{Nat}.\mathsf{b}_2\} : \mathsf{Z}$$

Remark 8.5.4 *In* λ_{CS} *à la Church, with constructors fully applied, tagged case-expressions and regular constructors one has minimal types. A formal proof of this statement is in [23]. It is possible to define a function* $\mathsf{Min}_\Gamma(e)$ *which computes, when it exists the minimal type of a term* e *in context* Γ*, i.e.,*

$$\Gamma \vdash e : \tau \ \Rightarrow\ \mathsf{Min}_\Gamma(e) \leq \tau$$

However, in λ_{CS} terms do not have any sort of type annotation and constructors are not required to be regular. Nevertheless, we still want to answer the question:

How can one decide whether a typing judgment $\Gamma \vdash e : \tau$ *is derivable in* λ_{CS} *?*

To overcome the problem of the non-existence of minimal types we describe all possible types of a term by a set of type schemes together with a set of subtyping constraints.

Given a term e, our strategy to identify all possible typings for e relies on the following steps. First of all, in order to disambiguate the overloading, constructors and case-expressions of e are tagged with their datatypes in all possible ways, producing the set $\mathsf{an}(e)$ of annotated terms built from e. The type system based on λ_{CS}, in which constructors and case-expressions are decorated with their datatypes is denoted by $\lambda^{\mathsf{a}}_{\mathsf{CS}}$.

Secondly, for each term $e' \in \mathsf{an}(e)$ one computes the most general typing for e'. Even with the issue of overloading solved by the datatype tags, one cannot represent the set of all possible types of a $\lambda^{\mathsf{a}}_{\mathsf{CS}}$-term by a single type alone, because one cannot guarantee the existence of least upper bounds in the subtyping order. To address this problem we use constrained types. A constrained type is a type scheme together with a set of subtyping constraints. The existence of least upper bounds and greatest lower bounds is no longer required. For example, case-expression (8.4), where the types of b_1 and b_2 have two upper bounds, Z and W, but no least upper bound, can now be assigned the type α together with the set of constraints $\{\mathsf{B}_1 \leq \alpha, \mathsf{B}_2 \leq \alpha\}$, stating that the expression (8.4) has any type which has both B_1 and B_2 as subtypes. The most general typing for e' will be of the form $C' \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e' : \tau'$ where C' is a set of subtyping constraints. We call $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ to the type system based on $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ where the typing judgments are enriched with subtyping constraints. The algorithm we define to infer such a typing judgment (the most general typing) for e is strongly inspired by the algorithm of Mitchell for type inference with subtyping [109, 78].

Constraints sets can be seen as a flexible form of bounded universal quantification. Constraints restrict the set of types over which the type variables in a type scheme range. Constraints can even restrict type variables to range over the empty set of types. This happens when there is no solution to the set of constraints and it should be treated as a type error.

Finally, to check if a judgment $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ is derivable, one has to match the most general typing for e in $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ (excluding the set of constraints) against the given judgment and then test if the resulting set of constraints is satisfiable.

Summarizing, our claim is that $\Gamma \vdash_{\lambda_{\mathsf{CS}}} e : \tau$ is derivable if and only if there is an annotated term $e' \in \mathsf{an}(e)$ for which $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e' : \tau$ is derivable. As the set of annotated terms $\mathsf{an}(e)$ is finite, if type-checking is decidable for $\lambda^{\mathsf{a}}_{\mathsf{CS}}$, we have a decision procedure. Our type checking algorithm for λ_{CS} (and all the subsidiary algorithms involved) is described at great length in the next subsections and has been implemented in Haskell[2]. Before addressing the problem of type-checking for λ_{CS} we give a formal description of the systems $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ and $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ and we present some of its properties.

8.5.2 The System $\lambda^{\mathsf{a}}_{\mathsf{CS}}$

Here we present system $\lambda^{\mathsf{a}}_{\mathsf{CS}}$, a system very similar to λ_{CS}. The main difference is that, in order to disambiguate the overloading, constructors and case-expressions come annotated with a datatype identifier. The set of $\mathcal{T}^{\mathsf{a}}_{\mathsf{cs}}$ of types of $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ is equal to $\mathcal{T}_{\mathsf{cs}}$.

Definition 8.5.5 (Expressions) *The set $\mathcal{E}^{\mathsf{a}}_{\mathsf{cs}}$ of* expressions *of $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ is given by the abstract syntax:*

$$a, b \ ::= \ x \mid \lambda x.a \mid a\ b \mid c_d \mid \mathsf{case}_d\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\} \mid \mathsf{letrec}_j(f_1 = e_1, \ldots, f_n = e_n)$$

where: in the fourth clause it is assumed that $c \in \mathsf{C}(d)$, in the fifth clause it is assumed that $\mathsf{C}(d) = \{c_1, \ldots, c_n\}$, and in the last clause $1 \leq j \leq n$.

The typing system of $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ just differs from the one for λ_{CS} in the typing rules for constructors and case-expressions. Here, we have

$$\text{(cons)} \qquad \frac{}{\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} c_d : \mathsf{Inst}^{\vec{\tau}}_d(c)}$$

$$\text{(case)} \qquad \frac{\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} a : d\,\vec{\tau} \quad \Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} b_i : \mathsf{Dom}^{\vec{\tau}}_d(c_i) \rightarrow \sigma \quad (1 \leq i \leq n)}{\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} \mathsf{case}_d\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\} : \sigma}$$

Definition 8.5.6 (Annotated terms) *The set of* annotated variants *of a term is given by the mapping $\mathsf{an} : \mathcal{E}_{\mathsf{cs}} \rightarrow \mathcal{P}(\mathcal{E}^{\mathsf{a}}_{\mathsf{cs}})$ defined as follows:*

$$\begin{array}{rcl}
\mathsf{an}(x) & = & \{x\} \\
\mathsf{an}(c) & = & \{c_d \mid c \in \mathsf{C}(d)\} \\
\mathsf{an}(a\,b) & = & \{a'\,b' \mid a' \in \mathsf{an}(a) \wedge b' \in \mathsf{an}(b)\} \\
\mathsf{an}(\lambda x.a') & = & \{\lambda x.a' \mid a' \in \mathsf{an}(a)\} \\
\mathsf{an}(\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}) & = & \{\mathsf{case}_d\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\} \mid a' \in \mathsf{an}(a) \wedge b'_i \in \mathsf{an}(b_i) \wedge \mathsf{C}(d) = \vec{c}\} \\
\mathsf{an}(\mathsf{letrec}_j(\vec{f} = \vec{e})) & = & \{\mathsf{letrec}_j(\vec{f} = \vec{e'}) \mid e'_i \in \mathsf{an}(e_i)\}
\end{array}$$

[2] `http://www.di.uminho.pt/~mjf/CS/`

Example 8.5.7 *Recall the definition of* Odd/Even/Nat *and* List/NeList *of examples 8.1.18 and 8.1.19. We have:*

$$\mathsf{an}(\mathsf{s}\,x) = \{\mathsf{s}_{\mathsf{Odd}}\,x, \mathsf{s}_{\mathsf{Even}}\,x, \mathsf{s}_{\mathsf{Nat}}\,x\}$$
$$\mathsf{an}(\mathsf{s}\,\mathsf{o}) = \{\mathsf{s}_{\mathsf{Odd}}\,\mathsf{o}_{\mathsf{Even}}, \mathsf{s}_{\mathsf{Even}}\,\mathsf{o}_{\mathsf{Even}}, \mathsf{s}_{\mathsf{Nat}}\,\mathsf{o}_{\mathsf{Even}}, \mathsf{s}_{\mathsf{Odd}}\,\mathsf{o}_{\mathsf{Nat}}, \mathsf{s}_{\mathsf{Even}}\,\mathsf{o}_{\mathsf{Nat}}, \mathsf{s}_{\mathsf{Nat}}\,\mathsf{o}_{\mathsf{Nat}}\}$$
$$\mathsf{an}(\lambda f.\lambda x.f\,(f\,x)) = \{\lambda f.\lambda x.f\,(f\,x)\}$$
$$\mathsf{an}(\lambda x.\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{cons} \Rightarrow \lambda h.\lambda t.\,\mathsf{s}\,h\}) = \{\lambda x.\mathsf{case}_{\mathsf{NeList}}\ x\ \mathsf{of}\ \{\mathsf{cons} \Rightarrow \lambda h.\lambda t.\,\mathsf{s}_{\mathsf{Nat}}\,h\},$$
$$\lambda x.\mathsf{case}_{\mathsf{NeList}}\ x\ \mathsf{of}\ \{\mathsf{cons} \Rightarrow \lambda h.\lambda t.\,\mathsf{s}_{\mathsf{Odd}}\,h\}, \lambda x.\mathsf{case}_{\mathsf{NeList}}\ x\ \mathsf{of}\ \{\mathsf{cons} \Rightarrow \lambda h.\lambda t.\,\mathsf{s}_{\mathsf{Even}}\,h\}\}$$

Lemma 8.5.8 *For any $e \in \mathcal{E}_{\mathsf{cs}}$, the set $\mathsf{an}(e)$ is finite.*

Proof. It is an immediate consequence of the fact that every term of $\mathcal{E}_{\mathsf{cs}}$ is finite and the set of datatypes $\mathcal{D}$ is also finite. □

Definition 8.5.9 (Erasure) *The erasure function $\mathsf{er} : \mathcal{E}^{\mathsf{a}}_{\mathsf{cs}} \to \mathcal{E}_{\mathsf{cs}}$ is defined inductively as follows:*

$$\begin{array}{rcl}
\mathsf{er}(x) & = & x \\
\mathsf{er}(c_d) & = & c \\
\mathsf{er}(a\,b) & = & \mathsf{er}(a)\,\mathsf{er}(b) \\
\mathsf{er}(\lambda x.a) & = & \lambda x.\mathsf{er}(a) \\
\mathsf{er}(\mathsf{case}_d\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}) & = & \mathsf{case}\ \mathsf{er}(a)\ \mathsf{of}\ \{\vec{c} \Rightarrow \mathsf{er}(\vec{b})\} \\
\mathsf{er}(\mathsf{letrec}_j(\vec{f} = \vec{e})) & = & \mathsf{letrec}_j(\vec{f} = \mathsf{er}(\vec{e}))
\end{array}$$

Lemma 8.5.10 *If $a \in \mathcal{E}_{\mathsf{cs}}$ and $a' \in \mathsf{an}(a)$, then $\mathsf{er}(a') = a$.*

Proof. By induction on the structure of a. □

Lemma 8.5.11 *If $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} a : \tau$ then, $\Gamma \vdash_{\lambda_{\mathsf{CS}}} \mathsf{er}(a) : \tau$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} a : \tau$. □

Lemma 8.5.12 $\Gamma \vdash_{\lambda_{\mathsf{CS}}} e : \tau$ *iff* $\exists\, e' \in \mathsf{an}(e).\ \Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e' : \tau$.

Proof.

⇒) By induction on the derivation of $\Gamma \vdash_{\lambda_{\mathsf{CS}}} e : \tau$.

⇐) Follows from lemmas 8.5.11 and 8.5.10.

□

Although the problem of overloading has been solved in $\lambda^{\mathsf{a}}_{\mathsf{CS}}$, this system does not enjoy the minimal type property and, consequently, not every typable $\lambda^{\mathsf{a}}_{\mathsf{CS}}$-term has a most general typing.

Example 8.5.13 *Under the assumptions of Example 8.5.3 one has two different typings for the same case-expression but no most general typing:*

$$y : \mathsf{Even} \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} \mathsf{case}_{\mathsf{Even}}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{b}_{1\mathsf{B}_1} \mid \mathsf{s} \Rightarrow \lambda x.\mathsf{b}_{2\mathsf{B}_2}\} : \mathsf{Z}$$

$$y : \mathsf{Even} \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} \mathsf{case}_{\mathsf{Even}}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{b}_{1\mathsf{B}_1} \mid \mathsf{s} \Rightarrow \lambda x.\mathsf{b}_{2\mathsf{B}_2}\} : \mathsf{W}$$

We remedy this situation in $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$, adding subtyping constraints to the type judgments. Alternative typings will then be constructed from most general typings using substitution and a proof system for subtypes.

8.5.3 The System $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$

Here we present a type system and a typing algorithm which adapts the subtyping ideas of [109, 78] to the subtyping relation of λ_{CS}. One has a fixed set of type constructors, $\mathcal{D} \cup \{\rightarrow\}$, and a partial order on $\mathcal{D}$. The subtyping relation is generated structurally from the partial order $\sqsubseteq_{\mathcal{D}}$ on datatypes and this partial order is also structural because $d \sqsubseteq_{\mathcal{D}} d'$ implies $\mathsf{ar}(d) = \mathsf{ar}(d')$. So, we work with what is it usually named *atomic subtyping*.

$\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ is an extension of $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ in which the typing judgments are enriched with a set C of subtyping constraints. We restrict C to consist only of atomic subtyping assertions. Regarding the nature of the subtyping relation involved, we divide C in two sets (Δ, Θ): Δ is a finite set of atomic subtype assertions, and Θ is a finite set of datatype inequalities. Let us introduce the system $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ formally.

Types and Subtyping

We assume now given a denumerable set $\mathcal{V}_{\mathcal{D}}$ of *datatype variables.* Let us partition $\mathcal{V}_{\mathcal{D}}$ into disjoint infinite sets $\mathcal{V}^0_{\mathcal{D}}, \mathcal{V}^1_{\mathcal{D}}, \ldots$, each $\mathcal{V}^n_{\mathcal{D}}$, with $n \in \mathbb{N}$, representing the set of datatype variables of arity n. We adopt the naming conventions that $\delta, \delta', \delta_i, \ldots$ range over $\mathcal{V}_{\mathcal{D}}$, and $\mathtt{d}, \mathtt{d}', \mathtt{d_i}, \ldots \in \mathcal{V}_{\mathcal{D}} \cup \mathcal{D}$.

A type expression is either a type variable, a function type expression or a datatype expression. We have $\mathcal{T}^{\mathsf{a}}_{\mathsf{cs}} \subseteq \mathcal{T}^{\mathsf{ac}}_{\mathsf{cs}}$.

Definition 8.5.14 (Types) *The set $\mathcal{T}^{\mathsf{ac}}_{\mathsf{cs}}$ of* types *is given by the abstract syntax:*

$$\sigma, \tau \ ::= \ \alpha \mid \sigma \rightarrow \tau \mid \mathtt{d}\,\vec{\tau}$$

where in the last clause, it is assumed that the length of $\vec{\tau}$ is exactly $\mathsf{ar}(\mathtt{d})$.

Definition 8.5.15 *The sets of* free type variables *and of* free datatype variables *of a type τ, denoted by $\mathsf{FTV}(\tau)$ and $\mathsf{FDV}(\tau)$, respectively, are defined by induction on the structure of τ as follows:*

$$\begin{array}{lcl}
\mathsf{FTV}(\alpha) & = & \{\alpha\} \\
\mathsf{FTV}(\sigma_1 \rightarrow \sigma_2) & = & \mathsf{FTV}(\sigma_1) \cup \mathsf{FTV}(\sigma_2) \\
\mathsf{FTV}(\mathtt{d}\,\vec{\tau}) & = & \bigcup_{i=1..\#\vec{\tau}} \mathsf{FTV}(\tau_i)
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{FDV}(\alpha) & = & \emptyset \\
\mathsf{FDV}(\sigma_1 \rightarrow \sigma_2) & = & \mathsf{FDV}(\sigma_1) \cup \mathsf{FDV}(\sigma_2) \\
\mathsf{FDV}(d\,\vec{\tau}) & = & \bigcup_{i=1..\#\vec{\tau}} \mathsf{FDV}(\tau_i) \\
\mathsf{FDV}(\delta\,\vec{\tau}) & = & \{\delta\} \cup \big(\bigcup_{i=1..\#\vec{\tau}} \mathsf{FDV}(\tau_i)\big)
\end{array}$$

We now turn to subtyping. The subtyping relation over $\mathcal{T}^{\mathsf{ac}}_{\mathsf{cs}}$ is generated structurally from the order on $\mathcal{D}$.

Definition 8.5.16 (Subtyping)

1. *A* subtype assertion *has the form $\sigma \leq \tau$, where σ and τ are types. We say that a subtype assertion $\sigma \leq \tau$ is* atomic *if both of σ and τ are type variables.*

2. *A* datatype inequality *has the form $\mathtt{d} \sqsubseteq \mathtt{d}'$ where $\mathtt{d}, \mathtt{d}' \in \mathcal{V}_{\mathcal{D}} \cup \mathcal{D}$ and $\mathsf{ar}(\mathtt{d}) = \mathsf{ar}(\mathtt{d}')$. Let Θ be a finite set of datatype inequalities; $\mathtt{d} \sqsubseteq \mathtt{d}'$ is* provable *from Θ, written $\Theta \vdash \mathtt{d} \sqsubseteq \mathtt{d}'$, if it can be inferred from the rules of Figure 8.8.*

3. *A* subtype context *is a pair (Δ, Θ), where Δ is a finite set of atomic subtype assertions and Θ is a finite set of datatype inequalities.*

4. *A subtype assertion $\sigma \leq \tau$ is* provable *from a subtype context (Δ, Θ), written $(\Delta, \Theta) \vdash \sigma \leq \tau$, if it can be inferred from the rules of Figure 8.9. We write $(\Delta, \Theta) \vdash (\Delta', \Theta')$ if $(\Delta, \Theta) \vdash \sigma \leq \tau$ for every $\sigma \leq \tau \in \Delta'$ and $\Theta \vdash \mathtt{d} \sqsubseteq \mathtt{d}'$ for every $\mathtt{d} \sqsubseteq \mathtt{d}' \in \Theta'$. Furthermore, we write $\vdash \sigma \leq \tau$ as an abbreviation of $(\emptyset, \emptyset) \vdash \sigma \leq \tau$.*

$$(\text{ax1})\quad \frac{d_1 \sqsubseteq_{\mathcal{D}} d_2}{\Theta \vdash d_1 \sqsubseteq d_2} \qquad (\text{ax2})\quad \frac{\mathtt{d} \sqsubseteq \mathtt{d}' \in \Theta}{\Theta \vdash \mathtt{d} \sqsubseteq \mathtt{d}'}$$

$$(\text{refl})\quad \frac{}{\Theta \vdash \mathtt{d} \sqsubseteq \mathtt{d}} \qquad (\text{trans})\quad \frac{\Theta \vdash \mathtt{d}_1 \sqsubseteq \mathtt{d}_2 \quad \Theta \vdash \mathtt{d}_2 \sqsubseteq \mathtt{d}_3}{\Theta \vdash \mathtt{d}_1 \sqsubseteq \mathtt{d}_3}$$

Figure 8.8: Rules for $\sqsubseteq$

$$(\text{data})\quad \frac{\Theta \vdash \mathtt{d} \sqsubseteq \mathtt{d}' \quad (\Delta, \Theta) \vdash \tau_i \leq \tau_i' \quad (1 \leq i \leq \mathsf{ar}(\mathtt{d}))}{(\Delta, \Theta) \vdash \mathtt{d}\,\vec{\tau} \leq \mathtt{d}'\,\vec{\tau'}}$$

$$(\text{ax})\quad \frac{\sigma \leq \tau \in \Delta}{(\Delta, \Theta) \vdash \sigma \leq \tau} \qquad (\text{trans})\quad \frac{(\Delta, \Theta) \vdash \sigma \leq \sigma' \quad (\Delta, \Theta) \vdash \sigma' \leq \sigma''}{(\Delta, \Theta) \vdash \sigma \leq \sigma''}$$

$$(\text{refl})\quad \frac{}{(\Delta, \Theta) \vdash \sigma \leq \sigma} \qquad (\text{func})\quad \frac{(\Delta, \Theta) \vdash \sigma' \leq \sigma \quad (\Delta, \Theta) \vdash \tau \leq \tau'}{(\Delta, \Theta) \vdash \sigma \to \tau \leq \sigma' \to \tau'}$$

Figure 8.9: Subtyping rules for $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$

Lemma 8.5.17 *If* $(\Delta, \Theta) \vdash (\Delta', \Theta')$ *and* $(\Delta', \Theta') \vdash (\Delta'', \Theta'')$, *then* $(\Delta, \Theta) \vdash (\Delta'', \Theta'')$.

Proof. For each $\mathtt{d} \sqsubseteq \mathtt{d}'$ in Θ'', there exists by the second hypothesis a derivation of $\Theta' \vdash \mathtt{d} \sqsubseteq \mathtt{d}'$. Now, it suffices to replace each use of rule (ax2) $\frac{\mathtt{d}_1 \sqsubseteq \mathtt{d}_2 \in \Theta'}{\Theta' \vdash \mathtt{d}_1 \sqsubseteq \mathtt{d}_2}$ in this derivation by a derivation of $\Theta \vdash \mathtt{d}_1 \sqsubseteq \mathtt{d}_2$, which must exist by the first hypothesis. For the assertions of Δ'' a similar sort of construction may be used. $\square$

Expressions and Typing

We conclude the definition of $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ by defining its expressions and providing them with a typing system. The set $\mathcal{E}^{\mathsf{ac}}_{\mathsf{cs}}$ of expressions of $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ is equal to $\mathcal{E}^{\mathsf{a}}_{\mathsf{cs}}$.

Definition 8.5.18 (Typing)

1. *A* variable context Γ *is a finite set of assumptions* $x_1 : \tau_1, \ldots, x_n : \tau_n$ *such that the* x_i*s are pairwise distinct elements of* $\mathcal{V}_{\mathcal{T}}$ *and* $\tau_i \in \mathcal{T}^{\mathsf{ac}}_{\mathsf{cs}}$. *Define* $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$.
2. *A* typing judgment *is a quadruple of the form* $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \tau$, *where* (Δ, Θ) *is a subtype context,* Γ *is a variable context,* $a \in \mathcal{E}^{\mathsf{ac}}_{\mathsf{cs}}$ *and* $\tau \in \mathcal{T}^{\mathsf{ac}}_{\mathsf{cs}}$.
3. *A typing judgment is* derivable *if it can be inferred from the rules of Figure 8.10.*
4. *A* $a \in \mathcal{E}^{\mathsf{ac}}_{\mathsf{cs}}$ *is* typable *if* $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \sigma$ *for some* Δ, Θ, Γ *and* σ.

(var) $$\frac{}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} x : \tau} \quad \text{if } (x : \tau) \in \Gamma$$

(abs) $$\frac{(\Delta, \Theta) \,|\, \Gamma, x : \tau \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.e : \tau \to \sigma}$$

(app) $$\frac{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau \to \sigma \qquad (\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e' : \tau}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e\, e' : \sigma}$$

(cons) $$\frac{}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c_d : \mathsf{Inst}^{\vec{\tau}}_d(c)}$$

(case) $$\frac{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : d\,\vec{\tau} \qquad (\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i : \mathsf{Dom}^{\vec{\tau}}_d(c) \to \sigma \quad (1 \le i \le n)}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{case}_d\; a \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \;\ldots\; \mid c_n \Rightarrow b_n\} : \sigma}$$

(rec) $$\frac{(\Delta, \Theta) \,|\, \Gamma,\, f_1 : \tau_1, \ldots,\, f_n : \tau_n \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i : \tau_i \qquad (1 \le i \le n)}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{letrec}_j(f_1 = e_1, \ldots, f_n = e_n) : \tau_j}$$

(sub) $$\frac{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau \qquad (\Delta, \Theta) \vdash \tau \le \sigma}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma}$$

Figure 8.10: Typing rules for $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$

Substitutions

The type expressions of $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ involve two sorts of variables: type variables and datatype variables. It is therefore natural that substitutions for types are a combination of two functions.

Definition 8.5.19 (Substitutions)

1. *A* type substitution *is a function from $\mathcal{V}_\mathcal{T}$ to $\mathcal{T}^{\mathsf{ac}}_{\mathsf{cs}}$. We write $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$ for the substitution mapping α_i to σ_i for $1 \leq i \leq n$, and mapping every other type variable to itself. If σ is a type expression and $S_\mathcal{T}$ is a substitution, then $S_\mathcal{T}(\sigma)$ is the type expression obtained by replacing each variable α in σ with $S_\mathcal{T}(\alpha)$.*

2. *A* datatype substitution *is a function from $\mathcal{V}_\mathcal{D}$ to $\mathcal{D} \cup \mathcal{V}_\mathcal{D}$, that respects the arity of the datatype variables*[3]*. We write $[\delta_1 := \mathtt{d}_1, \ldots, \delta_n := \mathtt{d_n}]$ for the datatype substitution mapping δ_i to d_i for $1 \leq i \leq n$, and mapping every other datatype variable to itself. If σ is a type expression and $S_\mathcal{D}$ is a datatype substitution, then $S_\mathcal{D}(\sigma)$ is the type expression obtained by replacing each datatype variable δ in σ with $S_\mathcal{D}(\delta)$.*

3. *A* substitution *is a pair $(S_\mathcal{T}, S_\mathcal{D})$, where $S_\mathcal{T}$ is a type substitution and $S_\mathcal{D}$ is a datatype substitution. If σ is a type expression and S a substitution, then $S(\sigma)$ is the type expression $S_\mathcal{T}(S_\mathcal{D}(\sigma))$.*

4. *The composition $S \circ R$ of substitutions S and R is defined by $(S \circ R)(\sigma) = S(R(\sigma))$.*

5. *Let $S = (S_\mathcal{T}, S_\mathcal{D})$ be a substitution, $\delta \in \mathcal{V}_\mathcal{D}$, Γ a set of variable assumptions, Δ a set of subtype assertions and Θ a set of datatype inequalities. We define:*

$$\begin{aligned}
&S(\delta) = S_\mathcal{D}(\delta)\\
&S(\Gamma) = \{x : S(\sigma) \mid x : \sigma \in \Gamma\}\\
&S(\Delta) = \{S(\tau) \leq S(\sigma) \mid \tau \leq \sigma \in \Delta\}\\
&S(\Theta) = \{S(\mathtt{d}) \sqsubseteq S(\mathtt{d}') \mid \mathtt{d} \sqsubseteq \mathtt{d}' \in \Theta\}
\end{aligned}$$

6. *The* support *of a substitution S, written $\mathsf{Supp}(S)$, is the set of variables not mapped to themselves by S, i.e.,*

$$\mathsf{Supp}(S) = (\{\alpha \in \mathcal{V}_\mathcal{T} \mid S(\alpha) \neq \alpha\}, \{\delta \in \mathcal{V}_\mathcal{D} \mid S(\delta) \neq \delta\})$$

7. *Let V be a set of type variables. A substitution S* preserves *V when, for every $\alpha \in V$ one has $S(\alpha) = \alpha$.*

8. *Let $V_\mathcal{T}$ be a set of type variables and let $V_\mathcal{D}$ be a set of datatype variables. A substitution S* chooses variables freely on *$(V_\mathcal{T}, V_\mathcal{D})$ if*

   (a) for each $\alpha \in V_\mathcal{T}$ no type variable and no datatype variable appears twice in $S(\alpha)$;

   (b) for distinct $\alpha, \alpha' \in V_\mathcal{T}$, no type variable and no datatype variable in $S(\alpha)$ appears in $S(\alpha')$;

   (c) for distinct $\delta, \delta' \in V_\mathcal{D}$, no datatype variable in $S(\delta)$ appears in $S(\delta')$;

   (d) for any $\alpha \in V_\mathcal{T}$ and $\delta \in V_\mathcal{D}$, no datatype variable in $S(\alpha)$ occurs in $S(\delta)$.

9. *A substitution S is* simple *if $\forall \alpha \in \mathsf{Supp}(S).\ S(\alpha) \in \mathcal{V}_\mathcal{T}$.*

[3]If $\mathsf{ds} : \mathcal{V}_\mathcal{D} \to \mathcal{D} \cup \mathcal{V}_\mathcal{D}$ is a datatype substitution then $\mathsf{ar}(\mathsf{ds}(\delta)) = \mathsf{ar}(\delta)$ for every $\delta \in \mathcal{V}_\mathcal{D}$.

10. *Let V be a set of datatype variables. G is a* ground datatype substitution for V *if for every $\delta \in V$, $G(\delta) \in \mathcal{D}$.*

11. *Let S and R be substitutions, $V_{\mathcal{T}}$ a set of type variables, and $V_{\mathcal{D}}$ a set of datatype variables. We write $S =_{(V_{\mathcal{T}}, V_{\mathcal{D}})} R$ if the substitutions S and R agree on all variables from $(V_{\mathcal{T}}, V_{\mathcal{D}})$.*

Lemma 8.5.20 *Assume substitution S chooses variables freely in $(V_{\mathcal{T}}, V_{\mathcal{D}})$ and R chooses variables freely on $(A \cup V_{\mathcal{T}}, B \cup V_{\mathcal{D}})$, where*

$$\begin{aligned} A &= \textstyle\bigcup_{\alpha \in V_{\mathcal{T}}} \mathsf{FTV}(S(\alpha)) \\ B &= \textstyle\bigcup_{\alpha \in V_{\mathcal{T}}} \mathsf{FDV}(S(\alpha)) \cup \bigcup_{\delta \in V_{\mathcal{D}}} \mathsf{FDV}(S(\delta)) \end{aligned}$$

Then $R \circ S$ chooses variables freely on $(V_{\mathcal{T}}, V_{\mathcal{D}})$.

Proof. The proof is straightforward from the definition. □

We now show how to factorize any substitution into the composition of one that chooses variables freely and one that replaces the freely-chosen variables to produce the original substitution.

Lemma 8.5.21 *Let A be a set of type variables, S a substitution (preserving A), $V_{\mathcal{T}}$ a set of type variables and $V_{\mathcal{D}}$ be a set of datatype variables, such that there are infinitely many type variables not in $V_{\mathcal{T}} \cup A$ and there are infinitely many datatype variables of any arity not in $V_{\mathcal{D}}$. There are substitutions S_1 and S_2 such that S_1 and S_2 are computable, S_1 chooses variables freely on $(V_{\mathcal{T}}, V_{\mathcal{D}})$ (and preserves A), substitution S_2 is simple, and $S =_{(V_{\mathcal{T}}, V_{\mathcal{D}})} S_2 \circ S_1$. Furthermore, if $S =_{(V_{\mathcal{T}}, V_{\mathcal{D}})} T_2 \circ T_1$ for some simple substitution T_2, then there exists a simple substitution R with $T_1 =_{(V_{\mathcal{T}}, V_{\mathcal{D}})} R \circ S_1$.*

Proof. The proof is similar to that of [109] and is omitted here. □

Matching, Instances and Most General Typings

An instance of a typing statement $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ may be obtained by applying a substitution S to all of its type expressions, and possibly choosing a "stronger" subtyping hypothesis or type assignment. However, $(S(\Delta), S(\Theta))$ may contain subtype assertions that are not atomic, and consequently it is not a well-formed subtype context. Thus, in our definition of instance we use a "minimal" atomic sets $S \bullet (\Delta, \Theta)$ that implies $(S(\Delta), S(\Theta))$, such that any atomic sets (Δ', Θ') that implies $(S(\Delta), S(\Theta))$ also implies $S \bullet (\Delta, \Theta)$. In order to define this operation $\bullet$, we begin by showing that a subtype context can only imply *matching* subtype assertions $\sigma \le \tau$, where σ and τ have the same "shape".

Definition 8.5.22 (Matching) *We define the* matching *relation on types by*

1. *α matches α', if $\alpha, \alpha' \in \mathcal{V}_{\mathcal{T}}$;*

2. *$\mathsf{d}\,\vec{\tau}$ matches $\mathsf{d}'\vec{\sigma}$, if $\mathsf{ar}(\mathsf{d}) = \mathsf{ar}(\mathsf{d}')$, and τ_i matches σ_i for $1 \le i \le \mathsf{ar}(\mathsf{d})$;*

3. *$\sigma \to \tau$ matches $\sigma' \to \tau'$, if σ matches σ' and τ matches τ'.*

Lemma 8.5.23 *Let (Δ, Θ) be a subtype context. Then*

1. *If $(\Delta, \Theta) \vdash \sigma \le \tau$, then σ matches τ.*

2. $(\Delta, \Theta) \vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2$ *iff* $(\Delta, \Theta) \vdash \tau_1 \leq \sigma_1$ *and* $(\Delta, \Theta) \vdash \sigma_2 \leq \tau_2$.

3. $(\Delta, \Theta) \vdash \mathtt{d}_1\vec{\sigma} \leq \mathtt{d}_2\vec{\tau}$ *iff* $\Theta \vdash \mathtt{d}_1 \sqsubseteq \mathtt{d}_2$ *and* $(\Delta, \Theta) \vdash \sigma_i \leq \tau_i$ *for* $1 \leq i \leq \mathsf{ar}(\mathtt{d}_1)$.

Proof.

1. By induction on the derivation of $(\Delta, \Theta) \vdash \sigma \leq \tau$
2. $\Rightarrow$) By induction on the derivation of $(\Delta, \Theta) \vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2$. $\Leftarrow$) Trivial.
3. $\Rightarrow$) By induction on the derivation of $(\Delta, \Theta) \vdash \mathtt{d}_1\vec{\sigma} \leq \mathtt{d}_2\vec{\tau}$. $\Leftarrow$) Trivial.

$\square$

Definition 8.5.24 *For any matching types σ and τ, we now define the operations* $\mathsf{atomic}(\sigma \leq \tau)$ *and* $\mathsf{dataIneq}(\sigma \leq \tau)$ *inductively as follows:*

$$\begin{array}{rcl}
\mathsf{atomic}(\sigma \leq \tau) & = & \{\sigma \leq \tau\} \quad \textit{if } \sigma \leq \tau \textit{ is an atomic subtype assertion} \\
\mathsf{atomic}(\sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2) & = & \mathsf{atomic}(\tau_1 \leq \sigma_1) \cup \mathsf{atomic}(\sigma_2 \leq \tau_2) \\
\mathsf{atomic}(\mathtt{d}\,\vec{\tau} \leq \mathtt{d}'\vec{\sigma}) & = & \bigcup_{1 \leq i \leq \mathsf{ar}(\mathtt{d})} \mathsf{atomic}(\tau_i \leq \sigma_i) \\
\\
\mathsf{dataIneq}(\sigma \leq \tau) & = & \emptyset \quad \textit{if } \sigma \leq \tau \textit{ is an atomic subtype assertion} \\
\mathsf{dataIneq}(\sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2) & = & \mathsf{dataIneq}(\tau_1 \leq \sigma_1) \cup \mathsf{dataIneq}(\sigma_2 \leq \tau_2) \\
\mathsf{dataIneq}(\mathtt{d}\,\vec{\tau} \leq \mathtt{d}'\vec{\sigma}) & = & \{\mathtt{d} \sqsubseteq \mathtt{d}'\} \cup \bigcup_{1 \leq i \leq \mathsf{ar}(\mathtt{d})} \mathsf{dataIneq}(\tau_i \leq \sigma_i)
\end{array}$$

Lemma 8.5.25 *Let σ and τ be matching types. Then* $\mathsf{atomic}(\sigma \leq \tau)$ *is a set of atomic subtype assertions,* $(\mathsf{atomic}(\sigma \leq \tau), \mathsf{dataIneq}(\sigma \leq \tau)) \vdash \sigma \leq \tau$ *and for all subtype context (Δ, Θ), if $(\Delta, \Theta) \vdash \sigma \leq \tau$ then* $(\Delta, \Theta) \vdash (\mathsf{atomic}(\sigma \leq \tau), \mathsf{dataIneq}(\sigma \leq \tau))$.

Proof. By induction on the structure of $\sigma \leq \tau$. $\square$

Definition 8.5.26 (Matching substitution) *We say that S* is a matching substitution for *a set Δ of possibly non-matching subtyping assertions, if for every $\sigma \leq \tau \in \Delta$, $S(\sigma)$ and $S(\tau)$ match.*

For any S that is a matching substitution for a set Δ of atomic subtyping assertions, we define the action of S on (Δ, Θ) *by*

$$S \bullet (\Delta, \Theta) = \Big(\bigcup_{\sigma \leq \tau \in \Delta} \mathsf{atomic}(S(\sigma) \leq S(\tau)) \ , \ S(\Theta) \cup \bigcup_{\sigma \leq \tau \in \Delta} \mathsf{dataIneq}(S(\sigma) \leq S(\tau) \ \Big)$$

The following lemma shows that this definition of $\bullet$ gives us precisely the desired behavior.

Lemma 8.5.27 *If S is a matching substitution for a set Δ of atomic subtyping assertions, then all subtyping assertions in $S \bullet (\Delta, \Theta)$ are atomic, $S \bullet (\Delta, \Theta) \vdash (S(\Delta), S(\Theta))$, and for all subtype context (Δ', Θ'), $(\Delta', \Theta') \vdash (S(\Delta), S(\Theta))$ implies $(\Delta', \Theta') \vdash S \bullet (\Delta, \Theta)$.*

Proof. Follows from Lemma 8.5.25. $\square$

We now define the instance relation the same way as it was done in [58, 109].

Definition 8.5.28 (Instance) *A typing statement* $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma'$ is an instance of $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ by S *if S is a matching substitution for Δ and*

$$(\Delta', \Theta') \vdash S \bullet (\Delta, \Theta), \quad S(\Gamma) \subseteq \Gamma', \quad \textit{and} \quad \sigma' = S(\sigma)$$

Definition 8.5.29 (Most general typing) *A judgment* $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ *is a* most general typing *for* e *iff its instances are exactly the derivable typings for* e.

We will prove that typing is closed under the instance relation. To this end, we need the following lemma showing that adding subtype assertions or adding typing assumptions preserves the provability of typing judgment.

Lemma 8.5.30 *Suppose that* $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$.

1. *If* $(\Delta', \Theta') \vdash (\Delta, \Theta)$ *and* (Δ', Θ') *is atomic, then* $(\Delta', \Theta') \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$.
2. *If* $x \in \mathsf{FV}(e)$, *then* $x : \tau \in \Gamma$ *for some* τ.
3. *If* $x : \tau \in \Gamma'$ *for every* $x : \tau \in \Gamma$ *with* $x \in \mathsf{FV}(e)$, *then* $(\Delta, \Theta) \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$.

Proof. By induction on the length of the derivation of $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$. □

We also need to show that the rules for deriving subtype assertions are closed under substitution.

Lemma 8.5.31 *If* (Δ, Θ) *is a subtype context such that* $(\Delta, \Theta) \vdash \sigma \leq \tau$, *and* S *is a matching substitution for* Δ, *then* $(S(\Delta), S(\Theta)) \vdash S(\sigma) \leq S(\tau)$.

Proof. By induction on the derivation of $(\Delta, \Theta) \vdash \sigma \leq \tau$. □

We finally prove that every instance of a derivable typing is also derivable. We will make use of this result to show that our type inference algorithm produce most general typings.

Theorem 8.5.32 *Assume that* $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma'$ *is an instance of* $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ *by some substitution* S. *Then*

$$(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma \qquad \Rightarrow \qquad (\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma'$$

Proof. By Lemma 8.5.30, it is sufficient to show that $S \bullet (\Delta, \Theta) \,|\, S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : S(\sigma)$, which is proved by induction on the derivation of $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$.

- Assume the last step is:

$$(\text{var}) \quad \frac{}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} x : \tau} \quad \text{with } (x : \tau) \in \Gamma$$

  One has $(x : S(\tau)) \in S(\Gamma)$, hence $S \bullet (\Delta, \Theta) \,|\, S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} x : S(\tau)$ by (var).

- Assume the last step is:

$$(\text{cons}) \quad \frac{}{(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c_d : \mathsf{Inst}_d^{\vec{\tau}}(c)}$$

  By definition, $\mathsf{Inst}_d^{\vec{\tau}}(c)$ has no datatype variables and the only type variables that may occur in it are the ones occurring in $\vec{\tau}$. Therefore $S(\mathsf{Inst}_d^{\vec{\tau}}(c)) = \mathsf{Inst}_d^{S(\vec{\tau})}(c)$. Hence we conclude by (cons).

- Assume the last step is:

$$\text{(abs)}\quad \frac{(\Delta,\Theta)\,|\,\Gamma, x:\tau \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e:\sigma}{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.e:\tau\to\sigma}$$

By induction hypothesis $S\bullet(\Delta,\Theta)\,|\,S(\Gamma), x:S(\tau) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a:S(\sigma)$. Moreover $S(\tau)\to S(\sigma) = S(\tau\to\sigma)$, so we may conclude by (abs).

- Assume the last step is:

$$\text{(app)}\quad \frac{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a:\tau\to\sigma \qquad (\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b:\tau}{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a\,b:\sigma}$$

By induction hypothesis $S\bullet(\Delta,\Theta)\,|\,S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a:S(\tau\to\sigma)$ and $S\bullet(\Delta,\Theta)\,|\,S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b:S(\tau)$. Moreover $S(\tau\to\sigma) = S(\tau)\to S(\sigma)$, so we may conclude by (app).

- Assume the last step is:

$$\text{(case)}\quad \frac{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a:d\,\vec{\tau} \qquad (\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i:\mathsf{Dom}^{\vec{\tau}}_d(c_i)\to\sigma \quad (1\le i\le n)}{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{case}_d\ a\ \mathsf{of}\ \{c_1\Rightarrow b_1 \mid \ldots \mid c_n\Rightarrow b_n\}:\sigma}$$

By induction hypothesis $S\bullet(\Delta,\Theta)\,|\,S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a:d\,S(\vec{\tau})$ and $S\bullet(\Delta,\Theta)\,|\,S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i : S(\mathsf{Dom}^{\vec{\tau}}_d(c_i))\to S(\sigma)$ for $1\le i\le n$. Since $\mathsf{Dom}^{\vec{\tau}}_d(c_i)$ has no datatype variables and the only type variables that may occur in it are the ones occurring in $\vec{\tau}$, we have $S(\mathsf{Dom}^{\vec{\tau}}_d(c_i)) = \mathsf{Dom}^{S(\vec{\tau})}_d(c_i)$. Therefore, we can apply rule (case) to conclude.

- Assume the last step is:

$$\text{(rec)}\quad \frac{(\Delta,\Theta)\,|\,\Gamma, f_1:\tau_1,\ldots,f_n:\tau_n \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i:\tau_i \quad (1\le i\le n)}{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{letrec}_j(f_1=e_1,\ldots,f_n=e_n):\tau_j}$$

By induction hypothesis $S\bullet(\Delta,\Theta)\,|\,S(\Gamma), f_1:S(\tau_1),\ldots,f_n:S(\tau_n) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i:S(\tau_i)$ for $1\le i\le n$. So, we conclude by (rec).

- Assume the last step is:

$$\text{(sub)}\quad \frac{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e:\tau \qquad (\Delta,\Theta) \vdash \tau\le\sigma}{(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e:\sigma}$$

By induction hypothesis $S\bullet(\Delta,\Theta)\,|\,S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e:S(\tau)$. Using lemmas 8.5.27, 8.5.31 and 8.5.17, we have $S\bullet(\Delta,\Theta)\vdash S(\tau)\le S(\sigma)$. Hence, by rule (sub), one has $S\bullet(\Delta,\Theta)\,|\,S(\Gamma) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e:S(\sigma)$.

□

Unification and Most General Matching Substitution

Unification is the act of making two types equal.

Definition 8.5.33 (Unifier)

1. *A* unifier *is a substitution that makes two types syntactically equal. More generally, if E is a set of pairs of types, then a substitution S* unifies E *if $S(\sigma) = S(\tau)$ for every pair $\langle\sigma,\tau\rangle \in E$. Since such pairs may be regarded as a set of equations to be solved, we often write the pairs $\langle\sigma,\tau\rangle \in E$ in the form $\sigma = \tau$.*

2. *Let V be a set of type variables, and E be a set of equations. We say that S is a* most general unifier for E (preserving V)*, if S unifies E (and S preserves V), and for all substitutions R that unify E (and preserves V), there exists a substitution U (preserving V) such that $R = U \circ S$.*

In this section several algorithms are presented. In order to make the pseudo-code of these algorithms more readable let us state some notational conventions. Algorithms are written in a pattern-matching style. Whenever it is written $A \cup B$ in the formal argument of an algorithm declaration, it is assumed that $A \cap B \neq \emptyset$. It is also assumed that only non-empty sets match $A \cup B$. Although the matching of a set to a pattern is nondeterministic, this does not affect the correctness of the algorithm. Algorithms are usually defined with the help of subsidiary algorithms and these are assumed to fail if the desired result is not well defined. Moreover, if a subsidiary algorithm fails the main algorithm terminates with an error.

The algorithm for unification presented is almost identical to the standard unification algorithm by Robinson [129]. Our algorithm makes explicit the set V of type variables that are not allowed to be modified. Although this feature is never used in the type inference algorithm for $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$, this notion of unification preserving a set of type variables will be useful in the type-checking algorithm for λ_{CS} (see Subsection 8.5.4) as we do no want the type variables that appear in the context to be changed.

Essentially, the idea is to recursively decompose equations between compound types of the same shape, substituting types for type variables when necessary and possible, since the type variables cannot occur in the type and cannot belong to the set of variables one wants to preserve (if it is not possible the algorithm fails). The algorithm also fails if it is asked to unify compound types with different shapes.

Definition 8.5.34 *The algorithm* unify *is defined in Figure 8.11; and* $\mathsf{unifyData}$ *is defined in Figure 8.12. Moreover, we define* $\mathsf{unify}(E) = \mathsf{unify}(E, \emptyset)$.

Lemma 8.5.35 *The algorithm* unify *takes a finite set E of equations between types and a finite set V of type variables, and produces a most general unifier for E preserving V. If no unifier preserving V exists for E, then* $\mathsf{unify}(E, V)$ *fails.*

Proof. The proof is quite similar to that of [129]. □

Using unification, we can find common substitution instances for our variable contexts. However these substitutions may not be matching for our subtype contexts. Therefore, we need to be able to compute a "most general matching substitution" for this resulting set of possibly non-matching subtype assertions.

Definition 8.5.36 (Most general matching substitution) *Let V be a set of type variables and Δ a set of subtype assertions. S is a* most general matching substitution for Δ (preserving V) *if S a matching substitution for Δ (preserving V); and every other matching substitution for Δ (preserving V), R, may be obtained as a composition of S with some substitution U, $R = U \circ S$.*

$$\begin{array}{l}
\mathsf{unify}(\emptyset, V) = (id, id) \\[1em]
\mathsf{unify}(E \cup \{\sigma = \tau\}, V) = \\
\qquad \mathsf{if}\ \ \sigma = \tau\ \ \mathsf{then}\ \ \mathsf{unify}(E, V) \\
\qquad \mathsf{else\ if}\ \ \tau \in \mathcal{V}_{\mathcal{T}} - V\ \ \mathsf{then}\ \ \mathsf{if}\ \ \tau\ \mathsf{nocc}\ \sigma\ \ \mathsf{then}\ \ \mathsf{let}\ S = ([\tau := \sigma], id) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{unify}(S(E), V) \circ S \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ fail \\
\qquad\qquad \mathsf{else\ if}\ \ \sigma \in \mathcal{V}_{\mathcal{T}} - V\ \ \mathsf{then}\ \ \mathsf{if}\ \ \sigma\ \mathsf{nocc}\ \tau\ \ \mathsf{then}\ \ \mathsf{let}\ S = ([\sigma := \tau], id) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{unify}(S(E), V) \circ S \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ fail \\
\qquad\qquad\quad \mathsf{else}\ fail \\[1em]
\mathsf{unify}(E \cup \{\sigma_1 \to \sigma_2 = \tau_1 \to \tau_2\}, V) = \mathsf{unify}(E \cup \{\sigma_1 = \tau_1, \sigma_2 = \tau_2\}, V) \\[1em]
\mathsf{unify}(E \cup \{\mathtt{d}\,\vec{\sigma} = \mathtt{d}'\vec{\tau}\}, V) = \mathsf{let}\ S = (id, \mathsf{unifyData}(\mathtt{d}, \mathtt{d}')) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{unify}(S(E \cup \{\sigma_i = \tau_i \mid 1 \le i \le \mathsf{ar}(\mathtt{d})\}), V) \circ S
\end{array}$$

Figure 8.11: The algorithm unify

$$\begin{array}{l}
\mathsf{unifyData}(\mathtt{d}, \mathtt{d}') = \mathsf{if}\ \ \mathtt{d} = \mathtt{d}'\ \ \mathsf{then}\ \ id \\
\qquad\qquad\qquad\quad \mathsf{else\ if}\ \ \mathtt{d} \in \mathcal{V}_{\mathcal{D}}^{\mathsf{ar}(\mathtt{d}')}\ \ \mathsf{then}\ [\mathtt{d} := \mathtt{d}'] \\
\qquad\qquad\qquad\qquad \mathsf{else\ if}\ \ \mathtt{d}' \in \mathcal{V}_{\mathcal{D}}^{\mathsf{ar}(\mathtt{d})}\ \ \mathsf{then}\ [\mathtt{d}' := \mathtt{d}] \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ fail
\end{array}$$

Figure 8.12: The algorithm unifyData

$$
\begin{array}{l}
\mathsf{match}\ (\Delta, V) = \\
\quad \mathsf{let}\ \mathsf{replace}\ \text{be a function that replaces each datatype constructor } d \in \mathcal{D} \\
\qquad\qquad \text{by special datatype variables } *_{\mathsf{ar}(d)} \in \mathcal{V}_{\mathcal{D}}^{\mathsf{ar}(d)} \\
\qquad E = \{\mathsf{replace}(\sigma) = \mathsf{replace}(\tau) \mid \sigma \leq \tau \in \Delta\} \\
\qquad S = \mathsf{unify}(E, V) \\
\quad \mathsf{in}\ \ \mathsf{if}\ \exists\ *_n\ .\ S(*_n) \notin \mathcal{V}_{\mathcal{D}}^{n}\ \ \mathsf{then}\ \ fail \\
\qquad\quad \mathsf{else\ let}\ A \text{ and } B \text{ be the sets of type and datatype variables of } \Delta \text{ respectively} \\
\qquad\qquad\qquad \text{compute substitutions } S_1 \text{ and } S_2 \text{ such that} \\
\qquad\qquad\qquad\qquad S_1 \text{ chooses variables freely on } (A, B \cup \bigcup_n \{*_n\}) \text{ and preserves } V, \\
\qquad\qquad\qquad\qquad S_2 \text{ is simple, and } S =_{(A, B \cup \bigcup_n \{*_n\})} S_2 \circ S_1 \\
\qquad\qquad \mathsf{in}\ \ S_1
\end{array}
$$

Figure 8.13: The algorithm match

Matching substitutions are related to unification by the following lemma.

Lemma 8.5.37 *Let V be a of set of type variables, Δ be a set of possibly non-matching subtype assertions and E be the set of equations $E = \{\sigma = \tau \mid \sigma \leq \tau \in \Delta\}$. Then S is a matching substitution for Δ (preserving V) iff there is a simple substitution R such that $R \circ S$ unifies E (and S preserves V).*

Proof. The proof is similar to that of [109]. □

The problem of finding a most general matching substitution for a set Δ of possibly non-matching subtyping assertions is reduced to finding a most general unifier for a set of equations E closely related with Δ. This set E is generated from Δ by converting each inequality in an equation and simultaneously replacing the occurrences of datatype constructors with special datatype variables (one for each distinct arity). This is because distinct datatype constructors (of the same arity) match but do not unify. The strategy is then, to compute the most general unifier for E, check that the resulting substitution merely renames this special datatype variables, and then replace all occurrences of type and datatype variables in the resulting types by distinct fresh type and datatype variables (since one just wants to ensure that the resulting types related by subtyping have the same shape, but are not necessarily equal). For this last task, we factorize the most general unifier into a substitution that chooses variables freely among the set of variables of E, composed with a simple substitution, as described in Lemma 8.5.21.

Definition 8.5.38 *The algorithm* match *is defined in Figure 8.13. Moreover, we define* $\mathsf{match}(\Delta) = \mathsf{match}(\Delta, \emptyset)$.

Lemma 8.5.39 *The algorithm* match, *given a finite set Δ of subtype assertions and a set V of type variables, produces a most general matching substitution for Δ preserving V if any matching substitution for Δ preserving V exists, and fails otherwise.*

Proof. This proof uses lemmas 8.5.37, 8.5.20 and 8.5.21, and is quite similar to that of [109]. □

Example 8.5.40 *Consider the sets of subtype assertions:*

$$\Delta_1 = \{\alpha_1 \to \mathsf{Nat} \leq \alpha_2,\ \alpha_3 \leq \mathsf{List}\,\alpha_4\} \qquad \Delta_2 = \{\mathsf{List}\,\alpha_1 \leq \mathsf{Nat}\}$$
$$\Delta_3 = \{\mathsf{NeList}\,\alpha_1 \leq \mathsf{List}\,(\alpha_2 \to \mathsf{Odd})\} \qquad \Delta_4 = \{\alpha_1 \to \alpha_2 \leq \mathsf{NeList}\,\alpha_3\}$$
$$\Delta_5 = \{\alpha_1 \to \mathsf{Nat} \leq \alpha_2 \to \delta_1,\ \delta_2\,\alpha_3 \leq \mathsf{List}\,\alpha_4,\ \mathsf{Even} \leq \delta_3\}$$

For each of these sets the algorithm match *produces the following results*

$$\mathsf{match}(\Delta_1) = ([\alpha_1 := \beta_1, \alpha_2 := \beta_2 \to \delta_3, \alpha_3 := \delta_4\,\beta_3, \alpha_4 := \beta_4], [\delta_1 := \delta_5, \delta_2 := \delta_6])$$
$$\mathsf{match}(\Delta_2) = fail$$
$$\mathsf{match}(\Delta_3) = ([\alpha_1 := \beta_1 \to \delta_3, \alpha_2 := \beta_2], [\delta_1 := \delta_4, \delta_2 := \delta_5])$$
$$\mathsf{match}(\Delta_4) = fail$$
$$\mathsf{match}(\Delta_5) = ([\alpha_1 := \beta_1, \alpha_2 := \beta_2, \alpha_3 := \beta_3, \alpha_4 := \beta_4], [\delta_1 := \delta_8, \delta_2 := \delta_9, \delta_3 := \delta_{10}, \delta_4 := \delta_6, \delta_5 := \delta_7])$$

Note that the support of the substitution produced by $\mathsf{match}(\Delta_5)$*, contains two datatype variables* (δ_4 *and* δ_5) *that do note occur in* Δ_5*. That is because the algorithm* match *introduces two datatype variables* ($*_0$ *and* $*_1$*, since we have* $\mathsf{Nat}, \mathsf{Even}$ *and* List) *and we have datatype variables in* Δ_5*. However,* $\mathsf{match}(\Delta_5) \bullet (\Delta_5, \emptyset) = (\{\beta_2 \leq \beta_1, \beta_3 \leq \beta_4\}, \{\mathsf{Nat} \sqsubseteq \delta_8, \delta_9 \sqsubseteq \mathsf{List}, \mathsf{Even} \sqsubseteq \delta_{10}\})$*.*

Moreover, $\mathsf{match}(\Delta_1) \bullet (\Delta_1, \emptyset) = (\{\beta_2 \leq \beta_1, \beta_3 \leq \beta_4\}, \{\mathsf{Nat} \sqsubseteq \delta_3, \delta_4 \sqsubseteq \mathsf{List}\})$ *and, if* $R = \mathsf{match}(\Delta_3)$ *and* Θ_2 *is a set of datatype inequalities,* $R \bullet (\Delta_2, \Theta_2) = (\{\beta_2 \leq \beta_1\}, R(\Theta_2) \cup \{\mathsf{NeList} \sqsubseteq \mathsf{List}, \delta_3 \sqsubseteq \mathsf{Odd}\})$*.*

The Algorithm for Most General Typings

We present an algorithm that compute, if it is possible, a most general typing of a term. This algorithm adapts Mitchell's typing algorithm [109] to the features of $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ type system (namely, the specificities of its subtyping relation that is generated structurally from a partial order on type constructors, and the existence of case-expressions and letrec-expressions).

Definition 8.5.41 *The algorithm* typeJudg *is defined in Figure 8.14.*

Given any term e, the algorithm $\mathsf{typeJudg}(e)$ produces a quadruple $(\Gamma, \sigma, \Delta, \Theta)$ such that $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ is a derivable typing judgment. This algorithm is defined inductively on the structure of terms. The algorithms unify and match play a crucial role in typeJudg, while a subsidiary role is played by the algorithms for constructing types, applying and composing substitutions, computing the union or the subtraction of contexts, and by the $\bullet$ operation. The algorithm typeJudg fails when the call to unify or match fails or any of the subsidiary algorithms fail. typeJudg is described informally below.

The behavior of typeJudg on variables and constructors is quite straightforward. For any variable x, typeJudg infers that x is of some supertype β of the assumed type α of x. For any constructor c_d, typeJudg infers that c_d is of some supertype of $\mathsf{Inst}^{\vec{\alpha}}_{d}(c)$.

For compound terms (abstractions, applications, case-expressions and letrec-expressions), the algorithm typeJudg computes a typing judgment for each of the top-level subterms by calling itself recursively. To make the algorithm more readable, we assume that type variables that occur in the typing judgments of different subterms are all distinct[4]. Using a series of steps, typeJudg combines these typings into a well-formed typing judgment for the compound term.

First, typeJudg uses unification to combine variable contexts. In particular, if a variable occurs free in different subterms, then the type assumptions about these distinct occurrences are unified. Furthermore,

[4]This assumption can be made, since we can always rename type variables.

$$\mathsf{typeJudg}(x) = \quad \big(\{x:\alpha\},\, \beta,\, \{\alpha \leq \beta\},\, \emptyset\big)$$

$$\begin{array}{lll}
\mathsf{typeJudg}(c_d) = & \mathsf{let} & S = \mathsf{match}\big(\{\mathsf{Inst}_d^{\vec{\alpha}}(c) \leq \beta\}\big) \\
 & & (\Delta_*, \Theta_*) = S \bullet \big(\{\mathsf{Inst}_d^{\vec{\alpha}}(c) \leq \beta\}, \emptyset\big) \\
 & \mathsf{in} & \big(\emptyset,\, S(\beta),\, \Delta_*,\, \Theta_*\big)
\end{array}$$

assuming that $\vec{\alpha}, \beta$ are fresh type variables.

$$\begin{array}{lll}
\mathsf{typeJudg}(a\,b) = & \mathsf{let} & (\Gamma_1, \rho_1, \Delta_1, \Theta_1) = \mathsf{typeJudg}(a) \\
 & & (\Gamma_2, \rho_2, \Delta_2, \Theta_2) = \mathsf{typeJudg}(b) \\
 & & R = \mathsf{unify}\big(\{\tau_1 = \tau_2 \mid x:\tau_1 \in \Gamma_1 \,\wedge\, x:\tau_2 \in \Gamma_2\} \cup \{\rho_1 = \rho_2 \to \alpha\}\big) \\
 & & S = \mathsf{match}\big(R(\Delta_1 \cup \Delta_2)\big) \circ R \\
 & & (\Delta', \Theta') = S \bullet (\Delta_1 \cup \Delta_2,\, \Theta_1 \cup \Theta_2) \\
 & & \Gamma' = S(\Gamma_1) \cup S(\Gamma_2) \\
 & \mathsf{in} & \big(\Gamma',\, S(\alpha),\, \Delta',\, \Theta'\big)
\end{array}$$

assuming the type variables introduced in $(\Gamma_1, \rho_1, \Delta_1, \Theta_1)$, $(\Gamma_2, \rho_2, \Delta_2, \Theta_2)$, and α are fresh.

$$\begin{array}{lll}
\mathsf{typeJudg}(\lambda x.a) = & \mathsf{let} & (\Gamma_1, \rho_1, \Delta_1, \Theta_1) = \mathsf{typeJudg}(a) \\
 & \mathsf{in} & \mathsf{if}\ \exists\, \tau.\ (x:\tau) \in \Gamma_1 \\
 & & \mathsf{then}\ \big(\Gamma_1 - \{x:\tau\},\, \tau \to \rho_1,\, \Delta_1,\, \Theta_1\big) \\
 & & \mathsf{else}\ \big(\Gamma_1,\, \alpha \to \rho_1,\, \Delta_1,\, \Theta_1\big)
\end{array}$$

assuming that α is a fresh type variable.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{typeJudg}(\mathsf{case}_d\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}) =} \\
 & \mathsf{let} & (\Gamma_0, \rho_0, \Delta_0, \Theta_0) = \mathsf{typeJudg}(a) \\
 & & (\Gamma_i, \rho_i, \Delta_i, \Theta_i) = \mathsf{typeJudg}(b_i) \quad \text{for} \ \ 1 \leq i \leq n \\
 & & \mathsf{A} = \big\{\tau_i = \tau_j \mid x:\tau_i \in \Gamma_i \,\wedge\, x:\tau_j \in \Gamma_j \,\wedge\, i,j \in \{0,...,n\} \,\wedge\, i \neq j\big\} \\
 & & \mathsf{B} = \big\{\rho_i = \mathsf{Dom}_d^{\vec{\alpha}}(c_i) \to \beta \ \mid\ 1 \leq i \leq n\big\} \\
 & & R = \mathsf{unify}\big(\{d\,\alpha_1 \ldots \alpha_{\mathsf{ar}(d)} = \rho_0\} \cup \mathsf{A} \cup \mathsf{B}\big) \\
 & & S = \mathsf{match}\big(R(\Delta_0 \cup \bigcup_{1 \leq i \leq n} \Delta_i)\big) \circ R \\
 & & (\Delta', \Theta') = S \bullet \big(\bigcup_{0 \leq i \leq n} \Delta_i\,,\, \bigcup_{0 \leq i \leq n} \Theta_i\big) \\
 & & \Gamma' = \bigcup_{0 \leq i \leq n} S(\Gamma_i) \\
 & \mathsf{in} & \big(\Gamma',\, S(\beta),\, \Delta',\, \Theta'\big)
\end{array}$$

assuming the type variables introduced in: $\alpha_1, \ldots, \alpha_{\mathsf{ar}(d)}, (\Gamma_i, \rho_i, \Delta_i, \Theta_i)$, for $0 \leq i \leq n$, and $\beta$are fresh.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{typeJudg}(\mathsf{letrec}_k(f_1 = e_1, \ldots, f_n = e_n)) =} \\
 & \mathsf{let} & (\Gamma_i, \rho_i, \Delta_i, \Theta_i) = \mathsf{typeJudg}(e_i) \quad \text{for} \ \ 1 \leq i \leq n \\
 & & \mathsf{A} = \big\{\tau_i = \tau_j \mid x:\tau_i \in \Gamma_i \,\wedge\, x:\tau_j \in \Gamma_j \,\wedge\, i,j \in \{1,...,n\} \,\wedge\, i \neq j\big\} \\
 & & \mathsf{B} = \big\{\rho_i = \sigma_j \ \mid f_i:\sigma_j \in \Gamma_j \,\wedge\, i,j \in \{1,...,n\}\big\} \\
 & & R = \mathsf{unify}(\mathsf{A} \cup \mathsf{B}) \\
 & & S = \mathsf{match}\big(\bigcup_{1 \leq i \leq n} R(\Delta_i)\big) \circ R \\
 & & (\Delta', \Theta') = S \bullet \big(\bigcup_{1 \leq i \leq n} \Delta_i\,,\, \bigcup_{1 \leq i \leq n} \Theta_i\big) \\
 & & \Gamma' = \bigcup_{1 \leq i \leq n} S(\Gamma_i) - \{f_i : S(\rho_i) \mid 1 \leq i \leq n\} \\
 & \mathsf{in} & \big(\Gamma',\, S(\rho_k),\, \Delta',\, \Theta'\big)
\end{array}$$

assuming the type variables introduced in $(\Gamma_i, \rho_i, \Delta_i, \Theta_i)$, for $1 \leq i \leq n$, are fresh.

Figure 8.14: The algorithm $\mathsf{typeJudg}$

- if the term is an application $a\,b$, then the type of a is unified with a function type that maps the type of b to some unspecified type α;
- if the term is a case-expression $\mathsf{case}_d\ a\ \mathsf{of}\,\{\vec{c} \Rightarrow \vec{b}\}$, then the type of a is unified with $d\,\vec{\alpha}$ (with $\vec{\alpha}$ fresh), and the type of each b_i is unified with $\mathsf{Dom}_d^{\vec{\alpha}}(c_i) \to \beta$, with β fresh;
- if the term is a letrec-expression $\mathsf{letrec}_k(f_1 = e_1, \ldots, f_n = e_n)$, then the type of each e_i is unified with the type assumptions made for f_i in the variable contexts.

Second, typeJudg applies the resulting unifiers to the subtype contexts in the original typing judgments and uses match to compute a substitution that causes the types of the new subtype assertions to match. Then a substitution S is generated by composing this matching substitution with the unifier. Finally, applying the substitution S to the original typing judgments (using $\bullet$ on the subtype contexts) and taking the union of the corresponding pieces yields the type judgment for the compound term, except for the case of letrec-expressions where one has still to eliminate from the variable context the declarations of the variables f_i.

We conclude proving the soundness and completeness of the typeJudg algorithm with respect to the $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ type system. To show that typeJudg is sound is to prove that for any term e, if $\mathsf{typeJudg}(e)$ returns $(\Gamma, \sigma, \Delta, \Theta)$, then every instance of $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ is derivable. Using Theorem 8.5.32 we just have to prove the next lemma.

Lemma 8.5.42 *If* $\mathsf{typeJudg}(e) = (\Gamma, \sigma, \Delta, \Theta)$*, then* $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ *is derivable.*

Proof. By induction on the structure of e.

- Assume $e \equiv x$. We have $\mathsf{typeJudg}(x) = (\{x : \alpha\}, \beta, \{\alpha \leq \beta\}, \emptyset)$ and $(\{\alpha \leq \beta\}, \emptyset) \,|\, x : \alpha \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} x : \beta$ is derivable using rule (var) and (sub).

- Assume $e \equiv c_d$ and $\mathsf{typeJudg}(c_d)$ is as presented in Figure 8.14. Since S is a matching substitution for $\{\mathsf{Inst}_d^{\vec{\alpha}}(c) \leq \beta\}$, we know that $S \bullet (\{\mathsf{Inst}_d^{\vec{\alpha}}(c) \leq \beta\}, \emptyset)$ is a well-formed subtype context and that $S \bullet (\{\mathsf{Inst}_d^{\vec{\alpha}}(c) \leq \beta, \emptyset) \vdash \mathsf{Inst}_d^{S(\vec{\alpha})}(c) \leq S(\beta)$, using Lemma 8.5.27 and the fact that $S(\mathsf{Inst}_d^{\vec{\alpha}}(c)) = \mathsf{Inst}_d^{S(\vec{\alpha})}(c)$. By rule (cons) $(\Delta_*, \Theta_*) \,|\, \emptyset \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c : \mathsf{Inst}_d^{S(\vec{\alpha})}(c)$ is derivable. So, by rule (sub), we can derive $(\Delta_*, \Theta_*) \,|\, \emptyset \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c : S(\beta)$.

- Assume $e \equiv a\,b$ and $\mathsf{typeJudg}(a\,b)$ is as presented in Figure 8.14. By induction hypothesis both $(\Delta_1, \Theta_1) \,|\, \Gamma_1 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \rho_1$ and $(\Delta_2, \Theta_2) \,|\, \Gamma_2 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b : \rho_2$ are derivable judgments. Since the substitution S is defined from the unifier R by composition, S must unify $\{\tau_1 = \tau_2 \mid x : \tau_1 \in \Gamma_1 \wedge x : \tau_2 \in \Gamma_2\} \cup \{\rho_1 = \rho_2 \to \alpha\}$. This implies that $S(\Gamma_1) \cup S(\Gamma_2)$ is a well-formed variable context. Since S is a matching substitution for $\Delta_1 \cup \Delta_2$, we know that $S \bullet (\Delta_1 \cup \Delta_2, \Theta_1 \cup \Theta_2)$ is a well-formed subtype context. By Theorem 8.5.32 it follows that the two judgments $S \bullet (\Delta_1 \cup \Delta_2, \Theta_1 \cup \Theta_2) \,|\, S(\Gamma_1) \cup S(\Gamma_2) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : S(\rho_1)$ and $S \bullet (\Delta_1 \cup \Delta_2, \Theta_1 \cup \Theta_2) \,|\, S(\Gamma_1) \cup S(\Gamma_2) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b : S(\rho_2)$ are derivable. As $S(\rho_1) = S(\rho_2) \to S(\alpha)$, we obtain $S \bullet (\Delta_1 \cup \Delta_2, \Theta_1 \cup \Theta_2) \,|\, S(\Gamma_1) \cup S(\Gamma_2) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a\,b : S(\alpha)$ by rule (app).

- Assume $e \equiv \lambda x.a$ and $\mathsf{typeJudg}(\lambda x.a)$ is as presented in Figure 8.14. By induction hypothesis $(\Delta_1, \Theta_1) \,|\, \Gamma_1 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \rho_1$ is derivable.
  - If $x \in \mathsf{FV}(a)$ then $x \in \mathsf{dom}(\Gamma_1)$, by Lemma 8.5.45. Let $\Gamma_1(x) = \tau$. By rule (abs), we may derive $(\Delta_1, \Theta_1) \,|\, \Gamma_1 - \{x : \tau\} \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.a : \tau \to \rho_1$.

– If $x \notin \mathsf{FV}(a)$ then $x \notin \mathsf{dom}(\Gamma_1)$, by Lemma 8.5.45. By Lemma 8.5.30 one has $(\Delta_1, \Theta_1) \,|\, \Gamma_1, x : \alpha \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \rho_1$, being α a fresh type variable. Hence, $(\Delta_1, \Theta_1) \,|\, \Gamma_1 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.a : \alpha \to \rho_1$ follows by rule (abs).

- Assume $e \equiv \mathsf{case}_d\ a\ \mathsf{of}\,\{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ and $\mathsf{typeJudg}(\mathsf{case}_d\ a\ \mathsf{of}\,\{\vec{c} \Rightarrow \vec{b}\})$ is as presented in Figure 8.14. By induction hypothesis $(\Delta_0, \Theta_0) \,|\, \Gamma_0 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \rho_0$ and $(\Delta_i, \Theta_i) \,|\, \Gamma_i \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i : \rho_i$, for $1 \leq i \leq n$. Since S is defined from the unifier R by composition, S must unify $\{d\,\vec{\alpha} = \rho_0\} \cup \mathsf{A} \cup \mathsf{B}$. This implies that $\Gamma' = \bigcup_{0 \leq i \leq n} S(\Gamma_i)$ is a well-formed variable context. Since S is a matching substitution for $\bigcup_{0 \leq i \leq n} \Delta_i$, we know that $(\Delta', \Theta') = S \bullet (\bigcup_{0 \leq i \leq n} \Delta_i, \bigcup_{0 \leq i \leq n} \Theta_i)$ is a well-defined subtype context. Moreover $S(\rho_i) = \mathsf{Dom}_d^{S(\vec{\alpha})}(c_i) \to S(\beta)$, for $1 \leq i \leq n$.

  Now, it is easy to check that, $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i : \mathsf{Dom}_d^{S(\vec{\alpha})}(c_i) \to S(\beta)$ is an instance of $(\Delta_i, \Theta_i) \,|\, \Gamma_i \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i : \rho_i$ by S, for $1 \leq i \leq n$. Furthermore, $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : d\,S(\vec{\alpha})$ is an instance of $(\Delta_0, \Theta_0) \,|\, \Gamma_0 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \rho_0$ by S. By Theorem 8.5.32 the following judgments are derivable: $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : d\,S(\vec{\alpha})$ and

  $$(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i : \mathsf{Dom}_d^{S(\vec{\alpha})}(c_i) \to S(\beta)\ ,\ \text{for } 1 \leq i \leq n$$

  So we can apply rule (case) and derive

  $$(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{case}_d\ a\ \mathsf{of}\,\{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : S(\beta)$$

- Assume $e \equiv \mathsf{letrec}_k(f_1 = e_1, \ldots, f_n = e_n)$ and $\mathsf{typeJudg}(\mathsf{letrec}_k(\vec{f} \Rightarrow \vec{e}))$ is as presented in Figure 8.14. By induction hypothesis $(\Delta_i, \Theta_i) \,|\, \Gamma_i \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i : \rho_i$, for $1 \leq i \leq n$. Since substitution S is defined from the unifier R by composition, S must unify $\{\tau_i = \tau_j \mid x : \tau_i \in \Gamma_i \wedge x : \tau_j \in \Gamma_j \wedge i, j \in \{1, ..., n\} \wedge i \neq j\}$. This implies that $\bigcup_{1 \leq i \leq n} S(\Gamma_i)$ is a well-formed variable context. Since S is a matching substitution for $\bigcup_{1 \leq i \leq n} \Delta_i$, we know that $(\Delta', \Theta') = S \bullet (\bigcup_{1 \leq i \leq n} \Delta_i, \bigcup_{1 \leq i \leq n} \Theta_i)$ is a well-formed subtype context. So, for each $i \in \{1, \ldots, n\}$, $(\Delta', \Theta') \,|\, \bigcup_{1 \leq i \leq n} S(\Gamma_i) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i : S(\rho_i)$ is an instance of $(\Delta_i, \Theta_i) \,|\, \Gamma_i \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i : \rho_i$ by S; and therefore, by Theorem 8.5.32 it follows that, for $1 \leq i \leq n$,

  $$(\Delta', \Theta') \,|\, \bigcup_{1 \leq i \leq n} S(\Gamma_i) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i : S(\rho_i)$$

  Let $\Gamma = \bigcup_{1 \leq i \leq n} S(\Gamma_i) \cup \{f_j : S(\rho_j) | 1 \leq j \leq n\}$. By Lemma 8.5.30, $(\Delta', \Theta') \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i : S(\rho_i)$ is derivable for every $1 \leq i \leq n$. Now, by rule (rec), we can derive

  $$(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{letrec}_k(f_1 = e_1, \ldots, f_n = e_n) : S(\rho_k)$$

□

Theorem 8.5.43 (Soundness of typeJudg) *If* $\mathsf{typeJudg}(e) = (\Gamma, \sigma, \Delta, \Theta)$*, then every instance of* $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ *is provable.*

Proof. By Theorem 8.5.32 and Lemma 8.5.42. □

We want now to prove that the algorithm typeJudg is complete, that is, any derivable typing judgment for a term e is an instance of the typing produced by $\mathsf{typeJudg}(e)$. To simplify this proof we first present a lemma stating that any typing derivation may be put in a "normal form", in which the subsumption rule is used only after the axioms in the typing system. This means that subtyping only enter into the base cases of the typing algorithm.

Lemma 8.5.44 *Suppose* $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ *is derivable. Then there is a derivation in which rule (sub) is only used immediately after the typing axioms (var) and (cons).*

Proof. Similar to the proof presented in [109]. $\square$

The type context produced by $\mathsf{typeJudg}(e)$ always contains exactly the free variables of e.

Lemma 8.5.45 *Assume* $\mathsf{typeJudg}(e) = (\Gamma, \sigma, \Delta, \Theta)$*. Then,* $x \in \mathsf{dom}(\Gamma)$ *iff* $x \in \mathsf{FV}(e)$*.*

Proof. By induction on the structure of e. $\square$

We also need the following property of substitutions.

Lemma 8.5.46 *Suppose that* Δ *is a set of subtype assertions between possibly non-matching types. Furthermore, suppose that* S*,* R *and* W *are substitutions with* $W =_{(V_{\mathcal{T}}, V_{\mathcal{D}})} S \circ R$*, where* $V_{\mathcal{T}}$ *and* $V_{\mathcal{D}}$ *contain all type and datatype variables, respectively, that occur in* Δ*. If* W *and* R *are matching substitutions for* (Δ, Θ)*, then* S *is a matching substitution for* $(R \bullet (\Delta, \Theta))$*, and* $S \bullet (R \bullet (\Delta, \Theta)) = W \bullet (\Delta, \Theta)$*.*

Proof. By induction on the structure of types. $\square$

Theorem 8.5.47 (Completeness of typeJudg) *If* $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$*, then* $\mathsf{typeJudg}(e) = (\Gamma', \sigma', \Delta', \Theta')$ *and* $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$ *is an instance of* $(\Delta', \Theta') \mid \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma'$*.*

Proof. By induction on the structure of terms. Assuming, for each case, $\mathsf{typeJudg}(e)$ as presented in Figure 8.14.

- Assume $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} x : \sigma$. By Lemma 8.5.44, we may assume the proof uses rule (var) followed by rule (sub). Since x must appear in Γ, we let $\Gamma(x) = \tau$ and note that $(\Delta, \Theta) \vdash \tau \leq \sigma$ must hold. By Lemma 8.5.23, τ matches σ.

  We have $\mathsf{typeJudg}(x) = (\{x : \alpha\}, \beta, \{\alpha \leq \beta\}, \emptyset)$, being α and β fresh type variables. Let S be the substitution $([\alpha := \tau, \beta := \sigma], id)$. It is easy to check that $(\Delta, \Theta) \vdash S \bullet \{\alpha \leq \beta\}$, $S(x : \tau) \subseteq \Gamma$ and $S(\beta) = \sigma$. Hence, it follows that $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} x : \sigma$ is an instance of $(\{\alpha \leq \beta\}, \emptyset) \mid x : \alpha \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} x : \beta$ by S.

- Assume $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c_d : \sigma$. By Lemma 8.5.44, we may assume the proof uses rule (cons) followed by rule (sub). We let $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c_d : \mathsf{Inst}^{\vec{\tau}}_d(c)$ for some $\vec{\tau}$ and note that $(\Delta, \Theta) \vdash \mathsf{Inst}^{\vec{\tau}}_d(c) \leq \sigma$ must hold. We have $\mathsf{typeJudg}(c_d) = (\emptyset, S(\beta), \Delta_*, \Theta_*)$ and S is a most general matching substitution for $\{\mathsf{Inst}^{\vec{\alpha}}_d(c) \leq \beta\}$, being $\vec{\alpha}, \beta$ fresh type variables.

  Let Z be a substitution such that $Z(\vec{\alpha}) = \vec{\tau}$ and $Z(\beta) = \sigma$. So, $Z(\mathsf{Inst}^{\vec{\alpha}}_d(c)) = \mathsf{Inst}^{\vec{\tau}}_d(c)$. Since (Δ, Θ) is atomic, $\mathsf{Inst}^{\vec{\tau}}_d(c)$ matches σ, by Lemma 8.5.39. Therefore, Z is a matching substitution for $\{\mathsf{Inst}^{\vec{\alpha}}_d(c) \leq \beta\}$. Using Lemma 8.5.46, $Z = W \circ S$ for some substitution W. Moreover, by Lemma 8.5.23, W is a matching substitution for (Δ_*, Θ_*) and $W \bullet (\Delta_*, \Theta_*) = Z \bullet (\{\mathsf{Inst}^{\vec{\alpha}}_d(c) \leq \beta\}, \emptyset)$.

We have $(\Delta,\Theta) \vdash (Z(\{\mathsf{Inst}_d^{\vec{\alpha}}(c) \leq \beta\}), Z(\emptyset))$ then, by Lemma 8.5.27, $(\Delta,\Theta) \vdash Z \bullet (\{\mathsf{Inst}_d^{\vec{\alpha}}(c) \leq \beta\}, \emptyset)$. By Lemma 8.5.46, we have $(\Delta,\Theta) \vdash W \bullet (\Delta_*, \Theta_*)$. Furthermore, $W(\emptyset) \subseteq \Gamma$ and $\sigma = W(S(\beta))$. Hence, we can conclude that $(\Delta,\Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c_d : \sigma$ is an instance of $(\Delta_*, \Theta_*) \,|\, \emptyset \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} c_d : S(\beta)$ by W.

- Assume $(\Delta,\Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.a : \sigma$. By Lemma 8.5.44, $(\Delta,\Theta) \,|\, \Gamma, x : \sigma_1 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \sigma_2$ is derivable with $\sigma = \sigma_1 \to \sigma_2$. By induction hypothesis $\mathsf{typeJudg}(a) = (\Gamma_1, \rho_1, \Delta_1, \Theta_1)$ and there is a matching substitution W such that

$$(\Delta,\Theta) \vdash W \bullet (\Delta_1, \Theta_1)\,, \quad W(\Gamma_1) \subseteq \Gamma, x : \sigma_1\,, \quad \text{and} \quad \sigma_2 = W(\rho_1)$$

  - If $x \in \mathsf{FV}(a)$ then $x \in \mathsf{dom}(\Gamma_1)$, by Lemma 8.5.45. Let $\Gamma_1(x) = \tau$, thus $W(\tau) = \sigma_1$. We have $\mathsf{typeJudg}(\lambda x.a) = (\Gamma_1 - \{x : \tau\}, \tau \to \rho_1, \Delta_1, \Theta_1)$ and it is easy to check that $W(\Gamma_1 - \{x : \tau\}) \subseteq \Gamma$ and $W(\tau \to \rho_1) = \sigma_1 \to \sigma_2 = \sigma$. Hence, $(\Delta,\Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.a : \sigma$ is an instance of $(\Delta_1, \Theta_1) \,|\, \Gamma_1 - \{x : \tau\} \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.a : \tau \to \rho_1$ by W.
  - If $x \notin \mathsf{FV}(a)$ then $x \notin \mathsf{dom}(\Gamma_1)$, by Lemma 8.5.45. We have $\mathsf{typeJudg}(\lambda x.a) = (\Gamma_1, \alpha \to \rho_1, \Delta_1, \Theta_1)$ with α fresh. Let $S = ([\alpha := \sigma_1], id) \circ W$. Since α is a fresh type variable w.r.t. $(\Gamma_1, \rho_1, \Delta_1, \Theta_1)$ and $x \notin \mathsf{dom}(\Gamma_1)$, it is easy to check that $(\Delta,\Theta) \vdash S \bullet (\Delta_1, \Theta_1)$, $S(\Gamma_1) \subseteq \Gamma$ and $S(\alpha \to \rho_1) = \sigma$. Therefore, $(\Delta,\Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.a : \sigma$ is an instance of $(\Delta_1, \Theta_1) \,|\, \Gamma_1 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.a : \alpha \to \rho_1$ by S.

- Assume $(\Delta,\Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a\, b : \sigma$. By Lemma 8.5.44, this must follow from $(\Delta,\Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : \tau \to \sigma$ and $(\Delta,\Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b : \tau$ by rule (app). By induction hypothesis $\mathsf{typeJudg}(a) = (\Gamma_1, \rho_1, \Delta_1, \Theta_1)$ and $\mathsf{typeJudg}(b) = (\Gamma_2, \rho_2, \Delta_2, \Theta_2)$ are most general typings for a and b. So, there exist substitutions W_1 and W_2 such that

$$\begin{array}{llll} (\Delta,\Theta) \vdash W_1 \bullet (\Delta_1, \Theta_1)\,, & W_1(\Gamma_1) \subseteq \Gamma\,, & \text{and} & W_1(\rho_1) = \tau \to \sigma \\ (\Delta,\Theta) \vdash W_2 \bullet (\Delta_2, \Theta_2)\,, & W_2(\Gamma_2) \subseteq \Gamma\,, & \text{and} & W_2(\rho_2) = \tau \end{array}$$

Since the algorithm $\mathsf{typeJudg}$ assures that the variables occurring in $(\Gamma_1, \rho_1, \Delta_1, \Theta_1)$ and $(\Gamma_2, \rho_2, \Delta_2, \Theta_2)$ are distinct, we can combine substitutions W_1 and W_2. Let W be any substitution such that

$$\begin{cases} W(\alpha) = \sigma \\ W(\beta) = W_1(\beta) & \text{if } \beta \text{ appears in the typing of } a \\ W(\beta) = W_2(\beta) & \text{if } \beta \text{ appears in the typing of } b \end{cases}$$

It is easy to see that

$$\begin{array}{llll} (\Delta,\Theta) \vdash (W(\Delta_1), W(\Theta_1))\,, & W(\Gamma_1) \subseteq \Gamma\,, & \text{and} & W(\rho_1) = \tau \to \sigma \\ (\Delta,\Theta) \vdash (W(\Delta_2), W(\Theta_2))\,, & W(\Gamma_2) \subseteq \Gamma\,, & \text{and} & W(\rho_2) = \tau \end{array}$$

By Lemma 8.5.30, Γ must give types to all free variables of a and b and, as stated in Lemma 8.5.45, a type context produced by $\mathsf{typeJudg}(e)$ always contains exactly the variables that occur free in e. Therefore, W must unify $\{\tau_1 = \tau_2 \mid x : \tau_1 \in \Gamma_1 \wedge x : \tau_2 \in \Gamma_2\}$. In addition, since $W(\rho_1) = \tau \to \sigma = W(\rho_2) \to W(\alpha)$, the substitution W unifies ρ_1 with $\rho_2 \to \alpha$. Because R is a most general unifier for these equations, there is a substitution Z such that $W = Z \circ R$. Since (Δ,Θ) is atomic, it follows by Lemma 8.5.23 that W is a matching substitution for Δ_1 and for Δ_2; so, W is a matching substitution for $\Delta_1 \cup \Delta_2$. Therefore, Z must be a matching substitution for $R(\Delta_1 \cup \Delta_2)$. But, as match computes a most general matching substitution, this implies that $Z = T \circ \mathsf{match}(R(\Delta_1 \cup \Delta_2))$ for some substitution T. We have

$$W = T \circ \mathsf{match}(R(\Delta_1 \cup \Delta_2)) \circ R = T \circ S$$

We have $(\Delta, \Theta) \vdash (W(\Delta_1 \cup \Delta_2), W(\Theta_1 \cup \Theta_2))$ so, by Lemma 8.5.27, $(\Delta, \Theta) \vdash W \bullet (\Delta_1 \cup \Delta_2, \Theta_1 \cup \Theta_2)$. Using Lemma 8.5.46, it follows that $(\Delta, \Theta) \vdash T \bullet (S \bullet (\Delta_1 \cup \Delta_2, \Theta_1 \cup \Theta_2))$. Moreover, it can be easily checked that $T(S(\Gamma_1) \cup S(\Gamma_2)) \subseteq \Gamma$ and $T(S(\alpha)) = \sigma$. Therefore, $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a\, b : \sigma$ is an instance of $S \bullet (\Delta_1 \cup \Delta_2, \Theta_1 \cup \Theta_2) \,|\, S(\Gamma_1) \cup S(\Gamma_2) \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a\, b : S(\alpha)$ by T.

- Assume $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{case}_d\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma$. By Lemma 8.5.44, this must follow, by rule (case), from $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} a : d\,\vec{\tau}$ and $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} b_i : \mathsf{Dom}^{\vec{\tau}}_d(c_i) \rightarrow \sigma$ for $1 \leq i \leq n$, where $\mathsf{ar}(d) = \#\vec{\tau}$ for some $\vec{\tau}$.

  By induction hypothesis $\mathsf{typeJudg}(a)$ and $\mathsf{typeJudg}(b_i)$ are most general typings for a and b_i for $1 \leq i \leq n$. So, there exist substitutions $W_0, W_1, \ldots, W_n$ such that

  $$\begin{array}{llll} (\Delta, \Theta) \vdash W_0 \bullet (\Delta_0, \Theta_0)\,, & W_0(\Gamma_0) \subseteq \Gamma\,, & \text{and} & W_0(\rho_0) = d\,\vec{\tau} \\ (\Delta, \Theta) \vdash W_i \bullet (\Delta_i, \Theta_i)\,, & W_i(\Gamma_i) \subseteq \Gamma\,, & \text{and} & W_i(\rho_i) = \mathsf{Dom}^{\vec{\tau}}_d(c_i) \rightarrow \sigma \quad (1 \leq i \leq n) \end{array}$$

  Since the algorithm $\mathsf{typeJudg}$ assures that type variables occurring in $(\Gamma_i, \rho_i, \Delta_i, \Theta_i)$, for $0 \leq i \leq n$, $\vec{\alpha}$ and β are all distinct, we can combine substitutions W_i, with $i \in \{0, \ldots, n\}$. Let F be any substitution such that

  $$\begin{cases} W(\beta) = \sigma \\ W(\alpha) = W_0(\alpha) & \text{if } \alpha \text{ appears in the typing of } a \\ W(\alpha) = W_i(\alpha) & \text{if } \alpha \text{ appears in the typing of } b_i \\ W(\vec{\alpha}) = \vec{\tau} \end{cases}$$

  By Lemma 8.5.30, Γ must give types to all free variables of a and b_i and, as stated in Lemma 8.5.45, a type context produced by $\mathsf{typeJudg}(e)$ always contains exactly the variables that occur free in e. Therefore, W must unify A. In addition, since $W(\rho_0) = d\,\vec{\tau} = d\,W(\vec{\alpha})$, the substitution W unifies ρ_0 with $d\,\vec{\alpha}$. Moreover, since $W(\rho_i) = \mathsf{Dom}^{\vec{\tau}}_d(c_i) \rightarrow \sigma = W(\mathsf{Dom}^{\vec{\alpha}}_d(c_i) \rightarrow \beta)$ for $1 \leq i \leq n$, then W must unify B.

  Because R is a most general unifier for $\{d\,\vec{\alpha} = \rho_0\} \cup \mathsf{A} \cup \mathsf{B}$, there is a substitution Z such that $W = Z \circ R$. Since (Δ, Θ) is atomic, it follows by Lemma 8.5.23 that W is a matching substitution for $\bigcup_{0 \leq i \leq n} \Delta_i$. Therefore, Z must be a matching substitution for $R(\bigcup_{0 \leq i \leq n} \Delta_i)$. But, as match computes a most general matching substitution, this implies that $Z = T \circ \mathsf{match}(R(\bigcup_{0 \leq i \leq n} \Delta_i))$ for some substitution T. We have

  $$W = T \circ \mathsf{match}(R(\bigcup_{0 \leq i \leq n} \Delta_i)) \circ R = T \circ S$$

  We can now prove that $(\Delta, \Theta) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{case}_d\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma$ is an instance of $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{case}_d\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : S(\beta)$ by T.

  - One has $(\Delta, \Theta) \vdash W \bullet (\bigcup_{0 \leq i \leq n} \Delta_i, \bigcup_{0 \leq i \leq n} \Theta_i)$, therefore $(\Delta, \Theta) \vdash (T \circ S) \bullet (\bigcup_{0 \leq i \leq n} \Delta_i, \bigcup_{0 \leq i \leq n} \Theta_i)$. Using Lemma 8.5.46, it follows that $(\Delta, \Theta) \vdash T \bullet (S \bullet (\bigcup_{0 \leq i \leq n} \Delta_i, \bigcup_{0 \leq i \leq n} \Theta_i))$. Hence $(\Delta, \Theta) \vdash T \bullet (\Delta', \Theta')$.
  - One has $W(\bigcup_{0 \leq i \leq n} \Gamma_i) \subseteq \Gamma$. As $W = T \circ S$, we can conclude

    $$T(\Gamma') = T\big(S(\bigcup_{0 \leq i \leq n} \Gamma_i)\big) \subseteq \Gamma$$

  - Finally, one has $T(S(\beta)) = W(\beta) = \sigma$.

- Assume $(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{letrec}_k(f_1 = e_1,\ldots,f_n = e_n) : \phi_k$. By Lemma 8.5.44, this must follow, by rule (rec), from $(\Delta,\Theta)\,|\,\Gamma, f_1:\phi_1,\ldots,f_n:\phi_n \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e_i : \phi_i$. By induction hypothesis $\mathsf{typeJudg}(e_i) = (\Gamma_i,\rho_i,\Delta_i,\Theta_i)$ are most general typings for e_i, for $1 \le i \le n$. So, there exist substitutions W_i, for $1 \le i \le n$, such that

$$(\Delta,\Theta) \vdash W_i \bullet (\Delta_i,\Theta_i)\,, \qquad W_i(\Gamma_i) \subseteq \Gamma, f_1:\phi_1,\ldots,f_n:\phi_n\,, \qquad \text{and} \qquad W_i(\rho_i) = \phi_i$$

  Since the algorithm $\mathsf{typeJudg}$ assures that the variables occurring in $(\Gamma_i,\rho_i,\Delta_i,\Theta_i)$, for $1 \le i \le n$, are distinct, we can combine substitutions W_i. Let W be any substitution such that

$$W(\alpha) = W_i(\alpha) \ \text{ if } \beta \text{ appears in the typing of } e_i$$

  It is easy to see that, for $1 \le i \le n$,

$$(\Delta,\Theta) \vdash (W(\Delta_i),W(\Theta_i))\,, \qquad W(\Gamma_i) \subseteq \Gamma, f_1:\phi_1,\ldots,f_n:\phi_n\,, \qquad \text{and} \qquad W(\rho_i) = \phi_i$$

  By lemmas 8.5.30 and 8.5.45 one concludes that W must unify A. Moreover, W must unify B because $W(\rho_i) = \phi_i$, for every $1 \le i \le n$. As R is a most general unifier for these equations, there is a substitution Z such that $W = Z \circ R$. Since (Δ,Θ) is atomic, it follows by Lemma 8.5.23 that W is a matching substitution for $\bigcup_{1\le i\le n}\Delta_i$. But, as match computes a most general matching substitution, this implies that $Z = T \circ \mathsf{match}(\bigcup_{1\le i\le n} R(\Delta_i))$ for some substitution T. So,

$$W = T \circ \mathsf{match}(\bigcup_{1\le i\le n} R(\Delta_i)) \circ R = T \circ S$$

  We have $(\Delta,\Theta) \vdash (W(\bigcup_{1\le i\le n}\Delta_i), W(\bigcup_{1\le i\le n}\Theta_i))$, hence by lemmas 8.5.27 and 8.5.46, it follows that $(\Delta,\Theta) \vdash T \bullet (\Delta',\Theta')$. We also have

$$T(\Gamma') = T(\bigcup_{1\le i\le n} S(\Gamma_i) - \{f_i : S(\rho_i)\,|\,1 \le i \le n\}) = \bigcup_{1\le i\le n} W(\Gamma_i) - \{f_i:\phi_i\,|\,1\le i\le n\} \subseteq \Gamma$$

  and $T(S(\rho_k)) = W(\rho_k) = \phi_k$. Consequently, $(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{letrec}_k(f_1 = e_1,\ldots,f_n = e_n) : \phi_k$ is an instance of $(\Delta',\Theta')\,|\,\Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{letrec}_k(f_1 = e_1,\ldots,f_n = e_n) : S(\rho_k)$ by T.

□

Let us show some examples of typings returned by the $\mathsf{typeJudg}$ algorithm.

Example 8.5.48 *The most general typings produced by the* $\mathsf{typeJudg}$ *algorithm for some terms:*

1. $(\emptyset, \{\delta_1 \sqsubseteq \delta_3, \delta_3 \sqsubseteq \mathsf{Even}, \mathsf{Odd} \sqsubseteq \delta_2\})\,|\,x:\delta_1 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{s}_{\mathsf{Odd}}\,x : \delta_2$
2. $(\emptyset, \{\delta_2 \sqsubseteq \mathsf{Even}, \mathsf{Odd} \sqsubseteq \delta_1, \mathsf{Nat} \sqsubseteq \delta_2\})\,|\, \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{s}_{\mathsf{Odd}}\,\mathsf{o}_{\mathsf{Nat}} : \delta_1$
3. $(\{\alpha_5 \le \alpha_1, \alpha_6 \le \alpha_1, \alpha_2 \le \alpha_6, \alpha_2 \le \alpha_4, \alpha_3 \le \alpha_5\}, \emptyset)\,|\,\emptyset \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda f.\lambda x.f\,(f\,x) : (\alpha_1 \to \alpha_2) \to \alpha_3 \to \alpha_4$
4. $(\emptyset, \{\delta_1 \sqsubseteq \delta_2, \delta_3 \sqsubseteq \mathsf{NeList}, \delta_2 \sqsubseteq \delta_4, \delta_4 \sqsubseteq \mathsf{Even}, \mathsf{Odd} \sqsubseteq \delta_5\})\,|\, \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \lambda x.\mathsf{case}_{\mathsf{NeList}}\ x\ \mathsf{of}\ \{\mathsf{cons} \Rightarrow \lambda h.\lambda t.\,\mathsf{s}_{\mathsf{Odd}}\,h\} : (\delta_3\,\delta_1) \to \delta_5$
5. $(\{\alpha_3 \le \alpha_6, \alpha_5 \le \alpha_6, \alpha_6 \le \alpha_1, \alpha_2 \le \alpha_3, \alpha_4 \le \alpha_5\}, \{\delta_3 \sqsubseteq \delta_1, \delta_1 \sqsubseteq \mathsf{List}, \mathsf{NeList} \sqsubseteq \delta_2\})\,|\,x:\alpha_2, l:\delta_3\,\alpha_4 \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} \mathsf{cons}_{\mathsf{NeList}}\,x\,l : \delta_2\,\alpha_1$

As can be observed in the example above, many of the constraints that appear in the typings returned by the $\mathsf{typeJudg}$ algorithm are redundant. The sets of constraints of such typing judgments can be simplified. However we do not address this problem, since it is not needed for the decidability of type checking in λ_{CS}.

8.5.4 Decidability of Type Checking

Given a context Γ, a term e, and a type τ, type checking consists of analyzing whether the judgment $\Gamma \vdash_{\lambda_{\mathsf{CS}}} e : \tau$ is derivable from the set of typing rules of λ_{CS}. Some features of λ_{CS}, namely: the overloading of constructors and the subtyping relation, make this a complex task. Since each term may be assigned more than one type and the minimal type property does not hold for λ_{CS}, our strategy to decide type checking in λ_{CS} is established in two steps:

1. an algorithm for producing the set of all annotated terms generated from a λ_{CS}-term;

2. an algorithm for deciding the derivability of a $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ typing judgment.

Essentially, the first step handles overloading and the second step handles the subtyping ordering problem. Our claim is that $\Gamma \vdash_{\lambda_{\mathsf{CS}}} e : \tau$ is derivable if and only if there is an annotated term $e' \in \mathsf{an}(e)$ for which $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e' : \tau$ is derivable. As the set of annotated terms $\mathsf{an}(e)$ is finite and type-checking is decidable in $\lambda^{\mathsf{a}}_{\mathsf{CS}}$, we have here a decision procedure.

The system $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ does not enjoy the minimal type property, so the type-checking algorithm for $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ relies on:

1. an algorithm for computing a most general typing in $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$;

2. an algorithm for deciding the satisfiability of a subtyping context.

Definition 8.5.49 *A subtype context* (Δ, Θ) *is* satisfiable *if there exists a substitution* S *such that* $\vdash (S(\Delta), S(\Theta))$

We begin showing the decidability of type checking for $\lambda^{\mathsf{a}}_{\mathsf{CS}}$. Before going into this proof, some lemmata involving the transition between $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ and $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$, are considered.

Lemma 8.5.50 *If* $\emptyset \vdash \mathtt{d} \sqsubseteq \mathtt{d}'$, *then* $(\mathtt{d} \equiv d \wedge \mathtt{d}' \equiv d' \wedge d \sqsubseteq_{\mathcal{D}} d')$ *or* $(\mathtt{d} \equiv \delta \wedge \mathtt{d}' \equiv \delta)$.

Proof. By induction on the derivation of $\emptyset \vdash \mathtt{d} \sqsubseteq \mathtt{d}'$. □

Lemma 8.5.51 $\vdash \tau \leq \sigma$ *in* $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ *and both* τ *and* σ *have no datatype variables iff* $\tau \leq \sigma$ *in* $\lambda^{\mathsf{a}}_{\mathsf{CS}}$.

Proof.

⇒) By induction on the derivation of $\vdash \tau \leq \sigma$.

⇐) By induction on the derivation of $\tau \leq \sigma$.

□

Lemma 8.5.52 $(\emptyset, \emptyset) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau$, *where there are no occurrences of datatype variables neither in* Γ *nor in* τ, *is derivable iff* $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ *is derivable.*

Proof.

⇒) By induction on the derivation of $(\emptyset, \emptyset) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau$.

⇐) By induction on the derivation of $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$.

□

Based on Lemma 8.5.52, we reduce the problem of checking if a typing judgment $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ is derivable to checking if $(\emptyset, \emptyset) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau$ is an instance of the most general typing judgment for e in $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$. The procedure of checking if $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ is derivable can be described briefly by the following steps: compute the most general typing for e in $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ (let it be $(\Delta', \Theta') \mid \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau'$); check if Γ' is not bigger than Γ; unify (preserving the type variables of Γ and τ) τ with τ' and the types assumed for the variables declared simultaneously in Γ and Γ'; check if the resulting subtype context (after applying the unifier) is satisfiable. We can have possibly non-matching subtype assertions after applying the unifier substitution to the subtype context but we check satisfiability over subtype contexts (with atomic assertions). So, before calling the algorithm that test the satisfiability of the subtype context, we compute the most general matching substitution (preserving the type variables of Γ and τ) and make it act over the sets of assertions, decomposing it in atomic subtype assertions.

Notation 8.5.53 *We use the following notation:*

$$\begin{array}{rcl} \mathsf{FTV}(\Gamma) & = & \bigcup_{(x:\sigma)\in\Gamma} \mathsf{FTV}(\sigma) \\ \mathsf{FDV}(\Theta) & = & \bigcup_{\mathsf{d}\sqsubseteq\mathsf{d}'\in\Theta} \mathsf{FDV}(\mathsf{d}) \cup \mathsf{FDV}(\mathsf{d}') \end{array}$$

Definition 8.5.54 *The algorithms* derivable *and* satisfiable *are defined in figures 8.15 and 8.16, respectively.*

$$\begin{array}{l}
\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau) = \\
\qquad \mathsf{let}\ (\Gamma', \tau', \Delta', \Theta') = \mathsf{typeJudg}(e) \\
\qquad \quad\ C = \mathsf{FTV}(\Gamma) \cup \mathsf{FTV}(\tau) \\
\qquad \mathsf{in}\ \ \mathsf{if}\ \mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma) \\
\qquad \quad\ \mathsf{then\ let}\ U = \mathsf{unify}\big(\{\sigma = \sigma' \mid x : \sigma \in \Gamma \wedge x : \sigma' \in \Gamma'\} \cup \{\tau = \tau'\}, C\big) \\
\qquad \qquad \qquad M = \mathsf{match}\big(U(\Delta'), C\big) \\
\qquad \qquad \qquad (\Delta, \Theta) = M \bullet \big(U(\Delta'), U(\Theta')\big) \\
\qquad \qquad \quad \mathsf{in}\ \ \mathsf{satisfiable}(\Delta, \Theta, C) \\
\qquad \quad\ \mathsf{else}\ \ \mathit{false}
\end{array}$$

assuming the type variables introduced in $(\Gamma', \tau', \Delta', \Theta')$ *are fresh.*

Figure 8.15: The algorithm derivable

The satisfiable algorithm performs the satisfiability check for subtype contexts. Let us prove that satisfiable is a sound and complete algorithm.

Lemma 8.5.55 *Given a set* Θ *of subtype inequalities,* dSatisfiable(Θ) *returns true iff there exists a ground datatype substitution* $G_{\mathcal{D}}$ *for* FDV(Θ) *such that for every* $\mathsf{d} \sqsubseteq \mathsf{d}' \in \Theta$, $G_{\mathcal{D}}(\mathsf{d}) \sqsubseteq_{\mathcal{D}} G_{\mathcal{D}}(\mathsf{d}')$.

Proof. Trivial. Just note that $\mathcal{D}^{\mathsf{FDV}(\Theta)}$ is the finite set of all possible ground datatype substitutions for the datatype variables in Θ. □

$$
\begin{array}{l}
\mathsf{satisfiable}(\Delta, \Theta, V) = \mathsf{tSatisfiable}(\Delta, V) \ \wedge \ \mathsf{dSatisfiable}(\Theta) \\[1em]
\mathsf{tSatisfiable}(\Delta, V) = \mathsf{unify}\big(\{\sigma = \sigma' \mid \sigma \leq \sigma' \in \Delta\}, V\big) \neq \mathit{fail} \\[1em]
\mathsf{dSatisfiable}(\Theta) = \mathsf{testSatisf}\big(\Theta, \mathcal{D}^{\mathsf{FDV}(\Theta)}\big) \\[1em]
\mathsf{testSatisf}(\{\}, S) = \mathit{true} \\
\mathsf{testSatisf}(\Theta, S \cup \{\rho\}) = \mathsf{if} \ \ \mathsf{solve}(\rho, \Theta) \ \ \mathsf{then} \ \ \mathit{true} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else} \ \ \mathsf{testSatisf}(\Theta, S) \\
\mathsf{testSatisf}(\Theta, \{\}) = \mathit{false} \\[1em]
\mathsf{solve}(\rho, \{\}) = \mathit{true} \\
\mathsf{solve}(\rho, \Theta \cup \{\mathtt{d} \sqsubseteq \mathtt{d}'\}) = \mathsf{if} \ \ \rho(\mathtt{d}) \sqsubseteq_{\mathcal{D}} \rho(\mathtt{d}') \ \ \mathsf{then} \ \ \mathsf{solve}(\rho, \Theta) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else} \ \ \mathit{false}
\end{array}
$$

Figure 8.16: The algorithm satisfiable

Lemma 8.5.56 *Given a subtype context (Δ, Θ) and a set V of type variables,* $\mathsf{satisfiable}(\Delta, \Theta, V)$ *returns true iff there exists a substitution R such that R preserves V and $\vdash (R(\Delta), R(\Theta))$.*

Proof. Let (Δ, Θ) be a subtype context and V a set of type variables.

$\Rightarrow$) If $\mathsf{satisfiable}(\Delta, \Theta, V)$ returns *true*, then both $\mathsf{tSatisfiable}(\Delta, V)$ and $\mathsf{dSatisfiable}(\Theta)$ return *true*. Then $\mathsf{unify}\big(\{\sigma = \sigma' \mid \sigma \leq \sigma' \in \Delta\}, V\big)$ does not fail, let $S = (S_{\mathcal{T}}, S_{\mathcal{D}})$ be the resulting substitution. As Δ is atomic we can conclude that $S_{\mathcal{D}} = id$. Furthermore, $\emptyset \vdash S(\alpha) \leq S(\alpha')$ for every $\alpha \leq \alpha' \in \Delta$. On the other hand, by Lemma 8.5.55, there exists a ground datatype substitution for $\mathsf{FDV}(\Theta)$, $G_{\mathcal{D}}$, such that $\emptyset \vdash G_{\mathcal{D}}(\mathtt{d}) \sqsubseteq G_{\mathcal{D}}(\mathtt{d}')$ for every $\mathtt{d} \sqsubseteq \mathtt{d}' \in \Theta$. Hence, $R = (S_{\mathcal{T}}, G_{\mathcal{D}})$ preserves V and $\vdash (R(\Delta), R(\Theta))$.

$\Leftarrow$) Assume R is a substitution such that R preserves V and $\vdash (R(\Delta), R(\Theta))$. Using Lemma 8.5.50, we have that for every $\mathtt{d} \sqsubseteq \mathtt{d}' \in \Theta$, $R(\mathtt{d}) = R(\mathtt{d}')$ or $R(\mathtt{d}) \sqsubseteq_{\mathcal{D}} R(\mathtt{d}')$. So, by Lemma 8.5.55, it easy to see that $\mathsf{dSatisfiable}(\Theta)$ returns *true*. Now we want to show that $\mathsf{tSatisfiable}(\Delta, V)$ returns *true*. This amounts to showing that $\mathsf{unify}\big(\{\sigma = \sigma' | \sigma \leq \sigma' \in \Delta\}, V\big)$ does not fail. As Δ is atomic, this unification can only fail if there is a $\alpha \leq \alpha' \in \Delta$ such that $\alpha \neq \alpha'$ and $\alpha, \alpha' \in V$. However, this can never happen because R preserves V and for every $\alpha, \alpha' \in \Delta$, $(\emptyset, \emptyset) \vdash R(\alpha) \leq R(\alpha')$. So, $\mathsf{tSatisfiable}(\Delta, V)$ succeeds. Hence $\mathsf{satisfiable}(\Delta, \Theta, V)$ returns *true*.

$\square$

Theorem 8.5.57 (Soundness of derivable) *If* $\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau)$ *returns true, then* $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ *is derivable.*

Proof. Let $\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau)$ as presented in Figure 8.15. If $\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau) = \mathit{true}$ then $\mathsf{satisfiable}(\Delta, \Theta, C) = \mathit{true}$, so by Lemma 8.5.56 there exists a substitution R such that R does not change the variables in C and $\vdash (R(\Delta), R(\Theta))$.

Let $S = R \circ M \circ U$. $(\emptyset, \emptyset) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau$ is an instance of $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau'$ by S since:

– $\vdash S \bullet (\Delta', \Theta')$ because $\vdash (R(\Delta), R(\Theta))$ and, by Lemma 8.5.27, $\vdash R \bullet (\Delta, \Theta)$. Moreover, by Lemma 8.5.46, $S \bullet (\Delta', \Theta') = R \bullet (\Delta, \Theta)$.

– $S(\Gamma') \subseteq \Gamma$ because $U(\Gamma') \subseteq \Gamma$ and $U(\Gamma') = R(M(U(\Gamma')))$ since neither M nor R changes the none of the free type variables of Γ.

– $\tau = U(\tau') = S(\tau')$ because neither M nor R changes the none of the free type variables of τ.

So, by Theorem 8.5.43, $(\emptyset, \emptyset) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau$. Hence, by Lemma 8.5.52, $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$. □

Theorem 8.5.58 (Completness of derivable) *If $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ is derivable, then* $\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau)$ *returns true.*

Proof. Assume $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ is derivable and let $\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau)$ as presented in Figure 8.15. By Lemma 8.5.52, if $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ then $(\emptyset, \emptyset) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau$. Therefore, by Theorem 8.5.47, $(\emptyset, \emptyset) \,|\, \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau$ is an instance of $(\Delta', \Theta') \,|\, \Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \tau'$ with $\mathsf{typeJudg}(e) = (\Gamma', \tau', \Delta', \Theta')$, i.e., there exists a matching substitution S such that $\vdash S \bullet (\Delta', \Theta')$, $S(\Gamma') \subseteq \Gamma$ and $S(\tau') = \tau$.

Since $S(\Gamma') \subseteq \Gamma$ and $S(\tau') = \tau$, S must unify $\{\sigma = \sigma' \mid x : \sigma \in \Gamma \wedge x : \sigma' \in \Gamma'\} \cup \{\tau = \tau'\}$ without changing $C = \mathsf{FTV}(\Gamma) \cup \mathsf{FTV}(\tau)$, then $S = Z \circ U$ for some substitution Z. Moreover, as S is a matching substitution for Δ', Z must be a matching substitution for $U(\Delta')$ and therefore $Z = W \circ M$ for some W because M is the most general matching substitution for $U(\Delta')$.

So, one has $S = W \circ M \circ U$ and, by Lemma 8.5.46, $S \bullet (\Delta', \Theta') = W \bullet ((M \circ U) \bullet (\Delta', \Theta')) = W \bullet (\Delta, \Theta)$. Hence $\vdash W \bullet (\Delta, \Theta)$ and, by Lemma 8.5.56, $\mathsf{satisfiable}(\Delta, \Theta, C) = \mathit{true}$. Therefore, $\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau) = \mathit{true}$. □

Corollary 8.5.59 (Decidability of type checking in $\lambda^{\mathsf{a}}_{\mathsf{CS}}$) *For any context Γ, and for any term e and type τ, it is decidable whether $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS}}} e : \tau$ is derivable.*

Proof. Immediate from theorems 8.5.57 and 8.5.58. □

Decidability of type-checking in λ_{CS} follows.

Corollary 8.5.60 (Decidability of type checking in λ_{CS}) *For any context Γ, and for any term e and type τ, it is decidable whether $\Gamma \vdash_{\lambda_{\mathsf{CS}}} e : \tau$ is derivable.*

Proof. It is an immediate consequence of Corollary 8.5.59, Lemma 8.5.12 and of the fact that the set $\mathsf{an}(e)$ is finite. □

Chapter 9

Extensible Overloaded Functions

Constructor subtyping improves the accuracy and the flexibility of type systems by coherently combining the subtyping between datatypes and the overloading of constructors, and it is adequate for extensible datatypes. This flexibility in the definition of datatypes should make functions on (extensible) datatypes more flexible. In this chapter we define a mechanism that allows functions to be overloaded (i.e. to have several types) and extensible (i.e. to be extensible from a datatype to another datatype with more constructors). The resulting framework provides a formal foundation for extensible datatypes with overloading of constructors and overloaded extensible recursive functions.

9.1 The System $\lambda_{\mathsf{CS+fun}}$

This section introduces the system $\lambda_{\mathsf{CS+fun}}$ which can be thought of as a simply typed λ-calculus with constructor subtyping and extensible recursive definitions defined by pattern-matching. The language of types and the subtyping relation used in $\lambda_{\mathsf{CS+fun}}$ are as defined for λ_{CS}. For the sake of simplicity, we do not have case-expressions and letrec-expressions in $\lambda_{\mathsf{CS+fun}}$. So, the term language of $\lambda_{\mathsf{CS+fun}}$ is simpler than the one of λ_{CS}. However, the calculus features overloaded constants and overloaded variables. The former are interpreted as constructors (without a computational meaning) and the latter are interpreted as functions (with a computational meaning).

9.1.1 Types and Terms

We assume the setting given for λ_{CS}. Hence we have $\mathcal{D}, \mathcal{C}, \mathcal{V}_{\mathcal{E}}, \mathcal{V}_{\mathcal{T}}$ and $\sqsubseteq_{\mathcal{D}}$ as in λ_{CS} and all constructors are strictly overloaded. Moreover, we assume a denumerable set $\mathcal{F}$ of *function symbols* and we let $f, g, \ldots$ range over $\mathcal{F}$. We assume that the sets $\mathcal{C}, \mathcal{V}_{\mathcal{E}}$ and $\mathcal{F}$ are pairwise disjoint. The set $\mathcal{T}_{\mathsf{fun}}$ of types of $\lambda_{\mathsf{CS+fun}}$ is equal to $\mathcal{T}_{\mathsf{cs}}$.

Definition 9.1.1 (Types) *The set $\mathcal{T}_{\mathsf{fun}}$ of* types *is given by the abstract syntax:*

$$\mathcal{T}_{\mathsf{fun}} \ni \tau, \sigma \quad ::= \quad \alpha \mid \tau \rightarrow \sigma \mid d\,\vec{\tau}$$

where in the last clause, it is assumed that $\#\vec{\tau} = \mathsf{ar}(d)$.

Terms are built up from variables, function symbols, constructors, abstractions and applications.

(refl) $$\frac{}{\sigma \leq \sigma}$$ (trans) $$\frac{\sigma \leq \sigma' \quad \sigma' \leq \sigma''}{\sigma \leq \sigma''}$$

(func) $$\frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$ (data) $$\frac{d \sqsubseteq_{\mathcal{D}} d' \quad \tau_i \leq \tau_i' \quad (1 \leq i \leq \mathsf{ar}(d))}{d\,\vec{\tau} \leq d'\,\vec{\tau'}}$$

Figure 9.1: Subtyping rules for $\lambda_{\mathsf{CS+fun}}$

Definition 9.1.2 (Terms) *The set $\mathcal{E}_{\mathsf{fun}}$ of terms is given by the abstract syntax:*

$$\mathcal{E}_{\mathsf{fun}} \ni a, b \quad ::= \quad x \mid f \mid c \mid \lambda x.\, a \mid a\; b$$

At this stage, the difference between constructors and function symbols is rather shallow. In the sequel, it should became clear that while constructors have no computational meaning, function symbols have a computational meaning given by environments of function definitions.

Note that as $\mathcal{F}$ and $\mathcal{V}_{\mathcal{E}}$ are disjoint sets, there is no interference of function symbols in the notions of free variables and substitution already defined for λ_{CS} that can thus remain unchanged (of course, $\mathsf{FV}(f) = \emptyset$).

9.1.2 Subtyping and Typing

As already mentioned, the subtyping relation of $\lambda_{\mathsf{CS+fun}}$ is the same defined for λ_{CS}. Let us recall it.

Definition 9.1.3 (Subtyping) *The subtyping relation $\leq$ over $\mathcal{T}_{\mathsf{fun}}$ is defined inductively by the rules of Figure 9.1.*

We now turn to the typing system. Typing judgments of $\lambda_{\mathsf{CS+fun}}$ are of the form $\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau$ where Γ, e and τ are as usual (respectively, a context of variable declarations, a term and a type) and ∇ is a context for declaring types to function symbols. More precisely we have the following definitions.

Definition 9.1.4 (Context, judgment)

1. *A* variable context *is a finite set $x_1 : \tau_1, \ldots, x_n : \tau_n$ where $x_1, \ldots, x_n$ are pairwise disjoint object variables and $\tau_1, \ldots, \tau_n$ are types. We let $\aleph_{\mathcal{V}}$ denote the set of variable contexts and let Γ, Γ' range over variable contexts.*

2. *A* function context *is a finite set $f_1 : \tau_1, \ldots, f_n : \tau_n$ where $f_1, \ldots, f_n$ are function symbols and $\tau_1, \ldots, \tau_n$ are types. We let $\aleph_{\mathcal{F}}$ denote the set of function contexts and let ∇, ∇' range over function contexts.*

3. *A* context *is a pair $\nabla \mid \Gamma$ where ∇ is a function context and Γ is a variable context. We let $\aleph$ denote the set of contexts.*

4. *A* typing judgment *is a quadruple of the form $\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau$ where ∇ is a function context, Γ is a variable context, e is a term and τ is a type.*

$$\text{(var)} \quad \frac{}{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} x : \tau} \quad \text{if } (x : \tau) \in \Gamma$$

$$\text{(abs)} \quad \frac{\nabla \mid \Gamma, x : \tau \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma}{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} \lambda x.\, e : \tau \to \sigma}$$

$$\text{(app)} \quad \frac{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau \to \sigma \qquad \nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e' : \tau}{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e\, e' : \sigma}$$

$$\text{(cons)} \quad \frac{}{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c : \mathsf{Inst}_d^{\vec{\tau}}(c)} \quad \text{if } c \in \mathsf{C}(d)$$

$$\text{(fun)} \quad \frac{}{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} f : S(\tau)} \quad \text{if } (f : \tau) \in \nabla \text{ and } S \text{ is a substitution}$$

$$\text{(sub)} \quad \frac{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau \quad \tau \leq \sigma}{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma}$$

Figure 9.2: Typing rules for $\lambda_{\mathsf{CS+fun}}$

Notice that whereas in variable contexts variables occur at most once, in function contexts the same function symbol can be declared to have different types, so that overloaded extensible functions can be modeled.

Definition 9.1.5 (Typing)

1. *A typing judgment is* derivable *if it can be inferred from the rules of Figure 9.2.*
2. *A term $e \in \mathcal{E}_{\mathsf{fun}}$ is* typable *if $\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ is derivable for some context $\nabla \mid \Gamma$ and type σ.*

Note that a function may be assigned polymorphic types, i.e., types that may contain free variables and that may be instantiated through some type substitution. Rule (fun) states that each legal typing for a function is obtained by instantiating some type assigned to the function in the function context. Rules (var), (abs), (app), (cons) and (sub) are directly adapted from λ_{CS}.

9.1.3 Definition of Functions

Functions are declared in a pattern matching style. A function is defined by a set of its possible types and a set of rewriting rules (modeled by pattern abstractions). For the sake of clarity, we restrict our attention to unary functions. This is not a real restriction, since functions of arbitrary arity may be encoded with unit and product types, and these types are easily defined in our system.

Example 9.1.6 (Product and Unit) *We define a parametric datatype for products as follows:*

$$\begin{array}{ll} \mathsf{Prod} \in \mathcal{D} & \mathsf{pair} \in \mathcal{C} \\ \mathsf{ar}(\mathsf{Prod}) = 2 & \mathsf{ar}(\mathsf{pair}) = 2 \\ \mathsf{C}(\mathsf{Prod}) = \{\mathsf{pair}\} & \mathsf{D}_{\mathsf{Prod}}(\mathsf{pair}) = \forall\, \alpha_1. \forall\, \alpha_2.\ \alpha_1 \to \alpha_2 \to \mathsf{Prod}\, \alpha_1 \alpha_2 \end{array}$$

We will use the usual infix notation $\mathsf{A} \times \mathsf{B}$ *to represent* $\mathsf{Prod}\,\mathsf{A}\,\mathsf{B}$*, and the out-fix notation* $\langle a, b\rangle$ *to represent* $(\mathsf{pair}\,a\,b)$*.*

The unit type is simply define by:

$$\begin{array}{ll} \mathsf{Unit} \in \mathcal{D} & \mathsf{unit} \in \mathcal{C} \\ \mathsf{ar}(\mathsf{Unit}) = 0 & \mathsf{ar}(\mathsf{unit}) = 0 \\ \mathsf{C}(\mathsf{Unit}) = \{\mathsf{unit}\} & \mathsf{D}_{\mathsf{Unit}}(\mathsf{unit}) = \mathsf{Unit} \end{array}$$

First, we define patterns and pattern abstractions, which act as rewrite rules. Second, we define environments as sets of function declarations. Patterns are expressions built up from constructors and variables; they are required to be *linear*, i.e. a variable can appear at most once in a pattern.

Definition 9.1.7 (Patterns)

1. *The set* $\mathcal{P}$ *of* patterns *is the set of terms given by the abstract syntax*

$$\mathcal{P} \ni p \ ::= \ x \mid c\,p_1 \ldots p_{\mathsf{ar}(c)}$$

   where in the last clause it is assumed that, for $1 \leq i \leq \mathsf{ar}(c)$*, the sets* $\mathsf{FV}(p_i)$ *are pairwise disjoint.*

2. *A* pattern abstraction *is an expression of the form* $\lambda p.\, e$ *where* $p \in \mathcal{P}$ *and* $e \in \mathcal{E}_{\mathsf{fun}}$*. We let* $\mathcal{R}$ *denote the set of pattern abstractions and* r, r', r_i *range over* $\mathcal{R}$*.*

Environments are sets of function definitions, which may be mutually recursive.

Definition 9.1.8 (Environments)

1. *A* function definition *is a triple* $f : \vec{\sigma} = \vec{r}$ *where* $f \in \mathcal{F}$ *is a function symbol,* $\vec{\sigma}$ *is a set of function types and* $\vec{r}$ *is a set of pattern-abstractions.*

2. *An* environment *is a finite set of function definitions. We let* $\mathcal{H}$ *denote the set of environments and* Σ, Σ' *range over* $\mathcal{H}$*.*

3. *Given an environment* Σ *and a function symbol* f*, we define the sets* FS_Σ*,* $\mathsf{Ty}_\Sigma(f)$ *and* $\mathsf{Ru}_\Sigma(f)$*, of function symbols in* Σ*,* Σ*-typings for* f*, and* Σ*-rewritings for* f*, respectively, as follows:*

$$\begin{array}{rcl} \mathsf{FS}_\Sigma & = & \{f \mid (f : \vec{\sigma} = \vec{r}) \in \Sigma\} \\ \mathsf{Ty}_\Sigma(f) & = & \{\sigma_i \mid (f : \vec{\sigma} = \vec{r}) \in \Sigma \wedge \sigma_i \in \vec{\sigma}\} \\ \mathsf{Ru}_\Sigma(f) & = & \{r_i \mid (f : \vec{\sigma} = \vec{r}) \in \Sigma \wedge r_i \in \vec{r}\} \end{array}$$

In the definition of a function f, whereas its typings are used in the typing of expressions, the rewritings for f influence the reduction of expressions. Thus, each environment induces a function context and a notion of reduction. The function context associated with an environment is obtained as follows.

Definition 9.1.9 (Erasure) *The erasure function* $\mathsf{er} : \mathcal{H} \to \aleph_{\mathcal{F}}$ *is defined inductively as follows:*

$$\begin{array}{rcl} \mathsf{er}(\emptyset) & = & \emptyset \\ \mathsf{er}(\Sigma, f : \{\sigma_1, \ldots, \sigma_n\} = \vec{r}) & = & \mathsf{er}(\Sigma), f : \sigma_1, \ldots, f : \sigma_n \end{array}$$

Let us illustrate the mechanism for defining functions, with the double function for natural numbers.

Example 9.1.10 (double) *Recall the definition of datatypes* $\mathsf{Odd}, \mathsf{Even}$ *and* Nat *given in Example 8.1.18. The* double *function for natural numbers can be defined as in the following environment:*

$$\begin{array}{rcl} \text{double} & : & \{\mathsf{Nat} \to \mathsf{Nat}\} \\ & = & \{\lambda \mathsf{o}.\, \mathsf{o}, \quad \lambda \mathsf{s}\, x.\, \mathsf{s}\,(\mathsf{s}\,(\text{double}\, x))\} \end{array}$$

We can expand this environment with another declaration, defining double *also for* Odd *and* Even*, as follows:*

$$\begin{array}{rcl} \text{double} & : & \{\mathsf{Even} \to \mathsf{Even},\ \mathsf{Odd} \to \mathsf{Even}\} \\ & = & \{\} \end{array}$$

Note that it is not necessary to define any extra equation (pattern abstraction), as we can reuse the equations already defined for Nat*.*

An example of a polymorphic overloaded function is the zipwith.

Example 9.1.11 (zipwith) *The* zipwith *function takes a binary function g and two lists, l_1 and l_2, whose element types match the domain of the function, and builds up the list with the results of successively applying g to the elements coming from l_1 and l_2. We can define an overloaded version of* zipwith *as follows:*

$$\begin{array}{rcl} \text{zipwith} & : & \{\alpha_1 \to \alpha_2 \to \alpha_3 \times (\mathsf{List}\,\alpha_1 \times \mathsf{List}\,\alpha_2) \to \mathsf{List}\,\alpha_3, \\ & & \ \alpha_1 \to \alpha_2 \to \alpha_3 \times (\mathsf{NeList}\,\alpha_1 \times \mathsf{NeList}\,\alpha_2) \to \mathsf{NeList}\,\alpha_3\} \\ & = & \{\lambda\langle g, \langle \mathsf{nil}, x\rangle\rangle.\mathsf{nil}, \quad \lambda\langle g, \langle x, \mathsf{nil}\rangle\rangle.\, \mathsf{nil}, \\ & & \ \lambda\langle g, \langle \mathsf{cons}\, x\, x', \mathsf{cons}\, y\, y'\rangle\rangle.\, \mathsf{cons}\,(g\, x\, y)\,(\text{zipwith}\,\langle g, \langle x', y'\rangle\rangle)\} \end{array}$$

9.1.4 Reduction Calculus

Pattern abstractions are to be used as rewrite rules and must be taken into account by the reduction calculus: pattern-reduction $\to_{\beta_{\text{pat}}}$ is defined as the compatible closure of the rule

$$(\lambda p.\, e)\, a \quad \to_{\beta_{\text{pat}}} \quad S(e)$$

where if existing, S is the unique substitution such that $S(p) = a$ and $\mathsf{Supp}(S) \subseteq \mathsf{FV}(p)$.

Example 9.1.12 *Consider the datatypes of natural numbers and products defined before. We have:*

$$\begin{array}{rcl} (\lambda \mathsf{s}\, x.\, f\, x)\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{o})) & \to_{\beta_{\text{pat}}} & f\,(\mathsf{s}\,\mathsf{o}) \\ (\lambda \mathsf{s}\, x.\, \lambda y.\, \mathsf{s}\,(f\, x\, y))\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{o})) & \to_{\beta_{\text{pat}}} & \lambda y.\, \mathsf{s}\,(f\,(\mathsf{s}\,\mathsf{o})\, y) \\ (\lambda\langle \mathsf{s}\, x, y\rangle.\, \mathsf{s}\,(f\,\langle x, y\rangle))\,\langle \mathsf{s}\,(\mathsf{s}\,\mathsf{o}), \mathsf{s}\, n\rangle & \to_{\beta_{\text{pat}}} & \mathsf{s}\,(f\,\langle \mathsf{s}\,\mathsf{o}, \mathsf{s}\, n\rangle) \end{array}$$

In some cases, patterns do not match and reduction is not possible, as for example in

$$(\lambda\langle \mathsf{s}x, y\rangle.\, \mathsf{s}(f\,\langle x, y\rangle))\,\langle \mathsf{o}, \mathsf{s}\, n\rangle$$

Thus, each environment determines a notion of reduction as follows.

Definition 9.1.13 (Reduction Calculus)

1. *β-reduction $\to_\beta$ is defined as the compatible closure of the rule*

$$(\lambda x.\, a)\; b \;\; \to_\beta \;\; a[x := b]$$

2. *Let $f : \vec{\sigma} = \vec{r}$ be a function definition. $\delta(f : \vec{\sigma} = \vec{r})$-reduction $\rightarrow_{\delta(f:\vec{\sigma}=\vec{r})}$ is defined as the compatible closure of the rule*
$$f\,a \quad \rightarrow_{\delta(f:\vec{\sigma}=\vec{r})} \quad S(e)$$
where is assumed that $\lambda p.\, e \in \vec{r}$ and S is the unique (if it exists) substitution such that $S(p) = a$ and $\mathsf{Supp}(S) \subseteq \mathsf{FV}(p)$.

3. *Let $\Sigma \equiv f_1 : \vec{\sigma_1} = \vec{r_1}, \ldots, f_n : \vec{\sigma_n} = \vec{r_n}$ be an environment. δ_Σ-reduction $\rightarrow_{\delta_\Sigma}$ is defined as*
$$\bigcup_{1 \leq i \leq n} \rightarrow_{\delta(f_i:\vec{\sigma_i}=\vec{r_i})}$$

4. *Let Σ be an environment. $\beta\delta_\Sigma$-reduction $\rightarrow_{\beta\delta_\Sigma}$ is defined as $\rightarrow_\beta \cup \rightarrow_{\delta_\Sigma}$. $\twoheadrightarrow_{\beta\delta_\Sigma}$ and $=_{\beta\delta_\Sigma}$ are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of $\rightarrow_{\beta\delta_\Sigma}$.*

The mechanics of this reduction calculus is illustrated by the following example.

Example 9.1.14 *Consider the function* double *defined in Example 9.1.10. The next reduction sequence computes the double of the double of one.*

$$\begin{array}{rll}
\text{double}\,(\text{double}\,(\mathsf{s}\,\mathsf{o})) & \rightarrow_{\delta_\Sigma} & \text{double}\,(\mathsf{s}\,(\mathsf{s}\,(\text{double}\,\mathsf{o}))) \\
& \rightarrow_{\delta_\Sigma} & \text{double}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{o})) \\
& \rightarrow_{\delta_\Sigma} & \mathsf{s}\,(\mathsf{s}\,(\text{double}\,(\mathsf{s}\,\mathsf{o}))) \\
& \rightarrow_{\delta_\Sigma} & \mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,(\text{double}\,\mathsf{o})))) \\
& \rightarrow_{\delta_\Sigma} & \mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{o})))
\end{array}$$

Observe that when we apply double *to* (double $(\mathsf{s}\,\mathsf{o})$) *we first evaluate* (double $(\mathsf{s}\,\mathsf{o})$) *to find out what sort of term it is, because* (double $(\mathsf{s}\,\mathsf{o})$) *does not match with any of the patterns in the Σ-rewritings of* double.

The computational behavior of expressions is specified by the rules for β-reduction and δ_Σ reduction. Of course, the δ_Σ reduction needs not be well-behaved, since we have not imposed any restrictions on environments. For example, assume we have defined the function double as it is done in Example 9.1.10 but without declaring the type $\mathsf{Even} \rightarrow \mathsf{Even}$ for it. In this situation, we have
$$\text{double} : \mathsf{Nat} \rightarrow \mathsf{Nat}, \text{double} : \mathsf{Odd} \rightarrow \mathsf{Even}\,|\,. \vdash_{\lambda_{\mathsf{CS+fun}}} \text{double}\,(\mathsf{s}\,\mathsf{o}) : \mathsf{Even}$$
and the reduction double $(\mathsf{s}\,\mathsf{o}) \rightarrow_{\delta_\Sigma} \mathsf{s}\,(\mathsf{s}\,(\text{double}\,\mathsf{o}))$. However, the judgment double : $\mathsf{Nat} \rightarrow \mathsf{Nat}$, double : $\mathsf{Odd} \rightarrow \mathsf{Even}\,|\,. \vdash_{\lambda_{\mathsf{CS+fun}}} \mathsf{s}\,(\mathsf{s}\,(\text{double}\,\mathsf{o})) : \mathsf{Even}$ does not hold since one has to use double : $\mathsf{Nat} \rightarrow \mathsf{Nat}$ to type (double o). Thus, we just have

$$\text{double} : \mathsf{Nat} \rightarrow \mathsf{Nat}, \text{double} : \mathsf{Odd} \rightarrow \mathsf{Even}\,|\,. \vdash_{\lambda_{\mathsf{CS+fun}}} \mathsf{s}\,(\mathsf{s}\,(\text{double}\,\mathsf{o})) : \mathsf{Nat}$$

So, if no restrictions on environments are imposed, the type of expressions will not be preserved under reduction, and we will have problems with subject reduction. The next section is devoted to the definition of well-formed environments.

9.1.5 Well-Formed Environments

There are several possible understandings of well-formedness for an environment. A minimal requirement is that δ_Σ-reduction should enjoy subject reduction. Other desirable requirements on recursive functions include unambiguousness, totality and possibly termination. Next we formalize these requirements as properties on environments. The decidability of such properties will be proved in Section 9.6.

As already mentioned, we are eventually interested in extending this mechanism, of extensible overloaded functions, to dependent types. So, we do not want the computational behavior of recursive functions to depend on typing, since it would create a circularity (in dependent type systems, typing depends on reduction through the conversion rule).

To achieve the subject reduction property for the reduction calculus, each reduction rule induced by the environment must be well-typed. The following definition formalizes this requirement.

Definition 9.1.15 (Well-typed environment) *An environment* Σ *is* well-typed *if, for every* $f \in \mathsf{FS}_\Sigma$ *and* $\tau_1 \to \tau_2 \in \mathsf{Ty}_\Sigma(f)$*, the following two conditions hold:*

1. $\exists\, \lambda p.e \in \mathsf{Ru}_\Sigma(f).\ \exists \Gamma.\ \ . \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} p : \tau_1$
2. $\forall\, \lambda p.e \in \mathsf{Ru}_\Sigma(f).\ \forall \Gamma.\ \ \mathsf{er}(\Sigma) \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} p : \tau_1 \ \Rightarrow\ \mathsf{er}(\Sigma) \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau_2$

These two conditions force every type declared for a function to be in accordance with the pattern abstractions involved in the definition of that function. That is, if it is possible to type the argument of the function with a domain type of the function, then it must be possible to type the body of the function (for that argument) with the corresponding codomain type. Moreover, every type assigned to a function must type at least one of its pattern abstractions. That is the reason why we require condition 1 to hold.

Observe that the environment defining double in Example 9.1.10 is well-typed. But if we just declare the types $\mathsf{Nat} \to \mathsf{Nat}$ and $\mathsf{Odd} \to \mathsf{Even}$ for double, the resulting environment is not well-typed, because the second pattern abstraction does not type-check. Let us give another example.

Example 9.1.16 (add) *Recall the definition of datatypes* $\mathsf{Odd}, \mathsf{Even}$ *and* Nat *given in Example 8.1.18. Addition of even numbers can be defined as in the following environment:*

$$\begin{array}{rcl} \mathrm{add} & : & \{\mathsf{Even} \to \mathsf{Even} \to \mathsf{Even},\ \mathsf{Odd} \to \mathsf{Even} \to \mathsf{Odd}\} \\ & = & \{\lambda \mathsf{o}.\lambda y.\, y,\quad \lambda \mathsf{s}\, x.\lambda y.\, \mathsf{s}\,(\mathrm{add}\, x\, y)\} \end{array}$$

This environment is well-typed. Notice how it is crucial that add *is overloaded. If we just declare type* $\mathsf{Even} \to \mathsf{Even} \to \mathsf{Even}$ *for* add *the second equation (pattern abstraction) does not type-check. Moreover, to expand the definition of odd and natural numbers we just have to overload* add *with the new types, as the equations can be completely reused.*

$$\begin{array}{rcl} \mathrm{add} & : & \{\mathsf{Even} \to \mathsf{Odd} \to \mathsf{Odd},\ \mathsf{Odd} \to \mathsf{Odd} \to \mathsf{Even},\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\} \\ & = & \{\} \end{array}$$

Again, note that for typing reasons, the first and second typings need to be declared together whereas the third typing could be declared on its own.

The next example illustrates the necessity of condition 1 in the definition of well-typed environments.

Example 9.1.17 *Consider the following environment* Σ*:*

$$\begin{array}{rcl} \mathrm{wrong} & : & \{\mathsf{Nat} \to \mathsf{Nat},\ \mathsf{Odd} \to \mathsf{Even},\ \alpha \to \beta\} \\ & = & \{\lambda \mathsf{o}.\, \mathsf{o},\quad \lambda \mathsf{s}\, x.\, x\} \end{array}$$

This environment is not well-typed because condition 1 fails. There is no pattern (in the pattern abstractions defined for wrong*) typable with type α for some context, since both* o *and* $\mathsf{s}\,x$ *are impossible to type with type α. However, condition 2 holds for the two patterns abstractions defining* wrong*.*

Clearly, the declaration of this function threatens subject reduction. For instance, we have

$$\frac{\text{wrong} : \alpha \to \beta \,|\, . \vdash_{\lambda_{\mathsf{CS+fun}}} \text{wrong} : \mathsf{Nat} \to \mathsf{List\,Nat} \qquad \text{wrong} : \alpha \to \beta \,|\, . \vdash_{\lambda_{\mathsf{CS+fun}}} \mathsf{o} : \mathsf{Nat}}{\text{wrong} : \alpha \to \beta \,|\, . \vdash_{\lambda_{\mathsf{CS+fun}}} \text{wrong}\,\mathsf{o} : \mathsf{List\,Nat}}$$

and the reduction wrong $\mathsf{o} \to_{\delta_\Sigma} \mathsf{o}$*; but it is impossible to give* o *the type* $\mathsf{List\,Nat}$*. Therefore, subject reduction does not hold.*

The reduction calculus should be confluent, so we only consider unambiguous functions. Functions defined in the environments must be non-overlapping. In other words, only one rule of the function should apply for a given term.

Definition 9.1.18 (Non-overlapping environment) *An environment Σ is* non-overlapping *if, for every $f \in \mathsf{FS}_\Sigma$ and distinct $\lambda p.e$, $\lambda p'.e' \in \mathsf{Ru}_\Sigma(f)$ there are no substitutions S and S' such that $S(p) = S'(p')$.*

An alternative approach would have been to opt for priority rewriting [12] and drop the requirement that functions should be non-overlapping.

It is easy to check that the environments defined till now are non-overlapping. However, sometimes this requirement complicates the writing of programs. Let us give an example.

Example 9.1.19 (MaybeNat) *Suppose we want to extend the datatype* Nat *with an extra constructor as follows:*

```
data MaybeNat extends Nat = Undef : MaybeNat
```

Hence, we have:

$$\begin{array}{ll}
\mathsf{MaybeNat} \in \mathcal{D} & \mathsf{Nat} \sqsubseteq_{\mathcal{D}} \mathsf{MaybeNat} \\
\mathsf{o}, \mathsf{s}, \mathsf{undef} \in \mathcal{C} & \\
\mathsf{ar}(\mathsf{MaybeNat}) = 0 & \mathsf{D}_{\mathsf{MaybeNat}}(\mathsf{undef}) = \mathsf{MaybeNat} \\
\mathsf{ar}(\mathsf{undef}) = 0 & \mathsf{D}_{\mathsf{MaybeNat}}(\mathsf{o}) = \mathsf{MaybeNat} \\
\mathsf{C}(\mathsf{MaybeNat}) = \{\mathsf{o}, \mathsf{s}, \mathsf{undef}\} & \mathsf{D}_{\mathsf{MaybeNat}}(\mathsf{s}) = \mathsf{Nat} \to \mathsf{MaybeNat}
\end{array}$$

Suppose also that we want to extend the function add *so that it works on the extended datatype* $\mathsf{MaybeNat}$*. To define a function* add : $\mathsf{MaybeNat} \to \mathsf{MaybeNat} \to \mathsf{MaybeNat}$ *by pattern matching can be a bit tricky because we restrict the environments to unary functions and case-expressions are not allowed in this system. Nevertheless we can use the product type constructor to deal with this problem. So, assume instead we have an uncurried version of addition defined by the following environment:*

$$\begin{array}{rcl}
\text{add} & : & \{\mathsf{Even} \times \mathsf{Even} \to \mathsf{Even},\ \mathsf{Odd} \times \mathsf{Even} \to \mathsf{Odd}, \\
& & \ \mathsf{Even} \times \mathsf{Odd} \to \mathsf{Odd},\ \mathsf{Odd} \times \mathsf{Odd} \to \mathsf{Even}, \\
& & \ \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}\} \\
& = & \{\lambda\langle \mathsf{o}, y\rangle.\, y,\quad \lambda\langle \mathsf{s}\,x, y\rangle.\, \mathsf{s}\,(\text{add}\,\langle x, y\rangle)\}
\end{array}$$

The first idea to expand addition for $\mathsf{MaybeNat}$ *is to extend the environment with the following declaration:*

$$\begin{array}{rcl}
\text{add} & : & \{\mathsf{MaybeNat} \times \mathsf{MaybeNat} \to \mathsf{MaybeNat}\} \\
& = & \{\lambda\langle \mathsf{undef}, y\rangle.\mathsf{undef},\quad \lambda\langle x, \mathsf{undef}\rangle.\mathsf{undef}\}
\end{array}$$

However, the resulting environment is overlapping. In fact, the last clause, $\lambda\langle x, \mathsf{undef}\rangle.\mathsf{undef}$ *overlaps with all the others. Hence the non-overlapping conditions fails. A correct version of addition for* $\mathsf{MaybeNat}$ *is the function* plus*:*

$$\begin{array}{lcl}
\text{plus} & : & \{\mathsf{Even} \times \mathsf{Even} \to \mathsf{Even},\ \mathsf{Odd} \times \mathsf{Even} \to \mathsf{Odd}\} \\
 & = & \{\lambda\langle \mathsf{o}, \mathsf{o}\rangle.\mathsf{o},\ \lambda\langle \mathsf{o}, \mathsf{s}\, x\rangle.\mathsf{s}\, x,\ \lambda\langle \mathsf{s}\, x, \mathsf{o}\rangle.\mathsf{s}\, x,\ \lambda\langle \mathsf{s}\, x, \mathsf{s}\, y\rangle.\mathsf{s}\,(\mathsf{s}\,(\text{plus}\,\langle x, y\rangle))\} \\
\text{plus} & : & \{\mathsf{Even} \times \mathsf{Odd} \to \mathsf{Odd},\ \mathsf{Odd} \times \mathsf{Odd} \to \mathsf{Even},\ \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}\} \\
 & = & \{\} \\
\text{plus} & : & \{\mathsf{MaybeNat} \times \mathsf{MaybeNat} \to \mathsf{MaybeNat}\} \\
 & = & \{\lambda\langle \mathsf{undef}, y\rangle.\mathsf{undef},\ \lambda\langle \mathsf{o}, \mathsf{undef}\rangle.\mathsf{undef},\ \lambda\langle \mathsf{s}\, x, \mathsf{undef}\rangle.\mathsf{undef}\}
\end{array}$$

Note that we are still using the facility of extending a previously defined function, but the definition of plus *over naturals is not very usual.*

We only consider total functions, so the functions defined by pattern matching must be exhaustive. Because functions can be polymorphic, to formalize this requirement of exhaustiveness we first introduce the notion of *quasi-closed pattern* w.r.t. a given type.

Definition 9.1.20 (Quasi-closed pattern) *A pattern p is* quasi-closed *w.r.t. σ if $\sigma \equiv d\,\vec{\tau}$, $p \equiv c\,p_1 \ldots p_{\mathsf{ar}(c)}$, $c \in \mathsf{C}(d)$ and each p_i is a quasi-closed w.r.t. $\mathsf{Dom}_d^{\vec{\tau}}(c)[i]$; or if $\sigma \not\equiv d\,\vec{\tau}$ and p is a variable.*

Obviously, if p is a pattern quasi-closed w.r.t. σ, then $. \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} p : \sigma$ for some Γ.

Definition 9.1.21 (Exhaustive environment) *An environment Σ is* exhaustive *if, for every $f \in \mathsf{FS}_\Sigma$, $\tau_1 \to \tau_2 \in \mathsf{Ty}_\Sigma(f)$ and for every pattern p quasi-closed w.r.t. τ_1, there exists $\lambda p'.e' \in \mathsf{Ru}_\Sigma(f)$ such that $p = S(p')$ for some substitution S.*

It is easy to check that the environments defined above are all exhaustive.

Summarizing, we can say that a function f is:

- well-typed, if all the rewrite rules defining it are well-typed;
- unambiguous, if for every pattern p, the expression $f\,p$ can be reduced in at most one way;
- total, if $f\,p$ is reducible for every quasi-closed pattern p of suitable type.

The well-formed environments will be those for which all functions defined in it satisfy these properties. The above properties are *global* in the sense that one needs to scan the whole environment to check whether they hold. As illustrated in the example bellow, it is crucial that the properties are global.

Example 9.1.22 *Consider the well-typed environment:*

$$\begin{array}{lcl}
\text{add} & : & \{\mathsf{Even} \to \mathsf{Even} \to \mathsf{Even},\ \mathsf{Odd} \to \mathsf{Even} \to \mathsf{Odd}\} \\
 & = & \{\lambda\mathsf{o}.\lambda y.\,\text{dummy}\, y,\ \lambda\mathsf{s}\, x.\lambda y.\,\mathsf{s}\,(\text{add}\, x\,(\text{dummy}\, y))\} \\
\text{dummy} & : & \{\mathsf{Even} \to \mathsf{Even}\} \\
 & = & \{\lambda x.x\}
\end{array}$$

If we extend this environment with the following declaration:

$$\begin{array}{lcl}
\text{add} & : & \{\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\} \\
 & = & \{\}
\end{array}$$

the resulting environment is not well-typed, whereas the original environment without this last definition is. This is because, for instance, we do not have

$$\nabla \,|\, . \vdash_{\lambda_{\mathsf{CS+fun}}} \lambda \mathsf{o}.\lambda y.\, \mathrm{dummy}\, y : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \quad , \ \textit{where}$$

$$\nabla \equiv \mathrm{add}\!:\!\mathsf{Even} \to \mathsf{Even} \to \mathsf{Even},\ \mathrm{add}\!:\!\mathsf{Odd} \to \mathsf{Even} \to \mathsf{Odd},\ \mathrm{dummy}\!:\!\mathsf{Even} \to \mathsf{Even},\ \mathrm{add}\!:\!\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}.$$

This example shows that when new types are assigned to a function, its equations need to be type-checked again.

We can now define the notion of well-formed environment.

Definition 9.1.23 (Well-formed environment) *An environment Σ is* well-formed *if Σ is well-typed, non-overlapping and exhaustive.*

Decidability of well-formedness for environments is proved in Section 9.6. The environments presented in examples 9.1.10, 9.1.11, 9.1.16 and 9.1.19 are all well-formed. We end this section with a last example of an environment well-formed.

Example 9.1.24 *Consider we enrich the environment described in Example 9.1.19 with the following definitions for the multiplication and the factorial functions:*

$$\begin{array}{lcl}
\mathrm{mult} & : & \{\mathsf{Even} \times \mathsf{Nat} \to \mathsf{Even},\ \mathsf{Nat} \times \mathsf{Even} \to \mathsf{Even},\ \mathsf{Odd} \times \mathsf{Odd} \to \mathsf{Odd},\ \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}\} \\
 & = & \{\lambda\langle \mathsf{o}, \mathsf{o}\rangle.\mathsf{o},\quad \lambda\langle \mathsf{o}, \mathsf{s}\, x\rangle.\mathsf{o},\quad \lambda\langle \mathsf{s}\, x, \mathsf{o}\rangle.\, \mathsf{o},\quad \lambda\langle \mathsf{s}\, x, \mathsf{s}\, y\rangle.\, \mathrm{plus}\, \langle \mathsf{s}\, y, \mathrm{mult}\, \langle x, \mathsf{s}\, y\rangle\rangle\} \\
\mathrm{fact} & : & \{\mathsf{Nat} \to \mathsf{Nat}\} \\
 & = & \{\lambda \mathsf{o}.\, \mathsf{s}\, \mathsf{o},\quad \lambda \mathsf{s}\, x.\, \mathrm{mult}\, \langle \mathsf{s}\, x, \mathrm{fact}\, x\rangle\} \\
\mathrm{mult} & : & \{\mathsf{MaybeNat} \times \mathsf{MaybeNat} \to \mathsf{MaybeNat}\} \\
 & = & \{\lambda\langle \mathsf{undef}, y\rangle.\, \mathsf{undef},\quad \lambda\langle \mathsf{o}, \mathsf{undef}\rangle.\, \mathsf{undef},\quad \lambda\langle \mathsf{s}\, x, \mathsf{undef}\rangle.\, \mathsf{undef}\} \\
\mathrm{fact} & : & \{\mathsf{MaybeNat} \to \mathsf{MaybeNat}\} \\
 & = & \{\lambda \mathsf{undef}.\mathsf{undef}\}
\end{array}$$

Let us name Σ the environment resulting of this expansion. Σ is a well-formed environment. Below we have a possible reduction sequence to compute fact (plus $\langle \mathsf{s}\,\mathsf{o}, \mathsf{s}\,\mathsf{o}\rangle$)*.*

$$\begin{array}{rcl}
\mathrm{fact}\, (\mathrm{plus}\, \langle \mathsf{s}\,\mathsf{o}, \mathsf{s}\,\mathsf{o}\rangle) & \to_{\delta_\Sigma} & \mathrm{fact}\, (\mathsf{s}\, (\mathsf{s}\, (\mathrm{plus}\, \langle \mathsf{o}, \mathsf{o}\rangle))) \\
 & \to_{\delta_\Sigma} & \mathrm{fact}\, (\mathsf{s}\, (\mathsf{s}\, \mathsf{o})) \\
 & \to_{\delta_\Sigma} & \mathrm{mult}\, \langle \mathsf{s}\, (\mathsf{s}\, \mathsf{o}), \mathrm{fact}\, \mathsf{o}\rangle \\
 & \to_{\delta_\Sigma} & \mathrm{mult}\, \langle \mathsf{s}\, (\mathsf{s}\, \mathsf{o}), \mathsf{s}\, \mathsf{o}\rangle \\
 & \to_{\delta_\Sigma} & \mathrm{plus}\, \langle \mathsf{s}\, \mathsf{o}, \mathrm{mult}\, \langle \mathsf{s}\, \mathsf{o}, \mathsf{s}\, \mathsf{o}\rangle\rangle \\
 & \to_{\delta_\Sigma} & \mathrm{plus}\, \langle \mathsf{s}\, \mathsf{o}, \mathrm{plus}\, \langle \mathsf{s}\, \mathsf{o}, \mathrm{mult}\, \langle \mathsf{o}, \mathsf{s}\, \mathsf{o}\rangle\rangle\rangle \\
 & \to_{\delta_\Sigma} & \mathrm{plus}\, \langle \mathsf{s}\, \mathsf{o}, \mathrm{plus}\, \langle \mathsf{s}\, \mathsf{o}, \mathsf{o}\rangle\rangle \\
 & \to_{\delta_\Sigma} & \mathrm{plus}\, \langle \mathsf{s}\, \mathsf{o}, \mathsf{s}\, \mathsf{o}\rangle \\
 & \to_{\delta_\Sigma} & \mathsf{s}\, (\mathsf{s}\, \mathsf{o})
\end{array}$$

9.2 Confluence

In this section we show that the reduction calculus generated by a non-overlapping environment is confluent. Observe that if Σ is non-overlapping, at most one rule of a function should apply for a given term. Therefore,

$$f\, a \to_{\delta_\Sigma} S(e) \quad \text{iff} \quad \exists!\, \lambda p.e \in \mathsf{Ru}_\Sigma(f).\, \exists!\, S\, .\ S(p) = a\ \wedge\ \mathsf{Supp}(S) \subseteq \mathsf{FV}(p)$$

The proof of confluence is done by the standard technique of Tait and Martin-Löf, similarly to what was done for λ_{CS} is Section 8.2.

Definition 9.2.1 *Let Σ be a non-overlapping environment. Define a binary relation $\twoheadrightarrow_{1_\Sigma}$ on $\mathcal{E}_{\mathsf{fun}}$ inductively as follows:*

1. $a \twoheadrightarrow_{1_\Sigma} a$
2. $a \twoheadrightarrow_{1_\Sigma} a' \quad \Rightarrow \quad \lambda x.a \twoheadrightarrow_{1_\Sigma} \lambda x.a'$
3. $a \twoheadrightarrow_{1_\Sigma} a' \ \wedge \ b \twoheadrightarrow_{1_\Sigma} b' \quad \Rightarrow \quad a\, b \twoheadrightarrow_{1_\Sigma} a'\, b'$
4. $a \twoheadrightarrow_{1_\Sigma} a' \ \wedge \ b \twoheadrightarrow_{1_\Sigma} b' \quad \Rightarrow \quad (\lambda x.a)\, b \twoheadrightarrow_{1_\Sigma} a'[x := b']$
5. $a \twoheadrightarrow_{1_\Sigma} a' \ \wedge \ e \twoheadrightarrow_{1_\Sigma} e' \quad \Rightarrow \quad f\, a \twoheadrightarrow_{1_\Sigma} S(e'),$ *whenever* $\exists\, \lambda p.e \in \mathsf{Ru}_\Sigma(f).\, \exists S.\ S(p) = a'$

Lemma 9.2.2 *Assume Σ is a non-overlapping environment. We have:*

$$a \twoheadrightarrow_{1_\Sigma} a' \ \wedge \ b \twoheadrightarrow_{1_\Sigma} b' \quad \Rightarrow \quad a[x := b] \twoheadrightarrow_{1_\Sigma} a'[x := b']$$

Proof. By induction on the definition of $a \twoheadrightarrow_{1_\Sigma} a'$. We treat here case 5.

Assume $a \twoheadrightarrow_{1_\Sigma} a'$ is $f\, a_1 \twoheadrightarrow_{1_\Sigma} S(e')$, and this is a direct consequence of $a_1 \twoheadrightarrow_{1_\Sigma} a'_1$, $e \twoheadrightarrow_{1_\Sigma} e'$ and $S(p) = a'_1$, for some substitution S and $\lambda p.e \in \mathsf{Ru}_\Sigma(f)$. By induction hypothesis $a_1[x := b] \twoheadrightarrow_{1_\Sigma} a'_1[x := b']$. Moreover, as $S(p) = a'_1$, we have $([x := b'] \circ S)(p) = a'_1[x := b']$. So, by rule 5 we have

$$(f\, a_1)[x := b] = f\,(a_1[x := b]) \twoheadrightarrow_{1_\Sigma} ([x := b'] \circ S)(e') = S(e')[x := b']$$

$\square$

Lemma 9.2.3 (Generation lemma for $\twoheadrightarrow_{1_\Sigma}$) *Let Σ be a non-overlapping environment. Then,*

1. *$\lambda x.a \twoheadrightarrow_{1_\Sigma} e$ implies $e \equiv \lambda x.a'$ with $a \twoheadrightarrow_{1_\Sigma} a'$.*
2. *$a_1\, a_2 \twoheadrightarrow_{1_\Sigma} e$ implies either:*
   - (a) *$e \equiv a'_1\, a'_2$ with $a_1 \twoheadrightarrow_{1_\Sigma} a'_1$ and $a_2 \twoheadrightarrow_{1_\Sigma} a'_2$;*
   - (b) *$a_1 \equiv \lambda x.b$, $e \equiv b'[x := a'_2]$ with $b \twoheadrightarrow_{1_\Sigma} b'$ and $a_2 \twoheadrightarrow_{1_\Sigma} a'_2$;*
   - (c) *or $a_1 \equiv f$, $e \equiv S(b')$ with $\lambda p.b \in \mathsf{Ru}_\Sigma(f)$, $b \twoheadrightarrow_{1_\Sigma} b'$, $a_2 \twoheadrightarrow_{1_\Sigma} a'_2$ and $S(p) = a'_2$.*

Proof. By induction on the definition of $\twoheadrightarrow_{1_\Sigma}$. $\square$

Lemma 9.2.4 *Let Σ be a non-overlapping environment. Then, $\twoheadrightarrow_{1_\Sigma}$ satisfies the* diamond property*, i.e.,*

$$a \twoheadrightarrow_{1_\Sigma} a_1 \ \wedge \ a \twoheadrightarrow_{1_\Sigma} a_2 \quad \Rightarrow \quad \exists\, a_3 \in \mathcal{E}_{\mathsf{fun}}.\ a_1 \twoheadrightarrow_{1_\Sigma} a_3 \ \wedge \ a_2 \twoheadrightarrow_{1_\Sigma} a_3$$

Proof. By induction on the definition of $a \twoheadrightarrow_{1_\Sigma} a_1$. $\square$

Lemma 9.2.5 *Let Σ be a non-overlapping environment. Then $\twoheadrightarrow_{\beta\delta_\Sigma}$ is the transitive closure of $\twoheadrightarrow_{1_\Sigma}$.*

Proof. $\twoheadrightarrow_{1_\Sigma}$ contains the reflexive closure of $\rightarrow_{\beta\delta_\Sigma}$. Moreover, $\twoheadrightarrow_{1_\Sigma} \subseteq \twoheadrightarrow_{\beta\delta_\Sigma}$. Since $\twoheadrightarrow_{\beta\delta_\Sigma}$ is the reflexive-transitive closure of $\rightarrow_{\beta\delta_\Sigma}$ it is also the transitive closure of $\twoheadrightarrow_{1_\Sigma}$. $\square$

Theorem 9.2.6 (Confluence) *Let Σ be a non-overlapping environment. Then $\rightarrow_{\beta\delta_\Sigma}$ is confluent:*

$$a_1 =_{\beta\delta_\Sigma} a_2 \quad \Rightarrow \quad \exists\, e \in \mathcal{E}_{\mathsf{fun}}.\ a_1 \twoheadrightarrow_{\beta\delta_\Sigma} e \ \wedge\ a_2 \twoheadrightarrow_{\beta\delta_\Sigma} e$$

Proof. Assume $a_1 =_{\beta\delta_\Sigma} a_2$, then $\exists\, a \in \mathcal{E}_{\mathsf{fun}}.\ a \twoheadrightarrow_{\beta\delta_\Sigma} a_1 \ \wedge\ a \twoheadrightarrow_{\beta\delta_\Sigma} a_2$. As $\twoheadrightarrow_{\beta\delta_\Sigma}$ is the transitive closure of $\twoheadrightarrow_{1_\Sigma}$, $\twoheadrightarrow_{\beta\delta_\Sigma}$ satisfies also the diamond property. So, we conclude. $\Box$

9.3 Subject Reduction

In this section we show that the type of an expression is preserved under reduction whenever the environment is well-typed and non-overlapping. The reduction rules introduced by the pattern abstractions and the overloading of functions make the proof a bit hard. Of course the difficulties appear when we want to show that typing is preserved under δ-reduction. For β-reduction the proof is quite standard.

Before going into the proof of subject reduction, we have to introduce some auxiliary definitions around the concepts of variable context and substitution, and some lemmata. As the set of types and the subtyping rules for $\lambda_{\mathsf{CS+fun}}$ and λ_{CS} are exactly the same, we have for $\lambda_{\mathsf{CS+fun}}$ the results about the subtyping relation already proved for λ_{CS} in Chapter 8, namely lemmas 8.3.3 and 8.3.4.

Lemma 9.3.1

1. *If $\Gamma \subseteq \Gamma'$ and $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ then $\nabla \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$.*

2. *If $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ then $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma)$.*

Proof. By induction on the derivation of $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$. $\Box$

Lemma 9.3.2 (Generation lemma for typing)

1. $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} x : \sigma \quad \Rightarrow \quad (x : \tau) \in \Gamma \ \wedge\ \tau \leq \sigma$

2. $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a\, b : \sigma \quad \Rightarrow \quad \nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a : \tau \rightarrow \sigma' \ \wedge\ \nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} b : \tau \ \wedge\ \sigma' \leq \sigma$

3. $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} \lambda x.e : \sigma \quad \Rightarrow \quad \sigma \equiv \tau_1 \rightarrow \tau_2 \ \wedge\ \nabla \,|\, \Gamma, x : \tau_1' \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau_2' \ \wedge\ \tau_1 \leq \tau_1' \ \wedge\ \tau_2' \leq \tau_2$

4. $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c : \sigma \quad \Rightarrow \quad \sigma \equiv \vec{\gamma} \rightarrow \theta \ \wedge\ \vec{\gamma} \leq \mathsf{Dom}_d^{\vec{\tau}}(c) \ \wedge\ d\, \vec{\tau} \leq \theta \ \wedge\ c \in \mathsf{C}(d)$

5. $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} f : \sigma \quad \Rightarrow \quad \sigma \equiv \tau_1 \rightarrow \tau_2 \ \wedge\ S(\tau) \leq \sigma \ \wedge\ (f : \tau) \in \nabla$

Proof. By inspection on the derivation of the antecedent judgments. $\Box$

The following lemma is a generation lemma for constructors fully applied.

Lemma 9.3.3

$$\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c\, a_1 \ldots a_{\mathsf{ar}(c)} : \theta \ \Rightarrow\ \theta \equiv d\, \vec{\tau} \ \wedge\ c \in \mathsf{C}(d) \ \wedge\ \nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a_i : \mathsf{Dom}_d^{\vec{\tau}}(c)[i], i = 1..\mathsf{ar}(c)$$

Proof. Assume $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c\, a_1 \ldots a_{\mathsf{ar}(c)} : \theta$. By Lemma 9.3.2 it follows that $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c : \vec{\gamma} \to \sigma$, $\sigma \leq \theta$ and also that, for $1 \leq j \leq \mathsf{ar}(c)$,

$$\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a_i : \gamma_i \;\wedge\; \gamma_i \leq \mathsf{Dom}_{d'}^{\vec{\psi}}(c)[i] \tag{9.1}$$

$$d'\vec{\psi} \leq \sigma \;\wedge\; c \in \mathsf{C}(d') \tag{9.2}$$

By transitivity we have $d'\vec{\psi} \leq \theta$. So, by generation lemma for subtyping one gets that $\theta \equiv d\,\vec{\tau}$, $d' \sqsubseteq_{\mathcal{D}} d$ and $\vec{\psi} \leq \vec{\tau}$. As every constructor is strictly overloaded $\mathsf{Dom}_{d'}^{\vec{\alpha}}(c) \leq \mathsf{Dom}_{d}^{\vec{\alpha}}(c)$. Since the parameters can only occur positively in the domains of constructors, by Lemma 8.3.4,

$$\mathsf{Dom}_{d'}^{\vec{\psi}}(c) \leq \mathsf{Dom}_{d}^{\vec{\tau}}(c)$$

From (9.2), as $d' \sqsubseteq_{\mathcal{D}} d$, follows that $c \in \mathsf{C}(d)$. From (9.1), by subsumption, one gets $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a_i : \mathsf{Dom}_{d}^{\vec{\tau}}(c)[i]$ for $i = 1..\mathsf{ar}(c)$. $\square$

Lemma 9.3.4 (Substitution lemma for typing)
If $\nabla \,|\, \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_{\lambda_{\mathsf{CS+fun}}} a : \sigma$ *and* $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} b_i : \tau_i$ *for* $i = 1..n$*, then* $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a[x_1 := b_1, \ldots, x_n := b_n] : \sigma$.

Proof. By induction on the derivation of $\nabla \,|\, \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n x : \tau \vdash_{\lambda_{\mathsf{CS+fun}}} a : \sigma$. Similar to the proof of Lemma 8.3.7 $\square$

We define a notion of *subcontext*, which is a partial order over variable contexts, and we introduce a notion of *minimal variable context* for typing a given pattern with a given type, and show how can they be obtained.

Definition 9.3.5 (Subcontext) *Let* Γ *and* Γ' *be variable contexts. We say that* Γ *is a* subcontext *of* Γ'*, and write* $\Gamma \trianglelefteq \Gamma'$*, if*

1. $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma')$
2. $\forall\, x \in \mathsf{dom}(\Gamma).\ \Gamma'(x) \leq \Gamma(x)$

Lemma 9.3.6 *The binary relation* $\trianglelefteq$ *is a partial order.*

Proof. It follows immediately from the fact that $\leq$ is a partial order. $\square$

Lemma 9.3.7

$$\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau \;\wedge\; \Gamma \trianglelefteq \Gamma' \quad\Rightarrow\quad \nabla \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau$$

Proof. By induction on the derivation of $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau$. $\square$

Definition 9.3.8 Γ *is the* minimal variable context *for typing the pattern* p *with the type* τ *if:*

1. $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} p : \tau$*; and*
2. *for every* Γ' *such that* $\nabla \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} p : \tau$*, one has* $\Gamma \trianglelefteq \Gamma'$.

Definition 9.3.9 *We define* $\mathsf{mvc}(p : \sigma)$ *inductively as follows:*

$$\begin{array}{rcl} \mathsf{mvc}(x : \tau) & = & \{x : \tau\} \\ \mathsf{mvc}(c\, p_1 \dots p_{\mathsf{ar}(c)} : d\, \vec{\tau}) & = & \bigcup_{i=1..\mathsf{ar}(c)} \mathsf{mvc}(p_i : \mathsf{Dom}_d^{\vec{\tau}}(c)[i]) \quad \textit{if} \quad c \in \mathsf{C}(d) \end{array}$$

Note that $\mathsf{mvc}(p : \sigma)$ is not defined whenever $p \equiv c\, p_1 \dots p_{\mathsf{ar}(c)}$ and σ is not a datatype, or is a datatype that does not have c as constructor. Otherwise $\mathsf{mvc}(p : \sigma)$ produces a well-defined variable context since patterns are linear. Bellow, when we write $\mathsf{mvc}(p : \sigma)$ we assume that $\mathsf{mvc}(p : \sigma)$ is defined.

Proposition 9.3.10 $\mathsf{mvc}(p : \tau)$ *is the minimal variable context for typing* $p : \tau$.

Proof. By induction on the structure of the pattern p. We must consider two cases:

- Case $p \equiv x$. Let $\Gamma = \mathsf{mvc}(x : \tau) = \{x : \tau\}$ and assume $\nabla \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} x : \tau$. By Lemma 9.3.2, $x : \sigma \in \Gamma'$ and $\sigma \leq \tau$, and thus $\Gamma \trianglelefteq \Gamma'$.
- Case $p \equiv c\, p_1 \dots p_{\mathsf{ar}(c)}$ and $\tau \equiv d\, \vec{\tau}$. Let $\Gamma = \bigcup_{i=1..\mathsf{ar}(c)} \mathsf{mvc}(p_i : \mathsf{Dom}_d^{\vec{\tau}}(c)[i])$ and assume $\nabla \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} c\, p_1 \dots p_{\mathsf{ar}(c)} : d\, \vec{\tau}$. By Lemma 9.3.3 follows that $c \in \mathsf{C}(d)$ and $\nabla \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} p_j : \mathsf{Dom}_d^{\vec{\tau}}(c)[j]$ for $j = 1..\mathsf{ar}(c)$. By induction hypothesis, $\mathsf{mvc}\big(p_j : \mathsf{Dom}_d^{\vec{\tau}}(c)[j]\big) \trianglelefteq \Gamma'$ for every $j = 1..\mathsf{ar}(c)$. Hence $\Gamma \trianglelefteq \Gamma'$, because
  1. $\mathsf{dom}(\Gamma) = \bigcup_{i=1..\mathsf{ar}(c)} \mathsf{dom}\big(\mathsf{mvc}(p_i : \mathsf{Dom}_d^{\vec{\tau}}(c)[i])\big) \subseteq \mathsf{dom}(\Gamma')$
  2. $\forall\, x \in \mathsf{dom}(\Gamma).\ x \in \mathsf{dom}\big(\mathsf{mvc}(p_j : \mathsf{Dom}_d^{\vec{\tau}}(c)[j])\big)$ for some $1 \leq j \leq \mathsf{ar}(c)$, and so, $\Gamma'(x) \leq \mathsf{mvc}(p_j : \mathsf{Dom}_d^{\vec{\tau}}(c)[j])(x) = \Gamma(x)$.

□

Lemma 9.3.11

$$\mathsf{mvc}(p : \sigma) \ \wedge\ \tau \leq \sigma \ \Rightarrow\ \mathsf{mvc}(p : \sigma) \trianglelefteq \mathsf{mvc}(p : \tau)$$

Proof. By induction on the structure of p. □

Definition 9.3.12 *Let* M *be a type substitution and* Γ *a variable context.* $M(\Gamma)$ *is defined inductively as follows:*

$$\begin{array}{rcl} M(\emptyset) & = & \emptyset \\ M(\Gamma, x\!:\!\tau) & = & M(\Gamma), x\!:\!M(\tau) \end{array}$$

Lemma 9.3.13 *Let* M *be a type substitution. Then,*

1. $\tau \leq \sigma \ \Rightarrow\ M(\tau) \leq M(\sigma)$
2. $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau \ \Rightarrow\ \nabla \,|\, M(\Gamma) \vdash_{\lambda_{\mathsf{CS+fun}}} e : M(\tau)$
3. $M(\mathsf{mvc}(p : \tau)) = \mathsf{mvc}(p : M(\tau))$

Proof.

1. By induction on the derivation of $\tau \leq \sigma$.
2. By induction on the derivation of $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau$ using 1.

3. By induction on the structure of p.

$\square$

Next we introduce the concept of compatible substitutions, and we define the union of compatible substitutions.

Definition 9.3.14

1. *Let S and R be type substitutions. We say that S and R are* compatible substitutions *if* $\forall\, x \in \mathsf{Supp}(S) \cap \mathsf{Supp}(R)\ .\ S(x) = R(x)$

2. *Let S and R be compatible type substitutions. The* union *of substitutions S and R, that we write as $S \oplus R$, can then be defined as follows:*

$$(S \oplus R)(x) = \begin{cases} S(x) & \textit{if } x \in \mathsf{Supp}(S) \\ R(x) & \textit{if } x \in \mathsf{Supp}(R) \\ x & \textit{otherwise} \end{cases}$$

Definition 9.3.15 *Let $A \subseteq \mathcal{V}_{\mathcal{E}}$, S a term substitution and Γ a variable context. We define $\Gamma|_A$ and $S|_A$ as follows:*

$$\Gamma|_A = \{\Gamma(x) \mid x \in A \cap \mathsf{dom}(\Gamma)\} \qquad \textit{and} \qquad S|_A(x) = \begin{cases} S(x) & \textit{if } x \in A \\ x & \textit{otherwise} \end{cases}$$

Lemma 9.3.16 *Let Σ be an environment well-typed and non-overlaping, and $f \in \mathsf{FS}_\Sigma$. If $\lambda c\,\vec{p}.e \in \mathsf{Ru}_\Sigma(f)$, then for every $\theta_1 \to \theta_2 \in \mathsf{Ty}_\Sigma(f)$, θ_1 is a datatype.*

Proof. Assume there is a $\lambda c\,\vec{p}.e \in \mathsf{Ru}_\Sigma(f)$. If some $\theta_1 \to \theta_2 \in \mathsf{Ty}_\Sigma(f)$ is such that θ_1 is a type variable or a function type, then it must exist a pattern abstraction $\lambda x.e' \in \mathsf{Ru}_\Sigma(f)$ because Σ is well-typed but that is impossible since Σ is non-overlapping. Hence the domain of every possible type for f must be a datatype. $\square$

Lemma 9.3.17 *Let $c\,p_1 \ldots p_{\mathsf{ar}(c)}$ be a pattern. If $\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c\,a_1 \ldots a_{\mathsf{ar}(c)} : d\,\vec{\tau}$, $S(p_i) = a_i$ for $i = 1..\mathsf{ar}(c)$ and $d\,\vec{\tau} \leq d'M(\vec{\rho})$, then $\exists\,\Gamma'.\ .\mid\Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} c\,p_1 \ldots p_{\mathsf{ar}(c)} : d'\vec{\rho}$.*

Proof. By induction on the structure of $c\,p_1 \ldots p_{\mathsf{ar}(c)}$.

We have $\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c\,a_1 \ldots a_{\mathsf{ar}(c)} : d\,\vec{\tau}$ so, by Lemma 9.3.3, we have that $c \in \mathsf{C}(d)$ and $\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a_i : \mathsf{Dom}_d^{\vec{\tau}}(c)[i]$ for $i = 1..\mathsf{ar}(c)$. From $d\,\vec{\tau} \leq d'M(\vec{\rho})$, by the generation lemma, we have $d \sqsubseteq_{\mathcal{D}} d'$ and $\vec{\tau} \leq M(\vec{\rho})$. As constructors are strictly overloaded and parameters can only occur in positive positions of the domain of constructors, it follows that $\mathsf{Dom}_d^{\vec{\tau}}(c) \leq \mathsf{Dom}_{d'}^{M(\vec{\rho})}(c)$. Moreover $c \in \mathsf{C}(d')$.

We want to prove that $\exists\,\Gamma'.\ .\mid\Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} c\,p_1 \ldots p_{\mathsf{ar}(c)} : d'\vec{\rho}$. Let us see how to obtain such a Γ'. For each p_i with $1 \leq i \leq \mathsf{ar}(c)$:

- If $p_i \equiv x_i$, then let $\Gamma'_i \equiv x : \mathsf{Dom}_{d'}^{\vec{\rho}}(c)[i]$. We have $.\mid\Gamma'_i \vdash_{\lambda_{\mathsf{CS+fun}}} p_i : \mathsf{Dom}_{d'}^{\vec{\rho}}(c)[i]$.

- If $p_i \equiv c_i\,p_{i,1} \ldots p_{i,\mathsf{ar}(c_i)}$, then $a_i \equiv c_i\,a_{i,1} \ldots a_{i,\mathsf{ar}(c_i)}$ and $S(p_{i,j}) = a_{i,j}$ for $j = 1..\mathsf{ar}(c_i)$. Moreover

$$\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c_i\,p_{i,1} \ldots p_{i,\mathsf{ar}(c_i)} : d_i\,\vec{\tau}_i \;\equiv\; \mathsf{Dom}_d^{\vec{\tau}}(c)[i] \;\leq\; \mathsf{Dom}_{d'}^{M(\vec{\rho})}(c)[i] \;\equiv\; d''M(\vec{\theta})$$

for some $d_i\,\vec{\tau_i}$ and $d''\vec{\theta}$. So, by induction hypothesis,

$$\exists\,\Gamma'_i.\quad .\,|\,\Gamma'_i \vdash_{\lambda_{\mathsf{CS+fun}}} p_i : d''\vec{\theta} \;\equiv\; \mathsf{Dom}^{\vec{\theta}}_{d''}(c)[i]$$

Let $\Gamma' \equiv \bigcup_{i=1}^{\mathsf{ar}(c)} \Gamma'_i|_{\mathsf{FV}(p_i)}$. Γ' is a well-formed context because the free variables of each p_i are always different, since $c\,p_1 \ldots p_{\mathsf{ar}(c)}$ is a linear pattern. As $c \in \mathsf{C}(d')$ we have by (cons) $\nabla\,|\,\Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} c : \mathsf{Dom}^{\vec{\rho}}_{d'}(c) \to d'\,\vec{\rho}$. Hence, applying (app) we get $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c\,p_1 \ldots p_{\mathsf{ar}}(c) : d'\,\vec{\rho}$. □

Lemma 9.3.18 *Let Σ be an environment well-typed and non-overlapping, and $\nabla = \mathsf{er}(\Sigma)$. If $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} f : \tau \to \sigma$, $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a : \tau$ and $S(p) = a$ for the unique $\lambda p.e \in \mathsf{Ru}_\Sigma(f)$, then*

$$\forall\,\theta_1 \to \theta_2 \in \mathsf{Ty}_\Sigma(f).\;\; M(\theta_1 \to \theta_2) \le \tau \to \sigma \;\Rightarrow\; \exists\,\Gamma'.\;\; .\,|\,\Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} p : \theta_1$$

Proof. By case analysis on p.

- Case $p \equiv x$, the for every $\theta_1 \to \theta_2 \in \mathsf{Ty}_\Sigma(f)$ we can always define $\Gamma' = x : \theta_1$ and we have $.\,|\,\Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} x : \theta_1$ by (var).
- Case $p \equiv c\,p_1 \ldots p_{\mathsf{ar}(c)}$ then $a \equiv c\,a_1 \ldots a_{\mathsf{ar}(c)}$ and $S(p_i) = a_i$, $i = 1..\mathsf{ar}(c)$. Moreover, from the hypothesis, by Lemma 9.3.3 we have $\tau \equiv d'\vec{\tau'}$. For every $\theta_1 \to \theta_2 \in \mathsf{Ty}_\Sigma(f)$ such that $M(\theta_1 \to \theta_2) \le \tau \to \sigma$. By Lemma 9.3.16, θ_1 must be a datatype. Let $\theta_1 \equiv d\,\vec{\rho}$. By the generation lemma for subtyping, one has $d'\vec{\tau'} \le d\,M(\vec{\rho})$. As $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} c\,a_1 \ldots a_{\mathsf{ar}(c)} : d'\vec{\tau'}$ we can conclude that
$$\exists\,\Gamma'.\;\; .\,|\,\Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} c\,p_1 \ldots p_{\mathsf{ar}(c)} : d\,\vec{\rho}$$
using Lemma 9.3.17.

□

Lemma 9.3.19 *Let $p_1, \ldots, p_n$ be a set of patterns such that $\mathsf{FV}(p_i) \cap \mathsf{FV}(p_j) = \emptyset$ for every $i, j \in \{1, \ldots, n\}$. If for every $i = 1..n$, $\nabla\,|\,\bigcup_{i=1}^n \mathsf{mvc}(p_i : \tau_i) \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$, $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a_i : \tau_i$ and $S_i(p_i) = a_i$ with $\mathsf{Supp}(S_i) \subseteq \mathsf{FV}(p_i)$, then $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} S(e) : \sigma$, where $S = \bigoplus_{i=1}^n S_i$.*

Proof. By induction on the multi-set of patterns $P = \{p_1, \ldots, p_n\}$.

Let us assume firstly that each pattern p_i is a variable x_i, then $\nabla\,|\,x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ and $S_i = [x_i := a_i]$. Using the variable convention, from Lemma 9.3.1 follows

$$\nabla\,|\,\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$$

Since $S = [x_1 := a_1, \ldots, x_n := a_n]$, by Lemma 9.3.4 we obtain $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} S(e) : \sigma$.

Otherwise, at least one pattern p_k (for some $1 \le k \le n$) is not a variable. Let us assume, without loss of generality, that $p_k \equiv c_k\,p_{k,1} \ldots p_{k,\mathsf{ar}(c_k)}$. Then $a_k \equiv c_k\,a_{k,1} \ldots a_{k,\mathsf{ar}(c_k)}$ and $S_{k,j}(p_{k,j}) = a_{k,j}$ for $j = 1..\mathsf{ar}(c_k)$, with $S_{k,j} = S_k|_{\mathsf{FV}(p_{k,j})}$. Moreover, by Lemma 9.3.3, we have $\tau_k \equiv d_k\vec{\tau_k}$ for some $d_k\vec{\tau_k}$, $c_k \in \mathsf{C}(d_k)$, and $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a_{k,j} : \mathsf{Dom}^{\vec{\tau_k}}_{d_k}(c_k)[j]$. From the hypothesis, by definition $\nabla\,|\,\bigcup_{j=1}^{\mathsf{ar}(c_k)} \mathsf{mvc}(p_{k,j} : \mathsf{Dom}^{\vec{\tau_k}}_{d_k}(c_k)[j]) \cup \bigcup_{i=1, i\neq k}^n \mathsf{mvc}(p_i : \tau_i) \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$. Let Q be the multi-set $\{p_1, \ldots, p_{k-1}, p_{k,1}, \ldots, p_{k,\mathsf{ar}(c_k)}, p_{k+1}, \ldots, p_n\}$. We have $Q \ll P$ for Q is obtained from P replacing p_k by the subpatterns $p_{k,1}, \ldots, p_{k,\mathsf{ar}(c_k)}$. So, from induction hypothesis follows

$$\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} \Big(\bigoplus_{j=1}^{\mathsf{ar}(c_k)} S_{k,j} \oplus \bigoplus_{i=1, i\neq k}^{n} S_i\Big)(e) : \sigma$$

Since, $\bigoplus_{j=1}^{\mathsf{ar}(c_k)} S_{k,j} = S_k$ we have $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} S(e) : \sigma$. □

We can now prove that computation preserves typing.

Theorem 9.3.20 (Subject reduction) *Let Σ be an environment well-typed and non-overlapping, and let $\nabla = \mathsf{er}(\Sigma)$. Then*

$$\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a : \sigma \quad \wedge \quad a \to_{\beta\delta_\Sigma} a' \quad \Rightarrow \quad \nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a' : \sigma$$

Proof. By induction on the derivation of $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} a : \sigma$. The interesting case is when the last rule is (app) and $a \equiv f\, b$. The remaining cases were already treated in the proof of subject reduction for λ_{CS}.

So, assume we have $a \equiv f\, b$, and the last step is

$$\frac{\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} f : \tau \to \sigma \qquad \nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} b : \tau}{\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} f\, b : \sigma}$$

Moreover assume $f\, b \to_{\beta\delta_\Sigma} S(e)$ for the unique $\lambda p.e \in \mathsf{Ru}_\Sigma(f)$ and the unique substitution S such that $b = S(p)$ and $\mathsf{Supp}(S) \subseteq \mathsf{FV}(p)$. We want to show that $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} S(e) : \sigma$.

By Lemma 9.3.2, we know that $\exists\, \theta_1 \to \theta_2 \in \mathsf{Ty}_\Sigma(f)$. $M(\theta_1 \to \theta_2) \leq \tau \to \sigma$ for some type substitution M. So, by the generation lemma for subtyping we get that $\tau \leq M(\theta_1)$ and $M(\theta_2) \leq \sigma$. Now, using Lemma 9.3.18, follows that $\exists\, \Gamma'.\ . \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} p : \theta_1$; and, as Σ is well-typed, $\nabla \,|\, \Gamma' \vdash_{\lambda_{\mathsf{CS+fun}}} e : \theta_2$. Therefore, $\mathsf{mvc}(p : \theta_1)$ is defined and

$$\nabla \,|\, \mathsf{mvc}(p : \theta_1) \vdash_{\lambda_{\mathsf{CS+fun}}} e : \theta_2$$

By Lemma 9.3.13, $\nabla \,|\, \mathsf{mvc}(p : M(\theta_1)) \vdash_{\lambda_{\mathsf{CS+fun}}} e : M(\theta_2)$. As $\tau \leq M(\theta_1)$, using lemmas 9.3.11 and 9.3.7 we obtain $\nabla \,|\, \mathsf{mvc}(p : \tau) \vdash_{\lambda_{\mathsf{CS+fun}}} e : M(\theta_2)$. From $M(\theta_2) \leq \sigma$, follows that $\nabla \,|\, \mathsf{mvc}(p : \tau) \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ by applying (sub). Finally, we are in conditions of using Lemma 9.3.19 and show that $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} S(e) : \sigma$. □

9.4 Strong Normalization

Clearly, the calculus is not normalizing as we do not impose any restriction on recursive definitions. For example, the following function does not terminate:

$$\begin{array}{rcl} \text{loop} & : & \{\mathsf{Nat} \to \mathsf{Nat}\} \\ & = & \{\lambda x.\, \text{loop}\, x\} \end{array}$$

In order to recover termination, we provide a simple criterion: the criterion for termination is inspired from [44]. In a nutshell, we ensure that a recursive function is terminating if, to compute $f\, p$, each recursive call of f is applied to patterns that are structurally smaller than p.

Definition 9.4.1 (Structurally smaller) *Let $p, p' \in \mathcal{P}$. p is* structurally smaller *than p', written $p \prec p'$, if it can be derived from the rules of Figure 9.3.*

Definition 9.4.2 *Let Σ be an environment and $f, g \in \mathsf{FS}_\Sigma$.*

1. *We say that g* occurs in *f, written g occur_Σ f, if exists a $\lambda p.e \in \mathsf{Ru}_\Sigma(f)$ such that g occ e.*

1. $$\frac{}{p_i \prec c\, p_1 \ldots p_{\mathsf{ar}(c)}} \quad \text{if} \quad 1 \leq i \leq \mathsf{ar}(c)$$

2. $$\frac{p_i \prec p'_i}{c\, p_1 \ldots p_i \ldots p_{\mathsf{ar}(c)} \prec c\, p_1 \ldots p'_i \ldots p_{\mathsf{ar}(c)}} \quad \text{if} \quad 1 \leq i \leq \mathsf{ar}(c)$$

3. $$\frac{p \prec p' \quad p' \prec p''}{p \prec p''}$$

Figure 9.3: Structurally smaller relation

2. *We define the set* $\mathsf{MD}_\Sigma(f)$ *of functions* mutually dependent *to* f *as follows:*

$$\mathsf{MD}_\Sigma(f) = \{g \mid g \ \mathsf{occur}_\Sigma \ f \ \wedge \ f \ \mathsf{occur}_\Sigma \ g\}$$

Definition 9.4.3 (Argument decreasing) *Let* Σ *be an environment.*

1. *A pattern abstraction* $\lambda p.e$ *is* argument decreasing w.r.t. f *if each occurrence of* f *in* e *is a term* $f\, p'$ *such that* $p' \prec p$.

2. *A function* f *is* argument decreasing *in* Σ *if all pattern abstractions of* $\mathsf{Ru}_\Sigma(f)$ *are argument decreasing w.r.t. each* $g \in \mathsf{MD}_\Sigma(f)$.

3. Σ *is* argument decreasing *if every function defined in it is argument decreasing.*

One can prove that all typable expressions are strongly normalizing for every environment where all function declarations are argument decreasing.

Theorem 9.4.4 (Strong normalization) *Let* Σ *be an well-formed environment and argument decreasing. If* $\mathsf{er}(\Sigma) \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$*, then* e *is strongly normalizing with respect to* $\beta\delta_\Sigma$*-reduction.*

Proof. The proof proceeds by the standards techniques based on saturated sets (or reducibility candidates). See [28] for such a proof for a more elaborate termination criteria. □

9.5 Type Checking

We want to decide whether or not a type judgment $\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ is derivable. We saw in Section 8.5 that type-checking is decidable for λ_{CS}. In $\lambda_{\mathsf{CS+fun}}$ we have an extra context of overloaded functions, but we can follow the same strategy to decide type-checking.

9.5.1 The System $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$

Similarly to what is done in Subsection 8.5.2, here we present system $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$. The set $\mathcal{T}^{\mathsf{a}}_{\mathsf{fun}}$ of types of $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$ is equal to $\mathcal{T}^{\mathsf{a}}_{\mathsf{cs}}$.

Definition 9.5.1 (Expressions) *The set* $\mathcal{E}^{\mathsf{a}}_{\mathsf{fun}}$ *of* expressions *of* $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$ *is given by the abstract syntax:*

$$a, b \ ::= \ x \mid c_d \mid f_\tau \mid \lambda x.a \mid a\, b$$

The typing system of $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$ is similar to the one of $\lambda^{\mathsf{a}}_{\mathsf{CS}}$. Here we do not have typing rules for case-expressions and letrec-expressions, and we have the following rule for functions:

$$(\text{fun}) \quad \frac{}{\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}} f_\tau : S(\tau)} \qquad \text{where } S \text{ is a substitution}$$

Definition 9.5.2 (∇-annotated terms) *Let* ∇ *be a context of overloaded variables. The set of* ∇-annotated *variants of a term is given by the mapping* $\mathsf{an}_\nabla : \mathcal{E}_{\mathsf{fun}} \to \mathcal{P}(\mathcal{E}^{\mathsf{a}}_{\mathsf{fun}})$ *defined as follows:*

$$\begin{array}{rcl}
\mathsf{an}_\nabla(x) & = & \{x\} \\
\mathsf{an}_\nabla(c) & = & \{c_d \mid c \in \mathsf{C}(d)\} \\
\mathsf{an}_\nabla(f) & = & \{f_\tau \mid (f : \tau) \in \nabla\} \\
\mathsf{an}_\nabla(a\, b) & = & \{a'\, b' \mid a' \in \mathsf{an}_\nabla(a) \wedge b' \in \mathsf{an}_\nabla(b)\} \\
\mathsf{an}_\nabla(\lambda x.a') & = & \{\lambda x.a' \mid a' \in \mathsf{an}_\nabla(a)\}
\end{array}$$

Observe that this definition follows very closely Definition 8.5.6. The results stated for $\lambda^{\mathsf{a}}_{\mathsf{CS}}$ are preserved in $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$, as it can be easily checked.

9.5.2 The System $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$

Similarly to what is done in Subsection 8.5.2, here we present system $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$c. The set $\mathcal{T}^{\mathsf{ac}}_{\mathsf{fun}}$ of types of $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$ is equal to $\mathcal{T}^{\mathsf{ac}}_{\mathsf{cs}}$, and the set $\mathcal{E}^{\mathsf{ac}}_{\mathsf{fun}}$ of terms of $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$ is equal to $\mathcal{E}^{\mathsf{ac}}_{\mathsf{cs}}$. The typing system of $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$ is equal to the one of $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ without (case) and (rec) rules and with the following rule for functions:

$$(\text{fun}) \quad \frac{}{(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} f_\tau : S(\tau)} \qquad \text{where } S \text{ is a substitution}$$

The algorithm to compute a most general typing to a term in $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$ is the one already defined in Figure 8.14 without the cases for letrec and case-expressions and with the extra case for functions, defined in Figure 9.4.

$$\begin{array}{rll}
\mathsf{typeJudg}(f_\tau) = & \mathsf{let} & S = \mathsf{match}\big(\{\tau \leq \beta\}\big) \\
& & (\Delta_*, \Theta_*) = S \bullet \big(\{\tau \leq \beta\}, \emptyset\big) \\
& \mathsf{in} & \big(\emptyset,\, S(\beta),\, \Delta_*,\, \Theta_*\big)
\end{array}$$
assuming that β *is a fresh type variable*

Figure 9.4: The algorithm $\mathsf{typeJudg}$ (for functions)

All the results obtained for $\lambda^{\mathsf{ac}}_{\mathsf{CS}}$ are preserved in $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$, as it can be easily checked. We just rewrite here the two main theorems.

Theorem 9.5.3 (Soundness of $\mathsf{typeJudg}$) *If* $\mathsf{typeJudg}(e) = (\Gamma, \sigma, \Delta, \Theta)$, *then every instance of* $(\Delta, \Theta) \mid \Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} e : \sigma$ *is provable.*

Proof. By Theorem 8.5.32 and Lemma 8.5.42 (both adapted for $\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}$). □

Theorem 9.5.4 (Completeness of typeJudg) *If* $(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS}}} e : \sigma$, *then* $\mathsf{typeJudg}(e) = (\Gamma', \sigma', \Delta', \Theta')$ *and* $(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} e : \sigma$ *is an instance of* $(\Delta',\Theta')\,|\,\Gamma' \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} e : \sigma'$.

Proof. By induction on the structure of terms. We just have to consider the case of functions. Assume $\mathsf{typeJudg}(f_\tau)$ as presented in Figure 9.4 and $(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} f_\tau : \sigma$. By Lemma 8.5.44, we may assume the proof uses rule (fun) followed by rule (sub). We let $(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} f_\tau : R(\tau)$ and $(\Delta,\Theta) \vdash R(\tau) \leq \sigma$ must hold. We have $\mathsf{typeJudg}(f_\tau) = (\emptyset, S(\beta), \Delta_*, \Theta_*)$ and S is a most general matching substitution for $\{\tau \leq \beta\}$, being β fresh.

Let Z be a substitution such that $Z(\beta) = \sigma$ and $Z(\tau) = R(\tau)$. Since (Δ,Θ) is atomic, $Z(\tau)$ matches $Z(\beta)$, by Lemma 8.5.39. Therefore, Z is a matching substitution for $\{\tau \leq \beta\}$. Using Lemma 8.5.46, $Z = W \circ S$ for some substitution W. Moreover, by Lemma 8.5.23, W is a matching substitution for (Δ_*, Θ_*) and $W \bullet (\Delta_*, \Theta_*) = Z \bullet (\{\tau \leq \beta\}, \emptyset)$.

We have $(\Delta,\Theta) \vdash (Z(\{\tau \leq \beta\}), Z(\emptyset))$ then, by Lemma 8.5.27, $(\Delta,\Theta) \vdash Z \bullet (\{\tau \leq \beta\}, \emptyset)$. By Lemma 8.5.46, we have $(\Delta,\Theta) \vdash W \bullet (\Delta_*, \Theta_*)$. Furthermore, $W(\emptyset) \subseteq \Gamma$ and $\sigma = W(S(\beta))$. Hence, we can conclude that $(\Delta,\Theta)\,|\,\Gamma \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} f_\tau : \sigma$ is an instance of $(\Delta_*,\Theta_*)\,|\,\emptyset \vdash_{\lambda^{\mathsf{ac}}_{\mathsf{CS+fun}}} f_\tau : S(\beta)$ by W. □

9.5.3 Decidability of Type Checking

We want to decide whether or not it is possible to type a term e with a type σ in a given context $\nabla\,|\,\Gamma$. Following what was done in Subsection 8.5.4, our claim is that $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ is derivable if and only if there is an ∇-annotated term $e' \in \mathsf{an}_\nabla(e)$ for which $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}} e' : \sigma$ is derivable. As the set of ∇-annotated terms $\mathsf{an}_\nabla(e)$ is finite, we have a decision procedure.

To test whether or not $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}} e' : \sigma$ is derivable we use the algorithm derivable defined in Figure 8.15.

Theorem 9.5.5 $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}} e : \tau$ *is derivable iff* $\mathsf{derivable}(\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}} e : \tau)$ *returns true.*

Proof. Immediate from theorems 8.5.57 and 8.5.58. □

Corollary 9.5.6 (Decidability of type checking in $\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}$) *For any context* $\nabla\,|\,\Gamma$, *and for any term* e *and type* σ, *it is decidable whether* $\Gamma \vdash_{\lambda^{\mathsf{a}}_{\mathsf{CS+fun}}} e : \sigma$ *is derivable.*

Proof. Immediate from Theorem 9.5.5. □

Decidability of type-checking in $\lambda_{\mathsf{CS+fun}}$ follows.

Corollary 9.5.7 (Decidability of type checking in $\lambda_{\mathsf{CS+fun}}$) *For any context* $\nabla\,|\,\Gamma$, *and for any term* e *and type* σ, *it is decidable whether* $\nabla\,|\,\Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \sigma$ *is derivable.*

Proof. It is an immediate consequence of Corollary 9.5.6, Lemma 8.5.12 and of the fact that the set $\mathsf{an}_\nabla(e)$ is finite. □

9.6 Decidability of Well-Formedness for Environments

An environment to be well-formed has to be well-typed, non-overlapping and exhaustive. In this section we prove that these requirements are decidable properties.

9.6.1 Decidability of Well-Typing

An environment Σ is well-typed if, for every $f \in \mathsf{FS}_\Sigma$ and $\tau_1 \to \tau_2 \in \mathsf{Ty}_\Sigma(f)$

1. $\exists\, \lambda p.e \in \mathsf{Ru}_\Sigma(f).\exists \Gamma.\ . \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} p : \tau_1$
2. $\forall\, \lambda p.e \in \mathsf{Ru}_\Sigma(f).\forall \Gamma.\ . \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+fun}}} p : \tau_1 \;\Rightarrow\; \mathsf{er}(\Sigma) \,|\, \Sigma \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau_2$

The decidability of this property relies on the following facts:

– FS_Σ, $\mathsf{Ru}_\Sigma(f)$ and $\mathsf{Ty}_\Sigma(f)$ are finite sets;

– we have minimal variable context for typing a pattern with a given type; and

– type-checking is decidable for $\lambda_{\mathsf{CS+fun}}$.

Theorem 9.6.1 (Decidability of well-typing) *It is decidable whether an environment Σ is well-typed or not.*

Proof. For each $f \in \mathsf{FS}_\Sigma$ and $\tau_1 \to \tau_2 \in \mathsf{Ty}_\Sigma(f)$:

1. $\mathsf{mvc}(p : \tau_1)$ has to be well-defined for at least one $\lambda p.e \in \mathsf{Ru}_\Sigma(f)$;
2. for each $\lambda p.e \in \mathsf{Ru}_\Sigma(f)$, whenever $\mathsf{mvc}(p : \tau_1)$ is well-defined, check if $\mathsf{er}(\Sigma) \,|\, \mathsf{mvc}(p : \tau_1) \vdash_{\lambda_{\mathsf{CS+fun}}} e : \tau_2$ is derivable.

As FS_Σ, $\mathsf{Ru}_\Sigma(f)$ and $\mathsf{Ty}_\Sigma(f)$ are finite sets, we have here a decision procedure. □

9.6.2 Decidability of Non-Overlapping

An environment Σ is non-overlapping if, for every $f \in \mathsf{FS}_\Sigma$ and distinct $\lambda p.e$, $\lambda p'.e' \in \mathsf{Ru}_\Sigma(f)$ there are no substitutions S and S' such that $S(p) = S'(p')$. We rely the test of this property on the test of whether two patterns are overlapping or not.

Definition 9.6.2 *Two patterns $p, p' \in \mathcal{P}$ are* overlapping *if there are substitutions S, S' such that $S(p) = S'(p')$.*

Definition 9.6.3 *Let $p, p' \in \mathcal{P}$. The predicate* $\mathsf{overlap}(p, p')$ *is derivable using the rules of Figure 9.5.*

Theorem 9.6.4 *Let $p, p' \in \mathcal{P}$.* $\mathsf{overlap}(p, p')$ *iff p and p' are overlapping.*

Proof.

$$1.\ \frac{}{\mathsf{overlap}(x,p)} \qquad 2.\ \frac{}{\mathsf{overlap}(p,x)} \qquad 3.\ \frac{\mathsf{overlap}(p_i,p'_i) \quad (1 \le i \le \mathsf{ar}(c))}{\mathsf{overlap}(c\,p_1 \ldots p_{\mathsf{ar}(c)}, c\,p'_1 \ldots p'_{\mathsf{ar}(c)})}$$

Figure 9.5: Rules for $\mathsf{overlap}$

$\Rightarrow$) By induction on the derivation of $\mathsf{overlap}(p,p')$. The only interesting case is when the last rule applied is rule 3. So, assume $\mathsf{overlap}(c\,p_1 \ldots p_{\mathsf{ar}(c)}, c\,p'_1 \ldots p'_{\mathsf{ar}(c)})$ comes from $\mathsf{overlap}(p_i,p'_i)$ for $i = 1..\mathsf{ar}(c)$. By induction hypothesis $\exists S_i, S'_i.\ S_i(p_i) = S'_i(p'_i)$. Define S and S' as follows:

$$S(x) = \begin{cases} S_i(x) & \text{if } x \in \mathsf{FV}(p_i) \\ x & \text{otherwise} \end{cases} \qquad S'(x) = \begin{cases} S'_i(x) & \text{if } x \in \mathsf{FV}(p'_i) \\ x & \text{otherwise} \end{cases}$$

Note that as we are working with linear patterns, S and S' are well-defined. It is now easy to see that $S(c\,p_1 \ldots p_{\mathsf{ar}(c)}) = c\,S_1(p_1) \ldots S_{\mathsf{ar}(c)}(p_{\mathsf{ar}(c)}) = c\,S'_1(p'_1) \ldots S'_{\mathsf{ar}(c)}(p'_{\mathsf{ar}(c)}) = S'(c\,p'_1 \ldots p'_{\mathsf{ar}(c)})$

$\Leftarrow$) Assume p and p' are overlapping, i.e., there are substitutions S, S' such that $S(p) = S'(p')$. The proof that $\mathsf{overlap}(p,p')$ is derivable follows by routine induction on the structure of p.

$\square$

The decidability of non-overlapping environments follows now as a corollary of this theorem.

Corollary 9.6.5 (Decidability of non-overlapping) *It is decidable whether an environment Σ is non-overlapping or not.*

Proof. It is an immediate consequence of Theorem 9.6.4 and of the fact that Σ and the set of Σ-rewritings for each function is finite. $\square$

9.6.3 Decidability of Exhaustiveness

An environment Σ is exhaustive if, for every $f \in \mathsf{FS}_\Sigma$, $\tau_1 \to \tau_2 \in \mathsf{Ty}_\Sigma(f)$ and for every quasi-closed pattern w.r.t. τ_1, p, there exists $\lambda p'.e' \in \mathsf{Ru}_\Sigma(f)$ such that $p = S(p')$ for some substitution S.

We begin by introducing the notion of exhaustive for a set of patterns and for a set of lists of patterns. Let us establish a notation for lists.

Notation 9.6.6 *We use the following notation for lists: $[]$ is the empty list; a list of length n, $[a_1, a_2, \ldots, a_n]$, can also be written as $a_1 :: [a_2, \ldots, a_n]$ or $a_1 :: a_2 :: \ldots :: a_n :: []$. The concatenation of lists is done by the infix operator $\bowtie$.*

Definition 9.6.7

1. *A set of patterns $p_1, \ldots, p_n$ is* τ-exhaustive *if for all q, quasi-closed patterns w.r.t. τ, there is a p_i, for $1 \le i \le n$, such that $q = S(p_i)$ for some substitution S.*

1.
$$\frac{}{\mathsf{exh}_{[]}\{[]\}}$$

2.
$$\frac{\mathsf{exh}_{\vec{\sigma}}\{\overrightarrow{q_1}, \ldots, \overrightarrow{q_n}\}}{\mathsf{exh}_{\tau::\vec{\sigma}}\{x_1 :: \overrightarrow{q_1}, \ldots, x_n :: \overrightarrow{q_n}\} \cup P}$$

3.
$$\frac{\begin{array}{l}\text{for } 1 \leq i \leq n,\\ \mathsf{exh}_{\mathsf{Dom}^{\vec{\tau}}_{d}(c_i) \bowtie \vec{\sigma}}\{\, \overrightarrow{p_{i,1}} \bowtie \overrightarrow{q_{i,1}}, \ldots, \overrightarrow{p_{i,k_i}} \bowtie \overrightarrow{q_{i,k_i}}, y_{1,1} :: \ldots :: y_{1,\mathsf{ar}(c_i)} :: \overrightarrow{q_1}, \ldots, y_{m,1} :: \ldots :: y_{m,\mathsf{ar}(c_i)} :: \overrightarrow{q_m} \,\}\end{array}}{\begin{array}{ll}\mathsf{exh}_{d\vec{\tau}::\vec{\sigma}}\{ & (c_1\, \overrightarrow{p_{1,1}}) :: \overrightarrow{q_{1,1}}, \ldots, (c_1\, \overrightarrow{p_{1,k_1}}) :: \overrightarrow{q_{1,k_1}}, \ldots,\\ & (c_r\, \overrightarrow{p_{r,1}}) :: \overrightarrow{q_{r,1}}, \ldots, (c_r\, \overrightarrow{p_{r,k_r}}) :: \overrightarrow{q_{r,k_r}}, y_1 :: \overrightarrow{q_1}, \ldots, y_m :: \overrightarrow{q_m} \quad \} \cup P\end{array}}$$

if $\emptyset \neq \{c_1, \ldots, c_r\} \subseteq \mathsf{C}(d) = \{c_1, \ldots, c_n\}$

Figure 9.6: Rules for exh

2. *Let P be a set of lists of length n of patterns and $\vec{\sigma}$ be a list of types of length n. P is* $\vec{\sigma}$-exhaustive *if, for every quasi-closed patterns $q_1, \ldots, q_n$ w.r.t. $\sigma_1, \ldots, \sigma_n$ respectively*

$$\exists \vec{p} \in P\ \exists S.\ S(p_1) = q_1 \wedge \ldots \wedge S(p_n) = q_n$$

Lemma 9.6.8

1. *If $P \subseteq P'$ and P is $\vec{\sigma}$-exhaustive, then P' is $\vec{\sigma}$-exhaustive.*

2. *Let P be a set of lists of length n of patterns, with $n > 0$, and $Q = \{\vec{p} \mid p :: \vec{p} \in P\}$. If P is $\tau :: \vec{\sigma}$-exhaustive, then Q is $\vec{\sigma}$-exhaustive*

Proof. It is an immediate consequence of Definition 9.6.7. □

Definition 9.6.9 *Let $\vec{\sigma}$ be a list of types and P a set of lists of length $\#\vec{\sigma}$ of patterns. The predicate $\mathsf{exh}_{\vec{\sigma}} P$ holds if and only if it is derivable using the rules of Figure 9.6.*

Definition 9.6.10 (Size of a pattern) *The size of a pattern p, written $\mathsf{size}(p)$, is defined as follows:*

$$\begin{array}{rcl}\mathsf{size}(x) & = & 0\\ \mathsf{size}(c\, p_1 \ldots p_{\mathsf{ar}(c)}) & = & 1 + \mathsf{size}(p_1) + \ldots + \mathsf{size}(p_{\mathsf{ar}(c)})\end{array}$$

The size of a list of patterns is defined by:

$$\begin{array}{rcl}\mathsf{size}([]) & = & 0\\ \mathsf{size}(p :: \vec{p}) & = & \mathsf{size}(p) + \mathsf{size}(\vec{p})\end{array}$$

The size of a set of lists of patterns is defined by:

$$\begin{array}{rcl}\mathsf{size}(\emptyset) & = & 0\\ \mathsf{size}(\{\vec{p_1}, \ldots, \vec{p_n}\}) & = & \mathsf{size}(\vec{p_1}) + \ldots + \mathsf{size}(\vec{p_n})\end{array}$$

Theorem 9.6.11 *Let $\vec{\gamma}$ be a list of types and P a set of lists of length $\#\vec{\gamma}$ of patterns. Then P is $\vec{\gamma}$-exhaustive if and only if $\mathsf{exh}_{\vec{\gamma}}P$ is derivable.*

Proof.

$\Leftarrow$) By induction on the derivation of $\mathsf{exh}_{\vec{\gamma}}P$.

1. Assume the last step is: $\mathsf{exh}_{[]}\{[]\}$. Trivially, $\{[]\}$ is $[]$-exhaustive.
2. Assume the last step is:

$$\frac{\mathsf{exh}_{\vec{\sigma}}\{\vec{q_1},\dots,\vec{q_n}\}}{\mathsf{exh}_{\tau:\vec{\sigma}}\{x_1 : \vec{q_1},\dots,x_n : \vec{q_n}\}\cup P'}$$

By induction hypothesis $\{\vec{q_1},\dots,\vec{q_n}\}$ is $\vec{\sigma}$-exhaustive. So, for every $p_1,\dots,p_{\#\vec{\sigma}}$ quasi-closed patterns w.r.t. $\vec{\sigma}$, $S(\vec{q_k}) = \vec{p}$ for some $k \in \{1,...,n\}$ and for some substitution S. But then, for every $p_0,p_1,\dots,p_{\#\vec{\sigma}}$ quasi-closed patterns w.r.t. $\tau : \vec{\sigma}$, there is a substitution R such that $R(x_k : \vec{q_k}) = p_0 : \vec{p}$. R is defined as follows:

$$R(x) = \begin{cases} S(x) & \text{if } x \in \vec{q_k} \\ p_0 & \text{if } x = x_k \\ x & \text{otherwise} \end{cases}$$

Hence $\{x_1 : \vec{q_1},\dots,x_n : \vec{q_n}\}$ is $(\tau : \vec{\sigma})$-exhaustive, and we conclude by item 1 of Lemma 9.6.8.

3. Assume the last step is:

$$\frac{\mathsf{exh}_{\mathsf{Dom}^{\vec{\tau}}_d(c_i)\bowtie\vec{\sigma}}A_i \quad \text{for } 1 \le i \le n}{\mathsf{exh}_{d\vec{\tau}\vec{\sigma}}B \cup P'}$$

with $A_i = \left\{\overrightarrow{p_{i,1}} \bowtie \overrightarrow{q_{i,1}},\dots,\overrightarrow{p_{i,k_i}} \bowtie \overrightarrow{q_{i,k_i}}, y_{1,1} : \dots : y_{1,\mathsf{ar}(c_i)} : \vec{q_1},\dots,y_{m,1} : \dots : y_{m,\mathsf{ar}(c_i)} : \overrightarrow{q_m}\right\}$, $B = \left\{(c_1\,\overrightarrow{p_{1,1}}) : \overrightarrow{q_{1,1}},\dots,(c_1\,\overrightarrow{p_{1,k_1}}) : \overrightarrow{q_{1,k_1}},\dots,(c_r\,\overrightarrow{p_{r,1}}) : \overrightarrow{q_{r,1}},\dots,(c_r\,\overrightarrow{p_{r,k_r}}) : \overrightarrow{q_{r,k_r}}, y_1 : \vec{q_1},\dots,y_m : \overrightarrow{q_m}\right\}$ and $\emptyset \neq \{c_1,\dots,c_r\} \subseteq \mathsf{C}(d) = \{c_1,\dots,c_n\}$.

By induction hypothesis each A_i is $(\mathsf{Dom}^{\vec{\tau}}_d(c_i) \bowtie \vec{\sigma})$-exhaustive. Thus, for every quasi-closed patterns $\vec{e}$ w.r.t. $\mathsf{Dom}^{\vec{\tau}}_d(c_i) \bowtie \vec{\sigma}$, there exists $\vec{p} \in A_i$ such that $S(\vec{p}) = \vec{e}$ for some substitution S.

Our goal is to prove that for every quasi-closed patterns $a_0 : \vec{a}$ w.r.t. $d\,\vec{\tau} : \vec{\sigma}$, there exists $\vec{q} \in B$ and a substitution R such that $R(\vec{q}) = a_0 : \vec{a}$. And we have $a_0 \equiv c_i\,\vec{b}$ with $\vec{b}$ being quasi-closed patterns w.r.t. $\mathsf{Dom}^{\vec{\tau}}_d(c_i)$. We have two alternatives for a list of patterns $\vec{p} \in A_i$:

(a) $\vec{p} \equiv \overrightarrow{p_{i,s}} \bowtie \overrightarrow{q_{i,s}}$ with $1 \le s \le k_i$. In this case, there is $\vec{q} \equiv (c_i\,\overrightarrow{p_{i,s}}) : \overrightarrow{q_{i,s}} \in B$ and by induction hypothesis there is a substitution S such that $S(\overrightarrow{p_{i,s}}) = \vec{b}$ and $S(\overrightarrow{q_{i,s}}) = \vec{a}$. So, $S(\vec{q}) = a_0 : \vec{a}$.

(b) $\vec{p} \equiv y_{s,1} : \dots : y_{s,\mathsf{ar}(c_i)} : \vec{q_s}$ with $1 \le s \le m$. In this case, there is $\vec{q} \equiv y_s : \overrightarrow{q_{i,s}} \in B$ and a substitution R such that $R(\vec{q}) = a_0 : \vec{a}$. Such a R is defined as follows

$$R(x) = \begin{cases} a_0 & \text{if } x = y_k \\ S(x) & \text{otherwise} \end{cases}$$

We have proved that B is $(d\,\vec{\tau} : \vec{\sigma})$-exhaustive. Now we conclude by item 1 of Lemma 9.6.8.

$\Rightarrow$) By induction on the definition of P is $\vec{\gamma}$-exhaustive, using as induction measure the lexicographic ordered pair $(\mathsf{size}(P), \#\vec{\sigma})$. Assume P is $\vec{\gamma}$-exhaustive.

1. Case $\vec{\gamma} \equiv []$. Then $P \equiv \{[]\}$ and we conclude by rule 1.
2. Case $\vec{\gamma} \equiv \alpha : \vec{\sigma}$ or $\vec{\gamma} \equiv \tau_1 \to \tau_2 : \vec{\sigma}$. Let $Q = \{\vec{p} \mid p : \vec{p} \in P\}$. By Lemma 9.6.8, Q is $\vec{\sigma}$-exhaustive so, by induction hypothesis, $\mathsf{exh}_{\vec{\sigma}}Q$ is derivable. Because P is $\vec{\gamma}$-exhaustive, for every $q_0 : \vec{q}$ quasi-closed pattern w.r.t. $\vec{\gamma}$, there exists $p_0 : \vec{p} \in P$ such that $S(p_0 : \vec{p}) = q_0 : \vec{q}$ for some S. As q_0 has to be a variable, we know that p_0 has also to be a variable. Hence $\mathsf{exh}_{\vec{\gamma}}P$ is derivable from $\mathsf{exh}_{\vec{\sigma}}Q$ by rule 2.
3. Case $\vec{\gamma} \equiv d\,\vec{\tau} : \vec{\sigma}$. Because P is $\vec{\gamma}$-exhaustive, for every $q_0 : \vec{q}$ quasi-closed patterns w.r.t. $d\,\vec{\tau} : \vec{\sigma}$, there exists $p_0 : \vec{p} \in P$ such that $S(p_0 : \vec{p}) = q_0 : \vec{q}$ for some S. Moreover $q_0 \equiv c\,\vec{e}$ with $c \in \mathsf{C}(d)$ and $\vec{e}$ quasi-closed patterns w.r.t. $\mathsf{Dom}_d^{\vec{\tau}}(c)$. Therefore, p_0 is either a variable or an applied constructor $c\,\vec{b}$ and $S(\vec{b}) = \vec{e}$.

   Let $Q_i = \{\, p_0 : \vec{p} \in P \mid \exists\, S.\ S(p_0 : \vec{p}) = (c_i\,\vec{e}) : \vec{q}$ for $c_i \in \mathsf{C}(d)$ and $\vec{e}$ quasi-closed patterns w.r.t. $\mathsf{Dom}_d^{\vec{\tau}}(c_i)$ and $\vec{q}$ quasi-closed patterns w.r.t. $\vec{\sigma}\}$. Let

   $$\begin{array}{rcl} Q_i' & = & \{x_1 : \ldots : x_{\mathsf{ar}(c_i)} : \vec{p} \mid x : \vec{p} \in Q_i,\ \text{with } x_1, \ldots, x_{\mathsf{ar}(c_i)} \text{ fresh}\} \\ & & \cup \{b_1 : \ldots : b_{\mathsf{ar}(c_i)} : \vec{p} \mid (c_i\, b_1 \ldots b_{\mathsf{ar}(c_i)}) : \vec{p} \in Q_i\} \end{array}$$

   We have that Q_i' is $(\mathsf{Dom}_d^{\vec{\tau}}(c_i) \bowtie \vec{\sigma})$-exhaustive and we have also that $\mathsf{size}(Q_i') < \mathsf{size}(P)$. Then, by induction hypothesis $\mathsf{exh}_{\mathsf{Dom}_d^{\vec{\tau}}(c_i) \bowtie \vec{\sigma}} Q_i'$ is derivable, for each $i = 1..n$. Now we can use rule 3 to derive $\mathsf{exh}_{d\,\vec{\tau}\vec{\sigma}}P$.

□

Theorem 9.6.12 *Given a set of patterns $\{p_1, \ldots, p_n\}$ and a type σ it is decidable whether $\{p_1, \ldots, p_n\}$ is σ-exhaustive or not.*

Proof. By Theorem 9.6.11 it is sufficient to decide whether $\mathsf{exh}_{[\sigma]}\{[p_1], \ldots, [p_n]\}$ is derivable. And this is easy, since the definition of exh is well-founded: the induction measure for $\mathsf{exh}_{\vec{\sigma}}P$ is the lexicographically ordered pair $(\mathsf{size}(P), \#\vec{\sigma})$. □

The decidability of exhaustive environments follows now as a corollary of this theorem.

Corollary 9.6.13 (Decidability of exhaustiveness) *It is decidable whether an environment Σ is exhaustive or not.*

Proof. It is an immediate consequence of Theorem 9.6.12 and of the fact that FS_Σ is finite and the sets of Σ-rewritings and Σ-typings for each function are finite. □

9.7 The System $\lambda_{\mathsf{CS+def}}$

The simplest way to think of pattern-matching is as trying to match each equation in turn. We want to compile function definitions with pattern-matching into case-expressions which can be evaluated more efficiently. In this section we define the target language $\lambda_{\mathsf{CS+def}}$, a simply typed λ-calculus with constructor subtyping, case-expressions and recursive definitions. The translation from $\lambda_{\mathsf{CS+fun}}$ to $\lambda_{\mathsf{CS+def}}$ will be described in Section 9.8.

9.7.1 Types and Terms

The language $\lambda_{\mathsf{CS+def}}$ is a mild variant of $\lambda_{\mathsf{CS+fun}}$. The set $\mathcal{T}_{\mathsf{def}}$ of types of $\lambda_{\mathsf{CS+def}}$ is equal to $\mathcal{T}_{\mathsf{fun}}$.

Definition 9.7.1 (Types) *The set $\mathcal{T}_{\mathsf{def}}$ of* types *is given by the abstract syntax:*

$$\mathcal{T}_{\mathsf{def}} \ni \tau, \sigma \quad ::= \quad \alpha \mid \tau \rightarrow \sigma \mid d\,\vec{\tau}$$

where in the last clause, it is assumed that $\#\vec{\tau} = \mathsf{ar}(d)$.

The set of terms of $\lambda_{\mathsf{CS+def}}$ is obtained from that of $\lambda_{\mathsf{CS+fun}}$ by replacing definitions by pattern-matching by case-expressions and recursive function definitions. Note that we do not have an explicit construct for recursive definitions. Indeed, recursive definitions will be treated as global, and stored in the environment.

Definition 9.7.2 (Terms) *The set $\mathcal{E}_{\mathsf{def}}$ of terms is given by the abstract syntax:*

$$\mathcal{E}_{\mathsf{def}} \ni a, b \quad ::= \quad x \mid f \mid c \mid \lambda x.\, a \mid a\, b \mid \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\}$$

where in the clause for case-expressions it is assumed that $\mathsf{C}(d) \subseteq \{c_1, \ldots, c_n\}$ for some $d \in \mathcal{D}$ and that all c_i's are pairwise distinct.

9.7.2 Subtyping and Typing

The subtyping relation of $\lambda_{\mathsf{CS+def}}$ is the same defined for $\lambda_{\mathsf{CS+fun}}$ (see Figure 9.1). The typing rules are extended with a new rule for case-expressions.

Definition 9.7.3 (Typing) *The typing rules for $\lambda_{\mathsf{CS+def}}$ are those of Figure 9.2 extended with following rule for case-expressions:*

$$(\text{case})\quad \frac{\begin{array}{l}\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} a : d\,\vec{\tau} \\ \nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} b_i : \mathsf{Dom}^{\vec{\tau}}_{d}(c_i) \rightarrow \sigma \qquad (1 \leq i \leq n) \\ \nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} b_j : \theta_j \qquad (n+1 \leq j \leq n+m)\end{array}}{\nabla \mid \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_{n+m} \Rightarrow b_{n+m}\} : \sigma} \quad \textit{if}\ \ \mathsf{C}(d) = \{c_1, \ldots, c_n\}$$

Note how the (case) rule allows to type case-expressions with more branches than the number of constructors of the datatype. This slightly unusual (case) rule is necessary to take care of overloading. For example, consider the following declaration:

$$\begin{array}{rcl}\mathrm{add} & : & \{\mathsf{Even} \rightarrow \mathsf{Even} \rightarrow \mathsf{Even},\ \mathsf{Odd} \rightarrow \mathsf{Even} \rightarrow \mathsf{Odd}\} \\ & = & \lambda x. \lambda y.\, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda z.\, \mathsf{s}\,(\mathrm{add}\, z\, y)\}\end{array}$$

is valid since the rule for case allows to type the case-expression with the type Odd for $x : \mathsf{Odd}$ (recall that $\mathsf{C}(\mathsf{Odd}) = \{\mathsf{s}\}$).

9.7.3 Environments

Functions are now defined by a single expression; still a function symbol is allowed to have several types.

Definition 9.7.4 (Environments)

1. *A* function definition *is a triple $h : \vec{\sigma} = e$ where $h \in \mathcal{F}$ is a function symbol, $\vec{\sigma}$ is a set of types and $e \in \mathcal{E}_{\mathsf{def}}$.*

2. *An* environment[1] *is a finite set of function definitions, where each function symbol is declared at most once. We let $\mathcal{K}$ denote the set of environments and Φ, Φ' range over $\mathcal{K}$.*

Each environment induces a function context. Environments are sets of function declarations which may be mutually recursive. An essential difference between the environments of $\lambda_{\mathsf{CS+def}}$ and those of $\lambda_{\mathsf{CS+fun}}$ is that in $\lambda_{\mathsf{CS+def}}$-environments a function is declared only once.

Definition 9.7.5 (Erasure) *The erasure function* $\mathsf{er} : \mathcal{K} \rightarrow \aleph_{\mathcal{F}}$ *is defined inductively as follows:*

$$\begin{array}{rcl} \mathsf{er}(\emptyset) & = & \emptyset \\ \mathsf{er}(\Phi, h : \{\sigma_1, \ldots, \sigma_n\} = e) & = & \mathsf{er}(\Phi), h : \sigma_1, \ldots, h : \sigma_n \end{array}$$

Definition 9.7.6 (Well-formed environment) *An environment $\Phi \equiv h_1 : \vec{\sigma_1} = e_1, \ldots, h_n : \vec{\sigma_n} = e_n$ is* well-formed *if, for every h_i and $\sigma_{i,j} \in \vec{\sigma_i}$, $\sigma_{i,j}$ is of the form $\sigma'_{i,j} \rightarrow \sigma''_{i,j}$ and* $\mathsf{er}(\Phi) \,|\, . \vdash_{\lambda_{\mathsf{CS+def}}} e_i : \sigma_{i,j}$.

Let us give a small example of a well-formed $\lambda_{\mathsf{CS+def}}$-environment.

Example 9.7.7 *Bellow we declare the function* double *in a well-formed $\lambda_{\mathsf{CS+def}}$-environment.*

$$\begin{array}{rcl} \mathrm{double} & : & \{\mathsf{Even} \rightarrow \mathsf{Even},\ \mathsf{Odd} \rightarrow \mathsf{Even},\ \mathsf{Nat} \rightarrow \mathsf{Nat}\} \\ & = & \lambda x.\, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \mid \mathsf{s} \Rightarrow \lambda y.\, \mathsf{s}\,(\mathsf{s}\,(\mathrm{double}\, y))\} \end{array}$$

The decidability of well-formedness for $\lambda_{\mathsf{CS+def}}$-environments is stated in Subsection 9.7.5.

9.7.4 Reduction Calculus

The computational behavior of case-expressions and definitions is given by ι and δ-reduction respectively.

Definition 9.7.8 (Reduction Calculus)

1. *β-reduction $\rightarrow_\beta$ is defined as the compatible closure of the rule*

$$(\lambda x.\, a)\, b \ \rightarrow_\beta \ a[x := b]$$

2. *ι-reduction $\rightarrow_\iota$ is defined as the compatible closure of the rule*

$$\mathsf{case}\ (c_i\, \vec{a})\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\} \ \rightarrow_\iota \ b_i\, \vec{a}$$

where $\vec{a}$ represents a vector of terms whose length is exactly $\mathsf{ar}(c_i)$ and $1 \leq i \leq n$.

3. *Let $h : \vec{\sigma} = e$ be a function definition. $\delta(h : \vec{\sigma} = e)$-reduction $\rightarrow_{\delta(h:\vec{\sigma}=e)}$ is defined as the compatible closure of the rule*

$$h\, a \ \rightarrow_{\delta(h:\vec{\sigma}=e)} \ e\, a$$

4. *Let $\Phi \equiv h_1 : \vec{\sigma_1} = e_1, \ldots, h_n : \vec{\sigma_n} = e_n$ be an environment. δ_Φ-reduction $\rightarrow_{\delta_\Phi}$ is defined as*

$$\bigcup_{1 \leq i \leq n} \rightarrow_{\delta(h_i:\vec{\sigma_i}=e_i)}$$

[1]We shall talk about $\lambda_{\mathsf{CS+fun}}$-environments and $\lambda_{\mathsf{CS+def}}$-environments when there are risks of confusion.

5. *Let Φ be an environment. $\beta\iota\delta_\Phi$-reduction $\rightarrow_{\beta\iota\delta_\Phi}$ is defined as $\rightarrow_\beta \cup \rightarrow_\iota \cup \rightarrow_{\delta_\Phi}$. $\twoheadrightarrow_{\beta\iota\delta_\Phi}$ and $=_{\beta\iota\delta_\Phi}$ are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of $\rightarrow_{\beta\iota\delta_\Phi}$.*

The mechanics of this reduction calculus is illustrated by the example below.

Example 9.7.9 *Assume Φ is the environment define in Example 9.7.7. Next is a reduction sequence to compute the double of one:*

$$\begin{array}{lll} \text{double}\,(\mathsf{s}\,\mathsf{o}) & \rightarrow_{\delta_\Phi} & (\lambda x.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \mid \mathsf{s} \Rightarrow \lambda y.\,\mathsf{s}\,(\mathsf{s}\,(\text{double}\,y))\})\,(\mathsf{s}\,\mathsf{o}) \\ & \rightarrow_\beta & \mathsf{case}\ (\mathsf{s}\,\mathsf{o})\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \mid \mathsf{s} \Rightarrow \lambda y.\,\mathsf{s}\,(\mathsf{s}\,(\text{double}\,y))\} \\ & \rightarrow_\iota & (\lambda y.\,\mathsf{s}\,(\mathsf{s}\,(\text{double}\,y)))\,\mathsf{o} \\ & \rightarrow_\beta & \mathsf{s}\,(\mathsf{s}\,(\text{double}\,\mathsf{o})) \\ & \twoheadrightarrow_{\beta\iota\delta_\Phi} & \mathsf{s}\,(\mathsf{s}\,\mathsf{o}) \end{array}$$

9.7.5 Meta-Theoretical Properties of $\lambda_{\mathsf{CS+def}}$

Here we summarize the main meta-theoretical of the $\lambda_{\mathsf{CS+def}}$ system.

Theorem 9.7.10 (Confluence) *Let Φ be a $\lambda_{\mathsf{CS+def}}$-environment. Then $\rightarrow_{\beta\iota\delta_\Phi}$ is confluent:*

$$a_1 =_{\beta\iota\delta_\Phi} a_2 \quad\Rightarrow\quad \exists\, e \in \mathcal{E}_{\mathsf{def}}.\ a_1 \twoheadrightarrow_{\beta\iota\delta_\Phi} e \ \wedge\ a_2 \twoheadrightarrow_{\beta\iota\delta_\Phi} e$$

Proof. By the standard technique of Tait and Martin-Löf. □

The proof of subject reduction for $\lambda_{\mathsf{CS+def}}$ is quite standard and simpler than the proof done for $\lambda_{\mathsf{CS+fun}}$.

Theorem 9.7.11 (Subject reduction) *Let Φ be a well-formed $\lambda_{\mathsf{CS+def}}$-environment and let $\nabla =$ $\mathsf{er}(\Phi)$. Then*

$$\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} a : \sigma \quad\wedge\quad a \rightarrow_{\beta\delta_\Sigma} a' \quad\Rightarrow\quad \nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} a' : \sigma$$

Proof. By induction on the derivation of $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} a : \sigma$, using generation and substitution lemmas for $\lambda_{\mathsf{CS+def}}$. □

Theorem 9.7.12 (Decidability of type checking in $\lambda_{\mathsf{CS+def}}$) *For any context $\nabla \,|\, \Gamma$, and for any term e and type σ, it is decidable whether $\nabla \,|\, \Gamma \vdash_{\lambda_{\mathsf{CS+def}}} e : \sigma$ is derivable.*

Proof. The proof is absolutely the same as the one of sections 9.5 and 8.5. Only slight modifications are needed to treat the new case-expressions (with more branches): one on the annotation map, and the other in the definition of the algorithm typeJudg. The new clause for defining the annotations of case-expressions must be

$$\mathsf{an}_\nabla(\mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}) = \{\mathsf{case}_d\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b'}\} \mid a' \in \mathsf{an}_\nabla(a) \wedge b'_i \in \mathsf{an}_\nabla(b_i) \wedge \mathsf{C}(d) \subseteq \vec{c}\}$$

In the new definition of typeJudg for case-expressions, the local declaration of set B (see Figure 8.14) must now be $\mathsf{B} = \{\rho_i = \mathsf{Dom}_d^{\vec{\sigma}}(c_i) \rightarrow \beta \ \mid\ 1 \leq i \leq n \ \wedge\ c_i \in \mathsf{C}(d)\}$. □

Theorem 9.7.13 (Decidability of well-formedness for $\lambda_{\mathsf{CS+def}}$-environments) *It is decidable whether a $\lambda_{\mathsf{CS+def}}$-environment is well-formed or not.*

Proof. The result follows from the decidability of type checking in $\lambda_{\mathsf{CS+def}}$. $\square$

Obviously, if no restrictions are imposed on the definitions of recursive functions, the $\lambda_{\mathsf{CS+def}}$ calculus is not normalizing. To recover strong normalization, we have to extend to $\lambda_{\mathsf{CS+def}}$ the argument decreasing criterion as it was done in Subsection 8.4.1.

The idea is to complement the check that each declared function is well-typed with another check ensuring that the body of the function satisfies a syntactical condition $\mathcal{G}$ that constraints the occurrences of recursive calls. The guard predicate $\mathcal{G}$ is basically the same described in Figure 8.5 (Subsection 8.4.1); the only difference is that we do not have here letrec-expressions.

The predicate $\mathcal{G}$ enforces termination by constraining all recursive calls to be applied to terms smaller than the formal argument of the function. Therefore, under the assumption that all the functions declared in the $\lambda_{\mathsf{CS+def}}$-environment are well-typed and satisfy the guard predicate $\mathcal{G}$, one can prove that all typable expressions are strongly normalizing.

9.8 Compiling $\lambda_{\mathsf{CS+fun}}$ into $\lambda_{\mathsf{CS+def}}$

The compilation of pattern abstractions into case-expressions proceeds pretty much in the standard way, as described e.g. in [80]. First, all pattern abstractions defining a function f are collected from the environment. Then pattern abstractions are transformed into nested simple pattern abstractions (e.g. $\lambda c\, p.\, e$ is transformed into $\lambda c\, y.\, (\lambda p.e)\, y$). Finally, nested pattern abstractions are classified according to their head pattern and transformed into case-expressions.

Let us call $\mathsf{compile}$ the compilation function that transforms a function definition into an expression in $\mathcal{E}_{\mathsf{def}}$, and $\mathsf{translate}$ the function that translates a $\lambda_{\mathsf{CS+fun}}$-environment into a $\lambda_{\mathsf{CS+def}}$-environment. In order to define $\mathsf{translate}$ we need to introduce first the two auxiliary function: Filter_Σ and Eqs.

Definition 9.8.1 *Let Σ be an environment and f a function symbol. We define the set* $\mathsf{Filter}_\Sigma(f)$ *as follows:*

$$\mathsf{Filter}_\Sigma(f) \quad = \quad \{(g : \vec{\sigma} = \vec{r}) \in \Sigma \mid g \neq f\}$$

Definition 9.8.2 Eqs *is a function that takes a set of pattern abstractions and converts each pattern abstraction, $\lambda p.e$, in a pair consisting of the list $[p]$ and the expression e. It is defined as follows:*

$$\begin{array}{l}\mathsf{Eqs}\big(\emptyset\big) = \emptyset \\ \mathsf{Eqs}\big(R \cup \{\lambda p.\, e\}\big) = \mathsf{Eqs}\big(R\big) \cup \big\{\ ([p], e)\ \big\}\end{array}$$

We present an algorithm that translates a $\lambda_{\mathsf{CS+fun}}$-environment into a $\lambda_{\mathsf{CS+def}}$-environment. This algorithm was designed to work only with well-formed $\lambda_{\mathsf{CS+fun}}$-environments.

Definition 9.8.3 *The algorithm* $\mathsf{translate}$ *transforms an well-formed $\lambda_{\mathsf{CS+fun}}$-environment into a $\lambda_{\mathsf{CS+def}}$-environment, as defined in Figure 9.7.*

$\mathsf{translate}$ picks up a function f from the environment, collects all pattern abstractions defining it, makes a variable x be the formal argument of f and calls the compilation function $\mathsf{compile}$ with the list $[x]$ and the set of equations that defines f.

The fact that all functions defined in the environment are non-overlapping, exhaustive and well-typed makes possible to built a simple compilation function.

Definition 9.8.4 *The algorithm* $\mathsf{compile}$ *is defined in Figure 9.8.*

$$\mathsf{translate}\big(\emptyset\big) = \emptyset$$

$$\begin{aligned}\mathsf{translate}\left(\Sigma \cup \{f : \vec{\sigma} = \vec{r}\}\right) = \;& \mathsf{translate}\left(\mathsf{Filter}_\Sigma(f)\right) \cup \\ & \Big\{ f : \mathsf{Ty}_\Sigma(f) = \lambda x.\, \mathsf{compile}\Big([x]\ ,\ \mathsf{Eqs}\big(\mathsf{Ru}_\Sigma(f)\big)\Big) \Big\}\end{aligned}$$

Figure 9.7: The algorithm translate

$$\begin{aligned}
&\mathsf{compile}\ \Big((x :: \vec{x})\ ,\ \big\{(y_1 :: \overrightarrow{q_1}, e_1), \ldots, (y_n :: \overrightarrow{q_n}, e_n)\big\}\Big) \\
&\qquad = \mathsf{compile}\Big(\vec{x}\ ,\ \big\{(\overrightarrow{q_1}, e_1[y_1 := x]), \ldots, (\overrightarrow{q_n}, e_n[y_n := x])\big\}\Big) \\[1em]
&\mathsf{compile}\ \Big([]\ ,\ \big\{([], e)\big\}\Big) = e \\[1em]
&\mathsf{compile}\ \Big((x :: \vec{x})\ ,\ \big\{(y_1 :: \overrightarrow{q_1}, e_1), \ldots, (y_m :: \overrightarrow{q_m}, e_m)\big\} \cup \\
&\qquad\quad \big\{(c_1\, \overrightarrow{p_{1,1}} :: \overrightarrow{q_{1,1}}, e_{1,1}), \ldots, (c_1\, \overrightarrow{p_{1,k_1}} :: \overrightarrow{q_{1,k_1}}, e_{1,k_1})\big\} \\
&\qquad\quad \cup \ldots \cup \\
&\qquad\quad \big\{(c_n\, \overrightarrow{p_{n,1}} :: \overrightarrow{q_{n,1}}, e_{n,1}), \ldots, (c_n\, \overrightarrow{p_{n,k_n}} :: \overrightarrow{q_{n,k_n}}, e_{n,k_n})\big\}\Big) \\
&\qquad = \mathsf{case}\ x\ \mathsf{of}\ \{ \\
&\qquad\qquad c_1 \Rightarrow \lambda x_{1,1}.\ldots\lambda x_{1,\mathsf{ar}(c_1)}.\ \mathsf{compile}\ \Big((x_{1,1} :: \ldots :: x_{1,\mathsf{ar}(c_1)} :: \vec{x})\ , \\
&\qquad\qquad\qquad \big\{\ (x_{1,1} :: \ldots :: x_{1,\mathsf{ar}(c_1)} :: \overrightarrow{q_1}, e_1[y_1 := c_1\, x_{1,1} \ldots x_{1,\mathsf{ar}(c_1)}]), \ldots, \\
&\qquad\qquad\qquad\quad (x_{1,1} :: \ldots :: x_{1,\mathsf{ar}(c_1)} :: \overrightarrow{q_m}, e_m[y_m := c_1\, x_{1,1} \ldots x_{1,\mathsf{ar}(c_1)}])\ \big\} \cup \\
&\qquad\qquad\qquad \big\{\ (\overrightarrow{p_{1,1}} \bowtie \overrightarrow{q_{1,1}}, e_{1,1}), \ldots, (\overrightarrow{p_{1,k_1}} \bowtie \overrightarrow{q_{1,k_1}}, e_{1,k_1})\ \big\}\Big) \\
&\qquad\qquad \vdots \\
&\qquad\qquad \mid c_n \Rightarrow \lambda x_{n,1}.\ldots\lambda x_{n,\mathsf{ar}(c_n)}.\ \mathsf{compile}\ \Big((x_{n,1} :: \ldots :: x_{n,\mathsf{ar}(c_n)} :: \vec{x})\ , \\
&\qquad\qquad\qquad \big\{\ (x_{n,1} :: \ldots :: x_{n,\mathsf{ar}(c_n)} :: \overrightarrow{q_1}, e_1[y_1 := c_n\, x_{n,1} \ldots x_{n,\mathsf{ar}(c_n)}]), \ldots, \\
&\qquad\qquad\qquad\quad (x_{n,1} :: \ldots :: x_{n,\mathsf{ar}(c_n)} :: \overrightarrow{q_m}, e_m[y_m := c_n\, x_{n,1} \ldots x_{n,\mathsf{ar}(c_n)}])\ \big\} \cup \\
&\qquad\qquad\qquad \big\{\ (\overrightarrow{p_{n,1}} \bowtie \overrightarrow{q_{n,1}}, e_{n,1}), \ldots, (\overrightarrow{p_{n,k_n}} \bowtie \overrightarrow{q_{n,k_n}}, e_{n,k_n})\ \big\}\Big) \\
&\qquad\quad \}
\end{aligned}$$

Figure 9.8: The algorithm compile

The function compile takes two arguments: a list of variables and a set of equations. Each equation is a pair, consisting of a list of patterns and an expression. Note that the list of variables and the list of patterns in the equations have all the same length.

The compile function is inspired from the **match** function defined for the pattern-matching compiler algorithm described in [80], but substantial simplifications were done:

- compile takes a set of equations instead of a list of equations (as in **match**). This is due to the non-overlapping property.
- compile does not need to distinguish the case where only constructors appear in the beginning of every equation, from the case where some equations begin with constructors and others with variables (as it is done in [80]), because the guarantee of non-overlapping and exhaustiveness makes possible to treat this case in an uniform way. Moreover, we do not need to enrich the λ-calculus expressions with the operator $[\!]$ (the alternative) and with the special value ERROR.

Let us illustrate the compilation process with two small examples.

Example 9.8.5 *Assume we have a datatype* $\mathsf{B} \in \mathcal{D}$ *such that* $\mathsf{ar}(\mathsf{B}) = 0$, $\mathsf{C}(\mathsf{B}) = \{\mathsf{b}\}$, $\mathsf{ar}(\mathsf{b}) = 0$ *and* $\mathsf{D_B}(\mathsf{b}) = \mathsf{B}$. *Consider the following* $\lambda_{\mathsf{CS+fun}}$*-environment defining the overloaded function* f.

$$\begin{array}{rcl}
\mathsf{f} & : & \{\mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}\} \\
 & = & \{\lambda\langle x, \mathsf{s}\,y\rangle.\, \mathsf{f}\,\langle sx, y\rangle, \quad \lambda\langle \mathsf{s}\,z, \mathsf{o}\rangle.\, z, \quad \lambda\langle \mathsf{o}, \mathsf{o}\rangle.\, \mathsf{o}\} \\
\mathsf{f} & : & \{\mathsf{Nat} \times \mathsf{B} \to \mathsf{B}\} \\
 & = & \{\lambda\langle x, \mathsf{b}\rangle.\, \mathsf{b}\}
\end{array}$$

Below, we present an exhaustive description of the several steps of the compilation of function f.

We begin with:

$$\lambda x_1.\, \mathsf{compile}\big([x_1], \{([\langle x, \mathsf{s}\,y\rangle], \mathsf{f}\,\langle \mathsf{s}\,x, y\rangle), ([\langle \mathsf{s}\,z, \mathsf{o}\rangle], z), ([\langle \mathsf{o}, \mathsf{o}\rangle], \mathsf{o}), ([\langle x, \mathsf{b}\rangle], \mathsf{b})\}\big)$$

Step 1:

$$\begin{array}{l}
\lambda x_1.\, \mathsf{case}\ x_1\ \mathsf{of}\, \{ \\
\qquad \mathsf{pair} \Rightarrow \quad \lambda x_2.\lambda x_3.\ \mathsf{compile}\big([x_2, x_3], \{([x, \mathsf{s}\,y], \mathsf{f}\,\langle \mathsf{s}\,x, y\rangle), ([\mathsf{s}\,z, \mathsf{o}], z), ([\mathsf{o}, \mathsf{o}], \mathsf{o}), ([x, \mathsf{b}], \mathsf{b})\}\big)\}
\end{array}$$

Step 2:

$$\begin{array}{l}
\lambda x_1.\, \mathsf{case}\ x_1\ \mathsf{of}\, \{ \\
\qquad \mathsf{pair} \Rightarrow \quad \lambda x_2.\lambda x_3.\ \mathsf{case}\ x_2\ \mathsf{of}\, \{ \\
\qquad\qquad\qquad \mathsf{o} \Rightarrow \mathsf{compile}\big([x_3], \{([\mathsf{s}\,y], \mathsf{f}\,\langle \mathsf{s}\,\mathsf{o}, y\rangle), ([\mathsf{o}], \mathsf{o}), ([\mathsf{b}], \mathsf{b})\}\big) \\
\qquad\qquad\qquad \mathsf{s} \Rightarrow \lambda x_4.\, \mathsf{compile}\big([x_4, x_3], \{([x_4, \mathsf{s}\,y], \mathsf{f}\,\langle \mathsf{s}\,(\mathsf{s}\,x_4), y\rangle), ([z, \mathsf{o}], z), ([x_4, \mathsf{b}], \mathsf{b})\}\big) \\
\qquad\quad \} \qquad\qquad\qquad \}
\end{array}$$

Step 3:

$$\begin{array}{l}
\lambda x_1.\, \mathsf{case}\ x_1\ \mathsf{of}\, \{ \\
\qquad \mathsf{pair} \Rightarrow \quad \lambda x_2.\lambda x_3.\ \mathsf{case}\ x_2\ \mathsf{of}\, \{ \\
\qquad\qquad\qquad\qquad \mathsf{o} \Rightarrow \quad \mathsf{case}\ x_3\ \mathsf{of}\, \{ \\
\qquad\qquad\qquad\qquad\qquad \mathsf{o} \Rightarrow \mathsf{compile}\big([], \{([], \mathsf{o})\}\big) \\
\qquad\qquad\qquad\qquad\qquad \mathsf{s} \Rightarrow \lambda x_4.\, \mathsf{compile}\big([x_4], \{([y], \mathsf{f}\,\langle \mathsf{s}\,\mathsf{o}, y\rangle)\}\big) \\
\qquad\qquad\qquad\qquad\qquad \mathsf{b} \Rightarrow \mathsf{compile}\big([], \{([], \mathsf{b})\}\big)\ \} \\
\qquad\qquad\qquad\qquad \mathsf{s} \Rightarrow \quad \lambda x_4.\, \mathsf{compile}\big([x_3], \{([\mathsf{s}\,y], \mathsf{f}\,\langle \mathsf{s}\,(\mathsf{s}\,x_4), y\rangle), ([\mathsf{o}], x_4), ([\mathsf{b}], \mathsf{b})\}\big) \\
\qquad\quad \} \qquad\qquad\qquad \}
\end{array}$$

Step 4:

$$
\begin{array}{l}
\lambda x_1.\,\mathsf{case}\ x_1\ \mathsf{of}\ \{\ \mathsf{pair} \Rightarrow \lambda x_2.\lambda x_3.\ \mathsf{case}\ x_2\ \mathsf{of}\ \{\\
\qquad \mathsf{o} \Rightarrow\ \ \mathsf{case}\ x_3\ \mathsf{of}\ \{\\
\qquad\qquad \mathsf{o} \Rightarrow \mathsf{o}\\
\qquad\qquad \mathsf{s} \Rightarrow \lambda x_4.\,\mathsf{compile}\big([], \{([], \mathsf{f}\,\langle \mathsf{s}\,\mathsf{o}, x_4\rangle)\}\big)\\
\qquad\qquad \mathsf{b} \Rightarrow \mathsf{b}\ \}\\
\qquad \mathsf{s} \Rightarrow\ \ \lambda x_4.\mathsf{case}\ x_3\ \mathsf{of}\ \{\\
\qquad\qquad\qquad \mathsf{o} \Rightarrow \mathsf{compile}\big([], \{([], x_4)\}\big)\\
\qquad\qquad\qquad \mathsf{s} \Rightarrow \lambda x_5.\mathsf{compile}\big([x_5], \{([y], \mathsf{f}\,\langle \mathsf{s}\,(\mathsf{s}\,x_4), y\rangle)\}\big)\\
\qquad\qquad\qquad \mathsf{b} \Rightarrow \mathsf{compile}\big([], \{([], \mathsf{b})\}\big)\\
\qquad \}\qquad \}\quad \}
\end{array}
$$

Step 5:

$$
\begin{array}{l}
\lambda x_1.\,\mathsf{case}\ x_1\ \mathsf{of}\ \{\ \mathsf{pair} \Rightarrow \lambda x_2.\lambda x_3.\ \mathsf{case}\ x_2\ \mathsf{of}\ \{\\
\qquad \mathsf{o} \Rightarrow\ \ \mathsf{case}\ x_3\ \mathsf{of}\ \{\\
\qquad\qquad \mathsf{o} \Rightarrow \mathsf{o}\\
\qquad\qquad \mathsf{s} \Rightarrow \lambda x_4.\,\mathsf{f}\,\langle \mathsf{s}\,\mathsf{o}, x_4\rangle\\
\qquad\qquad \mathsf{b} \Rightarrow \mathsf{b}\ \}\\
\qquad \mathsf{s} \Rightarrow\ \ \lambda x_4.\mathsf{case}\ x_3\ \mathsf{of}\ \{\\
\qquad\qquad\qquad \mathsf{o} \Rightarrow x_4\\
\qquad\qquad\qquad \mathsf{s} \Rightarrow \lambda x_5.\mathsf{compile}\big([], \{([], \mathsf{f}\,\langle \mathsf{s}\,(\mathsf{s}\,x_4), x_5\rangle)\}\big)\\
\qquad\qquad\qquad \mathsf{b} \Rightarrow \mathsf{b}\ \}\\
\qquad \}\qquad \}
\end{array}
$$

Step 6:

$$
\begin{array}{l}
\lambda x_1.\,\mathsf{case}\ x_1\ \mathsf{of}\ \{\ \mathsf{pair} \Rightarrow \lambda x_2.\lambda x_3.\ \mathsf{case}\ x_2\ \mathsf{of}\ \{\\
\qquad \mathsf{o} \Rightarrow\ \ \mathsf{case}\ x_3\ \mathsf{of}\ \{\\
\qquad\qquad \mathsf{o} \Rightarrow \mathsf{o}\\
\qquad\qquad \mathsf{s} \Rightarrow \lambda x_4.\,\mathsf{f}\,\langle \mathsf{s}\,\mathsf{o}, x_4\rangle\\
\qquad\qquad \mathsf{b} \Rightarrow \mathsf{b}\ \}\\
\qquad \mathsf{s} \Rightarrow\ \ \lambda x_4.\mathsf{case}\ x_3\ \mathsf{of}\ \{\\
\qquad\qquad\qquad \mathsf{o} \Rightarrow x_4\\
\qquad\qquad\qquad \mathsf{s} \Rightarrow \lambda x_5.\,\mathsf{f}\,\langle \mathsf{s}\,(\mathsf{s}\,x_4), x_5\rangle\\
\qquad\qquad\qquad \mathsf{b} \Rightarrow \mathsf{b}\ \}\\
\qquad \}\qquad \}
\end{array}
$$

The corresponding $\lambda_{\mathsf{CS+def}}$-environment is

$$
\begin{array}{lll}
\mathsf{f} & : & \{\mathsf{Nat} \times \mathsf{Nat} \rightarrow \mathsf{Nat},\ \ \mathsf{Nat} \times \mathsf{B} \rightarrow \mathsf{B}\}\\
 & = & \lambda x_1.\,\mathsf{case}\ x_1\ \mathsf{of}\ \{\ \mathsf{pair} \Rightarrow \lambda x_2.\lambda x_3.\ \mathsf{case}\ x_2\ \mathsf{of}\ \{\\
 & & \qquad \mathsf{o} \Rightarrow\ \ \mathsf{case}\ x_3\ \mathsf{of}\ \{\\
 & & \qquad\qquad \mathsf{o} \Rightarrow \mathsf{o}\\
 & & \qquad\qquad \mathsf{s} \Rightarrow \lambda x_4.\,\mathsf{f}\,\langle \mathsf{s}\,\mathsf{o}, x_4\rangle\\
 & & \qquad\qquad \mathsf{b} \Rightarrow \mathsf{b}\ \}\\
 & & \qquad \mathsf{s} \Rightarrow\ \ \lambda x_4.\mathsf{case}\ x_3\ \mathsf{of}\ \{\\
 & & \qquad\qquad\qquad \mathsf{o} \Rightarrow x_4\\
 & & \qquad\qquad\qquad \mathsf{s} \Rightarrow \lambda x_5.\,\mathsf{f}\,\langle \mathsf{s}\,(\mathsf{s}\,x_4), x_5\rangle\\
 & & \qquad\qquad\qquad \mathsf{b} \Rightarrow \mathsf{b}\ \}\\
 & & \qquad \}\qquad \}
\end{array}
$$

Example 9.8.6 *The translation of the $\lambda_{\mathsf{CS+fun}}$-environment of Example 9.1.19 is*

$$
\begin{array}{lll}
\text{plus} & : & \{\mathsf{Even}\times\mathsf{Even}\to\mathsf{Even},\ \mathsf{Odd}\times\mathsf{Even}\to\mathsf{Odd},\ \mathsf{Even}\times\mathsf{Odd}\to\mathsf{Odd},\ \mathsf{Odd}\times\mathsf{Odd}\to\mathsf{Even},\\
 & & \ \mathsf{Nat}\times\mathsf{Nat}\to\mathsf{Nat},\ \mathsf{MaybeNat}\times\mathsf{MaybeNat}\to\mathsf{MaybeNat}\}\\
 & = & \lambda x_1.\,\mathsf{case}\ x_1\ \mathsf{of}\ \{\ \mathsf{pair}\Rightarrow\lambda x_2.\lambda x_3.\ \mathsf{case}\ x_2\ \mathsf{of}\ \{\\
 & & \qquad \mathsf{o}\Rightarrow\ \mathsf{case}\ x_3\ \mathsf{of}\ \{\\
 & & \qquad\qquad \mathsf{o}\Rightarrow\mathsf{o}\\
 & & \qquad\qquad \mathsf{s}\Rightarrow\lambda x_4.\,\mathsf{s}\,x_4\\
 & & \qquad\qquad \mathsf{undef}\Rightarrow\mathsf{undef}\ \}\\
 & & \qquad \mathsf{s}\Rightarrow\ \lambda x_4.\mathsf{case}\ x_3\ \mathsf{of}\ \{\\
 & & \qquad\qquad \mathsf{o}\Rightarrow\mathsf{s}\,x_4\\
 & & \qquad\qquad \mathsf{s}\Rightarrow\lambda x_5.\,\mathsf{s}\,(\mathsf{s}\,(\text{plus}\,\langle x_4,x_5\rangle))\\
 & & \qquad\qquad \mathsf{undef}\Rightarrow\mathsf{undef}\ \}\\
 & & \qquad \mathsf{undef}\Rightarrow\ \mathsf{undef}\\
 & & \}\qquad\qquad \}
\end{array}
$$

We conjecture that the compilation is correct, in the sense that every well-typed, exhaustive, non-overlapping $\lambda_{\mathsf{CS+fun}}$-environment is translated into a well-formed $\lambda_{\mathsf{CS+def}}$-environment.

Conjecture 9.8.7 *Let Σ be a well-formed $\lambda_{\mathsf{CS+fun}}$-environment, and let $\Phi\equiv\mathsf{translate}(\Sigma)$. Then, for every $f\in\mathsf{FS}_\Sigma$ and $\sigma\in\mathsf{Ty}_\Sigma(f)$,*

$$\mathsf{er}(\Phi)\,|\,.\ \vdash_{\lambda_{\mathsf{CS+def}}}\lambda x.\,\mathsf{compile}\big([x],\mathsf{Eqs}(\mathsf{Ru}_\Sigma(f))\big):\sigma$$

From this conjecture, it would follow that:

Conjecture 9.8.8 *If Σ is a well-formed $\lambda_{\mathsf{CS+fun}}$-environment, then $\mathsf{translate}(\Sigma)$ is a well-formed $\lambda_{\mathsf{CS+def}}$-environment.*

We also conjecture that the translation preserves reduction in the following sense:

Conjecture 9.8.9 *Let Σ be a well-formed $\lambda_{\mathsf{CS+fun}}$-environment and $\Phi\equiv\mathsf{translate}(\Sigma)$. Then*

$$e\to_{\beta\delta_\Sigma}e'\quad\Rightarrow\quad e\twoheadrightarrow_{\beta\iota\delta_\Phi}\ e'$$

Chapter 10

Related Work and Conclusion

10.1 Related Work

While the approach taken in this thesis is original, there is a vast amount of literature on subtyping and overloading in higher-order typed languages.

Declarative subtyping

Declarative subtyping allows bounded declarations of the form $\alpha \leq A : *$ in contexts. Such an approach has been used in conjunction with related ideas, most notably bounded quantification [32]. Bounded quantification combines subtyping and parametric polymorphism, and it is written like universal quantification with the addition of an upper bound for the quantified type variable: $\forall\, \alpha \leq A.\, B$.

Barthe [21] studies $IF^{\omega}_{\leq}$ a higher-order typed λ-calculus with subtyping, bounded quantification, and order-sorted inductive types (i.e. with constructor subtyping). Compagnoni [40] studies a calculus combining higher-order bounded quantification and intersection types. The interaction between dependent types and declarative subtyping has been studied e.g. in [9, 117] for Logical Frameworks. Declarative subtyping is used by Pfenning to encode various formal languages in an extension of the Logical Frameworks with refinement types [117]. Constructor subtyping also shares some of the motivations of this work. However, the technicalities are rather different, as constructor subtyping uses overloading instead of intersection types.

The interaction between dependent types and declarative subtyping has been studied by Aspinall and Compagnoni [9] for the Logical Frameworks, by Chen [34] for the Calculus of Constructions, and by Zwanenburg [144] for Pure Type Systems. One major difference between [9] and [34, 144] is that the former lets subtyping depend on typing, which leads to substantial complications in the theoretical study of the system. More recently, Castagna and Chen [33] have extended Chen's variant of Aspinall and Compagnoni's $\lambda P_{\leq}$ with late-binding. Their calculus is a significant improvement over $\lambda P_{\leq}$ and allows to formalize the examples of [117]. However, declarative subtyping, even combined with late-binding, is not appropriate for the inductive approach to formalization.

Record subtyping and type classes

A record type is a finite unordered set of labeled types (i.e., is a generalization of labeled product type). Record subtyping [31] introduces a subtyping constraint $A \leq B$ whenever A and B are labeled tuples, and every component in B is also present in A, as is the case with colored points and points, where $\mathsf{CPoint} \leq \mathsf{Point}$, for $\mathsf{Point} \equiv \{\mathsf{x} : \mathbb{N},\ \mathsf{y} : \mathbb{N}\}$ and $\mathsf{CPoint} \equiv \{\mathsf{x} : \mathbb{N},\ \mathsf{y} : \mathbb{N},\ \mathsf{c} : \mathsf{Color}\}$. There are different forms of expressing subtyping for records: width subtyping and depth subtyping. Width subtyping means that a subtype has extra fields, while depth subtyping means that the types of the fields of a subtype are themselves subtypes. A subtyping rule for records that incorporates these two forms is

$$\frac{A_i \leq B_i \qquad\qquad m \leq n}{\{l_1 : A_1, \ldots, l_n : A_n\} \leq \{l_1 : B_1, \ldots, l_m : B_m\}}$$

If $m = n$ we have depth subtyping only, and if $A_i \equiv B_i$ we have width subtyping only.

While records are extensions of products, variant types are extensions of sums. A variant type is a labeled disjoint sum. One can also consider width subtyping and depth subtyping for variants. Width subtyping for variants means that a supertype can be obtained by adding new variants, whereas depth subtyping for variants is as for records. A subtyping rule for variants that incorporate this two forms is

$$\frac{A_i \leq B_i \qquad\qquad n \leq m}{[l_1 : A_1, \ldots, l_n : A_n] \leq [l_1 : B_1, \ldots, l_m : B_m]}$$

Notice that for variants we have $n \leq m$, while for records we have $m \leq n$.

The concept of record subtyping is of great importance in object calculi. The viewpoint developed by Cardelli [31] and adopted by other researchers [136, 127] is to consider an object as a record, using record subtyping to model subclass relations. In [136] Wand introduced the concept of row variable—a new kind of variable that ranges over entire "rows" of field labels and associated types—which allow more flexibility in the typing of records, and an extension of equational unification that supports row variables. This work was further developed by others [137, 138, 127, 78, 128]. Related methods have been developed for variants [61, 60]. These techniques have been extended to general notions of type classes [83], constraint types [99], and qualified types [79], which form the basis of Haskell's system of type classes. Typed functional programing languages with object-oriented extensions, such as OCaml [140] or O'Haskell [111] integrate record subtyping and variant subtyping.

Record subtyping is also relevant in the design of mechanisms for modular programming / formalization. One of the approaches used for expressing modular structures in type theory is based on dependent record types as signatures. There are several possible definitions for dependent records. Dependent records can be defined as labeled pairs, which can be left-associating [27, 122] (achieving extensibility) or right-associating [43] (achieving specialization of structures). In [10] dependent records are defined as labeled tuples of arbitrary length. [10, 43, 122] provide support for manifest fields, whereas [27] lacks manifest fields to express sharing. [27, 43] include record subtyping as primitive and [122] has no built-in notion of subtyping (but uses coercive subtyping to capture record subtyping). [43] features singleton types which are used to cast manifest fields.

Structural subtyping and order-sorted algebra

Structural subtyping is based on the structure of type expressions. In this approach, one assumes a subtyping relation on a set of base types; then one structurally extends the subtyping relation to the other types. Therefore, comparable types must have the same shape and can only differ by their atomic types. This contrast with non-structural subtyping, where types of different shapes may be comparable (e.g., record subtyping, or when a least type $\perp$ and a greatest type $\top$ are supplied).

The first type inference algorithm with structural subtyping was proposed by Mitchell [108, 109] and improved by Fuh and Mishra [58, 59]. Type systems associate not only a type to an expression but also a set of subtyping constraints, stating assumptions about the subtype order under which the typing is valid. As said in [75], the algorithm is inefficient, and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read. In order to minimize these problems, several researchers have investigated [126, 124] simplifying constraints in the typings generated by type inference algorithms.

Constructor subtyping is not exactly structural subtyping, but is closely related. In constructor subtyping, the subtyping order is defined by lifting a poset of type constructors (and type constructors related share the same arity) along the existing type structure. In this work, we have extended the algorithm given in [109] to our setting. However, we had no concerns about simplification issues, as our goal was just to prove the decidability of type checking.

Amadio and Cardelli [7] have investigated the interactions of subtyping and recursive types, in a simply typed λ-calculus, in which the subtyping relation is based on an ordering on infinite trees. A weak notion of extensible datatypes that rules out constructor overloading has been implemented and used by J. Nordlander [110, 111]. His approach allows to capture those instances of constructor subtyping which do not involve overloading, such as the datatype of lists/non-empty lists, but fails to capture those instances of constructor subtyping involving overloading, such as even/odd/natural numbers. Independently, E. Poll [121] has been studying subtyping between inductive types but the subtyping relation he considers is also weaker than ours and he does not provide a formal treatment of extensible recursive functions.

Order-sorted algebra was created by J.A. Goguen [70, 68] to solve the difficulties to deal with exceptions and partial functions using many-sorted algebra. Order-sorted algebra constitutes the basis of the OBJ3 [69] specification languages. The basic concept of order-sorted algebra is that of order sorted signature, which extends the traditional notion of many-sorted signature with subtyping and overloading, allowing a type to be declared as a subtype of another, and functions symbols to be assigned more than one domain and codomain.

Order-sorted algebra has been a source of inspiration for constructor subtyping systems. In [21] Barthe studies a system combining the system $F^{\omega}_{\leq}$ with the formalism of order-sorted algebra, and proves that the interaction between subtyping and recursion can be controlled in a satisfactory manner provided some mild restrictions are imposed on order-sorted signatures.

Inductive and recursive definitions for order-sorted datatypes have been studied in the context of the specification language ABEL, by Owe, Dahl, Bastiansen and Kristoffersen [50, 85, 113, 26]. Their work emphasizes the expressibility of the framework and suggests a paradigm, called terminating generator induction, which provides a pattern-matching-like facility for recursive definitions. However they do not study fundamental meta-theoretical properties, such as strong normalization or decidability of type checking.

Coercive subtyping

Coercive subtyping originates from early work by P. Aczel on the Galois project [13], and has been further developed by Z. Luo and his co-workers.

Coercive subtyping [18, 91] captures in a type-theoretical framework a convention that allows one to view a term a of type A as a term of type B whenever there is a previously agreed upon function, called coercion, from A to B. The coercion for the subtyping $A \leq B$ is a function $c : A \rightarrow B$, declared in a coercion context, which allows to view every element a of A as an element of B and a shorthand for $c\,a$ (e.g. if $f : B \rightarrow C$ then $f\,a$ is expanded to $f\,(c\,a)$ of type C). This approach leads to extremely powerful type systems, is implemented in several proof-development systems, and has proved useful. However, coercive subtyping yield intricate coherence problems: there should be only one coercion between two types A and B but, of course, coercions can be composed. A set of coercions is coherent if for every types A and B, every two coercions from A to B are convertible. Coherence is undecidable in presence of parametric coercions or of dependent coercions. Thus current implementations [125, 30] of coercive subtyping, do not check coherence.

There is no elegant way to capture constructor subtyping as a special instance of coercive subtyping. Luo has suggested the use of unit types in combination with implicit coercions to capture overloading—an idea of Luo described in detail in [13]. However, we have been unable to apply these ideas in Coq to overload the successor function in the example of the even/odd/natural numbers. In [93] this example is encoded using dependent coercions. However, constructor subtyping may therefore be viewed as a more primitive and adequate framework for formalizing natural semantics.

10.2 Conclusion

Constructor subtyping provides a solution to the problem of datatypes in typed λ-calculi with subtyping, and it is directly relevant both to functional programming languages and proof assistants. In this work we have introduced λ_{CS}, a simply typed λ-calculus, à la Curry, with mutually recursive parametric datatypes, supporting constructor subtyping, and we have shown the calculus is well-behaved.

The salient feature of constructor subtyping is the overloading of constructors. This form of subtyping combines coherently the subtyping between datatypes and constructor overloading and captures, in a type-theoretic context, the use of subtyping as inclusion between inductively defined sets. Moreover, it is adequate to work with mutually recursive datatypes. λ_{CS} provides a flexible type system fully compatible with the inductive approach to formalization—the syntax of a formal language is defined in terms of inductive types—, and its usefulness is largely demonstrated by its ability to express concisely and accurately a variety of formal languages.

Constructor subtyping is also of particular convenience for extensible datatypes and is specially adequate to re-usability. The mechanism of expanding and contracting datatypes has an associated form of code reuse for functions on datatypes, allowing also the definition of new functions by restricting or by expanding already defined ones. We have considered the problem of defining recursive functions on extensible datatypes. We have devised a general mechanism to define overloaded extensible recursive functions by pattern matching, and added it to λ_{CS}, defining the system $\lambda_{\mathsf{CS+fun}}$. Overloading has been studied most notably both in the context of records and type classes and in the context of type systems for object-oriented languages. In this latter context, overloading is often resolved through late binding or dynamic type-checks. However, both approaches force reduction to depend on typing. The salient feature of $\lambda_{\mathsf{CS+fun}}$ is that, unlike many object-oriented programming languages, the computational behavior of recursive functions does not depend on typing. We have proved that $\lambda_{\mathsf{CS+fun}}$ enjoys fundamental meta-theoretical properties, giving a formal foundation for extensible datatypes with overloading of constructors and of recursive functions. The resulting framework provides an elegant (core) language for practical functional programming with extensible datatypes and is appropriate to the incremental development of programs and proofs.

This work studies the interaction between constructor subtyping and inductive types, but many issues remain to be investigated. First of all, we intend to study the correctness of the compilation techniques for pattern-matching conjectured in Section 9.8, and study the properties of the calculus $\lambda_{\widehat{\mathsf{CS}}}$ sketched in Subsection 8.4.2.

In a different direction it might be of interest to investigate extensions of our results to systems with polymorphism and dependent types. It could also be of interest, to take the approach in [24] combining constructor subtyping with record subtyping and develop a mechanism for extensible overloaded functions for record types.

Having in mind the long term goal of adding constructor subtyping mechanisms to proof assistants, Barthe and van Raamsdonk [25] investigates a system where constructor subtyping is aggregated to the Calculus of Inductive Constructions, a powerful type system with dependent and inductive types that forms the basis of proof assistants as Coq and Lego. However, many issues deserve further study. This calculus is à la Church so, a line of work is to study the theory of canonical inhabitants. As pointed out in [21], the system is not well-behaved with respect to canonical inhabitants: e.g. both $\mathsf{nil}\,[\mathsf{Even}]$ and $\mathsf{nil}\,[\mathsf{Nat}]$ are closed normal inhabitants of $\mathsf{List}\,\mathsf{Nat}$. In [24], we show that an η-expansion rule for datatypes solves the problem in the simply typed case. It should be possible to adopt the same solution for the Calculus of Constructions, although the combination of η-expansion with dependent types is somewhat intricate [19, 20, 53, 63].

Another possible research direction is to consider constructor subtyping on inductive families. Inductive families [54], are a generalization of inductive types whereby it is possible to define families of types inductively. Typical examples include the enumeration sets and the type of lists of a given length. In particular, we aim to design a well-behaved framework that supports statements such as "the type of lists of length smaller than n is a subtype of the type of lists of length smaller than $n + 1$".

Bibliography

[1] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[2] A. Abel. Specification and verification of a formal system for structurally recursive functions. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proof and Programs, International Workshop, TYPES '99*, volume 1956 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2000.

[3] A. Abel and T. Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[4] A. Abel and R. Matthes. (Co-)iteration for higher-order nested datatypes. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 1–20, Berg en Dal, The Netherlands, 2003. Springer-Verlag.

[5] T. Altenkirch, V. Gaspes, B. Nordström, and B. von Sydow. *A user's guide to ALF*, June 1994. Available by ftp from `ftp.cs.chalmers.se` in directory `pub/provers` along with the implementation.

[6] T. Altenkirch. Logical relations and inductive/coinductive types. In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, *Proceedings 12th Int. Workshop on Computer Science Logic, CSL'98, Brno, Czech Republic, 24–28 Aug 1998*, volume 1584 of *Lecture Notes in Computer Science*, pages 343–354. Springer-Verlag, Berlin, October 1999.

[7] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proceedings of POPL'91*, pages 104–118. ACM Press, 1991.

[8] R. Amadio and S. Coupet-Grimal. Analysis of a guard condition in type theory (extended abstract). In Maurice Nivat, editor, *Proceedings 1st Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS'98, held as part of 1st Europ. Joint Confs. on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, 28 Mar - 4 Apr 1998*, volume 1378 of *Lecture Notes in Computer Science*, pages 48–62. Springer-Verlag, Berlin, 1998.

[9] D. Aspinall and A. Compagnoni. Subtyping dependent types. In *Proceedings of LICS'96*, pages 86–97. IEEE Computer Society Press, 1996.

[10] L. Augustsson. Cayenne: A language with dependent types. In *Proceedings of ICFP'98*, pages 239–250. ACM Press, 1998.

[11] A. Bac. *Un algorithme d'inférence de types pour les types coinductifs*. Memoire de dea, École Normale Supérieure de Lyon, June 1998.

[12] J. Baeten, J. Bergstra, and J.W. Klop. Priority rewrite systems. In P. Lescanne, editor, *Rewriting techniques and applications*, volume 256 of *Lecture Notes in Computer Science*, pages 83–94. Springer-Verlag, 1987.

[13] A. Bailey. *The Machine-Checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.

[14] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition, 1984.

[15] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992. Volume 2.

[16] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[17] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris 7, 1999.

[18] G. Barthe. Implicit coercions in type systems. In S. Berardi and M. Coppo, editors, *Proceedings of TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 1996.

[19] G. Barthe. Existence and uniqueness of normal forms in pure type systems with $\beta\eta$-conversion. In E. Grandjean G. Gottlob and K. Seyr, editors, *Proceedings of CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 241–259. Springer-Verlag, 1999.

[20] G. Barthe. Expanding the cube. In W. Thomas, editor, *Proceedings of FOSSACS'99*, volume 1578 of *Lecture Notes in Computer Science*, pages 90–103. Springer-Verlag, 1999.

[21] G. Barthe. Order-sorted inductive types. *Information and Computation*, 149(1):42–76, February 1999.

[22] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[23] G. Barthe and M.J. Frade. Constructor subtyping. Technical Report UMDITR9807, Department of Computer Science, University of Minho, 1998.

[24] G. Barthe and M.J. Frade. Constructor subtyping. In D. Swiestra, editor, *Proceedings of ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer-Verlag, 1999.

[25] G. Barthe and F. van Raamsdonk. Constructor subtyping in the calculus of inductive constructions. In J. Tuyrin, editor, *Proceedings of FOSSACS'00*, volume 1784 of *Lecture Notes in Computer Science*, pages 17–34. Springer-Verlag, 2000.

[26] T. Bastiansen. Parametric subtypes in ABEL. Technical Report 207, Department of Informatics, University of Oslo, 1995.

[27] G. Betarte. *Dependent Record Types and Algebraic Structures in Type Theory.* PhD thesis, Department of Computer Science, Chalmers Tekniska Högskola, 1998.

[28] F. Blanqui. *Theorie des Types et Recriture*. PhD thesis, LRI - Université de Paris-Sud, 2001.

[29] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive data type systems. *Theoretical Computer Science*, 272(1–2):41–68, 2002.

[30] P. Callaghan and Z. Luo. An implementation of LF with coercive subtyping & universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

[31] L. Cardelli. Semantics of multiple inheritance. *Information and Computation*, 76, 1985. Also published in/as: In 'Readings in Object-Oriented Database Systems" edited by S.Zdonik and D.Maier, Morgan Kaufman, 1990.

[32] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[33] G. Castagna and G. Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 1999. To appear.

[34] G. Chen. Subtyping calculus of constructions. In I. Prívara and P. Ruzicka, editors, *Proceedings of MFCS'97*, volume 1295 of *Lecture Notes in Computer Science*, pages 189–198. Springer-Verlag, 1997.

[35] W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2–3):261–300, September 2001.

[36] A. Church. A set of postulates for the foundation of logic part I. *Annals of Mathematics*, 33:346–366, 1932.

[37] A. Church. A set of postulates for the foundation of logic part II. *Annals of Mathematics*, 34:839–864, 1933.

[38] A. Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[39] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[40] A. Compagnoni. *Higher-order subtyping with intersection types*. PhD thesis, Department of Computer Science, University of Nijmegen, 1995.

[41] R. Constable et al. *Implementing Mathematics in the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[42] C. Coquand and T. Coquand. Structured type theory. In *Proceedings of LFM'99 (held in conjunction with PLI'99)*, 1999.

[43] T. Coquand, R. Pollack, and M. Takeyama. A logical framework with dependently typed records. In *Typed Lambda Calculus and Applications, TLCA'03*, volume 2701 of *LNCS*. Springer-Verlag, 2003.

[44] T. Coquand. Pattern matching with dependent types. In B. Nordström, K. Petters-son, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 71–84. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992. ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z.

[45] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Selected Papers 1st Int. Workshop on Types for Proofs and Programs, TYPES'93, Nijmegen, The Netherlands, 24–28 May 1993*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, Berlin, 1994.

[46] T. Coquand and C. Paulin-Mohring. Inductively defined types (preliminary version). In P. Martin-Löf and G. Mints, editors, *Proceedings Int. Conf. on Computer Logic, COLOG'88, Tallinn, USSR, 12–16 Dec 1988*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, Berlin, 1990.

[47] C. Cornes. *Conception d'un langage de haut niveau de representation de preuves: Récurrence par filtrage de motifs; Unification en présence de types inductifs primitifs; Synthèse de lemmes d'inversion.* PhD thesis, Université de Paris 7, 1997.

[48] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[49] H. Curry. Functionality in combinatory logic. *Proc. Nat. Acad. Science USA*, 20:584–590, 1934.

[50] O.-J. Dahl, O. Owe, and T.J. Bastiansen. Subtyping and constructive specification. *Nordic Journal of Computing*, 5(1), Spring 1998.

[51] R. Davies. A practical refinement-type checker for standard ml. In *Sixth International Conference on Algebraic Methodology and Software Technology*, 1997.

[52] N. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic, 1980.

[53] G. Dowek, G. Huet, and B. Werner. On the existence of long $\beta\eta$-normal forms in the cube. In H. Geuvers, editor, *Informal Proceedings of TYPES'93*, pages 115–130, 1993.

[54] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[55] H. Elbers. *Connecting formal and informal mathematics.* PhD thesis, Technische Universiteit Eindhoven, 1998.

[56] E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B, pages 995–1072. Elsevier Publishing, 1990.

[57] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of PLDI'91*, pages 268–277. ACM Press, 1991.

[58] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In *2nd European Symposium on Programming, Nancy*, pages 94–114. Springer-Verlag, New York, NY, 1988. Lecture Notes in Computer Science 300.

[59] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL)*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1989.

[60] J. Garrigue. Programming with polymorphic variants. In *Proc. of 1998 ACM SIGPLAN Workshop on ML, Baltimore, MD, USA, 26 Sept. 1998*, page ??? October 1998.

[61] J. Garrigue and H. Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *Conf. Record 21st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Oregon, PL, USA, 17–21 Jan. 1994*, page ??? ACM Press, New York, 1994.

[62] H. Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 193–217. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992. ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z.

[63] N. Ghani. Eta-expansions in dependent type theory—the calculus of constructions. In P. de Groote and J. Hindley, editors, *Proceedings of TLCA'97*, volume 1210 of *Lecture Notes in Computer Science*, pages 164–180. Springer-Verlag, 1997.

[64] J. Giesl. Termination analysis for functional programs using term orderings. *Lecture Notes in Computer Science*, 983:154–171, 1995.

[65] E. Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. PhD thesis, Ecole Normale Superieure de Lyon, 1996.

[66] E. Giménez. Structural recursive definitions in Type Theory. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.

[67] E. Giménez. Codifying guarded definitions with recursion schemes. In P. Dybjer and B. Nordström, editors, *Selected Papers 2nd Int. Workshop on Types for Proofs and Programs, TYPES'94, Båstad, Sweden, 6–10 June 1994*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer-Verlag, Berlin, 1995.

[68] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):216–273, 1992.

[69] J. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, August 1988.

[70] J. Goguen. Order sorted algebras: Exceptions and error sorts, coercions and overloaded operators. Semantics and Theory of Computation Report 14, UCLA, December 1978.

[71] B. Grobauer. Cost recurrences for DML programs. *ACM SIGPLAN Notices*, 36(10):253–264, October 2001.

[72] C. Gunter and J. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics and Language Design*. The MIT Press, 1994.

[73] T. Hallgren. *Subtypes in Polymorphic Functional Languages*. Licenciate Thesis, Department of Computer Science, Chalmers Tekniska Högskola, 1993.

[74] A. Haxthausen and M. Cerioli. The role of subsorts in subsort declarations and datatype declarations. Language design note for the Common Framework Initiative (COFI), 1997.

[75] M. Hoang and J. Mitchell. Lower bounds on type inference with subtypes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 176–185, San Francisco, California, January 1995.

[76] M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of LICS'94*, pages 208–212. IEEE Computer Society Press, 1994.

[77] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 410–423. ACM SIGACT and SIGPLAN, ACM Press, 1996.

[78] L. Jategaonkar and J. Mitchell. Type inference with extended pattern matching and subtypes. *Fund. Informaticae*, 1993.

[79] M. Jones. ML typing, explicit polymorphism and qualified types. *Lecture Notes in Computer Science*, 789:56–??, 1994.

[80] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.

[81] S. Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[82] J.-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 28 February 1997.

[83] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer Verlag, 1988.

[84] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[85] B. Kristoffersen and O.-J. Dahl. On introducing higher order functions in ABEL. *Nordic Journal of Computing*, 5(1):50–69, Spring 1998.

[86] T. Laan. *The Evolution of Type Theory in Logic and Mathematics*. PhD thesis, Techn. Univ. Eindhoven, 1997.

[87] C. Lee, N. Jones, and A. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, January 2001.

[88] D. Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Proceedings 24th Annual IEEE Symp. on Foundations of Computer Science, FOCS'83, Tucson, AZ, USA, 7–9 Nov 1983*, pages 460–469. Los Alamitos, CA, 1983.

[89] D. Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *APIC Studies in Data Processing*, pages 279–327. London, 1990.

[90] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

[91] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9:105–130, February 1999.

[92] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, University of Edinburgh, May 1992.

[93] Z. Luo and S. Soloviev. Dependent coercions. In Martin Hofmann, Giuseppe Rosolini, and Dusko Pavlovic, editors, *Proc. of 8th Conf. on Category Theory and Computer Science, CTCS'99, Edinburgh, UK, 10–12 Sept 1999*, volume 29 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, 1999.

[94] P. Manoury and M. Simonot. Automatizing termination proofs of recursively defined functions. *Theoretical Computer Science*, 135(2):319–343, 1994.

[95] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP'97*, pages 136–149. ACM Press, 1997.

[96] P. Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings 2nd Scandinavian Logic Symp., Oslo, Norway, 18–20 June 1970*, volume 63, pages 179–216. North-Holland Publishing, Amsterdam, 1971.

[97] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Proc. of Logic Colloquium '73, Bristol, UK, July 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, Amsterdam, 1975.

[98] P. Martin-Löf. *Intuitionistic Type Theory*. Bibioplois, Napoli, 1984. Notes of Giowanni Sambin on a series of lectues given in Padova.

[99] M. Sulzmann, M. Odersky, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.

[100] R. Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Fachbereich Mathematik, Ludwig-Maximilians-Universität München, 1998.

[101] R. Matthes. Monotone (co)inductive types and positive fixed-point types. *Theor. Inform. and Appl.*, 33(4–5):309–328, 1999.

[102] R. Matthes. Monotone fixed-point types and strong normalization. In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, *Proceedings 12th Int. Workshop on Computer Science Logic, CSL'98, Brno, Czech Republic, 24–28 Aug 1998*, volume 1584 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, Berlin, October 1999.

[103] R. Matthes. Tarski's fixed-point theorem and lambda calculi with monotone inductive types. In Benedikt Löwe and Florian Rudolph, editors, *Refereed Papers of Research Coll. on Foundations of the Formal Sciences, Berlin, Germany, 7–9 May 1999*, pages 91–112. Kluwer Academic Publishers, Dordrecht, 2000.

[104] C. McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh, 1999.

[105] N. Mendler. Recursive types and type constraints in second-order lambda-calculus. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987*, pages 30–36. Washington, DC, 1987.

[106] N. Mendler. Inductive types and type constraints in the second-order lambda-calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[107] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[108] J. Mitchell. Coercion and type inference. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[109] J. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991.

[110] J. Nordlander. Pragmatic subtyping in polymorphic languages. In *Proceedings of ICFP'98*. ACM Press, 1998.

[111] J. Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2–3):93–127, June 2002.

[112] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. Number 7 in International Series of Monographs on Computer Science. Oxford University Press, 1990.

[113] O. Owe and O.-J. Dahl. Generator induction in order sorted algebras. Research Report ISBN 82–7368–027–4, University Oslo, 1989.

[114] J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[115] L. Pareto. *Types for crash prevention*. PhD thesis, Chalmers Univ. Techn., Göteborg, 2000.

[116] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, Berlin, 1993.

[117] F. Pfenning. Refinement types for logical frameworks. In H. Geuvers, editor, *Informal Proceedings of TYPES'93*, pages 285–299, 1993.

[118] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings 5th Int. Conf. on Math. Foundations of Programming Semantics, MFPS'89, New Orleans, LA, USA, 29 Mar – 1 Apr 1989*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, Berlin, 1990.

[119] B. Pierce and D. Turner. Local type inference. In *Proceedings of POPL'98*, pages 252–265. ACM Press, 1998.

[120] B. Pierce, S. Dietzen, and S. Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, School of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, 1989.

[121] E. Poll. Subtyping and Inheritance for Inductive Types. In *Proceedings of TYPES'97 Workshop on Subtyping, inheritance and modular development of proofs, Durham, UK*, 1997.

[122] R. Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[123] F. Pottier. *Synthèse de types en présence de sous-typage: de la théorie la pratique*. PhD thesis, Université Paris VII, 1998.

[124] F. Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133, Philadelphia, Pennsylvania, 24–26 May 1996.

[125] The Coq Development Team LogiCal Project. *The Coq Proof Assistant Reference Manual. Version 7.4*, 1999-2003.

[126] J. Rehof. Minimal typings in atomic subtyping. In *Conference Record of POPL'97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 278–291. ACM SIGACT and SIGPLAN, ACM Press, 1997.

[127] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–88, Austin, Texas, January 1989.

[128] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–52, 1998.

[129] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[130] N. Shankar, S. Owre, and J. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.

[131] Z. Spławski and P. Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Proceedings 4th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'99, Paris, France, 27–29 Sept 1999*, pages 102–113. ACM Press, New York, 1999.

[132] A. Telford and D. Turner. Ensuring Streams Flow. In M. Johnson, editor, *Algebraic Methodology and Software Technology 1997*, pages 509–523. Springer-Verlag, 1997.

[133] T. Uustalu. *Natural Deduction for Intuitionistic Least and Greatest Fixedpoint Logics, with an Application to Program Construction*. PhD thesis (Dissertation TRITA-IT AVH 98:03), Dept. of Teleinformatics, Royal Inst. of Technology, Stockholm, May 1998.

[134] T. Uustalu and V. Vene. A cube of proof systems for the intuitionistic predicate μ,ν-logic. In Magne Haveraaen and Olaf Owe, editors, *Selected Papers 8th Nordic Workshop on Programming Theory, NPWT'96, Oslo, Norway, 4–6 Dec 1996*, Research Report 248, Dept. of Informatics, Univ. of Oslo, pages 237–246. May 1997.

[135] T. Uustalu and V. Vene. Least and greatest fixedpoints in intuitionistic natural deduction. *Theoretical Computer Science*, 272(1-2):315–339, 2002.

[136] M. Wand. Complete type inference for simple objects. In *Proceedings, Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, 22–25 June 1987. The Computer Society of the IEEE.

[137] M. Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, page 132, Edinburgh, Scotland, 5–8 July 1988. IEEE Computer Society.

[138] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.

[139] A. Whitehead and B. Russell. *Principia mathematica*. Cambridge University Press, Cambridge, 1910.

[140] J. Garrigue, D. Rémy, X. Leroy, D. Doligez, and J. Vouillon. *The Objective Caml system release 3.06 - Documentation and user's manual*, 2002.

[141] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of PLDI'98*, pages 249–257. ACM Press, 1998. *SIGPLAN Notices* 33(5), May 1998.

[142] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*, pages 214–227. ACM Press, 1999.

[143] H. Xi. Dependent types for program termination verification. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 231–246, Washington - Brussels - Tokyo, June 2001. IEEE.

[144] J. Zwanenburg. Pure type systems with subtyping. In J.-Y. Girard, editor, *Proceedings of TLCA'99*, volume 1581 of *Lecture Notes in Computer Science*, pages 381–396. Springer-Verlag, 1999.