Higher-Order Subtyping with Intersection Types

een wetenschappelijke proeve op het gebied van de
Wiskunde en Informatica

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen
op maandag 30 januari 1995,
des namiddags te 3.30 uur precies

door

ADRIANA BEATRIZ COMPAGNONI

geboren 23 december 1965 te Buenos Aires

Promotores: Professor M. Dezani-Ciancaglini,
Universit degli Studi di Torino, Turijn, Italië.
Professor dr. H. P. Barendregt.

Higher-Order Subtyping with Intersection Types

Adriana Beatriz Compagnoni

Cover design: David Aspinall and Adriana Compagnoni

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Compagnoni, Adriana Beatriz

Higher-order subtyping with intersection types
Adriana Beatriz Compagnoni. - [S.l. : s.n.]. - I11.
Thesis Nijmegen. - With ref.
ISBN 90-9007860-6
Subject headings: lambda calculus.

A Susana y Mario

Acknowledgements

Mariangiola Dezani is the most encouraging and enthusiastic computer scientist I have ever met. From her I learned to trust my intuition and follow it through to the very end. This research would not have been possible without her technical supervision and moral support. I would also like to thank my supervisor, Henk Barendregt. His lucid lectures on λ-calculus, his ability to hide irrelevant details, the clarity of his explanations, and his elegant technical prose have been constant guides.

I thank the people in the Netherlands, especially Erik Barendsen and Jan Kuper, for their help and friendship, and Steffen van Bakel and Paula Severi, for their comments on previous drafts of this thesis. Mieke Massink and Maria Fernández Ferreira started as colleagues and soon became two of my dearest friends.

I am grateful to Benjamin Pierce, who suggested the study of a λ-calculus combining higher-order polymorphism and intersection types. He and I defined the system $F^{\omega}_{\wedge}$, whose study is, by and large, the theme of this thesis; together we wrote [CP93], from which chapters 5 and 6 arose. His fertile mind and his passion for hard work make him inspiring company.

I am indebted to Rod Burstall, who opened to me the doors of the Laboratory for Foundations of Computer Science. There I found a highly motivating research environment where I developed most of the results in this thesis, and a handful of friends who made my life in Edinburgh a beautifully rich experience. I enjoyed technical discussions with Stuart Anderson, David Aspinall, Philippa Gardner, Martin Hofmann, Stefan Kahrs, Zhaohui Luo, Savi Maharaj, James McKinna, Randy Pollack, and Dilip Sequeira. My special thanks to Healfdene Goguen for many helpful comments on draft versions of this thesis.

It has been an honour to have Jan Willem Klop, Giuseppe Longo, and Rob Nederpelt as members of the manuscript commission.

From my water-color teacher, Joost van Moll, I learned a technique which actually changed my way of doing research; it consists of carefully examining what has been done, learning from the successful ideas, and being brave enough to start over again regardless of the time invested in the previous try.

In the non-academic world, I want to thank my parents, Mario Compagnoni and Susana Brunsch, for their unconditional love. I want to mention (in the order in which they appeared in my life) Silvana Cantoni, Karin Gottschalk, Claudio Massa, Maribel Fernández, Javier Blanco, Cristina Abbate, Martin de Zuviría, and Esther Gerrits, because life would not be the same without friends.

Edinburgh, 6th of December, 1994.

This research was supported by the Dutch organization for scientific research, NWO-SION project *Typed lambda calculus*, 612-316-030, and by EPSRC GRANT, GR/G 55792. Constructive logic as a basis for program development.

Contents

Chapter 1

Introduction

1.1 Types and programs

One of the basic ideas in programming is the notion of algorithm. An algorithm is a description of the rules one must follow to accomplish a task. But for a machine to be able to perform such a task, this description must be expressed in a formal language, and in particular programming languages serve this purpose.

Alan Turing introduced a formal language for describing computable functions, now called Turing machines, from which imperative programming arose. The lambda calculus was invented by Alonso Church to define the notion of computable functions [Chu36]. Since then a variety of lambda calculi have been defined and used in the study of programming languages. Lambda calculi can be seen as simple programming languages, since they are formal and describe computations. In that sense, a program is a term of the lambda calculus. From a different perspective, lambda calculi constitute metalanguages for analyzing other programming languages. Although λ-calculi are particularly well-suited to studying functional programming languages, they have also been used to study imperative programming disciplines [Lan65].

Types are an important tool in programming languages and logic which serve to classify terms according to basic properties, such as being a number or being a function. For example, if we think of the integer number 5, the term is 5 and its type is integer. The addition function has a type expressing that it takes two integer numbers as arguments and returns an integer number as result, which we write as follows.

$$+ \in (\text{integer} \times \text{integer}) \rightarrow \text{integer}$$

One immediate advantage of types is that nonsensical expressions can be considered *illegal*. For example, the expression

$$3 + \text{“good morning”}$$

will not be part of the language because “good morning” is not an integer number, but a string of characters. Although this is a rather coarse example, this sort of mistake is very frequent in the development of programs.

Further in that direction, AUTOMATH [dB80], Martin-Löf Type Theory [Mar73], Coquand and Huet's Calculus of Constructions [CH88], and Luo's Extended Calculus of Constructions [Luo90] are rich type systems in which a type can not only prevent the formation of nonsensical expressions but can also state properties of terms. For example, in the case of a term corresponding to a sorting algorithm for lists, the type can express the fact that the output is an ordered list.

Type structures help organizing ideas and structuring programs in such a way that disciplines of programming and type systems walk hand in hand. The development of ideas about programming motivates the design of type systems that encourage programming in a particular style. Ideally, one would like to have tailor-made type systems for each particular style of programming, so that bad programming style results in illegal terms.

We refer the reader to [Bar90, Mit90c] for a more detailed analysis of the relation between lambda calculi and programming languages.

1.2 Subtyping

Subtyping is a primitive relation that uniformly captures concepts from diverse areas of computer science. If S and T are sets, then $S \leq T$ (*S is a subtype of T*) means that elements of S are also elements of T. If S and T are specifications, then elements satisfying specification S also satisfy T. In object-oriented programming, if S and T are object descriptions, then $S \leq T$ states that where an object with interface T is expected, it is safe to use an object with interface S. When S and T are module interfaces in a software system, an implementation of S is also an implementation of T. If S and T are theorems, then a proof of S is also a proof of T. Understanding the essence, subtleties, and general properties of subtyping illuminates a wide area.

The idea of subtypes appears quite naturally in programming languages. If we think of types as sets, we can easily picture what a subtype could be. Informally, we can say that a type S is a subtype of T if any element of S can be seen as an element of T. We say *can be seen as* and not directly *is* because the act of considering an element of type S as an element of type T might hide some transformation. Consider for example the types *integer* and *real* of integers and real numbers respectively. Usually, on a computer, integers are represented in a different way than real numbers are; even if we might think of the integers as a subset of the real numbers, there is a translation to be performed. The act of considering an element of type S as an element of type T will be called *coercion*. In other words, we say that an element of type S is coerced into an element of type T. Somehow an element of type S has enough information to be seen as an element of type T.

While dealing with coercions we can distinguish between an explicit style and an implicit style. A style with explicit coercions means that coercions are explicitly indicated and in an implicit style, as the name suggests, coercions are left unstated. In systems including subtyping there is usually a rule for typing coerced terms, in other words, a rule that provided t has type S and S is a subtype of T allows us to derive that t can be coerced into T. In an explicit style, the coercion rule might look as follows.

$$\frac{t \in S \qquad S \leq T}{c_{S,T}{<}t{>} \in T} \qquad (\textsc{Coercion})$$

Similarly, in an implicit style the corresponding rule is as follows.

$$\frac{t \in S \qquad S \leq T}{t \in T} \qquad (\textsc{Subsumption})$$

An implicit coercion is motivated by the fact that the same term can be considered as belonging to two different types without performing any change in the term, as for example is the case when one of the types is included in the other (with the intuitive idea of set inclusion), while an explicit coercion gives explicit information about the transformation. We can think, for example, of a function f with the real numbers as domain, and a (sub)set A of real numbers. If x is a variable of type A, then we would like to use f on x as well, without performing any extra calculation to apply f to x. But if instead f is used with the integer number 3 as input and f happens to use the decimal part of its argument, then 3 should be mapped into 3.0 first. Therefore one can argue that the meaning of $f(3)$ is $f(c_{integer,real}{<}3{>})$.

One of the first applications of subtyping in λ-calculi was modeling the refinement of interfaces in object-oriented languages [Car88a]. The formal subtype relation $S \leq T$ models the assertion that the objects in some collection S provide more services than those in T, so that it is safe to use a member of S in any context where a member of T is expected.

1.3 Type inference and type checking

We can say that if a term has a type, it is, to a certain extent, correct. This correctness can be as simple as guaranteeing that computations will not fail by mismatch of the expected argument of a function, for example, and as elaborate as ensuring that certain specification or property is satisfied. With these ideas in mind, a first question one may ask is whether a given expression e is legal or correct, which in our framework means whether there exists a type T such that $e \in T$. This problem is traditionally called *type inference.* A related question, given a term e and a type T, is whether $e \in T$, known as *type checking.* In the presence of subtyping both problems, type checking and type inference, become more complicated because typing is defined in terms of subtyping. It is clear in the Subsumption and Coercion rules that in order to answer a question of the form $e \in T$ we should be able to answer questions of the form $S \leq T$. This shows that at the heart of the decidability of typing lies the question of whether the subtyping relation is decidable. In a system with the Subsumption rule, each term may be assigned more than one type. Then to answer the type checking and type inference questions we need a way to identify all possible types: for example, by finding some kind of representative of the types of each term. One plausible candidate is a *minimal type* with respect to the subtyping relation. Then type inference consists of finding a minimal type, and type checking whether $e \in T$ consists of finding a minimal type S such that $e \in S$ and checking if $S \leq T$. Without the minimal type property, type checking becomes algorithmically intractable. Imagine that instead of having just one representative we have a (finite) set of them, say $S_1, .., S_n$ such that for each type T of e there exists j such that $S_j \leq T$. Imagine that e is an application, say $e_1\, e_2$: then, to find the set of representative types of e, we need to match each representative of e_1 against each representative of e_2, which produces a combinatorial explosion.

1.4 Background

The formal study of subtyping in programming languages was begun by Reynolds [Rey80] and Cardelli [Car88a], who used a lambda-calculus with subtyping to model the refinement of interfaces in object oriented languages. This led to a considerable body of work, covering an increasing range of object-oriented features by combining subtyping with other type-theoretic constructs, including polymorphic functions [CW85, CG92, BCGS91]; records with update and extension operators [Car88a, CM91]; recursive types [AC93, BM92], and higher-order polymorphism [Car90, Mit90a].

Type systems with subtyping have also arisen from the study of lambda-calculi with intersection types at the University of Torino [CD80, BCD83]. Most of this work has been carried out in the setting of pure lambda-calculi, but it has also been applied to programming language design by Reynolds [Rey88]. Some work has begun on combining intersections with other typing features [Pie91, CDdL93].

The contribution of this thesis is to weave together these two threads by combining higher-order subtyping, which forms the cornerstone of several recent models of typed object-oriented programming [CHC90, Bru94, PT94], with intersection types, leading to an extended object model with multiple inheritance [CP93].

1.5 Results

This thesis is divided into two parts the first part consists of a detailed analysis of the meta-theory of a typed lambda calculus combining higher order bounded quantification and intersection types. Our research covers syntactic, semantic, and pragmatic aspects.

- Chapter 2 contains the definition of the system $F^{\omega}_{\wedge}$ and basic syntactic results.
    - We define the typed lambda calculus $F^{\omega}_{\wedge}$, a natural generalization of Girard's system F^{ω} with intersection types and bounded polymorphism. A novel aspect of our presentation is the use of term rewriting

techniques to present intersection types, which clearly splits the computational semantics (reduction rules) from the syntax (inference rules) of the system.

- The reduction rules of $F^{\omega}_{\wedge}$ can be divided into two main groups, reductions on types ($\rightarrow_{\beta\wedge}$) and reductions on terms ($\rightarrow_{\beta fors}$). Although confluence is not a modular property in general, in our case it is possible to provide a modular proof of it. In section 2.3, we combine the independent proofs of confluence for reductions on types and confluence for reduction on terms towards a proof of confluence of the reduction relation in the whole system.
- We prove the strong normalization property of $\rightarrow_{\beta\wedge}$ on well-formed types.

- Chapter 3 carries the most important result of this thesis. Our main contribution is the proof that subtyping in $F^{\omega}_{\wedge}$ is decidable. This yields as a corollary a solution to the previously open problem of the decidability of subtyping in $F^{\omega}_{\leq}$, its intersection free fragment, because $F^{\omega}_{\wedge}$ subtyping system is a conservative extension of that of $F^{\omega}_{\leq}$. Moreover, the decidability of subtyping is essential for the decidability of type checking and type inference. Another original feature is the use of a choice operator to model the behavior of variables during subtype checking. The proof of decidability is divided into the following steps.

  - We define an algorithmic presentation of the subtyping relation where only types in normal form are considered.
  - We prove that the algorithmic presentation is sound and complete with respect to the definition of subtyping, which means that it constitutes a deterministic procedure to check subtyping in $F^{\omega}_{\wedge}$.
  - Finally, we prove that the algorithmic presentation describes a terminating procedure. The proof of termination is reduced to the strong normalization property of the reduction on types enriched with a choice reduction which models the behavior of variables during subtype checking.

- In chapter 4, we prove that $F^{\omega}_{\wedge}$ satisfies the minimal type property, and we provide an algorithm for computing minimal types. We also prove that type inference and type checking in $F^{\omega}_{\wedge}$ are decidable. The minimal types property is used to prove that $F^{\omega}_{\wedge}$ satisfies the subject reduction property.

- In chapter 5, we define a model based on partial equivalence relations, and we prove that the subtyping relation and the type assignment system are sound with respect to the model.

- Although $F^{\omega}_{\wedge}$ was defined to provide a model of object-oriented programming with multiple inheritance, this thesis does not intend to provide an account on the foundations of object-oriented programming. In chapter 6, we show how to model multiple inheritance using intersection types. This is a continuation of the research on type-theoretic foundations of object-oriented programming by Pierce and Turner [PT94] where multiple inheritance is not captured.

The second part of this thesis is devoted to the study of two different styles of subtyping, subtyping with implicit coercions and subtyping with explicit coercions. We define and study two alternative presentations of subtyping for simply typed lambda calculus. The first one $\lambda_{\leqslant}$, a system with implicit coercions, and the second one λ_C, a system with explicit coercions. We show that the system $\lambda_{\leqslant}$ can be translated into λ_C, and that λ_C can be translated into $\lambda{\rightarrow}$. This means that from a pragmatic point of view, implicit or explicit coercions are just a matter of taste, and both disciplines can be compiled into the simply typed lambda calculus without subtyping.

Part I

Higher-Order Subtyping

Chapter 2

The $F^{\omega}_{\wedge}$ Calculus

2.1 Introduction

The system $F^{\omega}_{\wedge}$ was first introduced in [CP93], where it was shown to be rich enough to provide a typed model of object oriented programming with multiple inheritance. $F^{\omega}_{\wedge}$ is an extension of F^{ω} [Gir72] with bounded quantification and intersection types, which can be seen as a natural generalization of the type disciplines present in the current literature, for example in [CG92, Pie91, PT94, CP94]. Systems including either subtyping or intersection types or both have been widely studied for many years. What follows is not intended to be an exhaustive description, but a framework for the present work.

First-order type disciplines with intersection types have been investigated by the group in Torino [CDC78, BCD83] and elsewhere (see [CC90] for background and further references). A second-order λ-calculus with intersection types was studied in [Pie91]. Systems including subtyping were present in [CW85, Car88a]. Higher order generalizations of subtyping appear in [CCH^{+}89, CHC90, Mit90a, BM92]. $F_{\leq}$, a second-order λ-calculus with bounded quantification, was studied in [Ghe90], and in [Pie91] it was proved that subtyping in $F_{\leq}$ was undecidable and that undecidability was caused by the subtyping rule for bounded quantification.

In [CP94] an alternative rule for subtyping quantified types was presented and the decidability of subtyping was proved for an extension of system F with bounded polymorphism, where all bounds appearing in S-All have the ground kind $\star$, a main limitation of that system.

Allowing bounds of functional kind forces us to introduce a conversion rule to have invariance of subtyping under $\beta\wedge$-conversion of types. Therefore, our subtyping relation relates types of a more expressive type system than that presented in [CP94]. In fact, treating the interaction between interface refinement and encapsulation of objects, in object oriented programming, has required higher-order generalizations of subtyping–the F-bounded quantification of Canning, Cook, Hill, Olthoff and Mitchell [CCH^{+}89] or Cardelli and Mitchell's system $F^{\omega}_{\leq}$ [Car90, Mit90a, BM92].

Ghelli [Ghe94] remarked that the rule for subtyping between quantified types presented in [CP94] led to a well-behaved subtyping relation but that the typing relation fails to satisfy the minimal type property. This failure introduces serious problems in type checking and type inference, as we observed in chapter 1. At the moment it is not clear how to solve them or, even more problematic, whether the typing relation is decidable. A possible solution to overcome this problem is to replace the subtyping rule between quantifiers by the corresponding rule of Cardelli and Wegner's kernel fun [CW85].

In chapter 3 we give a positive answer to the decidability of subtyping in the presence of $\beta\wedge$-convertible types. We prove that subtyping in $F^\omega_\wedge$ is decidable, which *a fortiori* gives the decidability of subtyping for the $F^\omega_\leq$ fragment because the former is a conservative extension of the latter – namely, each subtyping statement derivable in $F^\omega_\wedge$ containing no intersections other than the empty ones is also derivable in $F^\omega_\leq$.

We present a definition of $F^\omega_\wedge$ that differs from the one introduced in [CP93] in two ways. First, Castagna and Pierce's quantifier rule has been replaced by the Cardelli and Wegner rule. Second, we introduce a richer notion of reduction on types, and thereby the four distributivity rules become particular cases of the conversion rule. This new reduction is shown to be confluent and strongly normalizing. The latter simplification was motivated by structural properties of the former presentation. Furthermore, this new presentation provides a different view of the system that is the key to proving the decidability of subtyping.

This new perspective suggests that to prove the decidability of subtyping it is enough to concentrate on types in normal form. Note that the solution cannot be as simple as to restrict the subtyping rules of $F^\omega_\wedge$ to handle only types in normal form and replace conversion by reflexivity. The following is a good example of the problem to be solved. Consider $\Gamma \equiv W{:}K, X \leq \Lambda Y{:}K.Y{:}K{\to}K,\ Z \leq X{:}K{\to}K$. Then $\Gamma \vdash X(Z\,W) \leq W$, which is not derivable without using conversion, i.e. without performing any β-reduction, even when the conclusion is in normal form.

The subtyping rules of $F^\omega_\wedge$ are not syntax directed, in the sense that the form of a derivable subtyping statement does not uniquely determine the last rule of its derivation (i.e. there might be more than one derivation of the same subtyping judgement). To develop a deterministic decision procedure to check subtyping, we need a new presentation of the subtyping relation that provides the foundations for a subtype-checking deterministic algorithm.

Our solution is divided in two main steps. First, we develop a *normal subtyping system*, $NF^\omega_\wedge$, in which only types in normal form are considered. We prove that derivations in $NF^\omega_\wedge$ can be normalized by eliminating transitivity and simplifying reflexivity. This simplification yields an algorithmic presentation, $AlgF^\omega_\wedge$. Moreover, we prove that $AlgF^\omega_\wedge$ is indeed an alternative presentation of the $F^\omega_\wedge$ subtyping relation, that is $\Gamma \vdash S \leq T$ if and only if $\Gamma^{nf} \vdash_{Alg} S^{nf} \leq T^{nf}$ (proposition 3.4.3).

The second and last step towards the decidability of subtyping in $F^\omega_\wedge$ is to prove that the algorithm described by $AlgF^\omega_\wedge$ terminates, which is equivalent to showing that the definition of the $AlgF^\omega_\wedge$ is well-founded. We discuss this further in section 3.5.

Checking whether $\Gamma \vdash_{Alg} S\,T \leq A$ is reduced to checking if $\Gamma \vdash_{Alg} lub_\Gamma(S\,T)^{nf} \leq$

A, where $lub_{\Gamma}(S\,T)$ substitutes the leftmost innermost variable of $S\,T$ by its bound in Γ. Such replacements may produce a term that is not in normal form, in which case we normalize it afterwards. The main problem here is that the size of the types to be examined in the recursive call does not decrease. This indicates that the proof of termination of the algorithm is not immediate. In particular, the proof of termination presented in [CP94] cannot be modified to serve our purposes, because of the interaction between $\beta\wedge$-reduction and the substitution of type variables by their bounds in our system. We discuss this further in section 3.5.

In this chapter we present the syntax of $F^{\omega}_{\wedge}$, we prove structural properties of the system, confluence, and the strong normalization property for the reduction on types.

2.2 Syntax of $F^{\omega}_{\wedge}$

We now present the rules for kinding, subtyping, and typing in $F^{\omega}_{\wedge}$. They are organized as proof systems for four interdependent judgement forms:

$\Gamma \vdash \mathrm{ok}$	well-formed context
$\Gamma \vdash T \in K$	well-kinded type
$\Gamma \vdash S \leq T$	subtype
$\Gamma \vdash e \in T$	well-typed term.

We sometimes use the metavariable Σ to range over statements (right-hand sides of judgements) of any of these four forms.

Syntactic Categories

The *kinds* of $F^{\omega}_{\wedge}$ are those of F^{ω}: the kind $\star$ of proper types and the kinds $K_1{\to}K_2$ of functions on types (sometimes called type operators).

$$\begin{array}{lll} \mathbb{K} & ::= & \star & \text{types} \\ & & \mathbb{K}{\to}\mathbb{K} & \text{type operators} \end{array}$$

The language of *types* of $F^{\omega}_{\wedge}$ is a straightforward higher-order extension of $F_{\leq}$, Cardelli and Wegner's second-order calculus of bounded quantification. Like $F_{\leq}$, it includes type variables (written X), function types ($T{\to}T'$), and polymorphic types ($\forall X{\leq}T{:}K.T'$), in which the bound type variable X ranges over all subtypes of the upper bound T. Moreover, like F^{ω}, we allow types to be abstracted on types ($\Lambda X{:}K.T$) and applied to argument types ($T\,T'$); in effect, these forms introduce a simply typed λ-calculus at the level of types. Finally, we allow arbitrary finite intersections ($\bigwedge^{K}[T_1..T_n]$), where all the T_i's are members of the same kind K.

$$\begin{array}{lll}
\mathbb{T} & ::= & X & \text{type variable} \\
 & & \mathbb{T}\rightarrow\mathbb{T} & \text{function type} \\
 & & \forall X{\leq}\mathbb{T}{:}\mathbb{K}.\mathbb{T} & \text{polymorphic type} \\
 & & \Lambda X{:}\mathbb{K}.\mathbb{T} & \text{operator abstraction} \\
 & & \mathbb{T}\,\mathbb{T} & \text{operator application} \\
 & & \bigwedge^{\mathbb{K}}[\mathbb{T}..\mathbb{T}] & \text{intersection at kind } \mathbb{K}
\end{array}$$

We use the abbreviation $\top^K$ for nullary intersections and sometimes $X{:}K$ for $X \leq \top^K{:}K$.

$$\begin{array}{lll}
\top^K & \triangleq & \bigwedge^K[\,] \\
X{:}K & \triangleq & X \leq \top^K{:}K
\end{array}$$

We drop the maximal type *Top* of $F_{\leq}$, since its role is played here by the empty intersection $\top^{\star}$. For technical convenience, we provide kind annotations on bound variables and intersections so that every type has an "obvious kind," which can be read off directly from its structure and the kind declarations in the context.

The language of terms includes the variables (x), applications ($e\,e$) , and functional abstractions ($\lambda x{:}T.e$) of the simply typed λ-calculus, plus the type abstraction ($\lambda X{\leq}T{:}K.e$) and application ($e\,T$) of F^{ω}. As in $F_{\leq}$, each type variable is given an upper bound at the point where it is introduced.

Intersection types are introduced by expressions of the form "for($X{\in}T_1..T_n$)e", which can be read as instructions to the type-checker to analyze the expression e separately under the assumptions $X \equiv T_1$, $X \equiv T_2$, …, $X \equiv T_n$ and conjoin the results. For example, if $+ \in \text{Int}\rightarrow\text{Int}\rightarrow\text{Int} \wedge \text{Real}\rightarrow\text{Real}\rightarrow\text{Real}$, then we can derive

$$\text{for}(X{\in}\text{Int, Real})\lambda x{:}X.x + x \quad \in \quad \text{Int}\rightarrow\text{Int} \wedge \text{Real}\rightarrow\text{Real}.$$

$$\begin{array}{llll}
e & ::= & x & \text{variable} \\
 & & \lambda x{:}\mathbb{T}.e & \text{abstraction} \\
 & & e\,e & \text{application} \\
 & & \lambda X{\leq}\mathbb{T}{:}\mathbb{K}.e & \text{type abstraction} \\
 & & e\,\mathbb{T} & \text{type application} \\
 & & \text{for}(X{\in}\mathbb{T}..\mathbb{T})e & \text{alternation}
\end{array}$$

The operational semantics of $F^{\omega}_{\wedge}$ is given by the following reduction rules on types and terms.

DEFINITION 2.2.1 (*Reduction rules for types*)

1. $(\Lambda X{:}K.T_1)T_2 \rightarrow_{\beta\wedge} T_1[X{\leftarrow}T_2]$

2. $S \to \bigwedge^{\star}[T_1..T_n] \to_{\beta\wedge} \bigwedge^{\star}[S{\to}T_1 \;..\; S{\to}T_n]$

3. $\forall X{\leq}S{:}K.\bigwedge^{\star}[T_1..T_n] \to_{\beta\wedge} \bigwedge^{\star}[\forall X{\leq}S{:}K.T_1 \;..\; \forall X{\leq}S{:}K.T_n]$

4. $\Lambda X{:}K_1.\bigwedge^{K_2}[T_1..T_n] \to_{\beta\wedge} \bigwedge^{K_1\to K_2}[\Lambda X{:}K_1.T_1 \;..\; \Lambda X{:}K_1.T_n]$

5. $(\bigwedge^{K_1\to K_2}[T_1..T_n])\, U \to_{\beta\wedge} \bigwedge^{K_2}[T_1\, U \;..\; T_n\, U]$

6. $\bigwedge^{K}[T_1 \;..\; \bigwedge^{K}[S_1..S_n] \;..\; T_m] \to_{\beta\wedge} \bigwedge^{K}[T_1 \;..\; S_1..S_n \;..\; T_m]$

The first rule is the usual β-reduction rule for types. Rules 2 through 5 express the fact that intersections in positive positions distribute with respect to the other type constructors. Rule 6 states that intersection is an associative operator. In section 2.5 we consider the reduction defined by rules 1 through 5 as $\to_{\beta\wedge^-}$ and the one defined by 6 as $\to_a$ (a comes from associativity). The left-hand side of each reduction rule is a *redex* and the right-hand side its *reduct*. The relation $\to_{\beta\wedge}$ is extended so as to become a compatible relation with respect to type formation, $\twoheadrightarrow_{\beta\wedge}$ is the transitive and reflexive closure of $\to_{\beta\wedge}$, and $=_{\beta\wedge}$ is the least equivalence relation containing $\to_{\beta\wedge}$. The capture-avoiding substitution of S for X in T is written $T[X{\leftarrow}S]$. Substitution is written similarly for terms, and is extended point-wise to contexts. The $\beta\wedge$-normal form of a type S is written S^{nf}, and is extended point-wise to contexts.

Definition 2.2.2 (*Reduction rules for terms*)

1. $(\lambda x{:}T_1.e_1)e_2 \to_{\beta fors} e_1[x{\leftarrow}e_2]$

2. $(\lambda X{\leq}T_1{:}K_1.e)T \to_{\beta fors} e[X{\leftarrow}T]$

3. $(\mathrm{for}(X{\in}T_1..T_n)e_1)e_2 \to_{\beta fors} \mathrm{for}(X{\in}T_1..T_n)(e_1\, e_2)$

4. $\mathrm{for}(X{\in}T_1..T_n)e \to_{\beta fors} e$, if $X \notin \mathrm{FV}(e)$

Rules 1 and 2 are the β-reductions on terms. Rule 3 says that the for constructor can be pushed to the outermost level. We consider the reduction defined by rules 1 through 3 as $\to_{\beta for}$ and the one defined by 4 as $\to_s$ (s comes from simplification). The left-hand side of each reduction rule is a *redex* and the right-hand side its *reduct*. The relation $\to_{\beta fors}$ is extended so as to become a compatible relation with respect to term formation, $\twoheadrightarrow_{\beta fors}$ is the transitive reflexive closure of $\to_{\beta fors}$, and $=_{\beta fors}$ is the least equivalence relation containing $\to_{\beta fors}$.

Contexts

A *context* Γ is a finite sequence of typing and subtyping assumptions for a set of term and type variables.

The empty context is written $\emptyset$. Term variable bindings have the form $x{:}T$; type variable bindings have the form $X{\leq}T{:}K$, where T is the upper bound of X and K is the kind of T.

$$\begin{array}{lll} \Gamma ::= & \emptyset & \text{empty context} \\ & \Gamma,\ x{:}T & \text{term variable declaration} \\ & \Gamma,\ X{\leq}T{:}K & \text{type variable declaration} \end{array}$$

When writing nonempty contexts, we omit the initial $\emptyset$. The domain of Γ is written $\mathrm{dom}(\Gamma)$. The functions FV(—) and FTV(—) give the sets of free term variables and free type variables of a term, type, or context. Since we are careful to ensure that no variable is bound more than once, we sometimes abuse notation and consider contexts as finite functions: $\Gamma(X)$ yields the bound of X in Γ, where X is implicitly asserted to be in $\mathrm{dom}(\Gamma)$.

Types, terms, contexts, statements, and derivations that differ only in the names of bound variables are considered identical. The underlying idea is that variables are de Bruijn indexes [dB72].

Definition 2.2.3 (*Closed*)

1. A term e is closed with respect to a context Γ if $\mathrm{FV}(e) \cup \mathrm{FTV}(e) \subseteq \mathrm{dom}(\Gamma)$.

2. A type T is closed with respect to a context Γ if $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\Gamma)$.

3. A typing statement $\Gamma \vdash e \in T$ is closed if e and T are closed with respect to Γ.

4. A kinding statement $\Gamma \vdash T \in K$ is closed if T is closed with respect to Γ.

5. A subtyping statement $\Gamma \vdash S \leq T$ is closed if S and T are closed with respect to Γ.

We consider only closed typing statements. Observe that in the limit case of the rule T-Meet, when $n = 0$, not having the closure convention would allow nonsensical terms to be typed. On the other hand, the free variable lemma (lemma 2.4.3) guarantees that kinding statements are closed and the well-kindedness of subtyping (lemma 2.4.19) ensures that subtyping statements are closed as well.

Context Formation

The rules for well-formed contexts are the usual ones: a start rule for the empty context and rules allowing a given well-formed context to be extended with either a term variable binding or a type variable binding.

$$\emptyset \vdash \mathrm{ok} \qquad \text{(C-Empty)}$$

$$\frac{\Gamma \vdash T \in \star \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma,\ x{:}T \vdash \mathrm{ok}} \qquad \text{(C-Var)}$$

$$\frac{\Gamma \vdash T \in K \qquad X \notin \mathrm{dom}(\Gamma)}{\Gamma,\ X{\leq}T{:}K \vdash \mathrm{ok}} \qquad \text{(C-TVar)}$$

Type Formation

For each type constructor, we give a rule specifying how it can be used to build well-formed type expressions. The critical rules are K-OAbs and K-OApp, which form type abstractions and type applications (essentially as in a simply typed λ-calculus).

The well-formedness premise $\Gamma \vdash \text{ok}$ in K-Meet (and in T-Meet below) is required for the case where $n = 0$.

$$\frac{\Gamma_1,\, X{\le}T{:}K,\, \Gamma_2 \vdash \text{ok}}{\Gamma_1,\, X{\le}T{:}K,\, \Gamma_2 \vdash X \in K} \qquad (\textsc{K-TVar})$$

$$\frac{\Gamma \vdash T_1 \in \star \qquad \Gamma \vdash T_2 \in \star}{\Gamma \vdash T_1{\to}T_2 \in \star} \qquad (\textsc{K-Arrow})$$

$$\frac{\Gamma,\, X{\le}T_1{:}K_1 \vdash T_2 \in \star}{\Gamma \vdash \forall X{\le}T_1{:}K_1.T_2 \in \star} \qquad (\textsc{K-All})$$

$$\frac{\Gamma,\, X{:}K_1 \vdash T_2 \in K_2}{\Gamma \vdash \Lambda X{:}K_1.T_2 \in K_1{\to}K_2} \qquad (\textsc{K-OAbs})$$

$$\frac{\Gamma \vdash S \in K_1{\to}K_2 \qquad \Gamma \vdash T \in K_1}{\Gamma \vdash S\,T \in K_2} \qquad (\textsc{K-OApp})$$

$$\frac{\Gamma \vdash \text{ok} \qquad \text{for each } i{\in}\{1..n\}\,,\ \Gamma \vdash T_i \in K}{\Gamma \vdash \bigwedge^{K}[T_1..T_n] \in K} \qquad (\textsc{K-Meet})$$

Subtyping

The rules defining the subtype relation are a natural extension of familiar calculi of bounded quantification. Aside from some extra well-formedness conditions, the rules S-Trans, S-TVar, and S-Arrow are the same as in the usual, second-order case. Rules S-OAbs and S-OApp extend the subtype relation point-wise to kinds other than $\star$. The rule of type conversion in F^{ω}, that is, if $\Gamma \vdash e \in T$ and $T =_{\beta} T'$ then $\Gamma \vdash e \in T'$, is captured here as the subtyping rule S-Conv, which also gives reflexivity as a special case. The rule S-All is the rule of Cardelli's Fun language [CW85] in which the bounds of the quantifiers are equal. Rules S-Meet-G and S-Meet-LB specify that an intersection of a set of types is the set's order-theoretic greatest lower bound.

$$\frac{\Gamma \vdash S \in K \qquad \Gamma \vdash T \in K \qquad S =_{\beta\wedge} T}{\Gamma \vdash S \le T} \qquad (\textsc{S-Conv})$$

$$\frac{\Gamma \vdash S \le T \qquad \Gamma \vdash T \le U}{\Gamma \vdash S \le U} \qquad (\textsc{S-Trans})$$

$$\frac{\Gamma_1,\, X{\le}T{:}K,\, \Gamma_2 \vdash \text{ok}}{\Gamma_1,\, X{\le}T{:}K,\, \Gamma_2 \vdash X \le T} \qquad (\textsc{S-TVar})$$

$$\frac{\Gamma \vdash T_1 \le S_1 \qquad \Gamma \vdash S_2 \le T_2 \qquad \Gamma \vdash S_1{\to}S_2 \in \star}{\Gamma \vdash S_1{\to}S_2 \le T_1{\to}T_2} \qquad (\textsc{S-Arrow})$$

$$\frac{\Gamma,\, X{\leq}U{:}K \vdash S \leq T \qquad \Gamma \vdash \forall X{\leq}U{:}K.S \in \star}{\Gamma \vdash \forall X{\leq}U{:}K.S \leq \forall X{\leq}U{:}K.T} \qquad \text{(S-All)}$$

$$\frac{\Gamma,\, X{:}K \vdash S \leq T}{\Gamma \vdash \Lambda X{:}K.S \leq \Lambda X{:}K.T} \qquad \text{(S-OAbs)}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash S\,U \in K}{\Gamma \vdash S\,U \leq T\,U} \qquad \text{(S-OApp)}$$

$$\frac{\text{for each } i{\in}\{1..n\}\,,\ \Gamma \vdash S \leq T_i \qquad \Gamma \vdash S \in K}{\Gamma \vdash S \leq \bigwedge^{K}[T_1..T_n]} \qquad \text{(S-Meet-G)}$$

$$\frac{\Gamma \vdash \bigwedge^{K}[T_1..T_n] \in K}{\Gamma \vdash \bigwedge^{K}[T_1..T_n] \leq T_i} \qquad \text{(S-Meet-LB)}$$

Term Formation

Except for T-Meet and T-For, the term formation rules are precisely those of the second-order calculus of bounded quantification. T-For provides for type checking under any of a set of alternate assumptions. For each S_i, the type derived for the instance of the body e when X is replaced by S_i is a valid type of the for expression itself. The T-Meet rule can then be used to collect these separate typings into a single intersection. Type-theoretically, T-Meet is the introduction rule for the $\wedge$ constructor; the corresponding elimination rule need not be given explicitly, since it follows from T-Subsumption and S-Meet-LB.

$$\frac{\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash \text{ok}}{\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash x \in T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\, x{:}T_1 \vdash e \in T_2}{\Gamma \vdash \lambda x{:}T_1.e \in T_1{\rightarrow}T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash f \in T_1{\rightarrow}T_2 \qquad \Gamma \vdash a \in T_1}{\Gamma \vdash f\,a \in T_2} \qquad \text{(T-App)}$$

$$\frac{\Gamma,\, X{\leq}T_1{:}K_1 \vdash e \in T_2}{\Gamma \vdash \lambda X{\leq}T_1{:}K_1.e \in \forall X{\leq}T_1{:}K_1.T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash f \in \forall X{\leq}T_1{:}K_1.T_2 \qquad \Gamma \vdash S \leq T_1}{\Gamma \vdash f\,S \in T_2[X{\leftarrow}S]} \qquad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash e[X{\leftarrow}S]{\in}T \qquad S \in \{S_1..S_n\}}{\Gamma \vdash \text{for}(X{\in}S_1..S_n)e \in T} \qquad \text{(T-For)}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \text{for each } i{\in}\{1..n\}\,,\ \Gamma \vdash e \in T_i}{\Gamma \vdash e \in \bigwedge^{\star}[T_1..T_n]} \qquad \text{(T-Meet)}$$

$$\frac{\Gamma \vdash e \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T} \qquad \text{(T-Subsumption)}$$

Most of the rules include premises which have two rather different sorts: *structural premises*, which play an essential role in giving the rule its intended semantic force, and *well-formedness premises*, which ensure that the entities named in the rule are of the expected sorts. In an algorithmic presentation of the system (on which an implementation might be based), the well-formation premises would be replaced by the meta-theoretic observation that "recursive calls" in the premises of all the rules preserve the well-formedness of the "arguments" named in the conclusion.

In the interest of brevity, we omit well-formation premises that can be derived from others. For example, in the rule S-Arrow, we drop the premise $\Gamma \vdash T_1 \to T_2 \in \star$, since it follows from $\Gamma \vdash S_1 \to S_2 \in \star$ using the properties proved in section 2.4.

2.2.1 Discussion

An equivalent presentation of intersection types uses binary intersections as in [CDC78]. The intersection of S and T is then written $S \wedge T$, and there is a maximal element at each kind, ω^K. The rules of the system have to be modified according to this alternative notation. In most cases, each of our rules about intersection types has to be replaced by two rules, one for the binary case and another for the maximal element. For example, the reduction rule

$$\forall X{\leq}S{:}K.\bigwedge^{\star}[T_1..T_n] \to_{\beta\wedge} \bigwedge^{\star}[\forall X{\leq}S{:}K.T_1 \,..\, \forall X{\leq}S{:}K.T_n]$$

is replaced by

$$\begin{array}{lll} \forall X{\leq}S{:}K.T_1 \wedge T_2 & \to_{\beta\wedge} & \forall X{\leq}S{:}K.T_1 \wedge \forall X{\leq}S{:}K.T_2 \quad \text{and} \\ \forall X{\leq}S{:}K.\omega^{\star} & \to_{\beta\wedge} & \omega^{\star}. \end{array}$$

Similar replacement takes place for rules 3 through 5 in definition 2.2.1. The term formation rule K-Meet is replaced by the two following rules.

$$\frac{\Gamma \vdash S \in K \qquad \Gamma \vdash T \in K}{\Gamma \vdash S \wedge T \in K} \qquad \text{(K-Int)}$$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \omega^K \in K} \qquad \text{(K-Max)}$$

The rule S-Meet-G is replaced by the following two rules.

$$\frac{\Gamma \vdash S \leq T_1 \qquad \Gamma \vdash S \leq T_2}{\Gamma \vdash S \leq T_1 \wedge T_2} \qquad \text{(S-Int-G)}$$

$$\frac{\Gamma \vdash S \in K}{\Gamma \vdash S \leq \omega^K} \qquad \text{(S-Max)}$$

In the λ-cube [Bar92], F^{ω}corresponds to λ_{ω}, the system defined by the rules $(\star,\star)$, $(\Box,\star)$, and $(\Box,\Box)$. If K is a kind defined by the grammar $\mathbb{K}$, then

$$\Gamma \vdash_{\lambda_{\omega}} K \in \Box.$$

The rule $(\Box,\Box)$ corresponds to the recursive step in the definition of $\mathbb{K}$; the rule $(\star,\star)$ corresponds to K-Arrow, and K-All is the parallel of rule $(\Box,\star)$ enriched with subtyping.

2.3 Confluence

In this section, we show that the system $F^{\omega}_{\wedge}$ is confluent. By that we mean that the reduction $\rightarrow_{\beta fors} \cup \rightarrow_{\beta\wedge}$ defined by putting together the reduction on terms, $\rightarrow_{\beta fors}$ (definition 2.2.2), and the reduction on types, $\rightarrow_{\beta\wedge}$ (definition 2.2.1), satisfies the Church-Rosser property. We use the Hindley-Rosen lemma (c.f. 3.3.5 [Bar84]) to establish this result. This factors the proof into two parts:

1. proving that $\twoheadrightarrow_{\beta fors}$ and $\twoheadrightarrow_{\beta\wedge}$ commute, and
2. proving that $\rightarrow_{\beta fors}$ and $\rightarrow_{\beta\wedge}$ satisfy the Church-Rosser property, the results of sections 2.3.1 and 2.3.2.

Remember that two binary relations $\rightarrow_1$ and $\rightarrow_2$ commute if the following diagram commutes.

$$\begin{tikzcd}
A \arrow[r, "1"'] \arrow[d, "2"'] & B \arrow[d, dotted, "2"] \\
C \arrow[r, dotted, "1"'] & D
\end{tikzcd}$$

In order to prove that $\twoheadrightarrow_{\beta fors}$ and $\twoheadrightarrow_{\beta\wedge}$ commute we use the following lemma.

Lemma 2.3.1 (3.3.6 [Bar84]) Let $\rightarrow_1$ and $\rightarrow_2$ be two binary relations on a set X. Suppose

$$\begin{tikzcd}
A \arrow[r, "1"'] \arrow[d, "2"'] & B \arrow[d, dotted, two heads, "2"] \\
C \arrow[r, dotted, "=1"'] & D
\end{tikzcd}$$

where $\rightarrow_{=1}$ is the reflexive closure of $\rightarrow_1$. Hence $\twoheadrightarrow_1$ and $\twoheadrightarrow_2$ commute.

We need the following auxiliary result to prove that $\twoheadrightarrow_{\beta\wedge}$ and $\twoheadrightarrow_{\beta fors}$ commute using the previous lemma.

Lemma 2.3.2 If $T \rightarrow_{\beta\wedge} T'$, then $e[X\leftarrow T] \twoheadrightarrow_{\beta\wedge} e[X\leftarrow T']$.

Proof: By induction on the structure of T. □

Lemma 2.3.3

$$\begin{tikzcd}
E \arrow[r, "\beta fors"'] \arrow[d, "\beta\wedge"'] & F \arrow[d, dotted, two heads, "\beta\wedge"] \\
G \arrow[r, dotted, "=\beta fors"'] & H
\end{tikzcd}$$

PROOF: By induction on the structure of E. Observe that if E is a type expression ($E \in \mathbb{T}$) then there can be only $\beta\wedge$-reductions starting from E, and the result holds vacuously. Consequently, the meaningful cases are when E is a term ($E \in \mathbb{E}$), and of those the interesting cases are when E is a $\beta$for$s$ redex.

1. $$\begin{array}{lll} E & \equiv & (\lambda x{:}T_1.e_1)e_2, \\ F & \equiv & e_1[x \leftarrow e_2], \\ G & \equiv & (\lambda x{:}T_1'.e_1)e_2, \text{ and} \\ T_1 & \to_{\beta\wedge} & T_1'. \end{array}$$

   Choose $H \equiv F$. Since $(\lambda x{:}T_1'.e_1)e_2 \to_{\beta fors} e_1[x \leftarrow e_2]$, the result follows.

2. $$\begin{array}{lll} E & \equiv & (\lambda X{\leq}T_1{:}K_1.e)S, \\ F & \equiv & e[X \leftarrow S], \\ G & \equiv & (\lambda X{\leq}T_1{:}e.)S', \text{ and} \\ S & \to_{\beta\wedge} & S'. \end{array}$$

   Choose $H \equiv e[X \leftarrow S']$. Since $(\lambda X{\leq}T_1{:}K_1.e)S' \to_{\beta fors} e[X \leftarrow S']$, the result follows by lemma 2.3.2.

3. $$\begin{array}{lll} E & \equiv & (\text{for}(X{\in}T_1..T_i..T_n)e_1)e_2, \\ F & \equiv & \text{for}(X{\in}T_1..T_i..T_n)(e_1e_2), \\ G & \equiv & (\text{for}(X{\in}T_1..T_i'..T_n)e_1)e_2, \text{ and} \\ T_i & \to_{\beta\wedge} & T_i'. \end{array}$$

   Choose $H \equiv \text{for}(X{\in}T_1..T_i'..T_n)(e_1e_2)$.

   $$\begin{array}{llll} \text{Since} & (\text{for}(X{\in}T_1..T_i'..T_n)e_1)e_2 & \to_{\beta fors} & \text{for}(X{\in}T_1..T_i'..T_n)(e_1e_2) \quad \text{and} \\ & \text{for}(X{\in}T_1..T_i..T_n)(e_1e_2) & \to_{\beta\wedge} & \text{for}(X{\in}T_1..T_i'..T_n)(e_1e_2) \end{array}$$

   the result follows.

4. $$\begin{array}{lll} E & \equiv & \text{for}(X{\in}T_1..T_i..T_n)e, \\ F & \equiv & e, \\ G & \equiv & \text{for}(X{\in}T_1..T_i'..T_n)e, \text{ and} \\ T_i & \to_{\beta\wedge} & T_i'. \end{array}$$

   Choose $H \equiv F$. Since $\text{for}(X{\in}T_1..T_i'..T_n)e \to_{\beta fors} e$, the result follows. □

COROLLARY 2.3.4 $\twoheadrightarrow_{\beta\wedge}$ and $\twoheadrightarrow_{\beta fors}$ commute.

2.3.1 The Church-Rosser theorem for $\rightarrow_{\beta\wedge}$

In this section we prove the Church-Rosser property for the reduction defined in 2.2.1. The strategy we use here is similar to the one used in chapter 11 section 1 of [Bar84] to prove the corresponding result for $\rightarrow_{\beta}$ in the type-free λ-calculus.

In order to prove the Church-Rosser property for $\rightarrow_{\beta\wedge}$ it is sufficient to show the following *strip* lemma. If S, T_1, T_2 in $\mathbb{T}$ are such that $S \rightarrow_{\beta\wedge} T_1$ and $S \twoheadrightarrow_{\beta\wedge} T_2$, then there exists T_3 such that $T_1 \twoheadrightarrow_{\beta\wedge} T_3$ and $T_2 \twoheadrightarrow_{\beta\wedge} T_3$. Graphically:

$$\begin{tikzcd}
S \arrow[r, two heads, "\beta\wedge"'] \arrow[d, "\beta\wedge"'] & T_2 \arrow[d, two heads, dotted, "\beta\wedge"] \\
T_1 \arrow[r, two heads, dotted, "\beta\wedge"'] & T_3
\end{tikzcd}$$

The idea of the proof is as follows. Let T_1 be the result of replacing the redex R in S by its reduct R'. If we keep track of what happens with R during the reduction $S \twoheadrightarrow_{\beta\wedge} T_2$, then we can find T_3. To be able to trace R we define a new set of terms $\underline{\mathbb{T}}$ where redexes can appear underlined. Consequently, if we underline R in S we only need to reduce all occurrences of the underlined R in T_2 to obtain T_3.

Definition 2.3.1.1 (*Underlining*)

1. $\underline{\mathbb{T}}$ is the set of terms defined by the following abstract syntax.

$$\begin{array}{rcl}
\underline{\mathbb{T}} & ::= & X \mid \underline{\mathbb{T}}{\rightarrow}\underline{\mathbb{T}} \\
& \mid & \forall X{\leq}\underline{\mathbb{T}}{:}\mathbb{K}.\underline{\mathbb{T}} \mid \Lambda X{:}\mathbb{K}.\underline{\mathbb{T}} \\
& \mid & \underline{\mathbb{T}}\,\underline{\mathbb{T}} \mid \bigwedge^{\mathbb{K}}[\underline{\mathbb{T}}..\underline{\mathbb{T}}] \\
& \mid & (\Lambda X{:}\mathbb{K}.\,\underline{\mathbb{T}})\,\underline{\mathbb{T}} \mid \bigwedge^{\mathbb{K}}[\underline{\mathbb{T}} \mathrel{..} \underline{\bigwedge^{\mathbb{K}}[\underline{\mathbb{T}} \mathrel{..} \underline{\mathbb{T}}]} \mathrel{..} \underline{\mathbb{T}}] \\
& \mid & \underline{\underline{\mathbb{T}} \rightarrow \bigwedge^{\star}[\underline{\mathbb{T}} \mathrel{..} \underline{\mathbb{T}}]} \mid \underline{\forall X{\leq}\underline{\mathbb{T}}{:}\mathbb{K}.\bigwedge^{\star}[\underline{\mathbb{T}} \mathrel{..} \underline{\mathbb{T}}]} \\
& \mid & \underline{\Lambda X{:}\mathbb{K}.\bigwedge^{\mathbb{K}}[\underline{\mathbb{T}} \mathrel{..} \underline{\mathbb{T}}]} \mid \underline{\bigwedge^{\mathbb{K}\rightarrow\mathbb{K}}[\underline{\mathbb{T}} \mathrel{..} \underline{\mathbb{T}}]\,\underline{\mathbb{T}}}
\end{array}$$

Observe that only redexes are underlined.

2. Underlined (one step) reduction $\rightarrow_{\underline{\beta\wedge}}$ is defined starting with the rewriting rules

(a) $(\Lambda X{:}K.T_1)\,T_2 \rightarrow_{\underline{\beta\wedge}} T_1[X{\leftarrow}T_2]$

(b) $S \rightarrow \bigwedge^{\star}[T_1..T_n] \rightarrow_{\underline{\beta\wedge}} \bigwedge^{\star}[S{\rightarrow}T_1 \mathrel{..} S{\rightarrow}T_n]$

(c) $\forall X{\leq}S{:}K.\bigwedge^{\star}[T_1..T_n] \rightarrow_{\underline{\beta\wedge}} \bigwedge^{\star}[\forall X{\leq}S{:}K.T_1 \mathrel{..} \forall X{\leq}S{:}K.T_n]$

(d) $\Lambda X{:}K_1.\bigwedge^{K_2}[T_1..T_n] \rightarrow_{\underline{\beta\wedge}} \bigwedge^{K_1\rightarrow K_2}[\Lambda X{:}K_1.T_1 \mathrel{..} \Lambda X{:}K_1.T_n]$

(e) $\bigwedge^{K_1\rightarrow K_2}[T_1..T_n]\,U \rightarrow_{\underline{\beta\wedge}} \bigwedge^{K_2}[T_1\,U \mathrel{..} T_n\,U]$

(f) $\bigwedge^{K}[T_1 \mathrel{..} \bigwedge^{K}[S_1..S_n] \mathrel{..} T_m] \rightarrow_{\underline{\beta\wedge}} \bigwedge^{K}[T_1 \mathrel{..} S_1..S_n \mathrel{..} T_m]$

(g) $\underline{(\Lambda X{:}K.T_1)\,T_2} \rightarrow_{\underline{\beta\wedge}} T_1[X{\leftarrow}T_2]$

(h) $\underline{S \rightarrow \bigwedge^{\star}[T_1..T_n]} \rightarrow_{\underline{\beta\wedge}} \bigwedge^{\star}[S{\rightarrow}T_1 \;..\; S{\rightarrow}T_n]$

(i) $\underline{\forall X{\leq}S{:}K.\bigwedge^{\star}[T_1..T_n]} \rightarrow_{\underline{\beta\wedge}} \bigwedge^{\star}[\forall X{\leq}S{:}K.T_1 \;..\; \forall X{\leq}S{:}K.T_n]$

(j) $\underline{\Lambda X{:}K_1.\bigwedge^{K_2}[T_1..T_n]} \rightarrow_{\underline{\beta\wedge}} \bigwedge^{K_1\rightarrow K_2}[\Lambda X{:}K_1.T_1 \;..\; \Lambda X{:}K_1.T_n]$

(k) $\underline{\bigwedge^{K_1\rightarrow K_2}[T_1..T_n]\,U} \rightarrow_{\underline{\beta\wedge}} \bigwedge^{K_2}[T_1\,U \;..\; T_n\,U]$.

(l) $\bigwedge^{K}[T_1 \;..\; \underline{\bigwedge^{K}[S_1..S_n]} \;..\; T_m] \rightarrow_{\underline{\beta\wedge}} \bigwedge^{K}[T_1 \;..\; S_1..S_n \;..\; T_m]$

$\rightarrow_{\underline{\beta\wedge}}$ is extended so as to become a compatible relation with respect to $\underline{\mathbb{T}}$, and $\twoheadrightarrow_{\underline{\beta\wedge}}$ is the reflexive and transitive closure of $\rightarrow_{\underline{\beta\wedge}}$.

3. If $T \in \underline{\mathbb{T}}$, then $|T| \in \mathbb{T}$ is obtained from T by erasing all underlinings.

4. The capture avoiding substitution for underlined terms is written as usual, $T[X{\leftarrow}S]$.

Definition 2.3.1.2 The map $\varphi : \underline{\mathbb{T}} \rightarrow \mathbb{T}$ is defined inductively as follows.

1. $\varphi(X) \equiv X$;

2. $\varphi(T_1 \rightarrow T_2) \equiv \varphi(T_1) \rightarrow \varphi(T_2)$;

3. $\varphi(\forall X{\leq}T_1{:}K.T_2) \equiv \forall X{\leq}\varphi(T_1){:}K.\varphi(T_2)$;

4. $\varphi(\Lambda X{:}K.T) \equiv \Lambda X{:}K.\varphi(T)$;

5. $\varphi(T_1\,T_2) \equiv \varphi(T_1)\,\varphi(T_2)$;

6. $\varphi(\bigwedge^{K}[T_1..T_n]) \equiv \bigwedge^{K}[\varphi(T_1) \;..\; \varphi(T_n)]$;

7. $\varphi(\underline{(\Lambda X{:}K.T_1)\,T_2}) \equiv \varphi(T_1)[X{\leftarrow}\varphi(T_2)]$;

8. $\varphi(\underline{S \rightarrow \bigwedge^{\star}[T_1..T_n]}) \equiv \bigwedge^{\star}[\varphi(S \rightarrow T_1) \;..\; \varphi(S \rightarrow T_n)]$;

9. $\varphi(\underline{\forall X{\leq}S{:}K_1.\bigwedge^{\star}[T_1..T_n]}) \equiv \bigwedge^{\star}[\varphi(\forall X{\leq}S{:}K_1.T_1) \;..\; \varphi(\forall X{\leq}S{:}K_1.T_n)]$;

10. $\varphi(\underline{\Lambda X{:}K_1.\bigwedge^{K_2}[T_1..T_n]}) \equiv \bigwedge^{K_1\rightarrow K_2}[\varphi(\Lambda X{:}K_1.T_1) \;..\; \varphi(\Lambda X{:}K_1.T_n)]$;

11. $\varphi(\underline{\bigwedge^{K_1\rightarrow K_2}[T_1..T_n]\,S}) \equiv \bigwedge^{K_2}[\varphi(T_1\,S) \;..\; \varphi(T_n\,S)]$;

12. $\varphi(\bigwedge^{K}[T_1 \;..\; \underline{\bigwedge^{K}[S_1..S_n]} \;..\; T_m]) \equiv \bigwedge^{K}[\varphi(T_1) \;..\; \varphi(S_1) \;..\; \varphi(S_n) \;..\; \varphi(T_m)]$.

Observe that φ reduces all underlined redexes.

Notation: $|T| \equiv S$ and $\varphi(T) \equiv S$ will be written:

$$T \xrightarrow{\;|-|\;} S \text{ and } T \xrightarrow{\;\varphi\;} S.$$

Lemma 2.3.1.3 If T, $S \in \mathbb{T}$ and $T' \in \underline{\mathbb{T}}$ are such that $|T'| \equiv T$ and $T \twoheadrightarrow_{\beta\wedge} S$, then there exists $S' \in \underline{\mathbb{T}}$ such that $T' \twoheadrightarrow_{\underline{\beta\wedge}} S'$ and $|S'| \equiv S$. Graphically:

$$\begin{tikzcd}
T' \arrow[r, dotted, two heads, "\underline{\beta\wedge}"'] \arrow[d, "|-|"'] & S' \arrow[d, "|-|"] \\
T \arrow[r, two heads, "\beta\wedge"'] & S
\end{tikzcd}$$

Proof: By induction on the definition of $\twoheadrightarrow_{\beta\wedge}$.

1. $T \rightarrow_{\beta\wedge} S$ (in one step). Since S is obtained by contracting a redex in T, S' can be obtained by contracting the corresponding redex in T'.

2. $T \twoheadrightarrow_{\beta\wedge} T$. Take $S' \equiv T'$.

3. $T \twoheadrightarrow_{\beta\wedge} U$ and $U \twoheadrightarrow_{\beta\wedge} S$. Finally, the result follows by the induction hypothesis and the transitivity of $\twoheadrightarrow_{\underline{\beta\wedge}}$. □

Lemma 2.3.1.4 Let S, T, and $U \in \underline{\mathbb{T}}$. Then

1. Suppose $X \not\equiv Y$ and $X \notin \mathrm{FV}(U)$. Then
   $S[X \leftarrow T][Y \leftarrow U] \equiv S[Y \leftarrow U][X \leftarrow T[Y \leftarrow U]]$.

2. $\varphi(S[X \leftarrow T]) \equiv \varphi(S)[X \leftarrow \varphi(T)]$.

3. If $S \twoheadrightarrow_{\beta\wedge} S'$, then $S[X \leftarrow U] \twoheadrightarrow_{\beta\wedge} S'[X \leftarrow U]$.

4. If T, $S \in \underline{\mathbb{T}}$ are such that $T \twoheadrightarrow_{\underline{\beta\wedge}} S$, then $\varphi(T) \twoheadrightarrow_{\beta\wedge} \varphi(S)$. Graphically:

$$\begin{tikzcd}
T \arrow[r, two heads, "\underline{\beta\wedge}"'] \arrow[d, "\varphi"'] & S \arrow[d, "\varphi"] \\
\varphi(T) \arrow[r, dotted, two heads, "\beta\wedge"'] & \varphi(S)
\end{tikzcd}$$

Proof:

1. By induction on the structure of S.

2. By induction on the structure of S using (1) in the cases $S \equiv \underline{(\Lambda X{:}K.S_1)\, S_2}$ and $S \equiv (\Lambda X{:}K.S_1)\, S_2$.

3. It is enough to show the result for $\rightarrow_{\beta\wedge}$; the rest follows by induction. The interesting cases are when S is a redex: if S is a β-redex, then the result follows easily using (1); otherwise the result follows easily by the definition of substitution.

4. By induction on the generation of $\twoheadrightarrow_{\underline{\beta\wedge}}$, using (2). □

LEMMA 2.3.1.5 Let $T \in \underline{\mathbb{T}}$. Then $|T| \twoheadrightarrow_{\beta\wedge} \varphi(T)$. Graphically:

$$\begin{tikzcd}
 & T \arrow[dl, "|-|"'] \arrow[dr, "\varphi"] & \\
|T| \arrow[rr, dotted, two heads, "\beta\wedge"] & & \varphi(T)
\end{tikzcd}$$

PROOF: By induction on the structure of T. □

LEMMA 2.3.1.6 (*Strip*) Let S, T_1, and $T_2 \in \mathbb{T}$. If $S \to_{\beta\wedge} T_1$ and $S \twoheadrightarrow_{\beta\wedge} T_2$, then there exists $T_3 \in \mathbb{T}$ such that $T_1 \twoheadrightarrow_{\beta\wedge} T_3$ and $T_2 \twoheadrightarrow_{\beta\wedge} T_3$. Graphically:

$$\begin{tikzcd}
S \arrow[r, two heads, "\beta\wedge"'] \arrow[d, "\beta\wedge"'] & T_2 \arrow[d, dotted, two heads, "\beta\wedge"] \\
T_1 \arrow[r, dotted, two heads, "\beta\wedge"'] & T_3
\end{tikzcd}$$

PROOF: Suppose that T_1 is obtained from S by replacing the occurrence redex R by its reduct R'. Then we can write $S \equiv S[R]$ and $T_1 \equiv S[R']$. Let $S[\underline{R}]$ be obtained from S by replacing R by its underlined version $\underline{R}$. Observe that $|S[\underline{R}]| \equiv S[R]$ and $\varphi(S[\underline{R}]) \equiv S[R']$. Then, by lemma 2.3.1.3, there exists T_2', by lemma 2.3.1.4(4), $S[R'] \twoheadrightarrow_{\beta\wedge} \varphi(T_2')$, and, by lemma 2.3.1.5, $T_2 \twoheadrightarrow_{\beta\wedge} \varphi(T_2')$, which justify the following diagram.

$$\begin{tikzcd}
S[R] \arrow[rr, two heads, "\beta\wedge"] \arrow[dd, "\beta\wedge"'] & & T_2 \arrow[dd, dotted, two heads, "\beta\wedge"] & \\
 & S[\underline{R}] \arrow[ul, "|-|"] \arrow[dl, "\varphi"'] \arrow[rr, dotted, two heads, "\underline{\beta\wedge}"] & & T_2' \arrow[ul, "|-|"] \arrow[dl, "\varphi"'] \\
S[R'] \arrow[rr, dotted, two heads, "\beta\wedge"] & & \varphi(T_2') &
\end{tikzcd}$$

To complete the proof, let $T_3 \equiv \varphi(T_2')$. □

THEOREM 2.3.1.7 (*Church-Rosser for* $\to_{\beta\wedge}$)

If S, T_1, and $T_2 \in \mathbb{T}$ are such that $S \twoheadrightarrow_{\beta\wedge} T_1$ and $S \twoheadrightarrow_{\beta\wedge} T_2$, then there exists $T_3 \in \mathbb{T}$ such that $T_1 \twoheadrightarrow_{\beta\wedge} T_3$ and $T_2 \twoheadrightarrow_{\beta\wedge} T_3$. Graphically:

$$\begin{tikzcd}
S \arrow[r, two heads, "\beta\wedge"'] \arrow[d, two heads, "\beta\wedge"'] & T_2 \arrow[d, dotted, two heads, "\beta\wedge"] \\
T_1 \arrow[r, dotted, two heads, "\beta\wedge"'] & T_3
\end{tikzcd}$$

Proof: By induction on the generation of $S \twoheadrightarrow_{\beta\wedge} T_1$.

1. $S \rightarrow_{\beta\wedge} T_1$. By the strip lemma (2.3.1.6).
2. $S \equiv T_1$. Take $T_3 \equiv T_2$.
3. $S \twoheadrightarrow_{\beta\wedge} T_1'$ and $T_1' \twoheadrightarrow_{\beta\wedge} T_1$. By the induction hypothesis, we can find first T_3' and then T_3, such that $T_1' \twoheadrightarrow_{\beta\wedge} T_3'$, $T_2 \twoheadrightarrow_{\beta\wedge} T_3'$, $T_1 \twoheadrightarrow_{\beta\wedge} T_3$, and $T_3' \twoheadrightarrow_{\beta\wedge} T_3$. Hence the result follows by the transitivity of $\twoheadrightarrow_{\beta\wedge}$. □

2.3.2 The Church-Rosser theorem for $\rightarrow_{\beta fors}$

In this section we prove the Church-Rosser property for the reduction defined in definition 2.2.2. The idea of the proof is as follows. We prove that $\rightarrow_{\beta for}$ and $\rightarrow_s$ are Church-Rosser; that $\rightarrow_s$ reduction steps can be postponed (see lemma 2.3.2.2); and that, if e, e_1, and $e_2 \in \mathbb{E}$ are such that $e \twoheadrightarrow_{\beta for} e_1$ and $e \twoheadrightarrow_s e_2$, there exists e_3 such that $e_1 \twoheadrightarrow_s e_3$ and $e_2 \twoheadrightarrow_{\beta for} e_3$ (see lemma 2.3.2.3).

Those four results allow us to prove the Church-Rosser theorem for $\rightarrow_{\beta fors}$. Let e, e_1, $e_2 \in \mathbb{E}$, such that $e \twoheadrightarrow_{\beta fors} e_1$ and $e \twoheadrightarrow_{\beta fors} e_2$. Then, by s-postponement, there exist f_1 and f_2; by Church-Rosser for $\rightarrow_{\beta for}$, there exists f_3; and, by lemma 2.3.2.3, there exist f_4 and f_5, and finally, by Church-Rosser for $\rightarrow_s$, there exists e_3 which completes the following diagram.

$$\begin{tikzcd}
e \arrow[r, two heads, "\beta for"'] \arrow[d, two heads, "\beta for"'] & f_1 \arrow[r, two heads, "s"'] \arrow[d, two heads, dotted, "\beta for"] & e_1 \arrow[d, two heads, dotted, "\beta for"] \\
f_2 \arrow[r, two heads, dotted, "\beta for"'] \arrow[d, two heads, "s"'] & f_3 \arrow[r, two heads, dotted, "s"'] \arrow[d, two heads, dotted, "s"] & f_4 \arrow[d, two heads, dotted, "s"] \\
e_2 \arrow[r, two heads, dotted, "\beta for"'] & f_5 \arrow[r, two heads, dotted, "s"'] & e_3
\end{tikzcd}$$

In order to prove the s-postponement property we need the following auxiliary lemma. We will consider $\mathrm{FV}(e)$ as the set of free term and type variables of e.

Lemma 2.3.2.1

1. $e_1 \rightarrow_{\beta fors} e_2$ implies $\mathrm{FV}(e_2) \subseteq \mathrm{FV}(e_1)$.
2. $e_1 \rightarrow_s e_2$ implies $e_1[X{\leftarrow}S] \rightarrow_s e_2[X{\leftarrow}S]$.
3. $e_1 \rightarrow_s e_2$ implies $e_1[x{\leftarrow}e] \rightarrow_s e_2[x{\leftarrow}e]$.
4. $e_1 \rightarrow_s e_2$ implies $e[x{\leftarrow}e_1] \twoheadrightarrow_s e[x{\leftarrow}e_2]$.

5. $e_1 \rightarrow_{\beta for} e_2$ implies $e_1[x{\leftarrow}e] \rightarrow_{\beta for} e_2[x{\leftarrow}e]$.

6. $e_1 \rightarrow_{\beta for} e_2$ implies $e[x{\leftarrow}e_1] \twoheadrightarrow_{\beta for} e[x{\leftarrow}e_2]$.

7. $e_1 \twoheadrightarrow_s e_2$ and $f_1 \twoheadrightarrow_s f_2$ implies $f_1[x{\leftarrow}e_1] \twoheadrightarrow_s f_2[x{\leftarrow}e_2]$.

8. $e_1 \twoheadrightarrow_{\beta for} e_2$ and $f_1 \twoheadrightarrow_{\beta for} f_2$ implies $f_1[x{\leftarrow}e_1] \twoheadrightarrow_{\beta for} f_2[x{\leftarrow}e_2]$.

Proof: Items 1 through 6 follow by induction on the structure of e_1; item 7 is a corollary of items 3 and 4, and item 8 is a corollary of items 5 and 6. □

Lemma 2.3.2.2 (*s-postponement*) If $e \rightarrow_s e_1$ and $e_1 \rightarrow_{\beta for} e_2$, then there exists e_3 such that $e \rightarrow_{\beta for} e_3$ and $e_3 \twoheadrightarrow_s e_1$.

Proof: By induction on the structure of e, using 2.3.2.1(1) for the case $e \equiv \text{for}(X{\in}T_1..T_n)f$; 2.3.2.1(3) and (4) for the case $e \equiv (\lambda x{:}T.f_1)f_2$; and 2.3.2.1(2) for the case $e \equiv (\lambda X{\leq}T{:}K.f)S$. □

Lemma 2.3.2.3 If e, e_1, and $e_2 \in \mathbb{E}$ are such that $e \twoheadrightarrow_{\beta for} e_1$ and $e \twoheadrightarrow_s e_2$ then there exists e_3 such that $e_1 \twoheadrightarrow_s e_3$ and $e_2 \twoheadrightarrow_{\beta for} e_3$. Graphically:

$$\begin{tikzcd}
e \arrow[r, two heads, "\beta for"'] \arrow[d, two heads, "s"'] & e_1 \arrow[d, dotted, two heads, "s"] \\
e_2 \arrow[r, dotted, two heads, "\beta for"'] & e_3
\end{tikzcd}$$

In order to prove this lemma, we prove first the corresponding result for a one step $\rightarrow_{\beta for}$ reduction.

Lemma 2.3.2.4 If e, e_1, and $e_2 \in \mathbb{E}$ are such that $e \rightarrow_{\beta for} e_1$ and $e \twoheadrightarrow_s e_2$, then there exists e_3 such that $e_1 \twoheadrightarrow_s e_3$ and $e_2 \twoheadrightarrow_{\beta for} e_3$. Graphically:

$$\begin{tikzcd}
e \arrow[r, "\beta for"'] \arrow[d, two heads, "s"'] & e_1 \arrow[d, dotted, two heads, "s"] \\
e_2 \arrow[r, dotted, two heads, "\beta for"'] & e_3
\end{tikzcd}$$

Proof of lemma 2.3.2.3: By induction on the derivation of $e \twoheadrightarrow_{\beta for} e_1$, using lemma 2.3.2.4. □

We now prove the Church-Rosser property for $\rightarrow_s$ using the Newman's proposition 3.1.25 in [Bar84], by proving that $\rightarrow_s$ is strongly normalizing and weak Church-Rosser.

Lemma 2.3.2.5 (*Strong normalization for* $\rightarrow_s$) Every s-reduction sequence starting from a term e terminates.

Proof: Straightforward, by induction on the number of symbols of the term being reduced. □

Lemma 2.3.2.6 (*Weak Church-Rosser for* $\to_s$) If e, e_1, and $e_2 \in \mathbb{E}$ are such that $e \to_s e_1$ and $e \to_s e_2$, then there exists e_3 such that $e_1 \twoheadrightarrow_s e_3$ and $e_2 \twoheadrightarrow_s e_3$. Graphically:

$$\begin{tikzcd}
e \arrow[r, "s"'] \arrow[d, "s"'] & e_1 \arrow[d, dotted, two heads, "s"] \\
e_2 \arrow[r, dotted, two heads, "s"'] & e_3
\end{tikzcd}$$

Proof: By induction on the structure of e, using 2.3.2.1(1) for the case $e \equiv \mathrm{for}(X{\in}T_1..T_n)f$. □

Corollary 2.3.2.7 (*Church-Rosser for* $\to_s$) If e, e_1, and $e_2 \in \mathbb{E}$ are such that $e \twoheadrightarrow_s e_1$ and $e \twoheadrightarrow_s e_2$, then there exists e_3 such that $e_1 \twoheadrightarrow_s e_3$ and $e_2 \twoheadrightarrow_s e_3$. Graphically:

$$\begin{tikzcd}
e \arrow[r, two heads, "s"'] \arrow[d, two heads, "s"'] & e_1 \arrow[d, dotted, two heads, "s"] \\
e_2 \arrow[r, dotted, two heads, "s"'] & e_3
\end{tikzcd}$$

We now prove the Church-Rosser theorem for the $\to_{\beta for}$ reduction. This result is obtained following a similar strategy to the one used to prove the corresponding properties for $\to_{\beta\wedge}$, the reduction on types, in section 2.3.1. In order to prove the Church-Rosser property for $\to_{\beta for}$, it is sufficient to show the following *strip* lemma. If e, f_1, and f_2 in $\mathbb{E}$ are such that $e \to_{\beta for} f_1$ and $e \twoheadrightarrow_{\beta for} f_2$, then there exists f_3 such that $f_1 \twoheadrightarrow_{\beta for} f_3$ and $f_2 \twoheadrightarrow_{\beta for} f_3$. Graphically:

$$\begin{tikzcd}
e \arrow[r, two heads, "\beta for"'] \arrow[d, "\beta for"'] & f_2 \arrow[d, dotted, two heads, "\beta for"] \\
f_1 \arrow[r, dotted, two heads, "\beta for"'] & f_3
\end{tikzcd}$$

The idea of the proof is as follows. Let f_1 be the result of replacing the redex R in e by its reduct R'. If we keep track of what happens with R during the reduction $e \twoheadrightarrow_{\beta for} f_2$, then we can find f_3. To be able to trace R we define a new set of terms $\underline{\mathbb{E}}$ where redexes can appear underlined. Then if we underline R in e we only need to reduce all occurrences of the underlined R in f_2 to obtain f_3.

Definition 2.3.2.8 (*Underlining*)

1. $\underline{\mathbb{E}}$ is the set of terms defined by the following abstract syntax.

$$\begin{array}{rcl}
\underline{\mathbb{E}} & ::= & x \\
& | & \lambda x{:}\mathbb{E}.\underline{\mathbb{E}} \\
& | & \underline{\mathbb{E}}\,\underline{\mathbb{E}} \\
& | & \lambda X{\leq}\mathbb{T}{:}\mathbb{K}.\underline{\mathbb{E}} \\
& | & \underline{\mathbb{E}}\,\mathbb{T} \\
& | & \mathrm{for}(X{\in}\mathbb{T}..\mathbb{T})\underline{\mathbb{E}} \\
& | & \underline{(\lambda x{:}\mathbb{T}.\underline{\mathbb{E}})\underline{\mathbb{E}}} \\
& | & \underline{(\lambda X{\leq}\mathbb{T}{:}\mathbb{K}.\underline{\mathbb{E}})\mathbb{T}} \\
& | & \underline{(\mathrm{for}(X{\in}\mathbb{T}..\mathbb{T})\underline{\mathbb{E}})\underline{\mathbb{E}}}
\end{array}$$

   Observe that only redexes are underlined.

2. Underlined (one step) reduction $\rightarrow_{\underline{\beta for}}$ is defined starting with the rewriting rules

   (a) $(\lambda x{:}T_1.e_1)e_2 \rightarrow_{\underline{\beta for}} e_1[x{\leftarrow}e_2]$

   (b) $(\lambda X{\leq}T_1{:}K_1.e)T \rightarrow_{\underline{\beta for}} e[X{\leftarrow}T]$

   (c) $(\mathrm{for}(X{\in}T_1..T_n)e_1)e_2 \rightarrow_{\underline{\beta for}} \mathrm{for}(X{\in}T_1..T_n)e_1\, e_2$

   (d) $\underline{(\lambda x{:}T_1.)e_1e_2} \rightarrow_{\underline{\beta for}} e_1[x{\leftarrow}e_2]$

   (e) $\underline{(\lambda X{\leq}T_1{:}K_1.e)T} \rightarrow_{\underline{\beta for}} e[X{\leftarrow}T]$

   (f) $\underline{(\mathrm{for}(X{\in}T_1..T_n)e_1)e_2} \rightarrow_{\underline{\beta for}} \mathrm{for}(X{\in}T_1..T_n)(e_1\, e_2)$

   $\rightarrow_{\underline{\beta for}}$ is extended so as to become a compatible relation with respect to $\underline{\mathbb{E}}$, and $\twoheadrightarrow_{\underline{\beta for}}$ is the transitive reflexive closure of $\rightarrow_{\underline{\beta for}}$.

3. If $e \in \underline{\mathbb{E}}$, then $|e| \in \mathbb{E}$ is obtained from e by erasing all underlinings.

4. The capture-avoiding substitution for underlined terms is written as usual, $e[X{\leftarrow}S]$ and $e[x{\leftarrow}f]$.

Definition 2.3.2.9 The map $\varphi : \underline{\mathbb{E}} \rightarrow \mathbb{E}$ is defined inductively as follows.

1. $\varphi(x) \equiv x$;

2. $\varphi(\lambda x{:}T.e) \equiv \lambda x{:}T.\varphi(e)$;

3. $\varphi(e_1e_2) \equiv \varphi(e_1)\varphi(e_2)$;

4. $\varphi(\lambda X{\leq}T{:}K.e) \equiv \lambda X{\leq}T{:}K.\varphi(e)$;

5. $\varphi(e_1T) \equiv \varphi(e_1)\,T$;

6. $\varphi(\mathrm{for}(X{\in}T_1..T_n)e) \equiv \mathrm{for}(X{\in}T_1..T_n)\varphi(e)$;

7. $\varphi(\underline{(\lambda x{:}T.e_1)e_2}) \equiv \varphi(e_1)[x{\leftarrow}\varphi(e_2)]$;

8. $\varphi(\underline{(\lambda X{\leq}T{:}K.e)T}) \equiv \varphi(e)[X{\leftarrow}T]$;

9. $\varphi(\underline{\text{for}(X{\in}T_1..T_n))e_1)e_2}) \equiv \text{for}(X{\in}T_1..T_n)\varphi(e_1\, e_2)$.

Observe that φ reduces all underlined redexes.

Notation: $|e_1| \equiv e_2$ and $\varphi(e_1) \equiv e_2$ will be written:

$$e_1 \xrightarrow{|-|} e_2 \text{ and } e_1 \xrightarrow{\varphi} e_2.$$

Lemma 2.3.2.10 If e, $f \in \mathbb{E}$ and $e' \in \underline{\mathbb{E}}$ are such that $|e'| \equiv e$ and $e \twoheadrightarrow_{\beta for} f$, then there exists $f' \in \underline{\mathbb{E}}$ such that $e' \twoheadrightarrow_{\underline{\beta for}} f'$ and $|f'| \equiv f$. Graphically:

$$\begin{tikzcd}
e' \arrow[r, dotted, two heads, "\underline{\beta for}"'] \arrow[d, "|-|"'] & f' \arrow[d, "|-|"] \\
e \arrow[r, two heads, "\beta for"'] & f
\end{tikzcd}$$

Proof: By induction on the definition of $\twoheadrightarrow_{\beta for}$.

1. $e \rightarrow_{\beta for} f$ (in one step). Then f is obtained by contracting a redex in e. f' can be obtained by contracting the corresponding redex in e'.

2. $e \twoheadrightarrow_{\beta for} e$. Take $f' \equiv e'$.

3. $e \twoheadrightarrow_{\beta for} f_1$ and $f_1 \twoheadrightarrow_{\beta for} f$. Hence the result follows by the induction hypothesis and the transitivity of $\twoheadrightarrow_{\underline{\beta for}}$. □

Lemma 2.3.2.11 Let e, f, and $g \in \underline{\mathbb{E}}$ and $S, T \in \mathbb{T}$. Then

1. (a) Suppose $x \neq y$ and $x \notin \text{FV}(g)$. Then
   $e[x{\leftarrow}f][y{\leftarrow}g] \equiv e[y{\leftarrow}g][x{\leftarrow}f[y{\leftarrow}g]]$.
   (b) Suppose $X \neq Y$ and $X \notin \text{FV}(S)$. Then
   $e[X{\leftarrow}T][Y{\leftarrow}S] \equiv e[Y{\leftarrow}S][X{\leftarrow}T[Y{\leftarrow}S]]$.
   (c) Suppose $X \neq Y$. Then
   $e[x{\leftarrow}f][X{\leftarrow}T] \equiv e[X{\leftarrow}T][x{\leftarrow}f[X{\leftarrow}T]]$.

2. (a) $\varphi(e[x{\leftarrow}f]) \equiv \varphi(e)[x{\leftarrow}\varphi(f)]$.
   (b) $\varphi(e[X{\leftarrow}T]) \equiv \varphi(e)[X{\leftarrow}T]$.

3. If e and $f \in \underline{\mathbb{E}}$ are such that $e \twoheadrightarrow_{\underline{\beta for}} f$, then $\varphi(e) \twoheadrightarrow_{\beta for} \varphi(f)$. Graphically:

$$\begin{tikzcd}
e \arrow[r, two heads, "\underline{\beta for}"'] \arrow[d, "\varphi"'] & f \arrow[d, "\varphi"] \\
\varphi(e) \arrow[r, dotted, two heads, "\beta for"'] & \varphi(f)
\end{tikzcd}$$

PROOF:

1. By induction on the structure of e.

2. By induction on the structure of e, using (1).

3. By induction on the generation of $\twoheadrightarrow_{\underline{\beta for}}$, using (2). □

LEMMA 2.3.2.12 If $e \in \underline{\mathbb{E}}$, then $|e| \twoheadrightarrow_{\beta for} \varphi(e)$. Graphically:

$$\begin{tikzcd}
 & e \arrow[dl, "|-|"'] \arrow[dr, "\varphi"] & \\
|e| \arrow[rr, dotted, two heads, "\beta for"] & & \varphi(e)
\end{tikzcd}$$

PROOF: By induction on the structure of e, using 2.3.2.1(8). □

LEMMA 2.3.2.13 (*Strip*) If e, f_1, and $f_2 \in \mathbb{E}$ are such that $e \rightarrow_{\beta for} f_1$ and $e \twoheadrightarrow_{\beta for} f_2$, then there exists $f_3 \in \mathbb{E}$ such that $f_1 \twoheadrightarrow_{\beta for} f_3$ and $f_2 \twoheadrightarrow_{\beta for} f_3$. Graphically:

$$\begin{tikzcd}
e \arrow[r, two heads, "\beta for"'] \arrow[d, "\beta for"'] & f_2 \arrow[d, dotted, two heads, "\beta for"] \\
f_1 \arrow[r, dotted, two heads, "\beta for"'] & f_3
\end{tikzcd}$$

PROOF: Suppose that f_1 is obtained from e by replacing the occurrence redex R with its reduct R'. Then we can write $e \equiv e[R]$ and $f_1 \equiv e[R']$. Let $e[\underline{R}]$ be obtained from e by replacing R by its underlined version $\underline{R}$. Observe that $|e[\underline{R}]| \equiv e[R]$ and $\varphi(e[\underline{R}]) \equiv e[R']$. Then, by lemma 2.3.2.10, there exists f_2'; by lemma 2.3.2.11(3), $e[R'] \twoheadrightarrow_{\beta for} \varphi(f_2')$, and, by lemma 2.3.2.12, $f_2 \twoheadrightarrow_{\beta for} \varphi(f_2')$, which justify the following diagram.

$$\begin{tikzcd}
e[R] \arrow[rr, two heads, "\beta for"] \arrow[dd, "\beta for"'] & & f_2 \arrow[dd, dotted, two heads, "\beta for"] & \\
 & e[\underline{R}] \arrow[ul, "|-|"'] \arrow[dl, "\varphi"] \arrow[rr, dotted, two heads, "\underline{\beta for}"] & & f_2' \arrow[ul, "|-|"'] \arrow[dl, "\varphi"] \\
e[R'] \arrow[rr, dotted, two heads, "\beta for"] & & \varphi(f_2') &
\end{tikzcd}$$

To complete the proof, let $f_3 \equiv \varphi(f_2')$. □

THEOREM 2.3.2.14 (*Church-Rosser for* $\rightarrow_{\beta for}$)
If e, f_1, and $f_2 \in \mathbb{E}$ are such that $e \twoheadrightarrow_{\beta for} f_1$ and $e \twoheadrightarrow_{\beta for} f_2$, then there exists $f_3 \in \mathbb{E}$ such that $f_1 \twoheadrightarrow_{\beta for} f_3$ and $f_2 \twoheadrightarrow_{\beta for} f_3$. Graphically:

$$\begin{tikzcd} e \arrow[r, two heads, "\beta for"'] \arrow[d, two heads, "\beta for"'] & f_2 \arrow[d, two heads, dotted, "\beta for"] \\ f_1 \arrow[r, two heads, dotted, "\beta for"'] & f_3 \end{tikzcd}$$

PROOF: By induction on the generation of $e \twoheadrightarrow_{\beta for} f_1$.

1. $e \rightarrow_{\beta for} f_1$. By the strip lemma.
2. $e \equiv f_1$. Take $f_3 \equiv f_2$.
3. $e \twoheadrightarrow_{\beta for} f_1'$ and $f_1' \twoheadrightarrow_{\beta for} f_1$. By the induction hypothesis we can find first f_3' and then f_3, such that $f_1' \twoheadrightarrow_{\beta for} f_3'$, $f_2 \twoheadrightarrow_{\beta for} f_3'$, $f_1 \twoheadrightarrow_{\beta for} f_3$, and $f_3' \twoheadrightarrow_{\beta for} f_3$. Hence the result follows by the transitivity of $\twoheadrightarrow_{\beta for}$. □

We have proved the confluence of the reduction $\rightarrow_{\beta fors}$ on terms.

THEOREM 2.3.2.15 (*Church-Rosser for* $\rightarrow_{\beta fors}$)
Let $e, f_1, f_2 \in \mathbb{E}$. If $e \twoheadrightarrow_{\beta fors} f_1$ and $e \twoheadrightarrow_{\beta fors} f_2$, then there exists $f_3 \in \mathbb{E}$ such that $f_1 \twoheadrightarrow_{\beta fors} f_3$ and $f_2 \twoheadrightarrow_{\beta fors} f_3$. Graphically:

$$\begin{tikzcd} e \arrow[r, two heads, "\beta fors"'] \arrow[d, two heads, "\beta fors"'] & f_2 \arrow[d, two heads, dotted, "\beta fors"] \\ f_1 \arrow[r, two heads, dotted, "\beta fors"'] & f_3 \end{tikzcd}$$

Finally, we can state and prove the confluence property for the reduction relation of $F^{\omega}_{\wedge}$.

THEOREM 2.3.2.16 (*Church-Rosser for* $\rightarrow_{\beta fors} \cup \rightarrow_{\beta\wedge}$)
If E, F, and $G \in \mathbb{T} \cup \mathbb{E}$ are such that $E \twoheadrightarrow_{\beta fors \cup \beta\wedge} F$ and $E \twoheadrightarrow_{\beta fors \cup \beta\wedge} G$, then there exists $H \in \mathbb{T} \cup \mathbb{E}$ such that $F \twoheadrightarrow_{\beta fors \cup \beta\wedge} H$ and $G \twoheadrightarrow_{\beta fors \cup \beta\wedge} H$. Graphically:

$$\begin{tikzcd} E \arrow[r, two heads, "\beta fors \cup \beta\wedge"'] \arrow[d, two heads, "\beta fors \cup \beta\wedge"'] & F \arrow[d, two heads, dotted, "\beta fors \cup \beta\wedge"] \\ G \arrow[r, two heads, dotted, "\beta fors \cup \beta\wedge"'] & H \end{tikzcd}$$

PROOF: By the commutativity of $\twoheadrightarrow_{\beta fors}$ and $\twoheadrightarrow_{\beta\wedge}$ (corollary 2.3.4) and the Church-Rosser property of $\rightarrow_{\beta fors}$ and $\rightarrow_{\beta\wedge}$ (theorems 2.3.1.7 and 2.3.2.15). □

The Church-Rosser theorem has interesting corollaries that we will use in the sequel.

COROLLARY 2.3.2.17 See chapter 3 [Bar84]. Let R be a reduction satisfying the Church-Rosser property. Then

1. If $T{=}_R S$, then there exists U such that $T \twoheadrightarrow_R U$ and $S \twoheadrightarrow_R U$.
2. If T is a normal form of S, then $S \twoheadrightarrow_R T$.
3. Each term has at most one R-normal form.

FACT 2.3.2.18

1. $\forall X{\leq}S{:}K.T =_{\beta\wedge} \top^\star$ if and only if $T =_{\beta\wedge} \top^\star$.
2. $\Lambda X{:}K.T =_{\beta\wedge} \top^\star$ if and only if $T =_{\beta\wedge} \top^\star$.
3. $S{\rightarrow}T =_{\beta\wedge} \top^\star$ if and only if $T =_{\beta\wedge} \top^\star$.
4. $T\,S =_{\beta\wedge} \top^\star$ if and only if $T =_{\beta\wedge} \top^\star$.

2.4 Structural properties

This section establishes a number of structural properties of $F^\omega_\wedge$. Except where noted, the proofs proceed by structural induction and are straightforward when performed in the order in which they appear.

LEMMA 2.4.1 If $\Gamma \vdash \Sigma$ and Γ_1 is a prefix of Γ, then $\Gamma_1 \vdash \mathrm{ok}$ as a subderivation. Moreover, except for the case $\Gamma_1 \equiv \Gamma$ and $\Sigma \equiv \mathrm{ok}$, the subderivation is strictly shorter.

LEMMA 2.4.2 (*Syntax-directedness of context judgements*)

1. If $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \mathrm{ok}$, then $\Gamma_1 \vdash T \in K$ by a proper subderivation.
2. If $\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash \mathrm{ok}$, then $\Gamma_1 \vdash T \in \star$ by a proper subderivation.

LEMMA 2.4.3 (*Free variables*)

1. If $\Gamma \vdash T \in K$, then $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\Gamma)$.
2. If $\Gamma \vdash \mathrm{ok}$, then each variable or type variable in $\mathrm{dom}(\Gamma)$ is declared only once.

LEMMA 2.4.4 (*Weakening/Permutation*) Let Γ and Γ' be contexts such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \mathrm{ok}$. Then $\Gamma \vdash \Sigma$ implies $\Gamma' \vdash \Sigma$.

Proof: By induction on the length of a derivation of $\Gamma \vdash \Sigma$.

K-OAbs We are given that $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K_2$. Applying K-Meet to $\Gamma' \vdash \mathrm{ok}$ we obtain $\Gamma' \vdash \top^{K_1} \in K_1$; we can assume, without loss of generality, that $X \notin \mathrm{dom}(\Gamma')$. Then, by C-TVar, $\Gamma',\ X{\leq}\top^{K_1}{:}K_1 \vdash \mathrm{ok}$. By the induction hypothesis, $\Gamma',\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K_2$, and the result follows applying K-OAbs.

T-Abs We are given that $\Gamma,\ x{:}T_1 \vdash e \in T_2$. By lemma 2.4.1 there exists a proper subderivation of $\Gamma,\ x{:}T_1 \vdash \mathrm{ok}$; by lemma 2.4.2, there is a yet shorter subderivation of $\Gamma \vdash T_1 \in \star$. We can now apply the induction hypothesis to obtain $\Gamma' \vdash T_1 \in \star$. As before, we can assume $x \notin \mathrm{dom}(\Gamma')$; by C-Var, $\Gamma',\ x{:}T_1 \vdash \mathrm{ok}$. By the induction hypothesis, we have $\Gamma',\ x{:}T_1 \vdash e \in T_2$, and applying T-Abs yields the desired result.

Other cases If $\Sigma \equiv \mathrm{ok}$ there is nothing to prove. K-TVar, S-TVar, T-Meet and T-Var applying the corresponding rule to $\Gamma' \vdash \mathrm{ok}$. S-OAbs similar to K-OAbs. K-All, S-All T-TAbs similar to T-Abs. All the other cases follow by straightforward application of the induction hypothesis. □

Lemma 2.4.5 (*Context, kind, and term strengthening*)

1. If $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \mathrm{ok}$ and $X \notin \mathrm{FTV}(\Gamma_2)$, then $\Gamma_1,\ \Gamma_2 \vdash \mathrm{ok}$.
2. If $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash S \in K'$ and $X \notin \mathrm{FTV}(\Gamma_2) \cup \mathrm{FTV}(S)$, then $\Gamma_1,\ \Gamma_2 \vdash S \in K'$.
3. If $\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash \Sigma$ and $x \notin \mathrm{FV}(\Sigma)$, then $\Gamma_1,\ \Gamma_2 \vdash \Sigma$.

Moreover, the derivations of the conclusions are strictly shorter than the derivation of the premises.

Proof: We prove statements 1 and 2 by simultaneous induction on the length of derivations, and statement 3 by induction on the derivation of $\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash \Sigma$.

1. C-Empty Vacuously true.

   C-Var By part 2 of the induction hypothesis and C-Var.

   C-TVar $\Gamma_2 \equiv \emptyset$. The result follows from lemma 2.4.1.

   $\Gamma_2 \not\equiv \emptyset$. By part 2 of the induction hypothesis and C-TVar.

2. K-TVar By part 1 of the induction hypothesis and K-TVar.

   K-Arrow By part 2 of the induction hypothesis and K-Arrow.

   K-All We are given that $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2,\ Y{\leq}T_1{:}K_1 \vdash T_2 \in \star$, and $X \notin \mathrm{FTV}(\Gamma_2) \cup \mathrm{FTV}(\forall(Y{\leq}T_1{:}K_1)T_2)$. In particular, $X \notin \mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2) - \{Y\}$. Observe that, by lemma 2.4.3, $X \not\equiv Y$. Then $X \notin \mathrm{FTV}(\Gamma_2,\ Y{\leq}T_1{:}K_1) \cup \mathrm{FTV}(T_2)$. Applying part 2 of the induction hypothesis and K-All the result follows.

K-OAbs Similar to the case K-All.

K-OApp By part 2 of the induction hypothesis and K-OApp.

K-Meet By parts 1 and 2 of the induction hypothesis and K-OApp.

3. Except for the cases we consider below and the case for C-Empty, which is trivially true, the result follows by straightforward application of the induction hypothesis and the corresponding rule in each case.

   C-Var $\Gamma_2 \equiv \emptyset$. The result follows by lemma 2.4.1.
   $\Gamma_2 \not\equiv \emptyset$. By the induction hypothesis and C-Var.

   T-Abs Using lemma 2.4.3, the induction hypothesis, and T-Abs.

   T-For Using that $\mathrm{FV}(\mathrm{for}(X \in T_1..T_n)e) = \mathrm{FV}(e) = \mathrm{FV}(e[X \leftarrow S])$, the induction hypothesis, and T-For. □

Proposition 2.4.6 (*Syntax-directedness of kinding/Generation for kinding*)

1. $\Gamma \vdash X \in K$ implies $\Gamma \equiv \Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2$ for some Γ_1, T, and Γ_2.
2. $\Gamma \vdash T_1 \rightarrow T_2 \in K$ implies $K \equiv \star$ and $\Gamma \vdash T_1, T_2 \in \star$.
3. $\Gamma \vdash \forall X{\leq}T_1{:}K_1.T_2 \in K$ implies $K \equiv \star$ and $\Gamma,\ X{\leq}T_1{:}K_1 \vdash T_2 \in \star$.
4. $\Gamma \vdash \Lambda(X{:}K_1)T_2 \in K$ implies $K \equiv K_1 \rightarrow K_2$ and $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K_2$, for some K_2.
5. $\Gamma \vdash S\,T \in K$ implies $\Gamma \vdash S \in K' \rightarrow K$ and $\Gamma \vdash T \in K'$, for some K'.
6. $\Gamma \vdash \bigwedge^K[T_1..T_n] \in K'$ implies $K \equiv K'$ and $\Gamma \vdash \mathrm{ok}$ and $\Gamma \vdash T_i \in K$ for each i.

Moreover, the proofs of the consequents are all strictly shorter than those of the antecedents.

Proof: In each case the antecedent uniquely determines the last rule of its derivation. The proof follows by inspection of the rules. □

Lemma 2.4.7 (*Uniqueness of kinds*) If $\Gamma \vdash T \in K$ and $\Gamma \vdash T \in K'$, then $K \equiv K'$.

Definition 2.4.8 (*Size*)

1. The size of a type expression T, $\mathit{size}_t(T)$, is defined as follows.

   (a) $\mathit{size}_t(X) = 2$,

   (b) $\mathit{size}_t(S \rightarrow T) = \mathit{size}_t(\forall X{\leq}S{:}K.T) = \mathit{size}_t(S\,T) = \mathit{size}_t(S) + \mathit{size}_t(T) + 1$,

   (c) $\mathit{size}_t(\Lambda X{:}K.T) = \mathit{size}_t(T) + 3$,

   (d) $\mathit{size}_t(\bigwedge^K[T_1..T_n]) = 2 + \Sigma_{1 \leq i \leq n} \mathit{size}_t(T_i)$.

2. The homomorphic extension to contexts, $size_c(\Gamma)$, is defined as follows.
   (a) $size_c(\emptyset) = 0$,
   (b) $size_c(\Gamma,\, X \leq T{:}K) = size_c(\Gamma,\, x{:}T) = size_c(\Gamma) + size_t(T)$.
3. The size of a subtyping, kinding, or ok judgement J, $size_j(J)$, is defined as follows.
   (a) $size_j(\Gamma \vdash \text{ok}) = size_c(\Gamma) + 1$,
   (b) $size_j(\Gamma \vdash T \in K) = size_c(\Gamma) + size_t(T)$.
   (c) $size_j(\Gamma \vdash S \leq T) = size_c(\Gamma) + size_t(S) + size_t(T)$.

Lemma 2.4.9 (*Well-foundedness of context formation and kinding rules*)

1. For every kinding or ok judgement J, $size_j(\emptyset \vdash \text{ok}) \leq size_j(J)$.
2. If $\dfrac{J_1 \,..\, J_n}{J}$ is a kinding rule or a context formation rule, then $size_j(J_i) < size_j(J)$ for each $i{\in}\{1..n\}$.

Corollary 2.4.10

1. For any context Γ it is decidable whether $\Gamma \vdash \text{ok}$.
2. For any context Γ, type expression T, and kind K, it is decidable whether $\Gamma \vdash T \in K$.

Proof: Lemma 2.4.2 and proposition 2.4.6 imply that context formation rules and kinding rules determine an algorithm to check context judgements and kinding judgements and lemma 2.4.9 implies that the algorithm terminates. □

Lemma 2.4.11 (*Type substitution*) Let $\Gamma_1 \vdash T \in K_U$. Then

1. If $\Gamma_1,\, X{\leq}U{:}K_U,\, \Gamma_2 \vdash S \in K_S$, then $\Gamma_1,\, \Gamma_2[X{\leftarrow}T] \vdash S[X{\leftarrow}T] \in K_S$.
2. If $\Gamma_1,\, X{\leq}U{:}K_U,\, \Gamma_2 \vdash \text{ok}$, then $\Gamma_1,\, \Gamma_2[X{\leftarrow}T] \vdash \text{ok}$.

Proof: By simultaneous induction on derivations of the premises. The proof of part 2 is straightforward using part 1 of the induction hypothesis. We consider the details of the proof of 1. The cases K-Arrow, K-All, K-OAbs, and K-OApp follow by straightforward application of part 1 of the induction hypothesis and the corresponding rule, while the case of K-Meet also uses part 2 of the induction hypothesis. We examine the case of K-TVar, where $S \equiv Y$ for some variable Y. By proposition 2.4.6(1) $Y{\leq}T_Y{:}K_S \in (\Gamma_1,\, X{\leq}U{:}K_U,\, \Gamma_2)$ for some T_Y. There are three cases to consider.

$Y{\leq}T_Y{:}K_S{\in}\Gamma_1$ Then we also have $Y{\leq}T_Y{:}K_S \in (\Gamma_1,\, \Gamma_2[X{\leftarrow}T])$. By part 2 of the induction hypothesis, $\Gamma_1,\, \Gamma_2[X{\leftarrow}T] \vdash \text{ok}$. Applying K-TVar, we get $\Gamma_1,\, \Gamma_2[X{\leftarrow}T] \vdash Y \in K_S$.

$Y{\leq}T_Y{:}K_S \equiv X{\leq}U{:}K_U$ We know that $\Gamma_1 \vdash T \in K_S \equiv K_U$. From the premise of K-TVar and part 2 of the induction hypothesis, we have $\Gamma_1,\, \Gamma_2[X{\leftarrow}T] \vdash \mathrm{ok}$. The result follows by weakening (lemma 2.4.4).

$Y{\leq}T_Y{:}K_S{\in}\Gamma_2$ Then we have $Y{\leq}T_Y[X{\leftarrow}T]{:}K_S \in (\Gamma_1,\, \Gamma_2[X{\leftarrow}T])$. By part 2 of the induction hypothesis, $\Gamma_1,\, \Gamma_2[X{\leftarrow}T] \vdash \mathrm{ok}$, from which the result follows by K-TVar. □

Lemma 2.4.12 (*Subject reduction for kinding judgements*) If $S \twoheadrightarrow_{\beta\wedge} T$ and $\Gamma \vdash S \in K$, then $\Gamma \vdash T \in K$.

Proof: In order to prove this result it is enough to prove the following statements by simultaneous induction on the derivation of $\Gamma \vdash S \in K$. The rest follows by induction on the definition of $\twoheadrightarrow_{\beta\wedge}$.

1. $\Gamma \vdash \mathrm{ok}$ and $\Gamma \rightarrow_{\beta\wedge} \Gamma'$ implies $\Gamma' \vdash \mathrm{ok}$.

2. $\Gamma \vdash S \in K$ and $S \rightarrow_{\beta\wedge} T$ implies $\Gamma \vdash T \in K$.

3. $\Gamma \vdash S \in K$ and $\Gamma \rightarrow_{\beta\wedge} \Gamma'$ implies $\Gamma' \vdash S \in K$. □

In chapter 4 we prove that the subject reduction property also holds for typing judgements.

Theorem 2.4.13 (*Kind invariance under type conversion*) If $\Gamma \vdash S \in K_S$ and $\Gamma \vdash T \in K_T$, with $S =_{\beta\wedge} T$, then $K_S \equiv K_T$.

Proof: By the Church-Rosser theorem 2.3.1.7, there exists U such that $S \twoheadrightarrow_{\beta\wedge} U$ and $T \twoheadrightarrow_{\beta\wedge} U$, and the result follows by subject reduction and unicity of kinds. □

Lemma 2.4.14 Let $\Gamma \vdash S_j \in K$ for each $j{\in}\{1..m\}$. Then if for every $i{\in}\{1..n\}$ there exists $j{\in}\{1..m\}$ such that $\Gamma \vdash S_j \leq T_i$, then $\Gamma \vdash \bigwedge^K[S_1..S_m] \leq \bigwedge^K[T_1..T_n]$.

A particular case of the previous lemma is the following.

Corollary 2.4.15 Let $\Gamma \vdash S_i \in K$ for each $i{\in}\{1..n\}$. Then $\Gamma \vdash S_i \leq T_i$, for every $i{\in}\{1..n\}$, implies $\Gamma \vdash \bigwedge^K[S_1..S_n] \leq \bigwedge^K[T_1..T_n]$.

Lemma 2.4.16 Let $\Gamma \vdash S, T \in K$. Then $\Gamma \vdash S \leq T$ if and only if $\Gamma \vdash S^{nf} \leq T^{nf}$.

Proof: We shall consider only one part the other is similar.

$\Rightarrow$) By subject reduction, we have that $\Gamma \vdash S^{nf}{\in}K$, then, by S-Conv, $\Gamma \vdash S^{nf} \leq S$. By similar reasoning we have $\Gamma \vdash T \leq T^{nf}$. The result follows by applying S-Trans twice. □

Lemma 2.4.17 (*Context modification*) If $\Gamma_1 \vdash U' \in K$ and Σ is either ok or $T \in K'$, then $\Gamma_1,\, X{\leq}U{:}K,\, \Gamma_2 \vdash \Sigma$ implies $\Gamma_1,\, X{\leq}U'{:}K,\, \Gamma_2 \vdash \Sigma$.

Lemma 2.4.18 Let $\Gamma \vdash S_i \in K$ for every $i{\in}\{1..n\}$. If for every j in $\{1..m\}$ there exists i in $\{1..n\}$ such that $\Gamma \vdash S_i \leq T_j$, then $\Gamma \vdash \bigwedge^K[S_1..S_n] \leq \bigwedge^K[T_1..T_m]$.

Proposition 2.4.19 (*Well-kindedness of subtyping*) If $\Gamma \vdash S \leq T$, then $\Gamma \vdash S \in K$ and $\Gamma \vdash T \in K$ for some K.

Proof: By induction on the derivation of $\Gamma \vdash S \leq T$.

S-Conv We are given that $\Gamma \vdash S \in K$ and $\Gamma \vdash T \in K'$ and $S =_\beta T$. By lemma 2.4.13, $K \equiv K'$.

S-Trans By the induction hypothesis and uniqueness of kinds (lemma 2.4.7).

S-TVar We are given that $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \text{ok}$. By K-TVar it follows that $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash X \in K$. Moreover, by lemma 2.4.2, we have $\Gamma_1 \vdash T \in K$, and by lemma 2.4.4, $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash T \in K$.

S-Arrow We are given $\Gamma \vdash T_1 \leq S_1$ and $\Gamma \vdash S_2 \leq T_2$ and $\Gamma \vdash S_1{\to}S_2 \in \star$. By proposition 2.4.6, $\Gamma \vdash S_1, S_2 \in \star$. Further, by the induction hypothesis together with uniqueness of kinds (lemma 2.4.7), we have $\Gamma \vdash T_1, T_2 \in \star$. Finally, the result follows by applying K-Arrow.

S-All We are given that $\Gamma,\ X{\leq}U{:}K_1 \vdash S_2 \leq T_2$ and $\Gamma \vdash \forall(X{\leq}U{:}K_1)S_2 \in \star$. By proposition 2.4.6, $\Gamma,\ X{\leq}U{:}K_1 \vdash S_2 \in \star$. Then, applying the induction hypothesis and lemma 2.4.7, we obtain $\Gamma \vdash T_1 \in K_1$ and $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in \star$, from which the result follows by applying K-All.

S-OAbs By the induction hypothesis and K-OAbs.

S-OApp Similar to S-All.

S-Meet-G Using the induction hypothesis, lemma 2.4.7, and K-Meet.

S-Meet-LB We are given $\Gamma \vdash \bigwedge^K[T_1..T_n] \in K$, which, by proposition 2.4.6, implies $\Gamma \vdash T_i \in K$ for each i. □

Proposition 2.4.20 (*Well-kindedness of typing*) If $\Gamma \vdash e \in T$, then $\Gamma \vdash T \in \star$.

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

T-Var We are given $\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash \text{ok}$. The result follows by lemma 2.4.2 and lemma 2.4.4.

T-Abs We are given $\Gamma,\ x{:}T_1 \vdash e \in T_2$. By the induction hypothesis, $\Gamma,\ x{:}T_1 \vdash T_2 \in \star$. By lemma 2.4.5, it follows that $\Gamma \vdash T_2 \in \star$. Furthermore, by lemmas 2.4.1 and 2.4.2, $\Gamma \vdash T_1 \in \star$. Hence, K-Arrow yields $\Gamma \vdash T_1{\to}T_2 \in \star$.

T-App By the induction hypothesis for $\Gamma \vdash f \in T_1{\to}T$ and proposition 2.4.6.

T-TAbs We are given $\Gamma,\, X{\leq}T_1{:}K_1 \vdash e \in T_2$. By the induction hypothesis, $\Gamma,\, X{\leq}T_1{:}K_1 \vdash T_2 \in \star$. We obtain $\Gamma \vdash \forall(X{\leq}T_1{:}K_1)T_2 \in \star$ by applying K-All .

T-TApp We know that $\Gamma \vdash f \in \forall(X{\leq}T_1{:}K_1)T_2$ and also $\Gamma \vdash S \leq T_1$. By the induction hypothesis, $\Gamma \vdash \forall(X{\leq}T_1{:}K_1)T_2 \in \star$ and, by proposition 2.4.6, $\Gamma,\, X{\leq}T_1{:}K_1 \vdash T_2 \in \star$. By lemmas 2.4.1 and 2.4.2, there exists a derivation of $\Gamma \vdash T_1 \in K_1$. By the well-kindedness of subtyping (proposition 2.4.19) and uniqueness of kinds (lemma 2.4.7), we have $\Gamma \vdash S \in K_1$. Then, by the type substitution lemma (lemma 2.4.11), $\Gamma \vdash T_2[X{\leftarrow}S] \in \star$.

T-For By the induction hypothesis.

T-Meet We are given that $\Gamma \vdash \text{ok}$ and that $\Gamma \vdash e \in T_i$ for each i. We have to consider two cases.

$n = 0$. Applying K-Meet to $\Gamma \vdash \text{ok}$ we obtain $\Gamma \vdash \top^{\star} \in \star$.

$n \neq 0$. By the induction hypothesis, $\Gamma \vdash T_i \in \star$ for every i and, then the result follows by applying K-Meet.

T-Sub By the induction hypothesis, proposition 2.4.19 and lemma 2.4.7. □

2.5 Strong normalization of $\rightarrow_{\beta\wedge}$

We prove that every type that has a kind in $F^{\omega}_{\wedge}$ is strongly normalizing in three steps. We first prove that $\rightarrow_a$ and also $\rightarrow_{\beta\wedge^-}$ are strongly normalizing. Then we prove that both reductions commute, i.e. if $T \rightarrow_a T_1$ and $T_1 \rightarrow_{\beta\wedge^-} T_2$, then there exists S such that $S \rightarrow_a T_2$ and $T \twoheadrightarrow_{\beta\wedge^-}^{>0} S$(in at least one step). Finally, using the previous two steps we prove that $\rightarrow_{\beta\wedge}$ is strongly normalizing.
A type T is called *strongly normalizing* if and only if all reduction sequences starting with T terminate. We write $\mathbb{T}$ for the set of all type expressions and *SN* for the subset of $\mathbb{T}$ of strongly normalizing type expressions. If A and B are subsets of $\mathbb{T}$, then $A \rightarrow B$ denotes the following subset of $\mathbb{T}$

$$A \rightarrow B = \{F \subseteq \mathbb{T} \,|\, \text{for all } a{\in}A \;\; Fa{\in}B\}.$$

Lemma 2.5.1 $\rightarrow_a$ is strongly normalizing.

Proof: By induction on the number of intersection symbols of the type expression being reduced. □

To prove strong normalization of $\rightarrow_{\beta\wedge^-}$ we use a model-theoretic argument interpreting kinds as sets of normalizing terms, and the soundness of the model gives, as a corollary, the strong normalization property. The interpretation of a kind K, notation $[\![K]\!]$, is defined as follows.

$$\begin{array}{lcl} [\![\star]\!] & = & \mathit{SN} \\ [\![K_1{\rightarrow}K_2]\!] & = & [\![K_1]\!] \rightarrow [\![K_2]\!]. \end{array}$$

Definition 2.5.2 (*Saturated set*) $\boldsymbol{S} \subseteq \mathit{SN}$ is *saturated* if is satisfies the following conditions:

1. If $R_1..R_n \in \mathit{SN}$, then $XR_1..R_n \in \boldsymbol{S}$.
2. If $R_1..R_n, Q \in \mathit{SN}$, then
   (a) if $P[X{\leftarrow}Q]R_1..R_n \in \boldsymbol{S}$, then $(\Lambda X{:}K.P)QR_1..R_n \in \boldsymbol{S}$, for every K and
   (b) if $(\bigwedge^{K_2}[T_1Q, .., T_mQ])R_1, .., R_n \in \boldsymbol{S}$,
   then $(\bigwedge^{K_1 \to K_2}[T_1, .., T_m])QR_1, .., R_n \in \boldsymbol{S}$, for every K_1.

Intuitively, a set of strongly normalizing type expressions is saturated if it contains all type variables and is closed under expansion of expressions which may have a kind of the form $K_1{\to}K_2$.

Lemma 2.5.3

1. *SN* is saturated.
2. If A, B are saturated, then $A \to B$ is saturated.
3. For any kind K, $[\![K]\!]$ is saturated.

Definition 2.5.4

1. A valuation ρ in $\mathbb{T}$ is a mapping from type variables to types.
2. The interpretation of a type with respect to ρ is

$$[\![T]\!]_\rho = T[X_1 \leftarrow \rho(X_1)..X_n \leftarrow \rho(X_n),$$

   where $\mathrm{FV}(T) = \{X_1..X_n\}$.
3. Let ρ be a valuation in $\mathbb{T}$. Then ρ *satisfies* $T \in K$, written $\rho \models T \in K$, if $[\![T]\!]_\rho \in [\![K]\!]$ and ρ *satisfies* $X{\leq}T{:}K$, written $\rho \models X{\leq}T{:}K$, if $\rho(X) \in [\![K]\!]$. We say that ρ *satisfies* a context Γ, $\rho \models \Gamma$, if $\rho \models X \leq S{:}K$ for all $X \leq S{:}K \in \Gamma$.
4. A context Γ *satisfies* $T \in K$, written $\Gamma \models T \in K$, if for every ρ such that $\rho \models \Gamma$, it follows that $\rho \models T \in K$.

Lemma 2.5.5

1. $\top^K {\in} [\![K]\!]$.
2. If $A_i {\in} [\![K]\!]$ for each $i {\in} \{1..n\}$, then $\bigwedge^K[A_1..A_n] {\in} [\![K]\!]$.

Proof:

1. By induction on the structure of K.

$K \equiv \star$ $\top^{\star}$ is in normal form. Hence, $\top^{\star} \in SN \equiv \llbracket K \rrbracket$.

$K \equiv K_1 \rightarrow K_2$ By the induction hypothesis, $\top^{K_2} \in \llbracket K_2 \rrbracket$. Moreover, if $B \in \llbracket K_1 \rrbracket$ then $\top^{K_1 \rightarrow K_2} B \in \llbracket K_2 \rrbracket$, by the saturation of $\llbracket K_2 \rrbracket$, which means that $\top^{K_1 \rightarrow K_2} \in \llbracket K_1 \rightarrow K_2 \rrbracket$.

2. By induction on the structure of K.

$K \equiv \star$ Then, by definition of $\llbracket K \rrbracket$, $A_i \in SN$ for each $i \in \{1..n\}$. Since every reduction starting from $\bigwedge^K [A_1..A_n]$ is a reduction consisting only of steps inside the $A_i{}'s$, one has $\bigwedge^K [A_1..A_n] \in SN \equiv \llbracket K \rrbracket$.

$K \equiv K_1 \rightarrow K_2$ Let $B \in \llbracket K_1 \rrbracket$. By the definition of $\rightarrow$, $A_i B \in \llbracket K_2 \rrbracket$, for each $i \in \{1..n\}$. By the induction hypothesis, $\bigwedge^{K_2} [A_1 B..A_n B] \in \llbracket K_2 \rrbracket$. Moreover, $\bigwedge^{K_1 \rightarrow K_2} [A_1..A_n] B \in \llbracket K_2 \rrbracket$ by the saturation of $\llbracket K_2 \rrbracket$, which means that $\bigwedge^{K_1 \rightarrow K_2} [A_1..A_n] \in \llbracket K_1 \rightarrow K_2 \rrbracket$. □

Proposition 2.5.6 (*Soundness*) If $\Gamma \vdash T \in K$, then $\Gamma \models T \in K$.

Proof: By induction on the derivation of $\Gamma \vdash T \in K$.

We consider the case for K-Meet. The other cases follow by similar reasoning. Let $T \equiv \bigwedge^K [T_1..T_n]$. We have to consider two cases.

$n \not\equiv 0$ We are given $\Gamma \vdash T_i \in K$ for each $i \in \{1..n\}$, and, by the induction hypothesis, $\Gamma \models T_i \in K$. Let ρ be a valuation such that $\rho \models \Gamma$. Then $\llbracket T_i \rrbracket_\rho \in \llbracket K \rrbracket$, for each $i \in \{1..n\}$. By lemma 2.5.5(2), $\bigwedge^K [\llbracket T_1 \rrbracket_\rho .. \llbracket T_n \rrbracket_\rho] \in \llbracket K \rrbracket$.

$n \equiv 0$ $T \equiv \top^K$. Since $\llbracket \top^K \rrbracket_\rho \equiv \top^K$, the result follows by 2.5.5(1). □

Theorem 2.5.7 (*Strong normalization for* $\rightarrow_{\beta\wedge^-}$) $\Gamma \vdash T \in K$ implies that every $(\beta\wedge^-)$-reduction sequence starting from T is finite.

Proof: By soundness, $\Gamma \models T \in K$. Choose ρ_0 such that $\rho_0(X) = X$. Observe that $\rho_0 \models \Gamma$ trivially. Hence $T \equiv \llbracket T \rrbracket_{\rho_0} \in \llbracket K \rrbracket \subseteq SN$. □

Lemma 2.5.8 If $T \rightarrow_a T_1$ and $T_1 \rightarrow_{\beta\wedge^-} T_2$, then there exists S such that $T \twoheadrightarrow_{\beta\wedge^-}{}^{>0} S$ and $S \rightarrow_a T_2$.

Proof: By induction on the structure of T. □

Corollary 2.5.9 If $T \twoheadrightarrow_a T_1$ and $T_1 \twoheadrightarrow_{\beta\wedge^-} T_2$, then there exists S such that $T \twoheadrightarrow_{\beta\wedge^-}{}^{>0} S$ and $S \twoheadrightarrow_a T_2$.

Proof: By induction on the generation of $T \twoheadrightarrow_a T_1$. □

Finally, we can prove strong normalization for $\rightarrow_{\beta\wedge}$.

Theorem 2.5.10 (*Strong normalization for* $\rightarrow_{\beta\wedge}$) $\Gamma \vdash T \in K$ implies that every $(\beta\wedge)$-reduction sequence starting from T is finite.

Proof: Let $\Gamma \vdash T \in K$. We reason by contradiction. Assume that there is an infinite $\beta\wedge$-reduction sequence starting from T. Then lemma 2.5.1 and theorem 2.5.7 imply that there are infinitely many alternations of $\rightarrow_a$ and $\rightarrow_{\beta\wedge^-}$ reduction sequences. Graphically:

$$\begin{tikzcd}
T \arrow[r, two heads, "a"'] & T_1 \arrow[d, two heads, "\beta\wedge^-"] & \\
& T_2 \arrow[r, two heads, "a"'] & T_3 \arrow[d, two heads, "\beta\wedge^-"] \\
& & T_4 \cdots\cdots
\end{tikzcd}$$

By corollary 2.5.9, we can construct an infinite $(\beta\wedge^-)$-reduction which contradicts theorem 2.5.7. Graphically:

$$\begin{tikzcd}
T \arrow[r, two heads, "a"'] \arrow[d, two heads, "\beta\wedge^-"] & T_1 \arrow[d, two heads, "\beta\wedge^-"] & \\
S_1 \arrow[r, two heads, "a"'] \arrow[d, two heads, "\beta\wedge^-"] & T_2 \arrow[r, two heads, "a"'] \arrow[d, two heads, "\beta\wedge^-"] & T_3 \arrow[d, two heads, "\beta\wedge^-"] \\
S_3 \arrow[r, two heads, "a"'] & S_2 \arrow[r, two heads, "a"'] & T_4 \cdots\cdots \\
\vdots & &
\end{tikzcd}$$

□

Chapter 3

Decidability of Subtyping in $F^{\omega}_{\wedge}$

In this chapter we show that the subtyping relation of $F^{\omega}_{\wedge}$ is decidable. The solution is divided into two main parts. First, we develop a *normal subtyping system*, $\mathit{NF}^{\omega}_{\wedge}$, in which only types in normal form are considered. We prove that proofs in $\mathit{NF}^{\omega}_{\wedge}$ can be normalized by eliminating transitivity and simplifying reflexivity. This simplification yields an algorithmic presentation, $\mathit{AlgF}^{\omega}_{\wedge}$, whose rules are syntax directed. Moreover, we prove that $\mathit{AlgF}^{\omega}_{\wedge}$ is indeed an alternative presentation of the $F^{\omega}_{\wedge}$ subtyping relation. Formally, $\Gamma \vdash S \leq T$ if and only if $\Gamma^{\mathit{nf}} \vdash_{\mathit{Alg}} S^{\mathit{nf}} \leq T^{\mathit{nf}}$ (proposition 3.4.3).

In the solution for the second order lambda calculus presented in [Pie91], the distributivity rules for intersection types are not considered as rewrite rules. For that reason, new syntactic categories have to be defined (composite and individual canonical types) and an auxiliary mapping (flattening) transforms a type into a canonical type. Our solution does not need either new syntactic categories or elaborate auxiliary mappings, since the role played there by canonical types is performed here by types in normal form.

Independently Steffen and Pierce proved a similar result for $F^{\omega}_{\leq}$ [SP94]. There are several differences between our work and the proof of decidability of subtyping in [SP94]. First, our result is for a stronger system which also includes intersection types. Our proof of termination has the novel idea of using a choice operator to model the behavior of type variables during subtype checking. A second major difference is the choice of the intermediate subtyping system. We define the normal system $\mathit{NF}^{\omega}_{\wedge}$ which is not only the key to proving decidability of subtyping but helped understand the fine structure of subtyping, yielding the algorithm $\mathit{AlgF}^{\omega}_{\wedge}$. In [SP94] the intermediate system, called a reducing system, leads to a much more complicated proof which involves dealing with several notions of reduction and further reformulation of the intermediate system.

3.1 Normal Subtyping

An important property of derivation systems is the information that a derivable judgement contains about its proofs. This information is essential to produce results which not only state properties about the subproofs, but also help identify ill formed judgements.

In $F^\omega_\wedge$ we can prove

$$W{:}K, X \leq \Lambda Y{:}K.Y{:}K{\to}K,\; Z \leq X{:}K{\to}K \vdash X(Z\,W) \leq W \tag{3.1}$$

This simple example already shows that S-Trans erases information obtained by S-Conv that is not present in the conclusion any longer (see 3.3.2 for a derivation). A first step towards an algorithm to check the subtyping relation is to design a set of rules in which the derivable judgements contain all the information about their derivations. To this end we define a set of rules, $NF^\omega_\wedge$, in which conversion is reduced to a minimum and, as we show in lemma 3.2.6, transitivity can be eliminated. Both results are proved with a standard cut-elimination argument. This yields a syntax directed subtyping relation, $AlgF^\omega_\wedge$, which constitutes a decision procedure for the original system.

In section 3.1, we present the subtyping system $NF^\omega_\wedge$, which uses the context and type formation rules of $F^\omega_\wedge$. We define rewriting rules for derivations in $NF^\omega_\wedge$ (definitions 3.2.3 and 3.2.4), and describe a terminating procedure to normalize proofs, which gives, as a consequence, the generation for subtyping (proposition 3.2.10) and an algorithmic presentation, $AlgF^\omega_\wedge$ (see definition 3.4.1).

Finally, in section 3.4, we show that there is an equivalence between subtyping in $F^\omega_\wedge$ and subtyping in $AlgF^\omega_\wedge$, which is essential to prove the decidability of subtyping in $F^\omega_\wedge$.

We now define the *normal subtyping system*, $NF^\omega_\wedge$. Subtyping statements in $NF^\omega_\wedge$ are written $\Gamma \vdash_n S \leq T$, and S, T, and all types appearing in Γ are in $\beta\wedge$-normal form.

Notation 3.1.1 A, B, and C range over types whose outermost constructor is not an intersection.

Remark 3.1.2 It is an immediate consequence of the $\beta\wedge$ reduction rules that, if T is in $\beta\wedge$ normal form, then T is either a variable X, $S{\to}A$, $\forall X{\leq}S{:}K.A$, $\Lambda X{:}K.A$, $A\,S$ where A is not an abstraction, or $\bigwedge^K[A_1..A_n]$. We frequently use this notation as a reminder of the shape of types in normal form. Note that we do not fully use this convention in definition 3.1.4 in order to highlight the fact that NS-Arrow, NS-All, and NS-OAbs have the same form as S-Arrow, S-All, and S-OAbs respectively.

We now define $lub_\Gamma(S)$. We prove in lemma 3.3.1 and corollary 3.3.1.2, that, when defined, it is the smallest type beyond S with respect to Γ.

Definition 3.1.3 (*Least strict Upper Bound*)

$$\begin{aligned} lub_\Gamma(X) &= \Gamma(X),\\ lub_\Gamma(T\,S) &= lub_\Gamma(T)\,S. \end{aligned}$$

Definition 3.1.4 (NF$^\omega_\wedge$ *subtyping rules*)

$$\frac{\Gamma \vdash S \in K}{\Gamma \vdash_n S \leq S} \qquad \text{(NS-Refl)}$$

$$\frac{\Gamma \vdash_n S \leq T \qquad \Gamma \vdash_n T \leq U}{\Gamma \vdash_n S \leq U} \qquad \text{(NS-Trans)}$$

$$\frac{\Gamma \vdash_n \Gamma(X) \leq A \qquad X \not\equiv A}{\Gamma \vdash_n X \leq A} \qquad \text{(NS-TVar)}$$

$$\frac{\Gamma \vdash_n T_1 \leq S_1 \qquad \Gamma \vdash_n S_2 \leq T_2 \qquad \Gamma \vdash S_1 \to S_2 \in \star}{\Gamma \vdash_n S_1 \to S_2 \leq T_1 \to T_2} \qquad \text{(NS-Arrow)}$$

$$\frac{\Gamma,\, X{\leq}U{:}K \vdash_n S \leq T \qquad \Gamma \vdash \forall X{\leq}U{:}K.S \in \star}{\Gamma \vdash_n \forall X{\leq}U{:}K.S \leq \forall X{\leq}U{:}K.T} \qquad \text{(NS-All)}$$

$$\frac{\Gamma,\, X{\leq}\top^K{:}K \vdash_n S \leq T}{\Gamma \vdash_n \Lambda X{:}K.S \leq \Lambda X{:}K.T} \qquad \text{(NS-OAbs)}$$

$$\frac{\Gamma \vdash_n (lub_\Gamma(T\,S))^{nf} \leq A \qquad \Gamma \vdash T\,S \in K \qquad T\,S \not\equiv A}{\Gamma \vdash_n T\,S \leq A} \qquad \text{(NS-OApp)}$$

$$\frac{\forall i{\in}\{1..m\}\, \Gamma \vdash_n A \leq T_i \qquad \Gamma \vdash A \in K}{\Gamma \vdash_n A \leq \bigwedge^K[T_1..T_m]} \qquad \text{(NS-}\forall\text{)}$$

$$\frac{\exists j{\in}\{1..n\}\, \Gamma \vdash_n S_j \leq A \qquad \forall k{\in}\{1..n\}\, \Gamma \vdash S_k \in K}{\Gamma \vdash_n \bigwedge^K[S_1..S_n] \leq A} \qquad \text{(NS-}\exists\text{)}$$

$$\frac{\forall i{\in}\{1..m\}\, \exists j{\in}\{1..n\}\, \Gamma \vdash_n S_j \leq T_i \qquad \forall k{\in}\{1..n\}\, \Gamma \vdash S_k \in K}{\Gamma \vdash_n \bigwedge^K[S_1..S_n] \leq \bigwedge^K[T_1..T_m]} \qquad \text{(NS-}\forall\exists\text{)}$$

As we mentioned in the introduction, an important factor to develop this system was to consider the distributivity rules of the presentation of $F^\omega_\wedge$ in [CP93] as reduction rules instead of subtyping rules. This new point of view suggested that an algorithmic system should, to a certain extent, concentrate on normal forms replacing the conversion rule by reflexivity. Consequently, a derivation of a subtyping statement should involve only types in normal form. But enlightened by the simple (counter)example (3.1) it is not possible to perform all reductions at once. In other words, the system does not satisfy an S-Conv postponement property. Without using S-Conv it is not possible to derive (3.1). Hence, the solution is not as simple as replacing S-Conv by NS-Refl.

In general, the interaction between S-Trans and S-Conv can be analyzed as follows. In S-Trans the metavariable T of the hypothesis is not present in the conclusion, but this is not a problem by itself (a similar situation appears in the simply typed lambda calculus in its application rule and the system is deterministic). The problem is that in the presence of S-Conv the *vanishing* T can be $\beta\wedge$-convertible to either S or U or to both S and U. What example (3.1) shows is that S and U may be different normal forms, which means that searching for T is inherently nondeterministic.

We cannot eliminate transitivity completely, we still need it on type variables and on type applications. In $F_{\leq}$ [Ghe90] transitivity is eliminated and hidden in a richer variable rule in which deciding whether $\Gamma \vdash X \leq T$ when $T \not\equiv X$ is reduced to deciding whether the bound of X is smaller than or equal to T. The bound of X has the particular property of being the least strict upper bound of X. This observation motivated the definition of our NS-OApp rule, in which we reduce the decision of whether $\Gamma \vdash T\,S \leq A$ when $A \not\equiv T\,S$, to check if the least strict upper bound of $T\,S$ is smaller than or equal to A (See lemma 3.3.1 and corollary 3.3.1.2). $lub_{\Gamma}(T\,S)$ is obtained from $T\,S$ by replacing its leftmost innermost variable by the corresponding bound in Γ. Consequently, $lub_{\Gamma}(T\,S)$ may be other than a normal form. That is the reason we normalize it. The strength of the conversion rule that is not captured by reflexivity is hidden in this normalization step. Since $T\,S$ is a well kinded type, by the free variables lemma (lemma 2.4.3), $\mathrm{FTV}(T\,S) \subseteq \mathrm{dom}(\Gamma)$. Therefore, $lub_{\Gamma}(T\,S)$ is defined. By lemma 3.3.1(1), $lub_{\Gamma}(T\,S)$ is well-kinded, and since well-kinded types are strongly normalizing, its normal form exists. The rules S-Meet-LB and S-Meet-G are replaced by NS-∃, NS-∀, and NS-∀∃.

3.2 Structural properties of $NF^{\omega}_{\wedge}$

This section establishes a number of structural properties of $NF^{\omega}_{\wedge}$. The proofs of lemmas 3.2.1 and 3.2.2 are similar to those of the corresponding properties for $F^{\omega}_{\wedge}$.

Lemma 3.2.1 If $\Gamma \vdash_n S \leq T$ and Γ_1 is a prefix of Γ, then $\Gamma_1 \vdash \mathrm{ok}$ as a subderivation. Moreover, the subderivation is strictly shorter.

Lemma 3.2.2 (*Weakening/Permutation*) Let Γ and Γ' be contexts such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \mathrm{ok}$. Then $\Gamma \vdash_n S \leq T$ implies $\Gamma' \vdash_n S \leq T$.

We present rewriting rules on derivations to simplify instances of NS-Refl and NS-Trans. We give a terminating strategy to transform a given derivation into a derivation with occurrences of NS-Refl only applied to type variables or type applications and without occurrences of NS-Trans. To improve readability we omit kinding judgements in the transitivity elimination rules which appear as hypothesis in the redex or in a proper subderivation of the missing ones, as we proved in generation for kinding (proposition 2.4.6). The derivations of the kinding judgements of each reduct of the reflexivity rules are proper subderivations of the kinding judgements in its redex.

Definition 3.2.3 (*Reflexivity simplification rules*)

$$1.\quad \boxed{\dfrac{\Gamma \vdash S{\to}A \in \star}{\Gamma \vdash_n S{\to}A \leq S{\to}A}\ \textsc{NS-Refl}}$$

$$\Rightarrow_R \boxed{\dfrac{\dfrac{\Gamma \vdash S \in \star}{\Gamma \vdash_n S \leq S}\ \textsc{NS-Refl} \qquad \dfrac{\Gamma \vdash A \in \star}{\Gamma \vdash_n A \leq A}\ \textsc{NS-Refl}}{\Gamma \vdash_n S{\to}A \leq S{\to}A}\ \textsc{NS-Arrow}}$$

2. $$\boxed{\dfrac{\Gamma \vdash \forall X{\leq}S{:}K.A \in \star}{\Gamma \vdash_n \forall X{\leq}S{:}K.A \leq \forall X{\leq}S{:}K.A}\,\textsc{NS-Refl}}$$

$$\Rightarrow_R \boxed{\dfrac{\dfrac{\Gamma,\, X{\leq}S{:}K \vdash A \in \star}{\Gamma,\, X{\leq}S{:}K \vdash_n A \leq A}\,\textsc{NS-Refl}}{\Gamma \vdash_n \forall X{\leq}S{:}K.A \leq \forall X{\leq}S{:}K.A}\,\textsc{NS-All}}$$

3. $$\boxed{\dfrac{\Gamma \vdash \Lambda X{:}K.A \in K{\to}K'}{\Gamma \vdash_n \Lambda X{:}K.A \leq \Lambda X{:}K.A}\,\textsc{NS-Refl}}$$

$$\Rightarrow_R \boxed{\dfrac{\dfrac{\Gamma,\, X{:}K \vdash A \in K'}{\Gamma,\, X{:}K \vdash_n A \leq A}\,\textsc{NS-Refl}}{\Gamma \vdash_n \Lambda X{:}K.A \leq \Lambda X{:}K.A}\,\textsc{NS-OAbs}}$$

4. $$\boxed{\dfrac{\Gamma \vdash \bigwedge^K[A_1..A_n] \in K}{\Gamma \vdash_n \bigwedge^K[A_1..A_n] \leq \bigwedge^K[A_1..A_n]}\,\textsc{NS-Refl}}$$

$$\Rightarrow_R \boxed{\dfrac{\dfrac{\Gamma \vdash A_i \in K}{\Gamma \vdash_n A_i \leq A_i \; \forall i{\in}\{1..n\}}\,\textsc{NS-Refl}}{\Gamma \vdash_n \bigwedge^K[A_1..A_n] \leq \bigwedge^K[A_1..A_n]}\,\textsc{NS-}\forall\exists}$$

Definition 3.2.4 (*Transitivity elimination rules*)

1. $$\boxed{\dfrac{\dfrac{\Gamma \vdash S \in K}{\Gamma \vdash_n S \leq S}\,\textsc{NS-Refl} \qquad \Gamma \vdash_n S \leq T}{\Gamma \vdash_n S \leq T}\,\textsc{NS-Trans}} \Rightarrow_T \boxed{\Gamma \vdash_n S \leq T}$$

2. $$\boxed{\dfrac{\Gamma \vdash_n S \leq T \quad \dfrac{\Gamma \vdash T \in K}{\Gamma \vdash_n T \leq T}\,\textsc{NS-Refl}}{\Gamma \vdash_n S \leq T}\,\textsc{NS-Trans}} \Rightarrow_T \boxed{\Gamma \vdash_n S \leq T}$$

3. $$\boxed{\dfrac{\dfrac{\Gamma \vdash_n \Gamma(X) \leq A}{\Gamma \vdash_n X \leq A}\,\textsc{NS-TVar} \qquad \Gamma \vdash_n A \leq B}{\Gamma \vdash_n X \leq B}\,\textsc{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\Gamma \vdash_n \Gamma(X) \leq A \quad \Gamma \vdash_n A \leq B}{\Gamma \vdash_n \Gamma(X) \leq B}\,\textsc{NS-Trans}}{\Gamma \vdash_n X \leq B}\,\textsc{NS-TVar}}$$

$$4.\quad \boxed{\dfrac{\dfrac{\Gamma \vdash_n T \leq S \quad \Gamma \vdash_n A \leq B}{\Gamma \vdash_n S{\to}A \leq T{\to}B}\,\text{NS-Arrow} \qquad \dfrac{\Gamma \vdash_n U \leq T \quad \Gamma \vdash_n B \leq C}{\Gamma \vdash_n T{\to}B \leq U{\to}C}\,\text{NS-Arrow}}{\Gamma \vdash_n S{\to}A \leq U{\to}C}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\Gamma \vdash_n U \leq T \quad \Gamma \vdash_n T \leq S}{\Gamma \vdash_n U \leq S}\,\text{NS-Trans} \qquad \dfrac{\Gamma \vdash_n A \leq B \quad \Gamma \vdash_n B \leq C}{\Gamma \vdash_n A \leq C}\,\text{NS-Trans}}{\Gamma \vdash_n S{\to}A \leq U{\to}C}\,\text{NS-Arrow}}$$

$$5.\quad \boxed{\dfrac{\dfrac{\Gamma,\, X{\leq}S{:}K \vdash_n A \leq B}{\Gamma \vdash_n \forall X{\leq}S{:}K.A \leq \forall X{\leq}S{:}K.B}\,\text{NS-All} \qquad \dfrac{\Gamma,\, X{\leq}S{:}K \vdash_n B \leq C}{\Gamma \vdash_n \forall X{\leq}S{:}K.B \leq \forall X{\leq}S{:}K.C}\,\text{NS-All}}{\Gamma \vdash_n \forall X{\leq}S{:}K.A \leq \forall X{\leq}S{:}K.C}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\Gamma,\, X{\leq}S{:}K \vdash_n A \leq B \quad \Gamma,\, X{\leq}S{:}K \vdash_n B \leq C}{\Gamma,\, X{\leq}S{:}K \vdash_n A \leq C}\,\text{NS-Trans}}{\Gamma \vdash_n \forall X{\leq}S{:}K.A \leq \forall X{\leq}U{:}K.C}\,\text{NS-All}}$$

$$6.\quad \boxed{\dfrac{\dfrac{\Gamma,\, X{:}K \vdash_n A \leq B}{\Gamma \vdash_n \Lambda X{:}K.A \leq \Lambda X{:}K.B}\,\text{NS-OAbs} \qquad \dfrac{\Gamma,\, X{:}K \vdash_n B \leq C}{\Gamma \vdash_n \Lambda X{:}K.B \leq \Lambda X{:}K.C}\,\text{NS-OAbs}}{\Gamma \vdash_n \Lambda X{:}K.A \leq \Lambda X{:}K.C}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\Gamma,\, X{:}K \vdash_n A \leq B \quad \Gamma,\, X{:}K \vdash_n B \leq C}{\Gamma,\, X{:}K \vdash_n A \leq C}\,\text{NS-Trans}}{\Gamma \vdash_n \Lambda X{:}K.A \leq \Lambda X{:}K.C}\,\text{NS-OAbs}}$$

$$7.\quad \boxed{\dfrac{\dfrac{\Gamma \vdash_n \mathit{lub}_{\Gamma}(A\,S)^{\mathit{nf}} \leq B}{\Gamma \vdash_n A\,S \leq B}\,\text{NS-OApp} \qquad \Gamma \vdash_n B \leq C}{\Gamma \vdash_n A\,S \leq C}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\Gamma \vdash_n (\mathit{lub}_{\Gamma}(A\,S))^{\mathit{nf}} \leq B \quad \Gamma \vdash_n B \leq C}{\Gamma \vdash_n \mathit{lub}_{\Gamma}(A\,S))^{\mathit{nf}} \leq C}\,\text{NS-Trans}}{\Gamma \vdash_n A\,S \leq C}\,\text{NS-OApp}}$$

$$8.\quad \boxed{\dfrac{\dfrac{\forall i{\in}\{1..n\}\ \Gamma \vdash_n A \leq A_i}{\Gamma \vdash_n A \leq \bigwedge^{K}[A_1..A_n]}\,\text{NS-}\forall \qquad \dfrac{\exists j{\in}\{1..n\}\ \Gamma \vdash_n A_j \leq B}{\Gamma \vdash_n \bigwedge^{K}[A_1..A_n] \leq B}\,\text{NS-}\exists}{\Gamma \vdash_n A \leq B}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\exists j{\in}\{1..n\}\ \Gamma \vdash_n A \leq A_j \quad \Gamma \vdash_n A_j \leq B}{\Gamma \vdash_n A \leq B}\,\text{NS-Trans}}$$

9. $$\boxed{\dfrac{\Gamma \vdash_n A \leq B \quad \dfrac{\forall i{\in}\{1..n\}\ \Gamma \vdash_n B \leq A_i}{\Gamma \vdash_n B \leq \bigwedge^K[A_1..A_n]}\,\text{NS-}\forall}{\Gamma \vdash_n A \leq \bigwedge^K[A_1..A_n]}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\forall i{\in}\{1..n\}\ \Gamma \vdash_n A \leq B \quad \Gamma \vdash_n B \leq A_i}{\forall i{\in}\{1..n\}\ \Gamma \vdash_n A \leq A_i}\,\text{NS-Trans}}{\Gamma \vdash_n A \leq \bigwedge^K[A_1..A_n]}\,\text{NS-}\forall}$$

10. $$\boxed{\dfrac{\dfrac{\exists j{\in}\{1..n\}\ \Gamma \vdash_n A_j \leq B}{\Gamma \vdash_n \bigwedge^K[A_1..A_n] \leq B}\,\text{NS-}\exists \qquad \Gamma \vdash_n B \leq A}{\Gamma \vdash_n \bigwedge^K[A_1..A_n] \leq A}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\exists j{\in}\{1..n\}\ \Gamma \vdash_n A_j \leq B \quad \Gamma \vdash_n B \leq A}{\exists j{\in}\{1..n\}\ \Gamma \vdash_n A_j \leq A}\,\text{NS-Trans}}{\Gamma \vdash_n \bigwedge^K[A_1..A_n] \leq A}\,\text{NS-}\exists}$$

11. $$\boxed{\dfrac{\dfrac{\exists j{\in}\{1..m\}\ \Gamma \vdash_n A_j \leq A}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq A}\,\text{NS-}\exists \qquad \dfrac{\forall i{\in}\{1..n\}\ \Gamma \vdash_n A \leq B_i}{\Gamma \vdash_n A \leq \bigwedge^K[B_1..B_n]}\,\text{NS-}\forall}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq \bigwedge^K[B_1..B_n]}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\exists j{\in}\{1..m\}\ \Gamma \vdash_n A_j \leq A \quad \forall i{\in}\{1..n\}\ \Gamma \vdash_n A \leq B_i}{\forall i{\in}\{1..n\}\,\exists j{\in}\{1..m\}\ \Gamma \vdash_n A_j \leq B_i}\,\text{NS-Trans}}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq \bigwedge^K[B_1..B_n]}\,\text{NS-}\forall\exists}$$

12. $$\boxed{\dfrac{\dfrac{\forall i{\in}\{1..n\}\,\exists j{\in}\{1..m\}\ \Gamma \vdash_n A_j \leq B_i}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq \bigwedge^K[B_1..B_n]} \quad \dfrac{\forall k{\in}\{1..r\}\,\exists i{\in}\{1..n\}\ \Gamma \vdash_n B_i \leq C_k}{\Gamma \vdash_n \bigwedge^K[B_1..B_n] \leq \bigwedge^K[C_1..C_r]}\,\text{NS-}\forall\exists}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq \bigwedge^K[C_1..C_r]}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\forall k{\in}\{1..r\}\,\exists i{\in}\{1..n\}\,\exists j{\in}\{1..m\} \quad \Gamma \vdash_n A_j \leq B_i \qquad \Gamma \vdash_n B_i \leq C_k}{\forall k{\in}\{1..r\}\,\exists j{\in}\{1..m\}\,\Gamma \vdash_n A_j \leq C_k}\,\text{NS-Trans}}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq \bigwedge^K[C_1..C_r]}\,\text{NS-}\forall\exists}$$

13. $$\boxed{\dfrac{\dfrac{\forall i{\in}\{1..n\}\,\exists j{\in}\{1..m\}\ \Gamma \vdash_n A_j \leq B_i}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq \bigwedge^K[B_1..B_n]}\,\text{NS-}\forall\exists \quad \dfrac{\exists i{\in}\{1..n\}\ \Gamma \vdash_n B_i \leq C}{\Gamma \vdash_n \bigwedge^K[B_1..B_n] \leq C}\,\text{NS-}\exists}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq C}\,\text{NS-Trans}}$$

$$\Rightarrow_T \boxed{\dfrac{\dfrac{\exists j{\in}\{1..m\}\ \exists i{\in}\{1..n\}\ \Gamma \vdash_n A_j \leq B_i \quad \Gamma \vdash_n B_i \leq C}{\exists j{\in}\{1..m\}\,\Gamma \vdash_n A_j \leq C}\,\text{NS-Trans}}{\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq C}\,\text{NS-}\exists}$$

$$14.\quad \boxed{\dfrac{\dfrac{\forall i{\in}\{1..n\}\ \Gamma\vdash_n A\le B_i}{\Gamma\vdash_n A\le \bigwedge^K[B_1..B_n]}\,\text{NS-}\forall \qquad \dfrac{\forall k{\in}\{1..r\}\ \exists i{\in}\{1..n\}\ \Gamma\vdash_n B_i\le C_k}{\Gamma\vdash_n \bigwedge^K[B_1..B_n]\le\bigwedge^K[C_1..C_r]}\,\text{NS-}\forall\exists}{\Gamma\vdash_n A\le\bigwedge^K[C_1..C_r]}\,\text{NS-}\textsc{Trans}}$$

$$\Rightarrow_T\quad \boxed{\dfrac{\dfrac{\forall k{\in}\{1..r\}\ \exists i{\in}\{1..n\}\quad \Gamma\vdash_n A\le B_i\qquad \Gamma\vdash_n B_i\le C_k}{\forall k{\in}\{1..r\}\ \Gamma\vdash_n A\le C_k}\,\text{NS-}\textsc{Trans}}{\Gamma\vdash_n A\le\bigwedge^K[C_1..C_r]}\,\text{NS-}\forall}$$

A derivation of a subtyping statement is in *refl-normal form* if it has no reflexivity redexes and it is in *trans-normal form* if it has no transitivity redexes, and it is in *normal form* if it has neither reflexivity nor transitivity redexes. The elimination of NS-Trans, and the simplification of NS-Refl follow a standard cut-elimination argument.

Lemma 3.2.5 (*Reflexivity simplification*) Let D be a derivation of a subtyping statement with only one application of NS-Refl. Then D has a refl-normal form.

Proof: Same argument as in lemma 3.2.6. □

Lemma 3.2.6 (*Transitivity elimination*) Let D be a derivation of a subtyping statement with only one application of NS-Trans. Then D has a trans-normal form.

Proof: By induction on the size of D following a case analysis of the last rule of D. If the last rule is not NS-Trans, then the result follows by the induction hypothesis. Otherwise we consider all possible last rules of the derivations of the premises and note that each possible configuration determines a trans-redex. Finally, observe that each reduction yields either a derivation in normal form or shorter derivations with only one occurrence of NS-Trans in which case the result follows by the induction hypothesis. □

An immediate corollary of this last result is that transitivity elimination terminates. Given a derivation D of $\Gamma\vdash_n S\le T$, iterate the previous lemma on all subderivations of D that have only one NS-Trans application. The number of times the lemma is applied is equal to the number of occurrences of NS-Trans in D. Furthermore, lemma 3.2.5 implies that reflexivity simplification terminates. The simplification rules are such that transitivity simplification rules do not create new reflexivity redexes. Therefore, we can reduce all instances of NS-Refl first and then all instances of NS-Trans, which is a terminating procedure to normalize a derivation. Consequently, we have proved the following corollary.

Corollary 3.2.7 (*Existence of normal derivations*) Given a derivation of $\Gamma\vdash_n S\le T$. Then there exists a derivation in normal form of $\Gamma\vdash_n S\le T$.

Lemma 3.2.8

1. A derivation in normal form whose last rule is NS-Refl is either a proof of $\Gamma \vdash_n X \leq X$ or $\Gamma \vdash_n A\,T \leq A\,T$.
2. If the last rule of a subtyping derivation D is NS-Trans, then D is not in normal form.

Proof:

1. According to the reflexivity elimination rules, any other possible NS-Refl application is a redex.
2. By case analysis of the last rules of the premises of the last rule of D. In each case the result follows either by the induction hypothesis or because the last rule of at least one of the derivations of the premises of D constitutes a redex. □

We can summarise the previous results as follows.

Corollary 3.2.9 If $\Gamma \vdash_n S \leq T$, then there exists a proof of the same judgement with no applications of NS-Trans and in which NS-Refl is only applied to type variables and type applications.

A consequence of the normalization of proofs is the following generation result.

Proposition 3.2.10 (*Generation for normal subtyping*)

1. $\Gamma \vdash_n X \leq B$ implies $X \equiv B$ and $\Gamma \vdash X \in K$ for some K, or $\Gamma \vdash_n \Gamma(X) \leq B$.
2. $\Gamma \vdash_n S{\to}A \leq B$ implies $B \equiv T{\to}C$, $\Gamma \vdash_n T \leq S$, $\Gamma \vdash_n A \leq C$, and $\Gamma \vdash S{\to}A \in \star$.
3. $\Gamma \vdash_n \forall X{\leq}S{:}K.A \leq B$ implies $B \equiv \forall X{\leq}S{:}K.C$, $\Gamma,\, X{\leq}S{:}K \vdash_n A \leq C$, and $\Gamma \vdash \forall X{\leq}S{:}K.A \in \star$.
4. $\Gamma \vdash_n \Lambda X{:}K.A \leq B$ implies $B \equiv \Lambda X{:}K.C$ and $\Gamma,\, X{\leq}\top^K{:}K \vdash_n A \leq C$.
5. $\Gamma \vdash_n A\,S \leq B$ implies $B \equiv A\,S$, or $\Gamma \vdash_n (lub_\Gamma(A\,S))^{nf} \leq B$, and $\Gamma \vdash A\,S \in K$.
6. $\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq B$ implies that there exists $j{\in}\{1..m\}$ such that $\Gamma \vdash_n A_j \leq B$ and $\forall k{\in}\{1..m\}\, \Gamma \vdash A_k \in K$.
7. $\Gamma \vdash_n A \leq \bigwedge^K[B_1..B_n]$ implies that for each $i{\in}\{1..n\}$ $\Gamma \vdash_n A \leq B_i$ and $\Gamma \vdash A \in K$.
8. $\Gamma \vdash_n \bigwedge^K[A_1..A_m] \leq \bigwedge^K[B_1..B_n]$ implies that for each $i{\in}\{1..n\}$ there exists $j{\in}\{1..m\}$ such that $\Gamma \vdash_n A_j \leq B_i$ and $\forall k{\in}\{1..m\}\, \Gamma \vdash A_k \in K$.

Moreover, given a normal proof of any of the antecedents, the proofs of the consequents are proper subderivations.

PROOF: In each case, given a proof of the antecedent, there is also a proof in normal form. Due to lemma 3.2.8(2), such a derivation cannot end with an application of NS-TRANS, and, because of lemma 3.2.8(1), if it ends with NS-REFL, then it is a derivation of a subtyping statement between type variables or type applications. Finally, the result follows by inspection of the other rules. □

LEMMA 3.2.11

1. $\Gamma \vdash_n T \leq \bigwedge^K[A_1..A_n]$ if and only if $\Gamma \vdash_n T \leq A_k$ for each $k \in \{1..n\}$.
2. $\Gamma \vdash_n T \leq \bigwedge^K[A_1..A_n]$ if and only if $\Gamma \vdash_n T \leq \bigwedge^K[A_k]$ for each $k \in \{1..n\}$.
3. Let $\Gamma \vdash \bigwedge^K[A_1..A_n] \in K$. Then $\Gamma \vdash_n \bigwedge^K[A_1..A_n] \leq T$ if and only if $\Gamma \vdash_n A_k \leq T$ for some $k \in \{1..n\}$.

PROOF: By induction on derivations, using lemma 3.2.7 and generation. □

3.3 Equivalence of ordinary and normal subtyping

In this section, we show that a subtyping statement is derivable in $F^{\omega}_{\wedge}$ if and only if the corresponding normalized statement is derivable in $NF^{\omega}_{\wedge}$. This equivalence is proved in theorem 3.3.9. As usual, we need some auxiliary properties and definitions, among which we can highlight propositions 3.3.2 and 3.3.8.

LEMMA 3.3.1 Let $lub_\Gamma(S)$ be defined. Then

1. $\Gamma \vdash S \in K$ implies $\Gamma \vdash lub_\Gamma(S) \in K$.
2. $\Gamma \vdash S \leq lub_\Gamma(S)$.

PROOF: Item 1 follows by induction on derivations, while item 2 follows by induction on the structure of S. □

PROPOSITION 3.3.2 (*Soundness*) If $\Gamma \vdash_n S \leq T$, then $\Gamma \vdash S \leq T$.

PROOF: By induction on the derivation of $\Gamma \vdash_n S \leq T$.

NS-REFL By S-CONV.

NS-TVAR By the induction hypothesis, S-TVAR and S-TRANS.

NS-OAPP By the induction hypothesis, lemma 3.3.1(2), S-CONV and S-TRANS.

NS-∃ We are given that for each k in $\{1..n\}$ $\Gamma \vdash A_k \in K$, and there is a j in $\{1..n\}$ such that $\Gamma \vdash_n A_j \leq B$. By K-MEET, $\Gamma \vdash \bigwedge^K[A_1..A_n] \in K$, and, by S-MEET-LB $\Gamma \vdash \bigwedge^K[A_1..A_n] \leq A_k$ for each k, in particular for j. Hence the result follows by the induction hypothesis and S-TRANS.

NS-$\forall$ We are given that $\Gamma \vdash A \in K$, and for each i in $\{1..m\}$ $\Gamma \vdash_n A \leq B_i$. Hence the result follows by the induction hypothesis and S-Meet-G.

NS-$\forall\exists$ We are given that for each k in $\{1..n\}$ $\Gamma \vdash A_k \in K$, and for each i in $\{1..m\}$ there is a j in $\{1..n\}$ such that $\Gamma \vdash_n A_j \leq B_i$. By K-Meet, $\Gamma \vdash \bigwedge^K[A_1..A_n] \in K$, and, by S-Meet-LB, $\Gamma \vdash \bigwedge^K[A_1..A_n] \leq A_k$ for each k. Hence the result follows by the induction hypothesis, S-Trans and S-Meet-G.

Other cases By the induction hypothesis and the corresponding rule in the other system. □

The following lemma says that empty intersections, $\top^K$, are maximal elements of the subtyping order.

Lemma 3.3.3

1. $\Gamma \vdash T \in K$ implies $\Gamma \vdash_n T \leq \top^K$.
2. $\Gamma \vdash T \in K$ implies $\Gamma \vdash T \leq \top^K$.

Proof: Statement 1 follows by the cases $m = 0$ in NS-$\forall$ and NS-$\forall\exists$. Statement 2 is the case $n = 0$ in S-Meet-G. □

Lemma 3.3.4

1. $\Gamma \vdash$ ok implies $\Gamma^{nf} \vdash$ ok.
2. $\Gamma \vdash T \in K$ implies $\Gamma^{nf} \vdash T \in K$.
3. $\Gamma \vdash S \leq T$ implies $\Gamma^{nf} \vdash S \leq T$.
4. Let $\Gamma_1, \Gamma_2 \vdash$ ok. Then $\Gamma_1^{nf}, \Gamma_2 \vdash T \in K$ implies $\Gamma_1, \Gamma_2 \vdash T \in K$.
5. Let $\Gamma_1, \Gamma_2 \vdash$ ok. Then $\Gamma_1^{nf}, \Gamma_2 \vdash S \leq T$ implies $\Gamma_1, \Gamma_2 \vdash S \leq T$.
6. Let $\Gamma \vdash S, T \in K$. Then $\Gamma^{nf} \vdash S^{nf} \leq T^{nf}$ if and only if $\Gamma \vdash S \leq T$.

Proof: Statements 1 and 2 follow by simultaneous induction on the size of derivations using lemma 2.4.17. Statement 3 follows by induction on the derivation of $\Gamma \vdash S \leq T$ using part 1, part 2, and lemma 2.4.17. Statement 4 follows by induction on the derivation of $\Gamma_1^{nf}, \Gamma_2 \vdash T \in K$. Item 5 follows by induction on the derivation of $\Gamma_1^{nf}, \Gamma_2 \vdash S \leq T$, using part 4. Item 6 is a corollary of part 3, part 5 and lemma 2.4.16. □

In the last lemma, items 1, 2, and 3 show that well formation of contexts, kinding judgements, and subtyping judgements are invariant under normalization of contexts, while items 4 and 5 are the converse of 2 and 3 respectively.

The following lemma states that S-TVar is an admissible rule in $NF^\omega_\wedge$.

Lemma 3.3.5 Let Γ be a context in normal form such that $\Gamma \vdash \text{ok}$ and $Y \in \text{dom}(\Gamma)$. Then $\Gamma \vdash_n Y \leq \Gamma(Y)$.

Proof: Let $\Gamma \equiv \Gamma_1, Y \leq T{:}K, \Gamma_2$. By lemma 2.4.2, $\Gamma_1 \vdash T \in K$. If T is not an intersection, then, by NS-Refl and NS-TVar, we have $\Gamma \vdash_n Y \leq T$. If $T \equiv \bigwedge^{K'}[B_1..B_m]$, then by generation for kinding and unicity of kinds, $\Gamma \vdash B_i \in K$ for each i and $K \equiv K'$. By NS-Refl, $\Gamma \vdash_n B_i \leq B_i$ for each i. Then, by NS-$\exists$ and NS-TVar, it follows that $\Gamma \vdash_n Y \leq B_i$ for each i, and, by NS-$\forall$, $\Gamma \vdash_n Y \leq T$. □

Lemma 3.3.6 (*Substitution*) If $\Gamma \vdash U \in K$ and $\Gamma, X{:}K, \Gamma' \vdash_n S \leq T$, then $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n (S[X{\leftarrow}U])^{nf} \leq (T[X{\leftarrow}U])^{nf}$.

Proof: By induction on the derivation of $\Gamma, X{:}K, \Gamma' \vdash_n S \leq T$. For the sake of clarity, we sometimes leave out kinding judgements and their justifications which follow easily from the structural properties in section 2.4. Let $\Gamma'' \equiv \Gamma, X{:}K, \Gamma'$.

NS-Refl By the type substitution lemma 2.4.11, lemma 3.3.4(2), subject reduction for kinds (lemma 2.4.12), and NS-Refl.

NS-Trans By the induction hypothesis and NS-Trans.

NS-TVar We are given $\Gamma'' \vdash_n \Gamma''(Y) \leq A$. We have to consider three cases.

1. $Y \equiv X$. By subject reduction, $\Gamma \vdash U^{nf} \in K$, and by lemma 3.3.3(1), it follows that $\Gamma \vdash_n U^{nf} \leq \top^K$. By weakening, it follows that $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n U^{nf} \leq \top^K$ and, by the induction hypothesis, it follows that $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n \top^K \leq (A[X{\leftarrow}U])^{nf}$. Finally, the result follows by NS-Trans.
2. $Y \in \text{dom}(\Gamma)$. By the free variables lemma, $X \notin \text{FV}(\Gamma(Y))$ and $X \not\equiv Y$. By lemmas 2.4.11, 3.3.4(1), and 3.3.5, it follows that $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n Y \leq \Gamma(Y)$, and, by the induction hypothesis, it follows that $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n \Gamma(Y) \leq (A[X{\leftarrow}U])^{nf}$. Finally, the result follows by NS-Trans.
3. $Y \in \text{dom}(\Gamma')$. By the induction hypothesis, it follows that
$$\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n (\Gamma'(Y)[X{\leftarrow}U])^{nf} \leq (A[X{\leftarrow}U])^{nf}.$$
By lemmas 2.4.11, 3.3.4(1), and 3.3.5,
$$\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n Y \leq (\Gamma, (\Gamma'[X{\leftarrow}U])^{nf})(Y).$$
Furthermore, $(\Gamma, (\Gamma'[X{\leftarrow}U])^{nf})(Y) = (\Gamma'(Y)[X{\leftarrow}U])^{nf}$. Hence the result follows by NS-Trans.

NS-Arrow We are given that $\Gamma'' \vdash_n T_1 \leq S_1$ and $\Gamma'' \vdash_n S_2 \leq T_2$. By the induction hypothesis, it follows that $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n (T_1[X{\leftarrow}U])^{nf} \leq (S_1[X{\leftarrow}U])^{nf}$ and $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n (S_2[X{\leftarrow}U])^{nf} \leq (T_2[X{\leftarrow}U])^{nf}$. There are four cases to consider, since $(T_2[X{\leftarrow}U])^{nf}$ and $(S_2[X{\leftarrow}U])^{nf}$ may be intersections or not. We shall consider only two of them to illustrate the proof method.

1. $(T_2[X{\leftarrow}U])^{nf}$ and $(S_2[X{\leftarrow}U])^{nf}$ are not intersections. Then the result follows by applying NS-Arrow.
2. $(S_2[X{\leftarrow}U])^{nf} = \bigwedge^{\star}[C_1..C_n]$ and $(T_2[X{\leftarrow}U])^{nf}$ is not an intersection. Then we have that
$$((T_1{\rightarrow}T_2)[X{\leftarrow}U])^{nf} = (T_1[X{\leftarrow}U])^{nf}{\rightarrow}(T_2[X{\leftarrow}U])^{nf} \quad \text{and}$$
$$((S_1{\rightarrow}S_2)[X{\leftarrow}U])^{nf}$$
$$= \bigwedge^{\star}[(S_1[X{\leftarrow}U])^{nf}{\rightarrow}C_1..(S_1[X{\leftarrow}U])^{nf}{\rightarrow}C_n].$$
By lemma 3.2.10, it follows that for some i $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n C_i \leq (T_2[X{\leftarrow}U])^{nf}$. Applying NS-Arrow, $\Gamma, (\Gamma'[X{\leftarrow}U])^{nf} \vdash_n (S_1[X{\leftarrow}U])^{nf}{\rightarrow}C_i \leq (T_1[X{\leftarrow}U])^{nf}{\rightarrow}(T_2[X{\leftarrow}U])^{nf}$. Finally, the result follows by NS-$\exists$.

Other cases NS-OApp is similar to the case for NS-TVar using lemma 2.3.1.4(3) and uniqueness of normal forms. All other cases are similar to that of NS-Arrow. □

This substitution lemma is the key result we use in proving that S-OApp has a corresponding admissible rule in $NF^{\omega}_{\wedge}$.

Lemma 3.3.7 $\Gamma \vdash S\,U \in K$. Then $\Gamma \vdash_n S \leq T$ implies $\Gamma \vdash_n (S\,U)^{nf} \leq (T\,U)^{nf}$.

Proof: By induction on the derivation of $\Gamma \vdash_n S \leq T$, assuming a derivation in normal form. The cases for NS-Arrow and NS-All are impossible because of the assumption $\Gamma \vdash S\,U \in K$.

NS-Refl By subject reduction for kinds and NS-Refl.

NS-TVar We are given $\Gamma \vdash_n \Gamma(X) \leq A$. By the induction hypothesis, $\Gamma \vdash_n (\Gamma(X)U)^{nf} \leq (AU)^{nf}$. We have to consider two cases.

$(AU)^{nf} \equiv B$	By NS-OApp.
$(AU)^{nf} \equiv \bigwedge^K[A_1..A_n]$	By lemma 3.2.11, $\Gamma \vdash_n (\Gamma(X)U)^{nf} \leq A_k$ for each k in $\{1..n\}$. By NS-OApp, $\Gamma \vdash_n X\,U \leq (A_k)$ for each k, which, by NS-$\forall$, implies $\Gamma \vdash_n X\,U \leq (AU)^{nf}$.

NS-OAbs We are given $\Gamma, X{:}K \vdash_n S_1 \leq T_1$. By the substitution lemma 3.3.6, it follows that $\Gamma \vdash_n (S_1[X{\leftarrow}U])^{nf} \leq (T_1[X{\leftarrow}U])^{nf}$. On the other hand, we have that $(\Lambda X{:}K.S_1)U \rightarrow_{\beta\wedge} S_1[X{\leftarrow}U]$ and $(\Lambda X{:}K.T_1)U \rightarrow_{\beta\wedge} T_1[X{\leftarrow}U]$. Finally, the result follows by the uniqueness of normal forms.

NS-OApp Similar to case NS-TVar.

NS-$\forall\exists$ By the induction hypothesis and NS-$\forall\exists$, using generation for subtyping.

NS-∃ and NS-∀ By the induction hypothesis, using lemma 3.2.11. □

PROPOSITION 3.3.8 (*Completeness*) If $\Gamma \vdash S \leq T$, then $\Gamma^{nf} \vdash_n S^{nf} \leq T^{nf}$.

PROOF: By induction on the derivation of $\Gamma \vdash S \leq T$, using lemma 3.3.7 for the case of S-OAPP. □

THEOREM 3.3.9 (*Equivalence of ordinary and normal subtyping*) Let $\Gamma \vdash S \in K$ and $\Gamma \vdash T \in K$. Then $\Gamma \vdash S \leq T$ if and only if $\Gamma^{nf} \vdash_n S^{nf} \leq T^{nf}$.

PROOF:

⇒) By completeness (3.3.8).

⇐) By soundness (3.3.2), it follows that $\Gamma^{nf} \vdash S^{nf} \leq T^{nf}$, and, by lemma 3.3.4(6), it follows that $\Gamma \vdash S \leq T$. □

3.3.1 Least strict upper bound

So far we only used that $lub_{\Gamma}(S)$ is an upper bound of S in the context Γ (See lemma 3.3.1(2)). We can now give the final motivation of the name we chose, showing that if $lub_{\Gamma}(S)$ is defined and $T \neq_{\beta\wedge} S$, then $\Gamma \vdash S \leq T$ implies $\Gamma \vdash lub_{\Gamma}(S) \leq T$. We first show that the corresponding property holds for the normalized system.

LEMMA 3.3.1.1 Let $lub_{\Gamma}(S)$ be defined. Then

1. If $S \twoheadrightarrow_{\beta\wedge} S'$ and $\Gamma \twoheadrightarrow_{\beta\wedge} \Gamma'$, then $lub_{\Gamma}(S) \twoheadrightarrow_{\beta\wedge} lub_{\Gamma'}(S')$.
2. If $\Gamma \vdash_n S \leq T$, then $\Gamma \vdash_n lub_{\Gamma}(S)^{nf} \leq T$ or $S \equiv T$.

PROOF:

1. By induction on the structure of S, observing that if $lub_{\Gamma}(S)$ is defined, so is $lub_{\Gamma'}(S')$.
2. By induction on the derivation of $\Gamma \vdash_n S \leq T$. It is immediate for the case NS-REFL; for NS-ARROW, NS-ALL, and NS-OABS $lub_{\Gamma}(S)$ is not defined; for the other rules the result follows using the induction hypothesis. □

COROLLARY 3.3.1.2 Let $lub_{\Gamma}(S)$ be defined. Then $\Gamma \vdash S \leq T$ and $T \neq_{\beta\wedge} S$ implies $\Gamma \vdash lub_{\Gamma}(S) \leq T$.

PROOF: By completeness, it follows that $\Gamma^{nf} \vdash_n S^{nf} \leq T^{nf}$. By lemma 3.3.1.1(2), $\Gamma^{nf} \vdash_n (lub_{\Gamma^{nf}}(S^{nf}))^{nf} \leq T^{nf}$, because $S^{nf} \not\equiv T^{nf}$. By soundness, it follows that $\Gamma^{nf} \vdash (lub_{\Gamma^{nf}}(S^{nf}))^{nf} \leq T^{nf}$, which is equivalent to $\Gamma \vdash lub_{\Gamma^{nf}}(S^{nf}) \leq T$ by lemma 3.3.4(6). Finally, (using lemmas 3.3.1(1) and 2.4.12, and proposition 2.4.19 to get the corresponding kinding judgements) it follows that $\Gamma \vdash lub_{\Gamma}(S) \leq T$ by lemma 3.3.1.1(1), S-CONV and S-TRANS. □

3.3.2 Example

In this section, we give the derivations in $F^{\omega}_{\wedge}$ and in $NF^{\omega}_{\wedge}$ of the example(3.1) mentioned in the introduction and in section 3.1.

Let $\Gamma \equiv X \le \Lambda Y{:}K.Y{:}K{\to}K,\ Z \le X{:}K{\to}K$. We present a proof of

$$\Gamma \vdash X(Z\,W) \le W$$

and a proof of its translation in the normal system

$$\Gamma^{nf} \vdash_n X(Z\,W)^{nf} \le W^{nf}.$$

Observe that $\Gamma^{nf} \equiv \Gamma,$
$(X(Z\,W))^{nf} \equiv X(Z\,W),$ and
$W^{nf} \equiv W.$

For the sake of readability we omit kinding judgements.

We have the following derivation in $F^{\omega}_{\wedge}$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \text{ok}}{\Gamma \vdash X \le \Lambda Y{:}K.Y}\ \text{S-TVar}}{\Gamma \vdash X(Z\,W) \le (\Lambda Y{:}K.Y)Z\,W}\ \text{S-OApp} \qquad \dfrac{(\Lambda Y{:}K.Y)Z\,W =_{\beta\wedge} Z\,W}{(\Lambda Y{:}K.Y)Z\,W \le Z\,W}\ \text{S-Conv}}{\Gamma \vdash X(Z\,W) \le Z\,W}\ \text{S-Trans}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \text{ok}}{\Gamma \vdash Z \le X}\ \text{S-TVar} \qquad \dfrac{\Gamma \vdash \text{ok}}{\Gamma \vdash X \le \Lambda Y{:}K.Y}\ \text{S-TVar}}{\Gamma \vdash Z \le (\Lambda Y{:}K.Y)}\ \text{S-Trans}}{\Gamma \vdash Z\,W \le (\Lambda Y{:}K.Y)W}\ \text{S-OApp} \qquad \dfrac{(\Lambda Y{:}K.Y)W =_{\beta\wedge} W}{(\Lambda Y{:}K.Y)W \le W}\ \text{S-Conv}}{\Gamma \vdash Z\,W \le W}\ \text{S-Trans}$$

$$\dfrac{\Gamma \vdash X(Z\,W) \le Z\,W \quad \Gamma \vdash Z\,W \le W}{\Gamma \vdash X(Z\,W) \le W}\ \text{S-Trans}$$

The corresponding derivation in normal form in $NF^{\omega}_{\wedge}$ is substantially shorter:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash W \in K}{\Gamma \vdash_n ((\Lambda Y{:}K.Y)W)^{nf} \le W}\ \text{NS-Refl}}{\Gamma \vdash_n X\,W \le W}\ \text{NS-OApp}}{\Gamma \vdash_n ((\Lambda Y{:}K.Y)(Z\,W))^{nf} \le W}\ \text{NS-OApp}}{\Gamma \vdash_n X(Z\,W) \le W}\ \text{NS-OApp}$$

3.4 A subtype checking algorithm, *Alg*$F^{\omega}_{\wedge}$

As it stands, $NF^{\omega}_{\wedge}$ as defined in section 3.1 is not a deterministic algorithm, because its rules are not syntax directed. Fortunately, we are not far away from an algorithmic presentation. In fact, corollary 3.2.9 is the bridge to the algorithmic presentation of the subtyping relation, $AlgF^{\omega}_{\wedge}$, which states that transitivity steps can be eliminated and reflexivity steps can be simplified. $AlgF^{\omega}_{\wedge}$ is obtained from $NF^{\omega}_{\wedge}$ by removing NS-Trans and restricting NS-Refl to type variables and type applications.

Definition 3.4.1 ($AlgF^{\omega}_{\wedge}$ *subtyping rules*)

$$\frac{\Gamma \vdash X \in K}{\Gamma \vdash_{Alg} X \leq X} \qquad \text{(AlgS-TVarRefl)}$$

$$\frac{\Gamma \vdash T\,S \in K}{\Gamma \vdash_{Alg} T\,S \leq T\,S} \qquad \text{(AlgS-OAppRefl)}$$

$$\frac{\Gamma \vdash_{Alg} \Gamma(X) \leq A \qquad X \not\equiv A}{\Gamma \vdash_{Alg} X \leq A} \qquad \text{(AlgS-TVar)}$$

$$\frac{\Gamma \vdash_{Alg} T_1 \leq S_1 \qquad \Gamma \vdash_{Alg} S_2 \leq T_2 \qquad \Gamma \vdash S_1 {\rightarrow} S_2 \in \star}{\Gamma \vdash_{Alg} S_1 {\rightarrow} S_2 \leq T_1 {\rightarrow} T_2} \qquad \text{(AlgS-Arrow)}$$

$$\frac{\Gamma,\, X{\leq}U{:}K \vdash_{Alg} S \leq T \qquad \Gamma \vdash \forall X{\leq}U{:}K.S \in \star}{\Gamma \vdash_{Alg} \forall X{\leq}U{:}K.S \leq \forall X{\leq}U{:}K.T} \qquad \text{(AlgS-All)}$$

$$\frac{\Gamma,\, X{\leq}\top^{K}{:}K \vdash_{Alg} S \leq T}{\Gamma \vdash_{Alg} \Lambda X{:}K.S \leq \Lambda X{:}K.T} \qquad \text{(AlgS-OAbs)}$$

$$\frac{\Gamma \vdash_{Alg} (lub_{\Gamma}(T\,S))^{nf} \leq A \qquad \Gamma \vdash T\,S \in K \qquad T\,S \not\equiv A}{\Gamma \vdash_{Alg} T\,S \leq A} \qquad \text{(AlgS-OApp)}$$

$$\frac{\forall i{\in}\{1..m\}\, \Gamma \vdash_{Alg} A \leq T_i \qquad \Gamma \vdash A \in K}{\Gamma \vdash_{Alg} A \leq \bigwedge^{K}[T_1..T_m]} \qquad \text{(AlgS-}\forall\text{)}$$

$$\frac{\exists j{\in}\{1..n\}\, \Gamma \vdash_{Alg} S_j \leq A \qquad \forall k{\in}\{1..n\}\, \Gamma \vdash S_k \in K}{\Gamma \vdash_{Alg} \bigwedge^{K}[S_1..S_n] \leq A} \qquad \text{(AlgS-}\exists\text{)}$$

$$\frac{\forall i{\in}\{1..m\}\, \exists j{\in}\{1..n\}\, \Gamma \vdash_{Alg} S_j \leq T_i \qquad \forall k{\in}\{1..n\}\, \Gamma \vdash S_k \in K}{\Gamma \vdash_{Alg} \bigwedge^{K}[S_1..S_n] \leq \bigwedge^{K}[T_1..T_m]} \qquad \text{(AlgS-}\forall\exists\text{)}$$

Lemma 3.4.2 (*Equivalence of normal and algorithmic subtyping*)
Let $\Gamma \vdash S, T \in K$. Then $\Gamma \vdash_n S \leq T$ if and only if $\Gamma \vdash_{Alg} S \leq T$.

Proof: ($\Rightarrow$) By corollary 3.2.9. ($\Leftarrow$) Immediate. □

We have thereby proved that $AlgF^{\omega}_{\wedge}$ is indeed a sound and complete algorithm to compute $F^{\omega}_{\wedge}$'s subtyping relation. We conclude the proof of decidability of subtyping in $F^{\omega}_{\wedge}$ by establishing in section 3.5 that $AlgF^{\omega}_{\wedge}$ always terminates.

Proposition 3.4.3 (*Equivalence of ordinary and algorithmic subtyping*)
Let $\Gamma \vdash S \in K$ and $\Gamma \vdash T \in K$. Then $\Gamma \vdash S \leq T$ if and only if $\Gamma^{nf} \vdash_{Alg} S^{nf} \leq T^{nf}$.

Proof: By the equivalence of ordinary and normal subtyping (theorem 3.3.9) and the equivalence of normal and algorithmic subtyping (lemma 3.4.2). □

3.5 Termination of subtype checking

The last step in proving the decidability of the subtyping relation of $F^{\omega}_{\wedge}$ is proving the termination or well-foundedness of the relation defined by the $AlgF^{\omega}_{\wedge}$ subtyping rules. We show this by reducing the well-foundedness of $AlgF^{\omega}_{\wedge}$ to the strong normalization property of the $\rightarrow_{\beta\wedge+}$ relation.

We begin by extending the language of types with the constructor + as follows.

$$\begin{array}{lcll}
\mathbb{T}^{+} & ::= & X & \text{type variable} \\
 & | & \mathbb{T}^{+}\rightarrow\mathbb{T}^{+} & \text{function type} \\
 & | & \forall(X{\leq}\mathbb{T}^{+}{:}\mathbb{K})\mathbb{T}^{+} & \text{polymorphic type} \\
 & | & \Lambda(X{:}\mathbb{K})\mathbb{T}^{+} & \text{operator abstraction} \\
 & | & \mathbb{T}^{+}\,\mathbb{T}^{+} & \text{operator application} \\
 & | & \bigwedge^{\mathbb{K}}[\mathbb{T}^{+}..\mathbb{T}^{+}] & \text{intersection at kind } \mathbb{K} \\
 & | & \mathbb{T}^{+}+\mathbb{T}^{+} & \text{choice}
\end{array}$$

Since we have enriched the language of types with a new type constructor, we need to extend our kinding judgements (section 2.2) with the following kinding rule.

$$\frac{\Gamma \vdash_{+} S \in K \qquad \Gamma \vdash_{+} T \in K}{\Gamma \vdash_{+} S+T \in K} \qquad \text{(K-Plus)}$$

$\rightarrow_{\beta\wedge+}$ is obtained from $\rightarrow_{\beta\wedge}$ by adding the reductions associated with the choice operator +, $S+T \rightarrow_{\beta\wedge+} S$ and $S+T \rightarrow_{\beta\wedge+} T$. We also need the corresponding kinding rule saying that $\Gamma \vdash S+T \in K$ whenever $\Gamma \vdash S, T \in K$. As far as we are aware, choice operators have not been used before to analyze subtyping.

Notation 3.5.1 We write + modulo commutativity and associativity.

We now define a new reduction $\rightarrow_{\beta\wedge+}$.

Definition 3.5.2 ($\rightarrow_{\beta\wedge+}$) The reduction on types $\rightarrow_{\beta\wedge+}$ is obtained from $\rightarrow_{\beta\wedge}$ (definition 2.2.1) by adding the following two rules:

1. $S+T \rightarrow_{\beta\wedge+} S$, and
2. $S+T \rightarrow_{\beta\wedge+} T$.

We also write $\rightarrow_{+}$ to refer to these two new reduction rules.

As usual, $\rightarrow_{\beta\wedge+}$ is extended to become a compatible relation with respect to type formation, $\twoheadrightarrow_{\beta\wedge+}$ is the reflexive, transitive closure of $\rightarrow_{\beta\wedge+}$, and $=_{\beta\wedge+}$ is the reflexive, symmetric, and transitive closure of $\rightarrow_{\beta\wedge+}$.

Proposition 3.5.3 (*Strong normalization for* $\rightarrow_{\beta\wedge+}$) If $\Gamma \vdash_{+} T \in K$, then every $\beta\wedge+$-reduction sequence starting from T is finite.

Proof: The result follows using the strategy used to prove that the reduction $\rightarrow_{\beta\wedge}$ is strongly normalizing on well kinded types (see theorem 2.5.10). We only need to modify the definition of saturated sets by adding the following closure condition:
if T, U, $R_1..R_n{\in}SN^+$, then $TR_1..R_n{\in}S$ and $UR_1..R_n{\in}S$ imply $(T{+}U)R_1..R_n{\in}S$.
□

Next, we define a measure for subtyping statements such that, given a subtyping rule, the measure of each hypothesis is smaller than that of the conclusion. Most measures for showing the well-foundedness of a relation defined by a set of inference rules involve a clever assignment of weights to judgements, often involving the number of symbols. We need a more sophisticated measure, since in AlgS-OApp it is not necessarily the case that the size of the hypothesis is smaller than the size of the conclusion.

We introduce a new mapping from types to types in the extended language in order to define a new measure on subtyping statements. To motivate the definition of this new measure, we analyze the behavior of type variables during subtype checking. Assume that we want to check if $\Gamma \vdash_{Alg} S \leq T$, where S is a variable or a type application. It can be the case that the judgement is obtained with an application of AlgS-TVar or AlgS-OApp, in which case we have to consider a new statement $\Gamma \vdash_{Alg} S' \leq T$, where S' is obtained from S by replacing a variable by its bound (and eventually normalizing). However, we do not replace every variable by its bound, as this would constitute an unsound operation with respect to subtyping.

Example 3.5.4 Two unrelated variables may have the same bound.

$X{\leq}\top^{\star}{:}\star,\ Y{\leq}\top^{\star}{:}\star \not\vdash X \leq Y,$ but

$X{\leq}\top^{\star}{:}\star,\ Y{\leq}\top^{\star}{:}\star \vdash \top^{\star} \leq \top^{\star}.$

Our new mapping, *plus*, includes in each type expression this nondeterministic behavior of its type variables.

Definition 3.5.5 (*plus*)

The mapping $plus_{\Gamma} : \mathbb{T}{\rightarrow}\mathbb{T}^{+}$ is defined as follows.

1. $plus_{\Gamma_1, X\leq T:K, \Gamma_2}(X) = X + plus_{\Gamma_1}(T)$,
2. $plus_{\Gamma}(T{\rightarrow}S) = plus_{\Gamma}(T){\rightarrow}plus_{\Gamma}(S)$,
3. $plus_{\Gamma}(\forall X{\leq}T{:}K.S) = \forall X{\leq}plus_{\Gamma}(T){:}K.plus_{\Gamma,X\leq T:K}(S)$,

4. $\mathit{plus}_{\Gamma}(\Lambda X{:}K.S) = \Lambda X{:}K.\mathit{plus}_{\Gamma,X:K}(S)$,

5. $\mathit{plus}_{\Gamma}(S\,T) = \mathit{plus}_{\Gamma}(S)\;\mathit{plus}_{\Gamma}(T)$,

6. $\mathit{plus}_{\Gamma}(\bigwedge^{K}[S_1..S_n]) = \bigwedge^{K}[\mathit{plus}_{\Gamma}(S_1)..\mathit{plus}_{\Gamma}(S_n)]$.

Example 3.5.6 $\mathit{plus}_{X\leq\top^{\star}:\star,\, Y\leq X:\star,\, Z\leq Y:\star}(Z) = Z + Y + X + \top^{\star}$.

We need to show that *plus* is well defined on well kinded arguments.

Lemma 3.5.7 (*Well-foundedness of plus*)
If $\Gamma \vdash T \in K$, then $\mathit{plus}_{\Gamma}(T)$ is defined.

Proof: Observe that the size_j of the kinding judgements of the arguments strictly decreases in each recursive call. Consider

$$\mathit{rank}(\mathit{plus}_{\Gamma}(S)) = \mathit{size}_j(\Gamma \vdash S \in \mathit{kind}(\Gamma, S)),$$

where $\mathit{size}_j(\Gamma \vdash S \in K)$ is the size of the derivation of the kinding judgement (see definition 2.4.8). The function *kind* can be defined straightforwardly using proposition 2.4.6, such that $\mathit{kind}(\Gamma, S) = K$ if $\Gamma \vdash S \in K$, and gives a constant *NoKind* otherwise. Moreover, lemma 2.4.9 implies that the function *kind* is total. Given that $\Gamma \vdash S \in K$, by lemmas 2.4.2(1) and 2.4.6, the rank decreases in each recursive call and the least value is that of $\mathit{size}(\vdash \top^{K} \in K)$. □

Lemma 3.5.8 If $\Gamma \vdash T \in K$, then $\Gamma \vdash_{+} \mathit{plus}_{\Gamma}(T) \in K$.

Proof: By induction on the derivation of $\Gamma \vdash T \in K$, observing that $\Gamma \vdash T \in K$ implies $\Gamma \vdash_{+} T \in K$. It is straightforward to verify that $\vdash_{+}$ satisfies weakening (see lemma 2.4.4). We consider here the case for K-TVar, the rest follows by straightforward induction. We are given, $\Gamma_1,\, X \leq T{:}K,\, \Gamma_2 \vdash \text{ok}$. By lemma 2.4.2, there is a proper subderivation of $\Gamma_1 \vdash T \in K$. Finally, the result follows by the induction hypothesis, weakening, and K-Plus. □

Lemma 3.5.9 (*Strengthening for plus*)

1. Let $X \notin \mathrm{FTV}(\Gamma_2) \cup \mathrm{FTV}(S)$. Then
$\Gamma_1,\, X{\leq}T_X{:}K_X,\, \Gamma_2 \vdash S \in K$ implies $\mathit{plus}_{\Gamma_1, X\leq T_X:K_X, \Gamma_2}(S) = \mathit{plus}_{\Gamma_1,\Gamma_2}(S)$.

2. $\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash S \in K$ implies $\mathit{plus}_{\Gamma_1, x:T, \Gamma_2}(S) = \mathit{plus}_{\Gamma_1,\Gamma_2}(S)$.

Proof:

1. By lemma 3.5.7, $\mathit{plus}_{\Gamma_1, X\leq T_X:K_X, \Gamma_2}(S)$ is defined, therefore we can reason by induction on the number of unfolding steps of *plus*. We proceed by case analysis on the form of S.

   $S \equiv Y$. We have to consider two cases.

(a) $\Gamma_1 \equiv \Delta_1,\ Y{\leq}T_1{:}K_1,\ \Delta_2$. Then, by definition,

$$\mathit{plus}_{\Gamma_1,\,X\leq T_X:K_X,\,\Gamma_2}(Y) = Y + \mathit{plus}_{\Delta_1}(T_1).$$

On the other hand, also by the definition of *plus*,

$$\mathit{plus}_{\Gamma_1,\Gamma_2}(Y) = Y + \mathit{plus}_{\Delta_1}(T_1).$$

(b) $\Gamma_2 \equiv \Delta_1,\ Y{\leq}T_1{:}K_1,\ \Delta_2$. By the definition of *plus*,

$$\mathit{plus}_{\Gamma_1,\,X\leq T_X:K_X,\,\Gamma_2}(Y) = Y + \mathit{plus}_{\Gamma_1,\,X\leq T_X:K_X,\,\Delta_1}(T_1).$$

By lemma 2.4.1,

$$\Gamma_1,\ X{\leq}T_X{:}K_X,\ \Gamma_2 \vdash \mathrm{ok},$$

and, by lemma 2.4.2(1),

$$\Gamma_1,\ X{\leq}T_X{:}K_X,\ \Delta_1 \vdash T_1 \in K_1.$$

Moreover, since $X \notin \mathrm{FTV}(\Gamma_2)$, it follows that $X \notin \mathrm{FTV}(\Delta_1) \cup \mathrm{FTV}(T_1)$. Then, applying the induction hypothesis we obtain

$$Y + \mathit{plus}_{\Gamma_1,\,X\leq T_X:K_X,\,\Delta_1}(T_1) = Y + \mathit{plus}_{\Gamma_1,\,\Delta_1}(T_1),$$

and the result follows by the definition of *plus*.

$S \equiv \forall Y{\leq}T_1{:}K_1.T_2$. By the definition of *plus*,

$$\begin{aligned}&\mathit{plus}_{\Gamma_1,\,X\leq T_X:K_X,\,\Gamma_2}(\forall Y{\leq}T_1{:}K_1.T_2)\\ &= \forall Y{\leq}\mathit{plus}_{\Gamma_1,\,\leq T_X:K_X,\,\Gamma_2}(T_1){:}K_1.\mathit{plus}_{\Gamma_1,\,X\leq T_X:K_X,\,\Gamma_2,\,Y\leq T_1:K_1}(T_2).\end{aligned}$$

By generation for kinding (proposition 2.4.6),

$$\Gamma_1,\ X{\leq}T_X{:}K_X,\ \Gamma_2,\ Y{\leq}T_1{:}K_1 \vdash T_2 \in \star,$$

and, since $X \notin \mathrm{FTV}(\Gamma_2,\ Y{\leq}T_1{:}K_1) \cup \mathrm{FTV}(T_2)$, by the induction hypothesis,

$$\begin{aligned}&\forall Y{\leq}\mathit{plus}_{\Gamma_1,\,\leq T_X:K_X,\,\Gamma_2}(T_1){:}K_1.\mathit{plus}_{\Gamma_1,\,X\leq T_X:K_X,\,\Gamma_2,\,Y\leq T_1:K_1}(T_2)\\ &= \forall Y{\leq}\mathit{plus}_{\Gamma_1,\,\leq T_X:K_X,\,\Gamma_2}(T_1){:}K_1.\mathit{plus}_{\Gamma_1,\,\Gamma_2,\,Y\leq T_1:K_1}(T_2).\end{aligned}$$

By lemma 2.4.1,

$$\Gamma_1,\ X{\leq}T_X{:}K_X,\ \Gamma_2,\ Y{\leq}T_1{:}K_1 \vdash \mathrm{ok},$$

by syntax directedness of context judgements (lemma 2.4.2(1)),

$$\Gamma_1,\ X{\leq}T_X{:}K_X,\ \Gamma_2 \vdash T_1 \in K_1.$$

Since $X \notin \mathrm{FTV}(\Gamma_2) \cup \mathrm{FTV}(T_1)$, by the induction hypothesis,

$$\begin{aligned}&\forall Y{\leq}\mathit{plus}_{\Gamma_1,\,\leq T_X:K_X,\,\Gamma_2}(T_1){:}K_1.\mathit{plus}_{\Gamma_1,\,\Gamma_2,\,Y\leq T_1:K_1}(T_2)\\ &= \forall Y{\leq}\mathit{plus}_{\Gamma_1,\Gamma_2}(T_1){:}K_1.\mathit{plus}_{\Gamma_1,\,\Gamma_2,\,Y\leq T_1:K_1}(T_2)\\ &= \mathit{plus}_{\Gamma_1,\Gamma_2}(\forall Y{\leq}T_1{:}K_1.T_2).\end{aligned}$$

For all the other cases, the result follows by straightforward application of the induction hypothesis, using generation for kinding (proposition 2.4.6).

2. the definition of *plus* does not depend on the assumptions of term variables. □

LEMMA 3.5.10 (*Weakening for plus*) If $\Gamma' \vdash \text{ok}$, $\Gamma \subseteq \Gamma'$, and $\Gamma \vdash S \in K$, then $\mathit{plus}_{\Gamma}(S) = \mathit{plus}_{\Gamma'}(S)$.

PROOF: The assumptions ensure that $\mathit{plus}_{\Gamma}(S)$ is defined, so we can proceed by induction on the number of unfolding steps of the definition of *plus*. We proceed by case analysis on the form of S.

$S \equiv X$. By generation for kinding (proposition 2.4.6) and the fact that $\Gamma \subseteq \Gamma'$,

$$\begin{aligned} &\Gamma \equiv \Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2 \quad \text{and} \\ &\Gamma' \equiv \Gamma_1',\, X{\leq}T{:}K,\, \Gamma_2'. \end{aligned}$$

There are two cases to consider.

1. If $\Gamma_1 \equiv \Gamma_1'$, then the result follows by the definition of *plus*.
2. If $\Gamma_1 \not\equiv \Gamma_1'$, then $\Gamma_1 \subseteq \Gamma_1' \cup \Gamma_2'$.
   By the definition of *plus*,

   $$\mathit{plus}_{\Gamma}(X) = X + \mathit{plus}_{\Gamma_1}(T).$$

   By lemmas 2.4.1 and 2.4.2(1), it follows that $\Gamma_1 \vdash T \in K$. Hence, by the induction hypothesis,

   $$X + \mathit{plus}_{\Gamma_1}(T) = X + \mathit{plus}_{\Gamma'}(T).$$

   Since $\Gamma' \vdash \text{ok}$, from lemma 2.4.2(1), it follows that $\Gamma_1' \vdash T \in K$. Consequently, $(\{X\} \cup \mathrm{FTV}(\Gamma_2')) \cap \mathrm{FTV}(T) = \emptyset$ by the free variables lemma (lemma 2.4.3). Hence, starting from the last declaration in Γ_2', we can iterate the strengthening lemma for *plus* (lemma 3.5.9 items 1 and 2) to obtain

   $$X + \mathit{plus}_{\Gamma'}(T) = X + \mathit{plus}_{\Gamma_1'}(T) = \mathit{plus}_{\Gamma'}(X).$$

$S \equiv \forall X{\leq}T_1{:}K_1.T_2$. By the definition of *plus*,

$$\mathit{plus}_{\Gamma}(\forall X{\leq}T_1{:}K_1.T_2) = \forall X{\leq}\mathit{plus}_{\Gamma}(T_1){:}K_1.\mathit{plus}_{\Gamma,\, X\leq T_1:K_1}(T_2).$$

By generation for kinding (proposition 2.4.6) and lemmas 2.4.1 and 2.4.2(1), it follows that $\Gamma \vdash T_1 \in K_1$. Then, by the induction hypothesis,

$$\forall X{\leq}\mathit{plus}_{\Gamma}(T_1){:}K_1.\mathit{plus}_{\Gamma,\, X\leq T_1:K_1}(T_2) = \forall X{\leq}\mathit{plus}_{\Gamma'}(T_1){:}K_1.\mathit{plus}_{\Gamma,\, X\leq T_1:K_1}(T_2).$$

By generation for kinding, $\Gamma,\, X{\leq}T_1{:}K_1 \vdash T_2 \in \star$. By weakening for kinding (lemma 2.4.4), $\Gamma' \vdash T_1 \in K_1$, and, by C-TVAR, $\Gamma',\, X{\leq}T_1{:}K_1 \vdash \text{ok}$. Applying again the induction hypothesis, it follows that

$$\begin{aligned} &\forall X{\leq}\mathit{plus}_{\Gamma'}(T_1){:}K_1.\mathit{plus}_{\Gamma,\, X\leq T_1:K_1}(T_2) \\ &= \forall X{\leq}\mathit{plus}_{\Gamma'}(T_1){:}K_1.\mathit{plus}_{\Gamma',\, X\leq T_1:K_1}(T_2) \\ &= \mathit{plus}_{\Gamma'}(\forall X{\leq}T_1{:}K_1.T_2). \end{aligned}$$

$S \equiv \Lambda X{:}K.T$. By the definition of *plus*,

$$plus_{\Gamma}(\Lambda X{:}K.T) = \Lambda X{:}K.plus_{\Gamma,\, X:K}(T).$$

By K-Meet, it follows that $\Gamma' \vdash \top^K \in K$, and, by C-TVar, Γ', $X{\leq}\top^K{:}K \vdash$ ok. Finally, the result follows by the induction hypothesis.

In all other cases, the proof follows by straightforward application of the induction hypothesis. □

The operation *plus* does not have the usual properties under substitution; as following example shows, the equality

$$plus_{\Gamma_1,\, X\leq S:K_1,\, \Gamma_2}(T_2)[X{\leftarrow}plus_{\Gamma_1}(T_1)] = plus_{\Gamma_1,\, \Gamma_2[X\leftarrow T_1]}(T_2[X{\leftarrow}T_1])$$

does not hold in general.

Example 3.5.11 Consider the case where

$\Gamma_1 \equiv Y{\leq}\top^{\star}{:}\star, \quad \Gamma_2 \equiv \emptyset, \quad S \equiv Y, \quad T_1 \equiv Y, \quad$ and $\quad T_2 \equiv X.$

Then

$$\begin{array}{rcl} plus_{Y\leq\top^{\star}:\star,\, X\leq Y:\star}(X)[X{\leftarrow}plus_{Y\leq\top^{\star}:\star}(Y)] & = & (X + Y + \top^{\star})[X{\leftarrow}(Y + \top^{\star})] \\ & = & Y + \top^{\star} + Y + \top^{\star}. \end{array}$$

On the other hand,

$$plus_{Y\leq\top^{\star}:\star}(X[X{\leftarrow}Y]) \;=\; plus_{Y\leq\top^{\star}:\star}(Y) \;=\; Y + \top^{\star}.$$

We therefore need a lemma which says that the well-typed types are well-behaved under substitution with respect to the *plus* operation.

Lemma 3.5.12 (*Substitution for plus*) If Γ_1, $X{\leq}S{:}K_1$, $\Gamma_2 \vdash T_2 \in K_2$ and $\Gamma_1 \vdash T_1 \in K_1$, then

$$plus_{\Gamma_1,\, X\leq S:K_1,\, \Gamma_2}(T_2)[X{\leftarrow}plus_{\Gamma_1}(T_1)] \twoheadrightarrow_{\beta\wedge+} plus_{\Gamma_1,\, \Gamma_2[X\leftarrow T_1]}(T_2[X{\leftarrow}T_1]).$$

Proof: By induction on the size of the derivation of Γ_1, $X{\leq}S{:}K_1$, $\Gamma_2 \vdash T_2 \in K_2$. We proceed by case analysis on the form of T_2.

$T_2 \equiv Y$. By the free variables lemma (lemma 2.4.3), $Y{\in}\mathrm{dom}(\Gamma_1,\, X{\leq}S{:}K_1,\, \Gamma_2)$. Then there are three cases to consider.

$Y{\in}\mathrm{dom}(\Gamma_1)$. Let $\Gamma_1 \equiv \Delta_1$, $Y{\leq}U{:}K$, Δ_2. Then

$$\begin{array}{l} plus_{\Gamma_1,\, X\leq S:K_1,\Gamma_2}(Y)[X{\leftarrow}plus_{\Gamma_1}(T_1)], \\ \qquad \text{by the definitions of } plus \text{ and substitution,} \\ = Y + (plus_{\Delta_1}(U)[X{\leftarrow}plus_{\Gamma_1}(T_1)]) \\ \qquad \text{since } X \notin \mathrm{FTV}(U) \cup \mathrm{FTV}(\Delta_1), X \notin \mathrm{FTV}(plus_{\Delta_1}(U)). \\ = Y + plus_{\Delta_1}(U), \\ = plus_{\Gamma_1,\Gamma_2[X\leftarrow T_1]}(Y[X{\leftarrow}T_1]). \end{array}$$

$Y \equiv X$. Then

$plus_{\Gamma_1,\,X\leq S:K_1,\Gamma_2}(X)[X\leftarrow plus_{\Gamma_1}(T_1)]$,

by the definitions of *plus* and substitution,

$= plus_{\Gamma_1}(T_1) + (plus_{\Delta_1}(U)[X\leftarrow plus_{\Gamma_1}(T_1)])$,

$\rightarrow_{+} plus_{\Gamma_1}(T_1)$.

On the other hand,

$plus_{\Gamma_1,\Gamma_2[X\leftarrow T_1]}(X[X\leftarrow T_1])$

$= plus_{\Gamma_1,\Gamma_2[X\leftarrow T_1]}(T_1)$,

since $\mathrm{FTV}(T_1) \cup \mathrm{dom}(\Gamma_1)$, by strengthening for *plus*(3.5.9),

$= plus_{\Gamma_1}(T_1)$.

$Y{\in}\mathrm{dom}(\Gamma_2)$. Let $\Gamma_2 \equiv \Delta_1,\ Y{\leq}U{:}K,\ \Delta_2$. Then

$plus_{\Gamma_1,\,X\leq S:K_1,\Gamma_2}(Y)[X\leftarrow plus_{\Gamma_1}(T_1)]$,

by the definitions of *plus* and substitution,

$= Y + (plus_{\Gamma_1,\,X\leq S:K_1,\,\Delta_1}(U)[X\leftarrow plus_{\Gamma_1}(T_1)])$,

by generation (2.4.6) and the induction hypothesis,

$\rightarrow_{\beta\wedge+} Y + plus_{\Gamma_1,\,\Delta_1[X\leftarrow T_1]}(U[X\leftarrow T_1])$,

$= plus_{\Gamma_1,\Gamma_2[X\leftarrow T_1]}(Y[X\leftarrow T_1])$.

$T_2 \equiv \forall Y{\leq}S_1{:}K.S_2$. Let $\Gamma \equiv \Gamma_1,\ X{\leq}S{:}K_1,\ \Gamma_2$. Then

$plus_{\Gamma}(\forall Y{\leq}S_1{:}K.S_2)[X\leftarrow plus_{\Gamma_1}(T_1)]$,

by the definitions of *plus* and substitution,

$= \forall Y{\leq}plus_{\Gamma}(S_1)[X\leftarrow plus_{\Gamma_1}(T_1)]{:}K.plus_{\Gamma,\,Y\leq S_1:K}(S_2)[X\leftarrow plus_{\Gamma_1}(T_1)]$,

by generation (proposition 2.4.6) and the induction hypothesis,

$\rightarrow_{\beta\wedge+} \forall Y{\leq}plus_{\Gamma_1,\Gamma_2[X\leftarrow T_1]}(S_1[X\leftarrow T_1]){:}K.$

$plus_{\Gamma_1,\,\Gamma_2[X\leftarrow T_1],\,Y\leq S_1[X\leftarrow T_1]:K}(S_2[X\leftarrow T_1])$

by the definitions of *plus* and substitution,

$= plus_{\Gamma_1,\Gamma_2[X\leftarrow T_1]}((\forall Y{\leq}S_1{:}K.S_2)[X\leftarrow T_1])$.

Other cases. All the other cases are similar to the case $T_2 \equiv \forall Y{\leq}S_1{:}K.S_2$. □

Lemma 3.5.13 (*Monotonicity of plus with respect to* $\rightarrow_{\beta\wedge}$) If $\Gamma \vdash T \in K$, then

1. $\Gamma \rightarrow_{\beta\wedge} \Gamma'$ implies $plus_{\Gamma}(T) \twoheadrightarrow_{\beta\wedge+} plus_{\Gamma'}(T)$.

2. $T \rightarrow_{\beta\wedge} T'$ implies $plus_{\Gamma}(T) \twoheadrightarrow_{\beta\wedge+}{}^{>0}\ plus_{\Gamma}(T')$.

Proof: By simultaneous induction on the size of the derivation of $\Gamma \vdash T \in K$. We proceed by case analysis on the form of T.

1. $\Gamma \rightarrow_{\beta\wedge} \Gamma'$.

$T \equiv X$. Let $\Gamma \equiv \Gamma_1,\ X{\leq}S{:}K_1,\ \Gamma_2$. Then we have to consider three cases.

(a) $\Gamma_1 \rightarrow_{\beta\wedge} \Gamma'_1$. Then

$$\begin{aligned}&\mathit{plus}_{\Gamma}(X) = X + \mathit{plus}_{\Gamma_1}(S)\\ &\qquad\text{by lemma 2.4.2 and part (1) of the induction hypothesis,}\\ &\twoheadrightarrow_{\beta\wedge} X + \mathit{plus}_{\Gamma'_1}(S) = \mathit{plus}_{\Gamma'_1}(X).\end{aligned}$$

(b) $S \rightarrow_{\beta\wedge} S'$. By lemma 2.4.2 and part (2) of the induction hypothesis.

(c) $\Gamma_2 \rightarrow_{\beta\wedge} \Gamma'_2$. By the definition of *plus*.

$T \equiv \forall X{\leq}T_1{:}K_1.T_2$. By generation for kinds (proposition 2.4.6), there are proper subderivations of $\Gamma \vdash T_1 \in K_1$ and $\Gamma,\ X{\leq}T_1{:}K_1 \vdash T_2 \in \star$. Then, by part (1) of the induction hypothesis, it follows that

$$\begin{aligned}\mathit{plus}_{\Gamma}(T_1) &\quad\twoheadrightarrow_{\beta\wedge}\quad \mathit{plus}_{\Gamma'}(T_1), \text{ and}\\ \mathit{plus}_{\Gamma,\,X\leq T_1:K_1}(T_2) &\quad\twoheadrightarrow_{\beta\wedge}\quad \mathit{plus}_{\Gamma',\,X\leq T_1:K_1}(T_2).\end{aligned}$$

The result follows by the definitions of *plus* and $\twoheadrightarrow_{\beta\wedge}$.

Other cases. The rest of the cases are similar to the case $T \equiv \forall X{\leq}T_1{:}K_1.T_2$, using generation for kinding (proposition 2.4.6) and part 1 of the induction hypothesis.

2. $T \rightarrow_{\beta\wedge} T'$.

$T \equiv \forall X{\leq}T_1{:}K_1.T_2$. We have to consider three cases.

(a) $T_1 \rightarrow_{\beta\wedge} T'_1$. By generation for kinding (proposition 2.4.6), there are proper subderivations of $\Gamma \vdash T_1 \in K_1$ and $\Gamma,\ X{\leq}T_1{:}K_1 \vdash T_2 \in \star$. Then, by parts (2) and (1) of the induction hypothesis respectively, it follows that

$$\begin{aligned}\mathit{plus}_{\Gamma}(T_1) &\quad{\twoheadrightarrow_{\beta\wedge}}^{>0}\quad \mathit{plus}_{\Gamma}(T'_1), \text{ and}\\ \mathit{plus}_{\Gamma,\,X\leq T_1:K_1}(T_2) &\quad\twoheadrightarrow_{\beta\wedge}\quad \mathit{plus}_{\Gamma,\,X\leq {T_1}':K_1}(T_2).\end{aligned}$$

The result follows by the definitions of *plus* and $\twoheadrightarrow_{\beta\wedge}$.

(b) $T_2 \rightarrow_{\beta\wedge} T'_2$. By part (2) of the induction hypothesis.

(c) $\forall X{\leq}T_1{:}K_1.\bigwedge^{\star}[S_1..S_n] \rightarrow_{\beta\wedge} \bigwedge^{\star}[\forall X{\leq}T_1{:}K_1.S_1..\forall X{\leq}T_1{:}K_1.S_n]$.

$$\begin{aligned}&\mathit{plus}_{\Gamma}(\forall X{\leq}T_1{:}K_1.\textstyle\bigwedge^{\star}[S_1..S_n])\\ &= \forall X{\leq}\mathit{plus}_{\Gamma}(T_1){:}K.\textstyle\bigwedge^{\star}[\mathit{plus}_{\Gamma,\,X\leq T_1:K_1}(S_1)..\mathit{plus}_{\Gamma,\,X\leq T_1:K_1}(S_n)]\\ &\rightarrow_{\beta\wedge+} \textstyle\bigwedge^{\star}[\forall X{\leq}\mathit{plus}_{\Gamma}(T_1){:}K.\mathit{plus}_{\Gamma,\,X\leq T_1:K_1}(S_1)..\\ &\qquad\qquad ..\forall X{\leq}\mathit{plus}_{\Gamma}(T_1){:}K.\mathit{plus}_{\Gamma,\,X\leq T_1:K_1}(S_n)]\\ &= \mathit{plus}_{\Gamma}(\textstyle\bigwedge^{\star}[\forall X{\leq}T_1{:}K_1.S_1..\forall X{\leq}T_1{:}K_1.S_n])\end{aligned}$$

$T \equiv T_1\,T_2$. We have to consider four cases.

(a) $T_1 \rightarrow_{\beta\wedge} T'_1$,

(b) $T_2 \rightarrow_{\beta\wedge} T'_2$,

(c) $T \equiv \bigwedge^{\star}[S_1..S_n]$ and $\bigwedge^{\star}[S_1..S_n]T_2 \rightarrow_{\beta\wedge} \bigwedge^{\star}[S_1\,T_2..S_n\,T_2]$.

(d) $T \equiv \Lambda X{:}K.S_1$ and $(\Lambda X{:}K.S_1)\,T_2 \rightarrow_{\beta\wedge} S_1[X{\leftarrow}T_2]$

Cases 2a, 2b, and 2c follow using similar arguments to those used for the case $T \equiv \forall X{\leq}T_1{:}K_1.T_2$. Consider case 2d.

$$\begin{array}{l} plus_\Gamma((\Lambda X{:}K.S_1)\, T_2) \\ = (\Lambda X{:}K.plus_{\Gamma,\, X\leq\top^K:K}(S_1))\, plus_\Gamma(T_2) \\ \to_{\beta\wedge}\; plus_{\Gamma,\, X\leq\top^K:K}(S_1)[X{\leftarrow}plus_\Gamma(T_2)], \\ \qquad \text{by lemma 3.5.12,} \\ \twoheadrightarrow_{\beta\wedge+}\; plus_\Gamma(S_1[X{\leftarrow}T_2]). \end{array}$$

Other cases. The rest of the cases follows using a similar argument to the one used in the case $T \equiv \forall X{\leq}T_1{:}K_1.T_2$. □

LEMMA 3.5.14 Let $lub_\Gamma(S)$ be defined and $\Gamma \vdash S \in K$. Then $plus_\Gamma(S) \twoheadrightarrow_+{}^{>0} plus_\Gamma(lub_\Gamma(S))$.

PROOF: By induction on the structure of S. Since $lub_\Gamma(S)$ is defined, it is enough to consider the following two cases.

$S \equiv X$. Let $\Gamma \equiv \Gamma_1,\, X \leq T{:}K,\, \Gamma_2$.

$$\begin{array}{rcll} plus_\Gamma(X) & = & X + plus_{\Gamma_1}(T) & \\ & = & X + plus_\Gamma(T) & \text{by weakening (lemma 3.5.10),} \\ & \to_+ & plus_\Gamma(T) & \\ & = & plus_\Gamma(lub_\Gamma(X)) & \end{array}$$

$S \equiv A\,T$. By the induction hypothesis. □

Our measure to show the well-foundedness of $AlgF^\omega_\wedge$ considers the $\beta\wedge+$-reduction paths of the *plus* versions of the types in the subtyping judgements. As we mentioned before, in ALGS-TVAR and ALGS-OAPP the types appearing in the hypothesis may be larger than those in their conclusions. Therefore, the well foundedness of the $AlgF^\omega_\wedge$ relation is not immediate. The next corollary gathers the previous results to serve our purposes.

COROLLARY 3.5.15

1. If $\Gamma \vdash X \in K$, then $plus_\Gamma(X) \twoheadrightarrow_{\beta\wedge+}{}^{>0} plus_\Gamma(\Gamma(X))$.
2. If $\Gamma \vdash A\,T \in K$ then $plus_\Gamma(A\,T) \twoheadrightarrow_{\beta\wedge+}{}^{>0} plus_\Gamma(lub_\Gamma(A\,T)^{nf})$.

PROOF: Item 1 is a particular case of the previous lemma (lemma 3.5.14), and item 2 is a consequence of lemma 3.5.14 and the monotonicity of *plus* with respect to $\to_{\beta\wedge+}$ (3.5.13(2)). □

Finally, we can define our measure.

DEFINITION 3.5.16 (*Weight*)

1. $\mathit{weight}(\Gamma \vdash_{Alg} S \leq T) = <\text{max-red}(\mathit{plus}_{\Gamma}(S)) + \text{max-red}(\mathit{plus}_{\Gamma}(T), \mathit{size}_j(\Gamma \vdash S \leq T))>$,
2. $\mathit{weight}(\Gamma \vdash T \in K) = <0, 0>$,

where max-red(S) is the length of a maximal $\beta\wedge+$-reduction path starting from S, and size_j is defined in definition 2.4.8.

Pairs are ordered lexicographically. Note that $<0, 0>$ is the least *weight*.

PROPOSITION 3.5.17 (*Well-foundedness of $AlgF^{\omega}_{\wedge}$*) If $\dfrac{J_1 \,..\, J_n}{J}$ is an $AlgF^{\omega}_{\wedge}$ rule, then $\mathit{weight}(J_i) < \mathit{weight}(J)$, for each $i{\in}\{1..n\}$.

PROOF: By inspection of the rules of $AlgF^{\omega}_{\wedge}$. □
Finally, we can state our main result.

THEOREM 3.5.18 (*Decidability of subtyping in $F^{\omega}_{\wedge}$*)
For any context Γ and for any two types S and T, it is decidable whether $\Gamma \vdash S \leq T$.

3.6 Our decidability proof and full $F_{\leq}$

In the introduction to chapter 2 we mentioned that subtyping in $F_{\leq}$, a second-order λ-calculus with bounded quantification defined by Curien and Ghelli in 1989, is undecidable. A question that comes to mind is: if we try to apply our proof of the decidability of subtyping in $F^{\omega}_{\wedge}$ to $F_{\leq}$, where will it fail?

If we consider the algorithm for the subtyping relation in [Ghe90], the place where our proof does not go through is when we try to prove that the algorithm terminates by calculating the maximal length of the *plus* versions of the types in the rule for subtyping quantified types. Remember that the subtyping rule for quantified types in full $F_{\leq}$ is:

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma,\, X{\leq}T_1 \vdash S_2 \leq T_2}{\Gamma \vdash \forall X{\leq}S_1.\, S_2 \leq \forall X{\leq}T_1.\, T_2} \qquad (F_{\leq}\text{- S-ALL})$$

Consider now the following case.

$$\begin{array}{lcl}
\Gamma & \equiv & Y_4{\leq}\top^{\star},\; Y_3{\leq}Y_4,\; Y_2{\leq}Y_3,\; Y_1{\leq}Y_2, \\
T_1 & \equiv & Y_1, \\
S_1 & \equiv & \top^{\star}, \\
T_2 & \equiv & X{\rightarrow}X, \quad \text{and} \\
S_2 & \equiv & X{\rightarrow}X.
\end{array}$$

The *plus* versions of the types in the subtyping statements of this example are as follows.

$$\begin{aligned}
&\mathit{plus}_{\Gamma,\,X\leq Y_1}(S_2) \equiv (X + Y_1 + Y_2 + Y_3 + Y_4 + \top^\star)\to(X + Y_1 + Y_2 + Y_3 + Y_4 + \top^\star)\\
&\mathit{plus}_{\Gamma,\,X\leq Y_1}(T_2) \equiv (X + Y_1 + Y_2 + Y_3 + Y_4 + \top^\star)\to(X + Y_1 + Y_2 + Y_3 + Y_4 + \top^\star)\\
&\mathit{plus}_{\Gamma}(\forall X{\leq}S_1.\ S_2) \equiv \forall X{\leq}\top^\star.\ (X + \top^\star)\to(X + \top^\star)\\
&\mathit{plus}_{\Gamma}(\forall X{\leq}T_1.\ T_2) \equiv \forall X{\leq}Y_1 + Y_2 + Y_3 + Y_4 + \top^\star.\\
&\qquad\qquad\qquad\qquad (X + Y_1 + Y_2 + Y_3 + Y_4 + \top^\star)\to(X + Y_1 + Y_2 + Y_3 + Y_4 + \top^\star)
\end{aligned}$$

The length of a maximal +-reduction in each case is:

$$\begin{array}{lcl}
\text{max-red}(\mathit{plus}_{\Gamma,\,X\leq Y_1}(S_2)) & = & 10\\
\text{max-red}(\mathit{plus}_{\Gamma,\,X\leq Y_1}(T_2)) & = & 10\\
\text{max-red}(\mathit{plus}_{\Gamma}(\forall X{\leq}S_1.\ S_2)) & = & 2\\
\text{max-red}(\mathit{plus}_{\Gamma}(\forall X{\leq}T_1.\ T_2)) & = & 14.
\end{array}$$

The *weight* of the conclusion $\Gamma \vdash \forall X{\leq}S_1.\ S_2 \leq \forall X{\leq}T_1.\ T_2$, as defined in definition 3.5.16, is smaller than the *weight* of the hypothesis $\Gamma,\ X{\leq}T_1 \vdash S_2 \leq T_2$, because the maximal length of a +-reduction starting from the *plus* version of the conclusion is shorter than the maximal length of a +-reduction starting from the *plus* version of that hypothesis. To be more precise,

$$\begin{array}{c}
\text{max-red}(\mathit{plus}_{\Gamma}(\forall X{\leq}S_1.\ S_2)) + \text{max-red}(\mathit{plus}_{\Gamma}(\forall X{\leq}T_1.\ T_2))\\
<\\
\text{max-red}(\mathit{plus}_{\Gamma,\,X\leq Y_1}(S_2)) + \text{max-red}(\mathit{plus}_{\Gamma,\,X\leq Y_1}(T_2)).
\end{array}$$

Chapter 4

Typing in $F^{\omega}_{\wedge}$

4.1 Type checking and type inference

Given a context Γ, a term e, and a type T, type checking consists of analyzing whether the judgement $\Gamma \vdash e \in T$ is derivable from a given set of inference rules. Type checking algorithms for lambda calculi, unless they are formulated using Gentzen's sequent calculus style, involve *guessing* the type of subterms. For example, when e is $e_1\, e_2$, the type of e_2 is not necessarily a subexpression of T, and in order to corroborate or to refute the assertion $\Gamma \vdash e \in T$ we need to *infer* a type for e_2.

In this section, we present an algorithm for inferring minimal types in $F^{\omega}_{\wedge}$. Given Γ and e, the type S constructed by the algorithm is a subtype of every T such that $\Gamma \vdash e \in T$. In this way, we reduce the problem of whether $\Gamma \vdash e \in T$ to that of inferring a type S such that $\Gamma \vdash e \in S$ and $\Gamma \vdash S \leq T$. Solving this problem involves not only the typing rules but all the inference rules of $F^{\omega}_{\wedge}$: the rule T-Subsumption depends on a subtyping judgement, the rule T-Var depends on an ok judgement, and the ok judgements depend on kinding judgements. Consequently, type checking uses the full power of the $F^{\omega}_{\wedge}$ system.

As an example, consider type checking the following judgement:

$$\Gamma,\, X \leq T_1 {\to} T_2,\, f{:}X,\, a{:}T_1 \vdash f\, a \in T_2.$$

The application $f\,a$ can only be formed if f has an arrow type. Using T-Var we can assign type X to f, which means that in order to obtain an arrow type for f we have to *replace* X by its bound, which has the right form. Observe how this *replacement* is performed by T-Subsumption in the following derivation.

$$\dfrac{\dfrac{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash \text{ok}}{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash \boxed{f \in X}} \quad \dfrac{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash \text{ok}}{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash X \leq T_1{\to}T_2}}{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash \boxed{f \in T_1{\to}T_2}}\ \textsc{T-Sub}$$

$$\dfrac{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash f \in T_1{\to}T_2 \quad \dfrac{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash \text{ok}}{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash a \in T_1}}{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}T_1 \vdash f\, a \in T_2}\ \textsc{T-App}$$

Note that, in the presence of T-Subsumption, we may actually perform the application when the type of a is a subtype of T_1. Namely, if

$$\frac{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, \boxed{a{:}U_1} \vdash a \in U_1 \qquad \Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}U_1 \vdash U_1 \leq T_1}{\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}U_1 \vdash \boxed{a \in T_1}}\ \textsc{T-Sub}$$

Moreover, we may want to check whether

$$\Gamma,\, X \leq T_1{\to}T_2,\, f{:}X,\, a{:}U_1 \vdash f\, a \in U_2,$$

where T_2 is a subtype of U_2.

The situation gets more complicated if f has an intersection type. Suppose that

$$\Gamma,\, X \leq T_1{\to}T_2,\, Y \leq S_1{\to}S_2,\, f{:}X \wedge Y \wedge \forall Z{\leq}V_1{:}K.V_2,\, a{:}U_1 \vdash f\, a \in U_2,$$

where U_1 is a subtype of T_1 and S_1. An algorithm should not consider the type $\forall Z{\leq}V_1{:}K.V_2$ for f since, in this case, f is applied to a term and not to a type. Then it has to replace X and Y by their bounds, $T_1{\to}T_2$ and $S_1{\to}S_2$. Moreover, given that the type of a, U_1, is a subtype of both S_1 and T_1, it should check whether $S_2 \wedge T_2$ is a subtype of U_2.

Another source of problems in the search for an algorithmic presentation of the typing rules is that types may not be in normal form. Consider the judgement

$$\Gamma,\, X \leq T_1{\to}T_2,\, Z \leq \Lambda Y{:}\star.Y,\, f{:}Z\,X,\, a{:}T_1 \vdash f\, a \in T_2, \tag{4.1}$$

In order to type the application, f should be assigned type $T_1{\to}T_2$. To do that, Z should be replaced by its bound in $Z\,X$. This replacement produces a type which is not in normal form, so $\Lambda Y{:}\star.Y\,X$ has to be normalised to obtain X. Finally, X is replaced by its bound and then the application can be typed.

The main new source of difficulty is the interaction between the need for normalization and the presence of intersection types.

An algorithm to infer types should proceed structurally on the form of the term whose type is to be inferred. This requires us to remove the rules which make our typing rules non-deterministic: we should eliminate T-Subsumption and T-Meet from the original presentation, and modify the other rules in such a way that we can still type the same set of terms.

We give some preliminary definitions and results before presenting the rules of our new system:

- We define the mapping *flub*, which performs the "replacements" which we motivated with the previous examples.

- We define the function *arrows*, to filter arrow types in order to deal with term application.

- We define the function *alls* to filter polymorphic types to deal with type application.

The function *lub* (definition 3.1.3) is a partial function which is only defined for type variables and type applications. Here, we extend the definition of *lub* to intersection types in such a way that it is defined if the least upper bound is defined for at least one of the types in the intersection.

DEFINITION 4.1.1 (*Homomorphic extension of lub to intersections, lub**)

$$\begin{array}{rcll} lub^*_\Gamma(X) & = & \Gamma(X), & \\ lub^*_\Gamma(S\,T) & = & lub^*_\Gamma(S)\,T, & \\ lub^*_\Gamma(\bigwedge^K[T_1..T_n]) & = & \bigwedge^K[T'_1..T'_n], & \text{if } \exists i{\in}\{1..n\} \text{ such that } lub^*_\Gamma(T_i){\downarrow}, \end{array}$$

where T'_i is $lub^*_\Gamma(T_i)$, if $lub^*_\Gamma(T_i){\downarrow}$, and T_i otherwise.

LEMMA 4.1.2 If $lub^*_\Gamma(T)$ is defined, then $\Gamma \vdash T \leq lub^*_\Gamma(T)$.

PROOF: By induction on the complexity of T, using corollary 2.4.15. □

LEMMA 4.1.3 Let $lub^*_\Gamma(T)$ be defined and $\Gamma \vdash T \in K$. Then $plus_\Gamma(T) \twoheadrightarrow_{\beta\wedge+}{}^{>0}$ $plus_\Gamma(lub^*_\Gamma(T))$.

PROOF: The proof follows by induction on the structure of T. If $T \equiv X$ or $T \equiv S\,T$, then the argument is the same as in lemma 3.5.14. The case remaining to be checked is when $T \equiv \bigwedge^K[T_1..T_n]$. Then

$$\begin{array}{rcl} plus_\Gamma(\bigwedge^K[T_1..T_n]) & = & \bigwedge^K[plus_\Gamma(T_1)..plus_\Gamma(T_n)] \\ plus_\Gamma(lub^*_\Gamma(\bigwedge^K[T_1..T_n])) & = & \bigwedge^K[plus_\Gamma(T'_1)..plus_\Gamma(T'_n)], \end{array}$$

where $T'_1 \equiv T_i$ or $T'_i = lub^*_\Gamma(T_i)$. Since $lub^*_\Gamma(T)$ is defined, there exists $j{\in}\{1..n\}$ such that $lub^*_\Gamma(T_j)$ is defined. Now, for every k such that $lub^*_\Gamma(T_k)$ is defined, by the induction hypothesis, we have that

$$plus_\Gamma(T_k) \twoheadrightarrow_{\beta\wedge+}{}^{>0}\ plus_\Gamma(lub^*_\Gamma(T_k)).$$

Hence,

$$plus_\Gamma(\bigwedge^K[T_1..T_n]) \twoheadrightarrow_{\beta\wedge+}{}^{>0}\ plus_\Gamma(lub^*_\Gamma(\bigwedge^K[T_1..T_n])).$$ □

We define the mapping *flub* which given a type T (and a context Γ) finds the smallest type larger than T (with respect to the subtype relation) having structural information to perform an application.

DEFINITION 4.1.4 (*Functional Least Upper Bound*) The functional least upper bound of a type T, in a context Γ, $flub_\Gamma(T)$ is defined as follows.

$$flub_\Gamma(T) = \begin{cases} flub_\Gamma(lub^*_\Gamma(T^{nf})), & \text{if } lub^*_\Gamma(T^{nf}){\downarrow}; \\ T^{nf}, & \text{otherwise.}^1 \end{cases}$$

The intuition behind the definition of the function *flub* is to find $T_1{\rightarrow}T_2$ starting form ZX in the example 4.1 above. In other words, $\mathit{flub}_\Gamma(Z\,X) = T_1{\rightarrow}T_2$. For simplicity we assume $T_1{\rightarrow}T_2$ in normal form. Step by step,

$$\begin{array}{rcl}
\mathit{flub}_\Gamma(Z\,X) & = & \mathit{flub}_\Gamma(\mathit{lub}^*_\Gamma(Z\,X)) \\
 & = & \mathit{flub}_\Gamma((\Lambda Y{:}K.Y)\,X) \\
 & = & \mathit{flub}_\Gamma(\mathit{lub}^*_\Gamma(((\Lambda Y{:}K.Y)\,X)^{\mathit{nf}})) \\
 & = & \mathit{flub}_\Gamma(\mathit{lub}^*_\Gamma(X)) \\
 & = & \mathit{flub}_\Gamma(T_1{\rightarrow}T_2) \\
 & = & T_1{\rightarrow}T_2.
\end{array}$$

More generally, *flub* climbs the subtyping hierarchy until it finds an arrow, a quantifier, or an intersection of these two. To show that *flub* is well-defined we use a similar argument to that used in section 3.5 to show that the relation defined by $\mathit{AlgF}^\omega_\wedge$ is well-founded. We show in lemma 4.1.5 that a maximal $\beta\wedge$+-reduction path of the *plus* version of the argument of *flub* is strictly longer than a maximal $\beta\wedge$+-reduction path of the *plus* version of the argument of its recursive call.

LEMMA 4.1.5 (*Well-foundedness of flub*)
If $\Gamma \vdash T \in K$, then $\mathit{flub}_\Gamma(T)$ is defined.

PROOF: If $\mathit{lub}^*_\Gamma(T^{\mathit{nf}})$ is undefined, *flub* terminates because $\rightarrow_{\beta\wedge}$ is strongly normalizing on well kinded types. Otherwise, define

$\mathit{weight}(\mathit{flub}_\Gamma(T)) = \mathit{max\text{-}red}(\mathit{plus}_\Gamma(T))$,

where $\mathit{max\text{-}red}(S)$ is the length of a maximal $\beta\wedge$+-reduction path starting from S. Lemma 3.5.8 and the strong normalization property of $\rightarrow_{\beta\wedge+}$ imply that *weight* is well defined and always positive on well kinded types. Since $\mathit{lub}^*_\Gamma(T^{\mathit{nf}})$ is defined,

$$\begin{array}{rcll}
\mathit{plus}_\Gamma(T) & \twoheadrightarrow_{\beta\wedge+} & \mathit{plus}_\Gamma(T^{\mathit{nf}}), & \text{by lemma 3.5.13(2),} \\
 & \twoheadrightarrow_{\beta\wedge+}{}^{>0} & \mathit{plus}_\Gamma(\mathit{lub}^*_\Gamma(T^{\mathit{nf}})), & \text{by lemma 4.1.3.}
\end{array}$$

Then the *weight* of the arguments of *flub* reduces in each recursive call, which proves that *flub* is well-founded. □

LEMMA 4.1.6 Let $\Gamma \vdash S, T \in \star$ and $S =_{\beta\wedge} T$. Then $\mathit{flub}_\Gamma(S) \equiv \mathit{flub}_\Gamma(T)$.

DEFINITION 4.1.7 (*arrows and alls*)

1. $$\begin{array}{rcl}
\mathit{arrows}\,(T_1{\rightarrow}T_2) & = & \{T_1{\rightarrow}T_2\}, \\
\mathit{arrows}\,(\bigwedge^\star[T_1..T_n]) & = & \cup_{i\in\{1..n\}}\,\mathit{arrows}\,(T_i), \\
\mathit{arrows}\,(T) & = & \emptyset, \quad \text{if } T \not\equiv T_1{\rightarrow}T_2 \text{ and } T \not\equiv \bigwedge^\star[T_1..T_n].
\end{array}$$

[1] This step can be optimised in an implementation of the type checking algorithm, allowing us to avoid the normalization of T when T is either an arrow type or a quantified type.

2. $$\begin{array}{lcl} \mathit{alls}\,(\forall X{\leq}T_1{:}K.T_2) & = & \{\forall X{\leq}T_1{:}K.T_2\}, \\ \mathit{alls}\,(\bigwedge^{\star}[T_1..T_n]) & = & \cup_{i\in\{1..n\}}\, \mathit{alls}\,(T_i), \\ \mathit{alls}\,(T) & = & \emptyset, \quad \text{if } T \not\equiv \forall X{\leq}T_1{:}K.T_2 \text{ and } T \not\equiv \bigwedge^{\star}[T_1..T_n]. \end{array}$$

The situation here is significantly more complex than in [Pie91] for $F_\wedge$, an extension of the second order λ-calculus. There it is enough to recursively search for arrows or polymorphic types in the context, because in $F_\wedge$ there is no reduction on types. The information to be searched for is explicit in the context, so the job done here by *flub* is simply an extra case in the definition of *arrows* and *alls*. Namely,

$$\begin{array}{rcl} \mathit{arrows}\,(X) & = & \mathit{arrows}\,(\Gamma(X)) \qquad \text{and} \\ \mathit{alls}\,(X) & = & \mathit{alls}\,(\Gamma(X)). \end{array}$$

Moreover, to prove that *flub* is well-founded is similar for us in complexity to proving termination of subtype checking. The similarity comes from the fact that computing *flub* involves replacing variables by their bounds in a given context and normalizing with respect to $\rightarrow_{\beta\wedge}$, as in lemma 4.1.5. In contrast, in [Pie91] it is enough to observe that well-formed contexts cannot contain cycles of variable references.

NOTATION 4.1.8 We introduce a new notation for intersection types. We write $\bigwedge^K[T \mid \phi(T)]$, meaning the intersection of all types T such that $\phi(T)$ holds. Note that this is an alternative notation to $\bigwedge^K[T_1..T_n]$ such that $\phi(T_i)$ holds if and only if $i{\in}\{1..n\}$.

We can now define a type inference algorithm for $F^\omega_\wedge$.

DEFINITION 4.1.9 (*A type inference algorithm,* inf)

$$\frac{\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash \text{ok}}{\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash_{\mathit{inf}} x \in T} \qquad \text{(AT-VAR)}$$

$$\frac{\Gamma,\, x{:}T_1 \vdash_{\mathit{inf}} e \in T_2}{\Gamma \vdash_{\mathit{inf}} \lambda x{:}T_1.e \in T_1{\rightarrow}T_2} \qquad \text{(AT-ABS)}$$

$$\frac{\Gamma \vdash_{\mathit{inf}} f \in T \qquad \Gamma \vdash_{\mathit{inf}} a \in S}{\Gamma \vdash_{\mathit{inf}} f\ a \in \bigwedge^{\star}[T_i \mid S_i{\rightarrow}T_i \in \mathit{arrows}\,(\mathit{flub}_\Gamma(T)) \text{ and } \Gamma \vdash S \leq S_i]} \qquad \text{(AT-APP)}$$

$$\frac{\Gamma,\, X{\leq}T_1{:}K_1 \vdash_{\mathit{inf}} e \in T_2}{\Gamma \vdash_{\mathit{inf}} \lambda X{\leq}T_1{:}K_1.e \in \forall X{\leq}T_1{:}K_1.T_2} \qquad \text{(AT-TABS)}$$

$$\frac{\Gamma \vdash_{\mathit{inf}} f \in T}{\Gamma \vdash_{\mathit{inf}} f\ S \in \bigwedge^{\star}[T_i[X{\leftarrow}S] \mid \forall X{\leq}S_i{:}K.T_i \in \mathit{alls}\,(\mathit{flub}_\Gamma(T)) \text{ and } \Gamma \vdash S \leq S_i]}$$

(AT-TAPP)

$$\frac{\text{for all } i{\in}\{1..n\} \quad \Gamma \vdash_{\mathit{inf}} e[X{\leftarrow}S_i] \in T_i}{\Gamma \vdash_{\mathit{inf}} \text{for}(X{\in}S_1..S_n)e \in \bigwedge^{\star}[T_1..T_n]} \qquad \text{(AT-FOR)}$$

The algorithmic information of rule AT-App is that in order to find a type for $f\,a$ in Γ, we need to infer a type S for a and a type T for f, and to take the intersection of all the $T_i's$ such that $T_i{\to}S_i \in \mathit{arrows}(\mathit{flub}_\Gamma(T))$ and $\Gamma \vdash S \leq S_i$.

4.2 Minimal typing

In this section we show that $F_\wedge^\omega$ satisfies the minimal typing property (theorem 4.2.11). We first prove that the algorithm *inf* is sound with respect to $F_\wedge^\omega$: if $\Gamma \vdash_{\mathit{inf}} e \in T$, then $\Gamma \vdash e \in T$ (proposition 4.2.4). We then prove that every closed term is typeable using either set of typing rules (lemma 4.2.8). Finally, we prove that *inf* computes minimal types for $F_\wedge^\omega$ terms (proposition 4.2.10).

Lemma 4.2.1 Let $\Gamma \vdash T \in \star$. Then $\Gamma \vdash T \leq \mathit{flub}_\Gamma(T)$.

Proof: Since *flub* is well-founded, we can proceed by induction on the number of unfolding steps in $\mathit{flub}_\Gamma(T)$. If $\mathit{flub}_\Gamma(T) = T^{\mathit{nf}}$, the result follows by S-Conv. Otherwise, $\mathit{flub}_\Gamma(T) = \mathit{flub}_\Gamma(\mathit{lub}_\Gamma^*(T^{\mathit{nf}}))$. By S-Conv,

$$\Gamma \vdash T \leq T^{\mathit{nf}}.$$

By lemma 4.1.2,

$$\Gamma \vdash T^{\mathit{nf}} \leq \mathit{lub}_\Gamma^*(T^{\mathit{nf}}).$$

By the induction hypothesis,

$$\Gamma \vdash \mathit{lub}_\Gamma^*(T^{\mathit{nf}}) \leq \mathit{flub}_\Gamma(\mathit{lub}_\Gamma^*(T^{\mathit{nf}})).$$

Finally, by S-Trans, the result follows. □

Lemma 4.2.2 Let $\Gamma \vdash T \in \star$. Then

1. $\Gamma \vdash T \leq \bigwedge^\star[S \mid S \in \mathit{arrows}(\mathit{flub}_\Gamma(T))]$.
2. $\Gamma \vdash T \leq \bigwedge^\star[S \mid S \in \mathit{alls}(\mathit{flub}_\Gamma(T))]$.

Proof:

1. Using lemma 4.2.1, we reduce our problem to proving that
$$\Gamma \vdash T \leq \bigwedge^\star[S \mid S \in \mathit{arrows}(T)],$$
which follows by induction on the structure of T.

2. Similar to 1. □

Lemma 4.2.3

1. If $\Gamma \vdash T \leq T_1{\to}T_2$, then $\Gamma \vdash \bigwedge^\star[S \mid S{\in}\mathit{arrows}(\mathit{flub}_\Gamma(T))] \leq T_1{\to}T_2$.
2. If $\Gamma \vdash T \leq \forall X{\leq}T_1{:}K.T_2$, then $\Gamma \vdash \bigwedge^\star[S \mid S{\in}\mathit{alls}(\mathit{flub}_\Gamma(T))] \leq \forall X{\leq}T_1{:}K.T_2$.

PROOF:

1. By induction on the derivation of $\Gamma \vdash T \leq T_1{\to}T_2$. The last rule of a derivation of this subtyping statement can only be S-CONV, S-TVAR, S-TRANS, or S-MEET-LB. The first three cases use similar arguments, therefore we consider here only the cases for S-CONV and S-MEET-LB.

   S-CONV We are given that $T =_{\beta\wedge} T_1{\to}T_2$. By lemma 4.1.6 and the definition of *flub*, we have that

   $$\begin{array}{rcl} \mathit{arrows}\,(\mathit{flub}_\Gamma(T)) & = & \mathit{arrows}\,(\mathit{flub}_\Gamma(T_1{\to}T_2)), \\ & = & \mathit{arrows}\,((T_1{\to}T_2)^{\mathit{nf}}) \end{array}$$

   We now have two cases to analyze.

   (a) If $(T_1{\to}T_2)^{\mathit{nf}} = T_1^{\mathit{nf}}{\to}T_2^{\mathit{nf}}$, then the result follows by S-MEET-LB and S-CONV.

   (b) Otherwise, let $(T_1{\to}T_2)^{\mathit{nf}} = \bigwedge^\star[T_1^{\mathit{nf}}{\to}U_1..T_1^{\mathit{nf}}{\to}U_n]$, where $T_2^{\mathit{nf}} = \bigwedge^\star[U_1..U_n]$. Then,

   $$\mathit{arrows}\,(\mathit{flub}_\Gamma(T)) = \{T_1^{\mathit{nf}}{\to}U_1..T_1^{\mathit{nf}}{\to}U_n\}.$$

   Consequently,

   $$\bigwedge{}^\star[S \mid S \in \mathit{arrows}\,(\mathit{flub}_\Gamma(T))] = (T_1{\to}T_2)^{\mathit{nf}},$$

   and the result follows by S-CONV.

   S-MEET-LB We are given that

   $$\Gamma \vdash \bigwedge{}^\star[S_1..T_1{\to}T_2..S_n] \leq T_1{\to}T_2.$$

   By the definition of *flub*,

   $$\begin{array}{l} \mathit{flub}_\Gamma(\bigwedge^\star[S_1..T_1{\to}T_2..S_n]) \\ = \bigwedge^\star[....T_1^{\mathit{nf}}{\to}A_1..T_1^{\mathit{nf}}{\to}A_m....], \end{array}$$

   where $T_2^{\mathit{nf}} = \bigwedge^\star[A_1..A_m]$ or
   $T_2^{\mathit{nf}} = A_1$

   Now,

   $$\begin{array}{l} \mathit{arrows}\,(\mathit{flub}_\Gamma(\bigwedge^\star[S_1..T_1{\to}T_2..S_n])) \\ \supseteq \{T_1^{\mathit{nf}}{\to}A_1..T_1^{\mathit{nf}}{\to}A_m\} \end{array}$$

   Then, if $T_2^{\mathit{nf}} = \bigwedge^\star[A_1..A_m]$, by lemma 2.4.18; and, if $T_2^{\mathit{nf}} = A_1$, by S-MEET-LB, we have that

   $$\Gamma \vdash \bigwedge{}^\star[S \mid S \in \mathit{arrows}\,(\mathit{flub}_\Gamma(\bigwedge{}^\star[S_1..T_1{\to}T_2..S_n]))] \leq (T_1{\to}T_2)^{\mathit{nf}}.$$

   Finally, the result follows by S-CONV.

2. Similar to previous item. □

PROPOSITION 4.2.4 (*Soundness of inf*) If $\Gamma \vdash_{\mathit{inf}} e \in T$, then $\Gamma \vdash e \in T$.

Proof: By induction on the derivation of $\Gamma \vdash_{\mathit{inf}} e \in T$. The interesting cases are when the last applied rule is either AT-App and AT-TApp.

AT-App We are given that

$$\Gamma \vdash_{\mathit{inf}} f\,a \in \bigwedge^{\star}[T_i \,|\, S_i{\to}T_i \in \mathit{arrows}\,(\mathit{flub}_{\Gamma}(T)) \text{ and } \Gamma \vdash S \leq S_i]$$

is derived from $\Gamma \vdash_{\mathit{inf}} f \in T$ and $\Gamma \vdash_{\mathit{inf}} a \in S$.

If $\bigwedge^{\star}[T_i \,|\, S_i{\to}T_i \in \mathit{arrows}\,(\mathit{flub}_{\Gamma}(T)) \text{ and } \Gamma \vdash S \leq S_i] =_{\beta\wedge} \top^{\star}$, then the result follows immediately using T-Meet. Otherwise, by the induction hypothesis, we have that $\Gamma \vdash f \in T$. By lemma 4.2.2(1), S-Meet-LB, S-Trans, and T-Subsumption, $\Gamma \vdash f \in S_i{\to}T_i$. By the induction hypothesis and T-Subsumption, $\Gamma \vdash a \in S_i$. By T-App, $\Gamma \vdash f\,a \in T_i$. Finally, by T-Meet,

$$\Gamma \vdash f\,a \in \bigwedge^{\star}[T_i \,|\, S_i{\to}T_i \in \mathit{arrows}\,(\mathit{flub}_{\Gamma}(T)) \text{ and } \Gamma \vdash S \leq S_i].$$

AT-TApp We are given that

$$\begin{array}{l}\Gamma \vdash_{\mathit{inf}} f\,S \in \\ \bigwedge^{\star}[T_i[X{\leftarrow}S] \,|\, \forall X{\leq}S_i{:}K.T_i \in \mathit{alls}\,(\mathit{flub}_{\Gamma}(T)) \text{ and } \Gamma \vdash S \leq S_i]\end{array}$$

is derived from $\Gamma \vdash_{\mathit{inf}} f \in T$.

If $\bigwedge^{\star}[T_i \,|\, S_i{\to}T_i \in \mathit{arrows}\,(\mathit{flub}_{\Gamma}(T)) \text{ and } \Gamma \vdash S \leq S_i] =_{\beta\wedge} \top^{\star}$, then the result follows immediately, using T-Meet. Otherwise, assume

$$\mathit{alls}\,(\mathit{flub}_{\Gamma}(T)) \equiv \bigwedge^{\star}[\forall X{\leq}S_1{:}K.T_1..\forall X{\leq}S_n{:}K.T_n].$$

By the induction hypothesis, we have that, $\Gamma \vdash f \in T$. By lemma 4.2.2(2), S-Meet-LB, S-Trans, and T-Subsumption, it follows that $\Gamma \vdash f \in \forall X{\leq}S_i{:}K.T_i$. Since $\Gamma \vdash S \leq S_i$, by T-App, $\Gamma \vdash f\,S \in T_i[X{\leftarrow}S]$. Finally, by T-Meet,

$$\begin{array}{ll}\Gamma \vdash & f\,S \in \\ & \bigwedge^{\star}[T_i[X{\leftarrow}S] \,|\, \forall X{\leq}S_i{:}K.T_i \in \mathit{alls}\,(\mathit{flub}_{\Gamma}(T)) \text{ and } \Gamma \vdash S \leq S_i]. \quad \square\end{array}$$

Lemma 4.2.5 (*Term application*)
If $\Gamma \vdash \bigwedge^{\star}[S_1{\to}T_1..S_n{\to}T_n] \leq S{\to}T$ and $\Gamma \vdash U \leq S$,
then $\Gamma \vdash \bigwedge^{\star}[T_j \,|\, \Gamma \vdash U \leq S_j] \leq T$.

Proof: There are two cases to be considered according to the normal form of $S{\to}T$. The case when $(S{\to}T)^{\mathit{nf}} \equiv S^{\mathit{nf}}{\to}T^{\mathit{nf}}$ is similar to but simpler than the one we consider here. Assume

$$(S{\to}T)^{\mathit{nf}} \equiv \bigwedge^{\star}[S^{\mathit{nf}}{\to}A_1..S^{\mathit{nf}}{\to}A_m], \text{ where } T^{\mathit{nf}} \equiv \bigwedge^{\star}[A_1..A_m].$$

By the equivalence of ordinary and normal subtyping (theorem 3.3.9),

$$\Gamma^{\mathit{nf}} \vdash_n \bigwedge^{\star}[S_1^{\mathit{nf}}{\to}B_1^1..S_1^{\mathit{nf}}{\to}B_1^{k_1}..S_n^{\mathit{nf}}{\to}B_n^1..S_n^{\mathit{nf}}{\to}B_n^{k_n}] \leq \bigwedge^{\star}[S^{\mathit{nf}}{\to}A_1..S^{\mathit{nf}}{\to}A_m],$$

$$\text{where } T_i^{\mathit{nf}} = \begin{cases} B_i^1, & \text{if it is not an intersection;} \\ \bigwedge^{\star}[B_i^1..B_i^{k_i}], & \text{otherwise.} \end{cases}$$

By generation for normal subtyping (proposition 3.2.10), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ and $l_j{\in}\{i..k_j\}$ such that

$$\Gamma^{nf} \vdash_n S_j^{nf}{\to}B_j^{l_j} \leq S^{nf}{\to}A_i$$

Again, by generation for normal subtyping (proposition 3.2.10), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ and $l_j{\in}\{i..k_j\}$ such that

$$\begin{array}{l} \Gamma^{nf} \vdash_n S^{nf} \leq S_j^{nf} \quad \text{and} \\ \Gamma^{nf} \vdash_n B_j^{l_j} \leq A_i \end{array}$$

By NS-Trans and the equivalence of ordinary and normal subtyping (theorem 3.3.9), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ and $l_j{\in}\{i..k_j\}$ such that

$$\begin{array}{l} \Gamma \vdash U \leq S_j \quad \text{and} \\ \Gamma^{nf} \vdash_n B_j^{l_j} \leq A_i. \end{array}$$

By NS-$\exists$, for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ such that

$$\begin{array}{l} \Gamma \vdash U \leq S_j \quad \text{and} \\ \Gamma^{nf} \vdash_n \bigwedge^{\star}[B_j^1..B_j^{k_j}] \leq A_i. \end{array}$$

By the equivalence of ordinary and normal subtyping (theorem 3.3.9), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ such that

$$\begin{array}{l} \Gamma \vdash U \leq S_j \quad \text{and} \\ \Gamma \vdash T_j \leq A_i. \end{array}$$

By lemma 2.4.14, S-Conv, and S-Trans,

$$\Gamma \vdash \bigwedge^{\star}[T_j \,|\, \Gamma \vdash U \leq S_j] \leq T. \qquad \square$$

Lemma 4.2.6 (*Substitution for subtyping*)
If $\Gamma_1 \vdash S_1 \leq T_1$ and $\Gamma_1,\, X{\leq}T_1{:}K_1,\, \Gamma_2 \vdash S_2 \leq T_2$, then $\Gamma_1,\, \Gamma_2[X{\leftarrow}S_1] \vdash S_2[X{\leftarrow}S_1] \leq T_2[X{\leftarrow}S_1]$.

Proof: By straightforward induction on the derivation of $\Gamma_1,\, X{\leq}T_1{:}K_1,\, \Gamma_2 \vdash S_2 \leq T_2$, using the weakening lemma (lemma 2.4.4), the type substitution lemma (lemma 2.4.11), and lemma 2.3.1.4(3). □

Lemma 4.2.7 (*Type application*)
If $\Gamma \vdash \bigwedge^{\star}[\forall X{\leq}S_1{:}K_1.T_1..\forall X{\leq}S_n{:}K_n.T_n] \leq \forall X{\leq}S{:}K.T$ and $\Gamma \vdash U \leq S$, then $\Gamma \vdash \bigwedge^{\star}[T_j[X{\leftarrow}U] \,|\, \Gamma \vdash U \leq S_j] \leq T[X{\leftarrow}U]$.

Proof: There are two cases to be considered according to the normal form of $\forall X{\leq}S{:}K.T$. The case when $(\forall X{\leq}S{:}K.T)^{nf} \equiv \forall X{\leq}S^{nf}{:}K.T^{nf}$ is similar to but simpler than the one we consider here. Assume

$$(\forall X{\leq}S{:}K.T)^{nf} \equiv \bigwedge^{\star}[\forall X{\leq}S^{nf}{:}K.A_1..\forall X{\leq}S^{nf}{:}K.A_m]$$

where $T^{nf} \equiv \bigwedge^{\star}[A_1..A_m]$. By the equivalence of ordinary and normal subtyping (theorem 3.3.9),

$$\Gamma^{nf} \vdash_n \bigwedge^{\star}[\forall X{\leq}S_1^{nf}{:}K_1.B_1^1..\forall X{\leq}S_1^{nf}{:}K_1.B_1^{k_1}..\forall X{\leq}S_n^{nf}{:}K_n.B_n^1..\forall X{\leq}S_n^{nf}{:}K_n.B_n^{k_n}]$$
$$\leq$$
$$\bigwedge^{\star}[\forall X{\leq}S^{nf}{:}K.A_1..\forall X{\leq}S^{nf}{:}K.A_m],$$

where $T_i^{nf} = \begin{cases} B_i^1, & \text{if it is not an intersection;} \\ \bigwedge^{\star}[B_i^1..B_i^{k_i}], & \text{otherwise.} \end{cases}$

By generation for normal subtyping (proposition 3.2.10), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ and $l_j{\in}\{i..k_j\}$ such that

$$\Gamma^{nf} \vdash_n \forall X{\leq}S_j^{nf}{:}K_j.B_j^{l_j} \leq \forall X{\leq}S^{nf}{:}K.A_i$$

Again, by generation for normal subtyping (proposition 3.2.10), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ and $l_j{\in}\{i..k_j\}$ such that

$$\begin{array}{l} K \equiv K_j, \\ S^{nf} \equiv S_j^{nf}, \quad \text{and} \\ \Gamma^{nf},\, X{\leq}S_j^{nf}{:}K \vdash_n B_j^{l_j} \leq A_i. \end{array}$$

By NS-∀∃, for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ such that

$$\Gamma^{nf},\, X{\leq}S_j^{nf}{:}K \vdash_n T_j^{nf} \leq A_i.$$

By the equivalence of ordinary and normal subtyping (theorem 3.3.9), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ such that

$$\Gamma,\, X{\leq}S_j{:}K \vdash T_j \leq A_i.$$

Furthermore, by S-Conv and S-Trans,

$$\Gamma \vdash U \leq S_j.$$

Then, by the substitution lemma for subtyping (lemma 4.2.6), for each $i{\in}\{1..m\}$ there exist $j{\in}\{1..n\}$ such that

$$\Gamma \vdash T_j[X{\leftarrow}U] \leq A_i[X{\leftarrow}U].$$

By NS-∀∃,

$$\Gamma \vdash \bigwedge^{\star}[T_j[X{\leftarrow}U] \,|\, \Gamma \vdash U \leq S_j] \leq \bigwedge^{\star}[A_1[X{\leftarrow}U]..A_m[X{\leftarrow}U]].$$

By the definition of substitution,

$$\Gamma \vdash \bigwedge^{\star}[T_j[X{\leftarrow}U] \,|\, \Gamma \vdash U \leq S_j] \leq T^{nf}[X{\leftarrow}U].$$

Finally, by lemma 2.3.1.4(3), S-Conv, and S-Trans,

$$\Gamma \vdash \bigwedge^{\star}[T_j[X{\leftarrow}U] \,|\, \Gamma \vdash U \leq S_j] \leq T[X{\leftarrow}U], \qquad \square$$

Usually, the next step to prove the accuracy of an algorithm, *inf* in our case, would be to prove a completeness result: if the term e has type T with respect to the context Γ in the the typing system $F^{\omega}_{\wedge}$ then the algorithm *inf* finds a type T' for e in Γ. In the present situation this result is not strong enough, since every closed term is typeable in both systems. One easily proves that

Lemma 4.2.8

1. If e is closed in Γ, then there exists T such that $\Gamma \vdash e \in T$.

2. If e is closed in Γ, then there exists T such that $\Gamma \vdash_{\mathit{inf}} e \in T$.

We use the fact that *inf* is deterministic, which means that the rules are invertible, to prove the important property that it finds a minimal type.

Proposition 4.2.9 (*Generation for inf*)

1. If $\Gamma \vdash_{\mathit{inf}} x \in T$, then $T \equiv \Gamma(x)$.

2. If $\Gamma \vdash_{\mathit{inf}} \lambda x{:}T_1.e \in T$, then $T \equiv T_1{\to}T_2$, where $\Gamma,\ x{:}T_1 \vdash_{\mathit{inf}} e \in T_2$ as a subderivation.

3. If $\Gamma \vdash_{\mathit{inf}} f\ a \in T$, then
   $$T \equiv \bigwedge^{\star}[T_i \,|\, S_i{\to}T_i \in \mathit{arrows}(\mathit{flub}_{\Gamma}(U)) \text{ and } \Gamma \vdash S \leq S_i],$$
   where $\Gamma \vdash_{\mathit{inf}} f \in U$ and $\Gamma \vdash_{\mathit{inf}} a \in S$ as subderivations.

4. If $\Gamma \vdash_{\mathit{inf}} \lambda X{\leq}T_1{:}K_1.e \in T$, then
   $$T \equiv \forall X{\leq}T_1{:}K_1.T_2,$$
   where $\Gamma,\ X{\leq}T_1{:}K_1 \vdash_{\mathit{inf}} e \in T_2$ as a subderivation.

5. If $\Gamma \vdash_{\mathit{inf}} f\ S \in T$, then
   $$T \equiv \bigwedge^{\star}[T_i[X{\leftarrow}S] \,|\, \forall X{\leq}S_i{:}K.T_i \in \mathit{alls}(\mathit{flub}_{\Gamma}(U)) \text{ and } \Gamma \vdash S \leq S_i],$$
   where $\Gamma \vdash_{\mathit{inf}} f \in U$ as a subderivation.

6. If $\Gamma \vdash_{\mathit{inf}} \mathrm{for}(X{\in}S_1..S_n)e \in T$, then $T \equiv \bigwedge^{\star}[T_1..T_n]$, where $\Gamma \vdash_{\mathit{inf}} e[X{\leftarrow}S_i] \in T_i$, for each $i{\in}\{1..n\}$, as subderivations.

Proof: The form of the term in the antecedent uniquely determines the last rule of its derivation.

Proposition 4.2.10 (*Minimal typing*)
If $\Gamma \vdash e \in T$ and $\Gamma \vdash_{\mathit{inf}} e \in T'$, then $\Gamma \vdash T' \leq T$.

PROOF: By induction on the derivation of $\Gamma \vdash e \in T$.

T-VAR By generation for *inf* (proposition 4.2.9), $T' \equiv T$, and then the result follows by S-CONV.

T-ABS We are given that

$$\begin{aligned}&e \equiv \lambda x{:}T_1.e_2,\\ &T \equiv T_1{\to}T_2, \quad \text{and}\\ &\Gamma,\, x{:}T_1 \vdash e_2 \in T_2.\end{aligned}$$

By generation for *inf* (proposition 4.2.9),

$$\begin{aligned}&T' \equiv T_1{\to}T_2' \quad \text{and}\\ &\Gamma,\, x{:}T_1 \vdash_{\mathit{inf}} e_2 \in T_2'.\end{aligned}$$

By the induction hypothesis, $\Gamma,\, x{:}T_1 \vdash T_2' \leq T_2$, and by strengthening (lemma 2.4.5), $\Gamma \vdash T_2' \leq T_2$, from which it follows that $\Gamma \vdash T' \leq T$.

T-APP We are given that

$$\begin{aligned}&e \equiv f\ a,\\ &\Gamma \vdash f \in V{\to}T, \quad \text{and}\\ &\Gamma \vdash a \in V.\end{aligned}$$

By generation for *inf* (proposition 4.2.9),

$$\begin{aligned}&\Gamma \vdash_{\mathit{inf}} f \in U,\\ &\Gamma \vdash_{\mathit{inf}} a \in S, \quad \text{and}\\ &T' \equiv \bigwedge^\star[T_i \mid S_i{\to}T_i \in \mathit{arrows}(\mathit{flub}_\Gamma(U)) \text{ and } \Gamma \vdash S \leq S_i].\end{aligned}$$

By the induction hypothesis,

$$\begin{aligned}&\Gamma \vdash U \leq V{\to}T, \quad \text{and}\\ &\Gamma \vdash S \leq V.\end{aligned}$$

By lemma 4.2.3(1),

$$\Gamma \vdash \bigwedge^\star[S_i{\to}T_i \mid S_i{\to}T_i \in \mathit{arrows}(\mathit{flub}_\Gamma(U))] \leq V{\to}T.$$

Finally, by the term application lemma (lemma 4.2.5), it follows that

$$\Gamma \vdash \bigwedge^\star[T_i \mid \Gamma \vdash S \leq S_i] \leq T,$$

where $S_i{\to}T_i \in \mathit{arrows}(\mathit{flub}_\Gamma(U))$.

In other words,

$$\Gamma \vdash \bigwedge^\star[T_i \mid S_i{\to}T_i \in \mathit{arrows}(\mathit{flub}_\Gamma(U))] \leq T.$$

T-TAbs We are given that

$$e \equiv \lambda X{\leq}T_1{:}K_1.e_2,\\ T \equiv \forall X{\leq}T_1{:}K_1.T_2, \quad \text{and}\\ \Gamma,\ X{\leq}T_1{:}K_1 \vdash e \in T_2.$$

By generation for *inf* (proposition 4.2.9),

$$T' \equiv \forall X{\leq}T_1{:}K_1.T_2' \quad \text{and}\\ \Gamma,\ X \leq T_1{:}K_1 \vdash_{\mathit{inf}} e_2 \in T_2'.$$

By the induction hypothesis, $\Gamma,\ X \leq T_1{:}K_1 \vdash T_2' \leq T_2$. Then, by S-All it follows that $\Gamma \vdash T' \leq T$.

T-TApp We are given that

$$e \equiv f\ S,\\ T \equiv T_2[X{\leftarrow}S]\Gamma \vdash f \in \forall X{\leq}T_1{:}K.T_2, \quad \text{and}\\ \Gamma \vdash S \leq T_1.$$

By generation for *inf* (proposition 4.2.9),

$$T' \equiv \bigwedge{}^{\star}[U_i \mid \forall X{\leq}S_i{:}K_i.U_i \in \mathit{alls}(\mathit{flub}_\Gamma(U)) \text{ and } \Gamma \vdash S \leq S_i], \text{ and}\\ \Gamma \vdash_{\mathit{inf}} f \in U.$$

By the induction hypothesis, $\Gamma \vdash U \leq \forall X{\leq}T_1{:}K.T_2$. By lemma 4.2.3(2),

$$\Gamma \vdash \bigwedge{}^{\star}[\forall X{\leq}S_i{:}K_i.U_i \mid \forall X{\leq}S_i{:}K_i.U_i \in \mathit{alls}(\mathit{flub}_\Gamma(U))] \leq \forall X{\leq}T_1{:}K.T_2.$$

Then, by the type application lemma (lemma 4.2.7), it follows that

$$\Gamma \vdash \bigwedge{}^{\star}[U_i[X{\leftarrow}S] \mid \Gamma \vdash S \leq S_i] \leq T_2[X{\leftarrow}S],$$

where $\forall X{\leq}S_i{:}K_i.U_i \in \mathit{alls}(\mathit{flub}_\Gamma(U))$.

Namely,

$$\Gamma \vdash \bigwedge{}^{\star}[U_i[X{\leftarrow}S] \mid \forall X{\leq}S_i{:}K_i.U_i \in \mathit{alls}(\mathit{flub}_\Gamma(U)) \text{ and } \Gamma \vdash S \leq S_i]\\ \leq T_2[X{\leftarrow}S].$$

T-For We are given that

$$e \equiv \text{for}(X{\in}S_1..S_n)e_2\\ \Gamma \vdash e_2[X{\leftarrow}S] \in T, \quad \text{and}\\ S \in \{S_1..S_n\}.$$

By generation for *inf* (proposition 4.2.9),

$$\begin{array}{ll} T' \equiv \bigwedge^{\star}[T_1..T_m], & \text{and} \\ \Gamma \vdash_{\mathit{inf}} e_2[X{\leftarrow}S] \in T_i & \text{for each i in } \{1..m\}\,. \end{array}$$

By the induction hypothesis, S-Meet-LB, and S-Trans, we have that

$$\Gamma \vdash \bigwedge^{\star}[T_1..T_m] \leq T.$$

T-Meet By the induction hypothesis and S-Meet-G.

T-Sub By the induction hypothesis and S-Trans. □

Finally, we have proved the following result.

Theorem 4.2.11 (*Minimal typing for $F^{\omega}_{\wedge}$*) Given a term e and a context Γ, there exists T such that for every T', if $\Gamma \vdash e \in T'$, then $\Gamma \vdash T \leq T'$.

4.3 Decidability of type checking and type inference

In the previous section we proved that the algorithm *inf* is sound and computes minimal types for the $F^{\omega}_{\wedge}$ typing system. The next step is to prove that the algorithm *inf* always terminates. This result completes the proof of decidability of type checking and type inference in $F^{\omega}_{\wedge}$.

We first define a measure for terms such that the type information inside the terms is considered to have constant value. The intuition behind the definition is to find a measure on terms which is invariant under type substitution (see lemma 4.3.2).

Definition 4.3.1 (*size* $||{-}||$)

$$\begin{array}{lcl} ||x|| & = & 1, \\ ||\lambda x{:}T.e|| & = & 1 + ||e||, \\ ||e_1\, e_2|| & = & ||e_1|| + ||e_2||, \\ ||\lambda X{\leq}T{:}K.e|| & = & 1 + ||e||, \\ ||e\,T|| & = & 1 + ||e||, \\ ||\mathrm{for}(X{\in}T_1..T_n)e|| & = & 1 + ||e||. \end{array}$$

Lemma 4.3.2 $||e|| = ||e[X{\leftarrow}T]||$.

Proposition 4.3.3 (*Well-foundedness of* inf)
The inference rules for *inf* define a terminating algorithm.

PROOF: In the case of AT-VAR, the termination follows from the decidability of ok judgements (see corollary 2.4.10(1)). Furthermore, for each rule R of *inf*, if $\Gamma \vdash e \in T$ is a hypothesis and $\Gamma \vdash e' \in T'$ is the conclusion of R, then $\|e\| < \|e'\|$. Moreover, in the cases for AT-APP and AT-TAPP, $\Gamma \vdash f \in T$ by the soundness of *inf* (proposition 4.2.4), $\Gamma \vdash T \in \star$ by well-kindedness of typing (proposition 2.4.20). Hence $\mathit{flub}_\Gamma(T)$ is defined by lemma 4.1.5. Furthermore, *arrows* and *alls* define finite sets, and, as we proved in section 3.5, subtyping is decidable. Hence, the algorithm *inf* always terminates. □

We can now state and prove that type checking in $F^\omega_\wedge$ is decidable.

THEOREM 4.3.4 (*Decidability of type checking in* $F^\omega_\wedge$)
For any context Γ, and for any term e and type T closed in Γ, it is decidable whether $\Gamma \vdash e \in T$.

PROOF: Infer a minimal type T' for e in Γ using *inf*, which is decidable by proposition 4.3.3, and check whether $\Gamma \vdash T' \leq T$, which is also decidable by theorem 3.5.18. □

Every term e closed in a context Γ has type $\top^\star$. We are interested in finding types other than $\top^\star$, namely non-trivial types. Since *inf* computes minimal types and $\top^\star$ is the largest type (modulo $=_{\beta\wedge}$) , if a term has a non trivial type in a given context, then the algorithm *inf* finds it.

THEOREM 4.3.5 (*Decidability of type inference in* $F^\omega_\wedge$)
For any context Γ and for any term e closed in Γ, it is decidable whether there exists a type T such that $\Gamma \vdash e \in T$ and $T \neq_{\beta\wedge} \top^\star$.

PROOF: Infer a minimal type T for e in Γ using *inf*, which is decidable by proposition 4.3.3, and reduce T to normal form which is decidable because $\rightarrow_{\beta\wedge}$ is strongly normalising (see theorem 2.5.10). Finally, check whether $T^{\mathit{nf}} \equiv \top^\star$. □

4.4 Subject reduction

The $F^\omega_\wedge$ system is layered in three syntactic categories: kinds, types, and terms. Since terms do not appear in either types or kinds, reductions in type expressions can be studied independently from the reductions of terms. In section 2.2, we proved that reduction on types preserves kinding properties: the sub-language of types and kinds satisfies the subject reduction property (lemma 2.4.12):

$$\text{if} \quad \Gamma \vdash S \in K \quad \text{and} \quad S \twoheadrightarrow_{\beta\wedge} T, \quad \text{then} \quad \Gamma \vdash T \in K.$$

In this section, we show the subject reduction property for typing judgements (proposition 4.4.7):

$$\text{if} \quad \Gamma \vdash e \in T \quad \text{and} \quad e \twoheadrightarrow_{\beta for} e', \quad \text{then} \quad \Gamma \vdash e' \in T.$$

In other words, reductions on terms are also safe.

Lemma 4.4.1 If $Y \notin \mathrm{FV}(S)$, then

1. $e[Y{\leftarrow}T][X{\leftarrow}S] \equiv e[X{\leftarrow}S][Y{\leftarrow}T[X{\leftarrow}S]]$
2. $U[Y{\leftarrow}T][X{\leftarrow}S] \equiv U[X{\leftarrow}S][Y{\leftarrow}T[X{\leftarrow}S]]$

Proof: By induction on the structure of e and U respectively. □

Lemma 4.4.2 (*Substitution for typing*)

1. If $\Gamma_1 \vdash e_1 \in S_1$ and $\Gamma_1,\, x{:}S_1,\, \Gamma_2 \vdash e_2 \in S_2$, then $\Gamma_1,\, \Gamma_2 \vdash e_2[x{\leftarrow}e_1] \in S_2$.
2. If $\Gamma_1 \vdash S{\leq}S_1$ and $\Gamma_1,\, X{\leq}S_1{:}K_1,\, \Gamma_2 \vdash e_2 \in S_2$, then $\Gamma_1,\, \Gamma_2[X{\leftarrow}S] \vdash e_2[X{\leftarrow}S] \in S_2[X{\leftarrow}S]$.

Proof:

1. By induction on the derivation of $\Gamma_1,\, x{:}S_1,\, \Gamma_2 \vdash e_2 \in S_2$.
2. By induction on the derivation of $\Gamma_1,\, X{\leq}S_1{:}K_1,\, \Gamma_2 \vdash e_2 \in S_2$, using the type substitution lemma (lemma 2.4.11) in the T-Var and T-Meet cases; the substitution lemma for subtyping (lemma 4.2.6) and lemma 4.4.1 in the case for T-TApp; lemma 4.4.1 in the T-For case, and the substitution lemma for subtyping (lemma 4.2.6) in the T-Subsumption case. □

Lemma 4.4.3 $\Gamma \vdash \top^\star \leq T$ if and only if $T =_{\beta\wedge} \top^\star$ and $\Gamma \vdash T \in \star$.

Proof: If $T =_{\beta\wedge} \top^\star$, then the result follows by S-Conv. Otherwise, if $\Gamma \vdash \top^\star \leq T$, by the well-kindedness of subtyping (proposition 2.4.19), T-Meet, and uniqueness of kinds (lemma 2.4.7), $\Gamma \vdash T \in \star$. By the equivalence of ordinary and algorithmic subtyping (proposition 3.4.3), $\Gamma^{nf} \vdash_{Alg} \top^\star \leq T^{nf}$, which can only be derived using AlgS-∀∃ where T^{nf} is the empty intersection. □

Given $\Gamma \vdash S \leq T$, generation for normal subtyping (proposition 3.2.10) and the equivalence of ordinary and normal subtyping (theorem 3.3.9) provide subtyping information about the normal forms of S and T. We can also show that subtyping is structural for arrow types, quantified types and type operators, without reducing the terms in the subtyping relation to normal form. An implementation of a subtyping algorithm for $F^\omega_\wedge$ could take advantage of this fact by delaying normalizing steps, which might result in having to consider fewer recursive calls or calls with smaller arguments.

Lemma 4.4.4 (*Generation for ordinary subtyping*)

1. $\Gamma \vdash T_1{\rightarrow}T_2 \leq S_1{\rightarrow}S_2$ and $S_2 \neq_{\beta\wedge} \top^\star$ if and only if $\Gamma \vdash S_1 \leq T_1$ and $\Gamma \vdash T_2 \leq S_2$
2. $\Gamma \vdash \forall X{\leq}T_1{:}K_T.T_2 \leq \forall X{\leq}S_1{:}K_S.S_2$ and $S_2 \neq_{\beta\wedge} \top^\star$ if and only if $K_S \equiv K_T$, $T_1 =_{\beta\wedge} S_1$, and $\Gamma,\, X{\leq}T_1{:}K_T \vdash T_2 \leq S_2$.

3. $\Gamma \vdash \Lambda X{:}K_T.T_2 \leq \Lambda X{:}K_S.S_2$ and $S_2 \neq_{\beta\wedge} \top^\star$ if and only if $\Gamma,\, X{:}K_S \vdash T_2 \leq S_2$ and $K_T \equiv K_S$.

Proof: The three statements are proved using a similar argument. We consider here the proof of part 2. If $K_S \equiv K_T$, $T_1 =_{\beta\wedge} S_1$, and $\Gamma,\, X{\leq}T_1{:}K_T \vdash T_2 \leq S_2$, then, by S-All and S-Conv, $\Gamma \vdash \forall X{\leq}T_1{:}K_T.T_2 \leq \forall X{\leq}S_1{:}K_S.S_2$. Conversely, let

$$\Gamma \vdash \forall X{\leq}T_1{:}K_T.T_2 \leq \forall X{\leq}S_1{:}K_S.S_2 \quad \text{and} \quad S_2 \neq_{\beta\wedge} \top^\star.$$

Lemma 4.4.3 implies that $T_2^{nf} \neq_{\beta\wedge} \top^\star$. Then we have to consider four cases according to whether S_2^{nf} and T_2^{nf} are intersection types or not. We illustrate the proof argument considering just one case. Let

$$\begin{array}{l}(\forall X{\leq}T_1{:}K_T.T_2)^{nf} \equiv \forall X{\leq}{T_1}^{nf}{:}K_T.T_2^{nf}, \quad \text{and}\\ (\forall X{\leq}S_1{:}K_S.S_2)^{nf} \equiv \bigwedge^\star[\forall X{\leq}{S_1}^{nf}{:}K_S.A_1..\forall X{\leq}{S_1}^{nf}{:}K_S.A_n],\end{array}$$

where $S_2^{nf} \equiv \bigwedge^\star[A_1..A_n]$. By the equivalence of ordinary and normal subtyping (theorem 3.3.9) and generation for normal subtyping (proposition 3.2.10), for each $i{\in}\{1..n\}$

$$\Gamma^{nf} \vdash_n \forall X{\leq}{T_1}^{nf}{:}K_T.T_2^{nf} \leq \forall X{\leq}{S_1}^{nf}{:}K_S.A_i$$

and, again generation for normal subtyping (proposition 3.2.10) implies that

$$\begin{array}{l}\Gamma^{nf},\, X{\leq}T_1^{nf}{:}K_T \vdash_n T_2^{nf} \leq A_i, \quad \text{and}\\ T_1^{nf} \equiv S_1^{nf}.\end{array}$$

By NS-$\forall$,

$$\Gamma^{nf},\, X{\leq}T_1^{nf}{:}K_T \vdash_n T_2^{nf} \leq S_2^{nf}$$

and, by the equivalence of ordinary and normal subtyping (theorem 3.3.9),

$$\Gamma,\, X{\leq}T_1{:}K_T \vdash T_2 \leq S_2. \qquad \square$$

Lemma 4.4.5 (*Generation for typing*)

1. If $\Gamma \vdash \lambda x{:}S_1.e \in S$, then there exists S_2 such that $\Gamma,\, x{:}S_1 \vdash e \in S_2$ and $\Gamma \vdash S_1{\rightarrow}S_2 \leq S$.

2. If $\Gamma \vdash \lambda X{\leq}S_1{:}K_1.e \in S$, then there exists S_2 such that $\Gamma,\, X{\leq}S_1{:}K_1 \vdash e \in S_2$ and $\Gamma \vdash \forall X{\leq}S_1{:}K_1.S_2 \leq S$.

3. If $\Gamma \vdash \text{for}(X{\in}\{U_1..U_n\})e \in T$, then there exist $T_1..T_n$ such that, for each i in $\{1..n\}$, $\Gamma \vdash e[X{\leftarrow}U_i] \in T_i$ and $\Gamma \vdash \bigwedge^\star[T_1..T_n] \leq T$.

Proof: Each statement is proved by induction on the derivation of the typing statement in the antecedent. We exhibit here the proof of part 3. We proceed by case analysis on the last rule of the derivation of $\Gamma \vdash \text{for}(X{\in}\{U_1..U_n\})e \in T$.

T-For We are given that $\Gamma \vdash e[X \leftarrow U] \in T$ for some $U \in \{U_1..U_n\}$. Since every closed term has a type, we have that, for each i in $\{1..n\}$, $\Gamma \vdash e[X \leftarrow U_i] \in T_i$, and, by S-Meet-LB, $\Gamma \vdash \bigwedge^{\star}[T_1..T..T_n] \leq T$.

T-Meet Let $T \equiv \bigwedge^{\star}[S_1..S_k]$. We are given that,

$$\begin{array}{l} \Gamma \vdash \text{ok} \quad \text{and} \\ \Gamma \vdash \text{for}(X \in \{U_1..U_n\})e \in S_j, \quad \text{for each } j \text{ in } \{1..k\}. \end{array}$$

By the induction hypothesis, for each $j \in \{1..k\}$ and each $i \in \{1..n\}$, there exist T_{j_i} such that

$$\begin{array}{l} \Gamma \vdash e[X \leftarrow U_i] \in T_{j_i}, \qquad \text{and} \\ \Gamma \vdash \bigwedge^{\star}[T_{j_1}..T_{j_n}] \leq S_j, \end{array}$$

and, by the minimal type property (theorem 4.2.11), there exist $T_1..T_n$ such that

$$\begin{array}{l} \Gamma \vdash e[X \leftarrow U_i] \in T_i, \quad \text{and} \\ \Gamma \vdash T_i \leq T_{j_i}, \end{array}$$

by lemma 2.4.18, it follows that $\Gamma \vdash \bigwedge^{\star}[T_1..T_n] \leq \bigwedge^{\star}[T_{j_1}..T_{j_n}]$, and by S-Trans, $\Gamma \vdash \bigwedge^{\star}[T_1..T_n] \leq S_j$. Finally, by S-Meet-G, it follows that $\Gamma \vdash \bigwedge^{\star}[T_1..T_n] \leq \bigwedge^{\star}[S_1..S_k]$.

T-Sub We are given that

$$\begin{array}{l} \Gamma \vdash \text{for}(X \in \{U_1..U_n\})e \in S, \quad \text{and} \\ \Gamma \vdash S \leq T. \end{array}$$

The result follows by the induction hypothesis and S-Trans. □

Since terms cannot occur in types, subject reduction for terms does not need to consider reductions in contexts.

Proposition 4.4.6 (*One step subject reduction for typing judgements*)
If $\Gamma \vdash e \in T$ and $e \rightarrow_{\beta for} e'$, then $\Gamma \vdash e' \in T$.

Proof: Since every term has type $\top^{\star}$, the interesting case is when $T \neq_{\beta\wedge} \top^{\star}$. This proposition follows by induction on the derivation of $\Gamma \vdash e \in T$. We consider the cases where e is a redex; the other cases follow by straightforward application of the induction hypothesis.

T-App There are two possibilities for e to be a redex.

1. $e \equiv (\lambda x{:}S_1.e_1)\, e_2$, $e' \equiv e_1[x{\leftarrow}e_2]$, and $T \equiv T_2$. We are given that

   $$\Gamma \vdash \lambda x{:}S_1.e_1 \in T_1{\rightarrow}T_2 \quad \text{and} \quad \Gamma \vdash e_2 \in T_1.$$

   By the generation lemma for typing (lemma 4.4.5), there exists S_2 such that,

   $$\Gamma,\, x{:}S_1 \vdash e_1 \in S_2 \quad \text{and} \quad \Gamma \vdash S_1{\rightarrow}S_2 \leq T_1{\rightarrow}T_2.$$

   Since $T_2 \neq_{\beta\wedge} \top^\star$, by the generation lemma for ordinary subtyping (lemma 4.4.4),

   $$\Gamma \vdash T_1 \leq S_1 \quad \text{and} \quad \Gamma \vdash S_2 \leq T_2.$$

   Then, by T-Subsumption, it follows that

   $$\Gamma,\, x{:}S_1 \vdash e_1 \in T_2 \quad \text{and} \quad \Gamma \vdash e_2 \in S_1.$$

   Finally, by the substitution lemma for typing (lemma 4.4.2(1)),

   $$\Gamma \vdash e_1[x{\leftarrow}e_2] \in T_2.$$

2. $e \equiv (\text{for}(X{\in}U_1..U_n)e_2)\, e_1$, $e' \equiv \text{for}(X{\in}U_1..U_n)(e_2\, e_1)$, and $T \equiv T_2$. We are given that

   $$\Gamma \vdash \text{for}(X{\in}U_1..U_n)e_2 \in T_1{\rightarrow}T_2 \quad \text{and} \quad \Gamma \vdash e_1 \in T_1.$$

   By the generation lemma for typing (lemma 4.4.5), there exist $V_1..V_n$ such that

   $$\Gamma \vdash e_2[X{\leftarrow}U_i] \in V_i \quad \text{for each } i{\in}\{1..n\}\text{ , and}$$
   $$\Gamma \vdash \textstyle\bigwedge^\star[V_1..V_n] \leq T_1{\rightarrow}T_2.$$

   We write $V_i^{nf} \equiv A_{i_1},$ if it is not an intersection,
   $V_i^{nf} \equiv \bigwedge^\star[A_{i_1}..A_{i_{k_i}}],$ otherwise.

   Note that $\bigwedge^\star[V_1..V_n]^{nf} \equiv \bigwedge^\star[A_{1_1}..A_{1_{k_1}}..A_{n_1}..A_{n_{k_n}}]$. By the equivalence of ordinary and normal subtyping (theorem 3.3.9),

   $$\Gamma^{nf} \vdash_n \textstyle\bigwedge^\star[A_{1_1}..A_{1_{k_1}}..A_{n_1}..A_{n_{k_n}}] \leq (T_1{\rightarrow}T_2)^{nf}.$$

   We have to consider two cases according to the form of $(T_1{\rightarrow}T_2)^{nf}$.

   (a) $(T_1{\rightarrow}T_2)^{nf} \equiv T_1^{nf}{\rightarrow}T_2^{nf}$. By generation for normal subtyping (proposition 3.2.10), there exist $l{\in}\{1..n\}$ and $j{\in}\{1..k_l\}$ such that

   $$\Gamma^{nf} \vdash_n A_{l_j} \leq T_1^{nf}{\rightarrow}T_2^{nf},$$

   and, by NS-∃ or NS-Refl,

   $$\Gamma^{nf} \vdash_n V_l^{nf} \leq A_{l_j},$$

   by NS-Trans,

   $$\Gamma^{nf} \vdash_n V_l^{nf} \leq T_1^{nf}{\rightarrow}T_2^{nf},$$

and, by the equivalence of ordinary and normal subtyping (theorem 3.3.9),

$$\Gamma \vdash V_l \leq T_1 \to T_2.$$

Then, by T-Subsumption,

$$\Gamma \vdash e_2[X \leftarrow U_l] \in T_1 \to T_2.$$

By T-App,

$$\Gamma \vdash (e_2[X \leftarrow U_l])\, e_1 \in T_2,$$

and since X is not a free variable of e_1 we have that,

$$\Gamma \vdash e_2\, e_1[X \leftarrow U_l] \in T_2.$$

Finally, applying T-For, we have that

$$\Gamma \vdash \text{for}(X \in U_1..U_n)e_2\, e_1 \in T_2.$$

(b) $(T_1 \to T_2)^{nf} \equiv \bigwedge^\star[T_1^{nf} \to B_1..T_1^{nf} \to B_r]$, where $T_2^{nf} \equiv \bigwedge^\star[B_1..B_r]$. By generation for normal subtyping (proposition 3.2.10), for every $s \in \{1..r\}$ there exist $l \in \{1..n\}$ and $j \in \{1..k_l\}$ such that

$$\Gamma^{nf} \vdash_n A_{l_j} \leq T_1^{nf} \to B_s,$$

and, by NS-∃ or NS-Refl,

$$\Gamma^{nf} \vdash_n V_l^{nf} \leq A_{l_j},$$

by NS-Trans, for every $s \in \{1..r\}$ there exists $l \in \{1..n\}$ such that

$$\Gamma^{nf} \vdash_n V_l^{nf} \leq T_1^{nf} \to B_s,$$

and, by the equivalence of ordinary and normal subtyping (theorem 3.3.9),

$$\Gamma \vdash V_l \leq T_1 \to B_s.$$

By T-Subsumption, for every $s \in \{1..r\}$ there exists $l \in \{1..n\}$

$$\Gamma \vdash e_2[X \leftarrow S_l] \in T_1 \to B_s.$$

By T-App, for every $s \in \{1..r\}$ there exists $l \in \{1..n\}$

$$\Gamma \vdash (e_2[X \leftarrow S_l])\, e_1 \in B_s,$$

and since X is not a free variable of e_1 we have that, for every $s \in \{1..r\}$ there exists $l \in \{1..n\}$

$$\Gamma \vdash e_2\, e_1[X \leftarrow S_l] \in B_s.$$

Applying T-For, we get that for every $s \in \{1..r\}$

$$\Gamma \vdash \text{for}(X \in U_1..U_n)e_2\, e_1 \in B_s,$$

by T-Meet,

$$\Gamma \vdash \text{for}(X \in U_1..U_n)e_2\, e_1 \in T_2^{nf}.$$

Finally, by S-Conv and T-Subsumption,

$$\Gamma \vdash \text{for}(X \in U_1..U_n)e_2\, e_1 \in T_2.$$

T-TApp There are two possibilities for e to be a redex. The case when $e \equiv (\text{for}(X \in U_1..U_n)e_2)\, S$ follows a similar argument to the one used for the case $e \equiv (\text{for}(X \in U_1..U_n)e_2)\, e_1)$ in T-App.

If $e \equiv (\lambda X{\le}S_1{:}K_S.e_2)\,S \quad e' \equiv e_2[X{\leftarrow}S]$, and $T \equiv T_2[X{\leftarrow}S]$, we have that $\Gamma \vdash \lambda X{\le}S_1{:}K_S.e_1 \in \forall X{\le}T_1{:}K_T.T_2$ and $\Gamma \vdash S \le T_1$. By the generation lemma for typing (lemma 4.4.5), there exists S_2 such that

$$\Gamma,\, X{\le}S_1{:}K_S \vdash e_2 \in S_2 \quad \text{and}$$
$$\Gamma \vdash \forall X{\le}S_1{:}K_S.S_2 \le \forall X{\le}T_1{:}K_T.T_2.$$

Since $T_2[X{\leftarrow}S] \neq_{\beta\wedge} \top^\star$, lemma 2.3.1.4(3) implies that $T_2 \neq_{\beta\wedge} \top^\star$. Then, by the generation lemma for ordinary subtyping (lemma 4.4.4),

$$\Gamma,\, X{\le}S_1{:}K_S \vdash S_2 \le T_2, \quad S_1 =_{\beta\wedge} T_1, \quad \text{and} \quad K_S \equiv K_T.$$

By T-Subsumption, $\Gamma,\, X{\le}S_1{:}K_S \vdash e_2 \in T_2$, and, by S-Trans and S-Conv, $\Gamma \vdash S \le S_1$. Finally, by the substitution lemma for typing (lemma 4.4.2(2)), $\Gamma \vdash e_2[X{\leftarrow}S] \in T_2[X{\leftarrow}S]$.

T-For Let $e \equiv \text{for}(X{\in}U_1..U_n)e_1$, where $X \notin \text{FTV}(e_1)$ and $e' \equiv e_1$. We are given that $\Gamma \vdash e_1[X{\leftarrow}U] \in T$, with $U{\in}\{U_1..U_n\}$. Since $e_1 \equiv e_1[X{\leftarrow}U]$, the result holds. □

We now have all the results needed in order to prove that reduction on terms preserves typing. The following proposition, the subject reduction property for $F^\omega_\wedge$ terms, is a consequence of the previous one.

Proposition 4.4.7 (*Subject reduction for typing judgements*)
If $\Gamma \vdash e \in T$ and $e \twoheadrightarrow_{\beta for} e'$, then $\Gamma \vdash e' \in T$.

Proof: By induction on the derivation of $e \twoheadrightarrow_{\beta for} e'$, using proposition 4.4.6. □

Chapter 5

A PER Model for $F^{\omega}_{\wedge}$

5.1 Introduction

This chapter is based on [CP93]. The differences come from having replaced the distributivity subtyping rules by reduction rules. Among simplest models for typed λ-calculi are those based on partial equivalence relations (PERs). A model in this style is essentially untyped: terms are interpreted by erasing all type information and interpreting the resulting pure λ-term as an element of the model. A type, in this setting, is just a subset of the model along with an appropriate notion of equivalence of elements. Coercions between types are interpreted as inclusion of PERs.

Our PER model for $F^{\omega}_{\wedge}$ extends the model of $F_{\wedge}$ given in [Pie91], which is based on Bruce and Longo's model for $F_{\leq}$ [BL90]. The usual interpretation of a quantified type $\forall X.\mathbb{T}$ in a PER model is the PER-indexed intersection of all possible instances of T. Bruce and Longo refined this definition to interpret $\forall X{\leq}S.\mathbb{T}$ as the intersection of all the instances of T where X is interpreted as a sub-PER of the interpretation of S. This intuition also serves for intersection types: $\bigwedge^{\star}[T_1..T_n]$ is interpreted as the intersection of the PERs interpreting each of the T_i's. We generalize this model to ω-order polymorphism (and subtyping) by interpreting type operators as functions over PERs.

To deal correctly with intersection types, we need to make one significant technical departure here from PER models of ordinary bounded quantification: instead of allowing the elements of our PERs to be drawn from the carrier of an arbitrary partial combinatory algebra $\mathcal{D}$, we require that $\mathcal{D}$ be a *total* combinatory algebra. This restriction is needed to validate instances of S-Conv, which have the form $\Gamma \vdash \top^{\star} \leq S \to \top^{\star}$. For example, let $S = \top^{\star}$. The empty intersection $\top^{\star}$ is interpreted by the everywhere-defined PER, i.e., $[\![\top^{\star}]\!]$ relates every m to itself. To validate the distributivity law, it must therefore be the case that $[\![\top^{\star} \to \top^{\star}]\!]$ relates every element to itself. But this will only be true if the application of any element to any other element is defined. This observation is due to QingMing Ma.

Cardelli and Longo [CL91] and Bruce and Mitchell [BM92] have given related models for variants of F^{ω} including subtyping, but without intersections.

The notation and fundamental definitions used here are based on papers of

Bruce and Longo [BL90], Freyd, Mulry, Rosolini, and Scott [FMRS90], and others. A helpful basic reference for PER models of second-order λ-calculi is [Mit90b]; also see [BMM90] for more general discussion of second-order models and [Bar84, HS86] for general discussion of combinatory models.

5.2 Total combinatory algebras

A *total combinatory algebra* is a tuple $\mathcal{D} = \langle D, \cdot, k, s\rangle$ comprising a set D of *elements*, an *application function* $\cdot$ with type $D \to (D \to D)$, and distinguished elements k, $s \in D$ such that, for all d_1, d_2, $d_3 \in D$,

$$k \cdot d_1 \cdot d_2 = d_1$$
$$s \cdot d_1 \cdot d_2 \cdot d_3 = (d_1 \cdot d_3) \cdot (d_2 \cdot d_3).$$

Throughout this section, we work with a fixed, but unspecified, total combinatory algebra $\mathcal{D}$. (C.f. [Sco76] for examples.)

The set of *pure λ-terms* is defined by the following grammar:

$$M ::= x \mid \lambda(x)M \mid M_1\ M_2$$

The set of *combinator terms* is:

$$C ::= x \mid C_1\ C_2 \mid K \mid S$$

The *bracket abstraction* of a combinator term C with respect to a variable x, written $\mathrm{fun}^\star\,(x)\,C$, is defined as follows:

$$\begin{array}{lcll} \mathrm{fun}^\star\,(x)\,C & = & K\,C & \text{when } x \notin \mathrm{FV}(C) \\ \mathrm{fun}^\star\,(x)\,x & = & S\,K\,K & \\ \mathrm{fun}^\star\,(x)\,C_1\,C_2 & = & S\,(\mathrm{fun}^\star\,(x)\,C_1)\,(\mathrm{fun}^\star\,(x)\,C_2) & \text{when } x{\in}\mathrm{FV}(C_1\,C_2) \end{array}$$

The *combinator translation* of a pure λ-term M, written $|M|$, is defined as follows:

$$\begin{array}{lcl} |x| & = & x \\ |\lambda(x)M| & = & \mathrm{fun}^\star\,(x)\,|M| \\ |M_1\,M_2| & = & |M_1|\,|M_2| \end{array}$$

A *term environment* η is a finite function from term variables to elements of D. When $x \notin \mathrm{dom}(\eta)$, we write $\eta[x{\leftarrow}d]$ for the environment that maps x to d and agrees with η everywhere else. We write $\eta\backslash x$ for the environment like η except that $\eta(x)$ is undefined; $\eta\backslash\Gamma$ is like η but undefined on all the variables in $\mathrm{dom}(\Gamma)$. We say that η' *extends* η when $\mathrm{dom}(\eta) \subseteq \mathrm{dom}(\eta')$ and η and η' agree on $\mathrm{dom}(\eta)$.

Let C be a combinator term and η a term environment such that $\mathrm{FV}(C) \subseteq \mathrm{dom}(\eta)$. Then the *interpretation* of C under η, written $[\![C]\!]_\eta$, is defined as follows:

$$\begin{array}{lcl} [\![x]\!]_\eta & = & \eta(x) \\ [\![C_1\,C_2]\!]_\eta & = & [\![C_1]\!]_\eta \cdot [\![C_2]\!]_\eta \\ [\![K]\!]_\eta & = & k \\ [\![S]\!]_\eta & = & s. \end{array}$$

LEMMA 5.2.1

1. If η' extends η and $\mathrm{FV}(C) \subseteq \mathrm{dom}(\eta)$, then $[\![C]\!]_\eta = [\![C]\!]_{\eta'}$.

2. $[\![\mathrm{fun}^\star\,(x)\,C]\!]_\eta \cdot m = [\![C]\!]_{\eta[x\leftarrow m]}$.

PROOF: Standard. □

5.3 Higher-order partial equivalence relations

A *partial equivalence relation* (PER) on a combinatory algebra $\mathcal{D}$ is a symmetric and transitive relation A on D. We write $m\ \{A\}\ n$ when A relates m and n. The *domain* of A, written $\mathrm{dom}(A)$, is the set $\{n \mid n\ \{A\}\ n\}$. (Note that $m\ \{A\}\ n$ implies $m \in \mathrm{dom}(A)$.) We write **PER** for the class of all PERs.

If A and B are relations, then $A \to B$ is the relation where $m\ \{A \to B\}\ n$ iff, for all $p, q{\in}D$, $p\ \{A\}\ q$ $m{\cdot}p\ \{B\}\ n{\cdot}q$. It is not hard to show that $A \to B$ is a PER when A and B are PERs, and that the intersection of any set of PERs is a PER. To interpret type operators, we need to consider not only PERs, but arbitrary function spaces built on PER. An element of such a function space (including, as a special case, an element of PER itself), is called a *higher-order PER* (HOPER). The interpretation of a kind K is a suitable space of HOPERs:

$$\begin{array}{lcl} [\![\,\star\,]\!] & = & \mathbf{PER} \quad \text{and} \\ [\![K_1{\to}K_2]\!] & = & [\![K_1]\!] \to [\![K_2]\!]. \end{array}$$

We generalize the familiar graph-inclusion of relations to HOPERs as follows:

$$\begin{array}{lll} A \subseteq^\star B & \text{iff} & A, B \in [\![\,\star\,]\!] \text{ and} \\ & & m\ \{A\}\ n \text{ implies } m\ \{B\}\ n \text{ for all } m, n{\in}D; \\ A \subseteq^{K_1\to K_2} B & \text{iff} & A, B \in [\![K_1{\to}K_2]\!] \text{ and} \\ & & A\,P \subseteq^{K_2} B\,P \text{ for all } P \in [\![K_1]\!]. \end{array}$$

Let $\{A_i \in [\![K]\!]\}_{i\in I}$ be a set of HOPERs indexed by a set I. Then $\bigcap^K_{i\in I} A_i$ is the HOPER defined by

$$\begin{array}{lll} m\ \{\bigcap^\star_{i\in I} A_i\}\ n & \text{iff} & \text{for every } i,\ m\ \{A_i\}\ n \\ \bigcap^{K_1\to K_2}_{i\in I} A_i & = & \lambda P{\in}[\![K_1]\!].\, \bigcap^{K_2}_{i\in I} A_i\,P \end{array}$$

LEMMA 5.3.1

1. Each $\subseteq^K$ is transitive.

2. If $A_j \in [\![K]\!]$ for each $j{\in}I$, then $\bigcap^K_{i\in I} A_i \in [\![K]\!]$.

3. If $A \subseteq^K B_j$ for each j, then $A \subseteq^K \bigcap^K_{i\in I} B_i$.

4. $\bigcap^{K}_{i\in I} A_i \subseteq^{K} A_j$ for each j.

5. $\bigcap^{\star}_{i\in I} A{\rightarrow}B_i \subseteq^{\star} A \rightarrow \bigcap^{\star}_{i\in I} B_i$.

6. $\bigcap^{\star}_{i\in I} \bigcap^{\star}_{P\subseteq^K A} B_i \subseteq^{\star} \bigcap^{\star}_{P\subseteq^K A} \bigcap^{\star}_{i\in I} B_i$.

7. $\bigcap^{K_1\rightarrow K_2}_{i\in I} \lambda P{\in}[\![K_1]\!].\, B_i\, P \subseteq^{K_1\rightarrow K_2} \lambda P{\in}[\![K_1]\!].\, \bigcap^{K_2}_{i\in I} B_i\, P$, if each $B_i \in [\![K_1{\rightarrow}K_2]\!]$.

8. $\bigcap^{K_2}_{i\in I} B_i\ A \subseteq^{K_2} (\bigcap^{K_1\rightarrow K_2}_{i\in I} B_i)\ A$, where $A{\in}[\![K_1]\!]$.

Indeed, in cases 4 through 8 the inclusions are equalities.

Proof: Straightforward. □

Each collection $[\![K]\!]$ of HOPERs has a maximal element under the ordering $\subseteq^K$. This element is written $\top^K$ and can be calculated as follows: $\top^\star$ is the total relation on $\mathcal{D}$ and $\top^{K_1\rightarrow K_2} = \lambda P{\in}[\![K_1]\!].\, \top^{K_2}$.

Fact 5.3.2 Let $A \in [\![K]\!]$. Then:

1. $A \subseteq^K \top^K$.

2. $\top^K \subseteq^K A$ implies $A = \top^K$.

5.4 HOPER interpretation of $F^\omega_\wedge$

An *environment* η is a finite function from type variables to HOPERs and from term variables to elements of D. The notations for environment extension, restriction, and agreement are carried over from term environments. By an abuse of notation, type environments are used in place of term environments from now on.

The *erasure* of an $F^\omega_\wedge$ term e, written erase(e), is the pure λ-term defined as follows:

$$\begin{array}{lcl}
\text{erase}(x) & = & x \\
\text{erase}(\lambda x{:}T.e) & = & \lambda(x)\text{erase}(e) \\
\text{erase}(e_1\ e_2) & = & \text{erase}(e_1)\ \text{erase}(e_2) \\
\text{erase}(\lambda X{\leq}T{:}K.e) & = & \text{erase}(e) \\
\text{erase}(e\ T) & = & \text{erase}(e) \\
\text{erase}(\text{for}(X{\in}T_1..T_n)e) & = & \text{erase}(e).
\end{array}$$

Let η be a term environment and e an expression such that $\text{FV}(e) \subseteq \text{dom}(\eta)$. Then the *interpretation* of e under η, written $[\![e]\!]_\eta$, is $[\![|\text{erase}(e)|]\!]_\eta$.

If η is an environment and T a type expression such that $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\eta)$, then the *interpretation* of T under η, written $\llbracket T \rrbracket_\eta$, is the HOPER defined as follows:

$$\begin{array}{lcl}
\llbracket X \rrbracket_\eta & = & \eta(X) \\
\llbracket T_1 \rightarrow T_2 \rrbracket_\eta & = & \llbracket T_1 \rrbracket_\eta \rightarrow \llbracket T_2 \rrbracket_\eta \\
\llbracket \forall X {\leq} T_1{:}K_1 . T_2 \rrbracket_\eta & = & \bigcap^{\star}_{P \subseteq^{K_1} \llbracket T_1 \rrbracket_\eta} \llbracket T_2 \rrbracket_{\eta[X \leftarrow P]} \\
\llbracket \bigwedge^K [T_1..T_n] \rrbracket_\eta & = & \bigcap^{K}_{1 \leq i \leq n} \llbracket T_i \rrbracket_\eta \\
\llbracket S\, T \rrbracket_\eta & = & \llbracket S \rrbracket_\eta \, \llbracket T \rrbracket_\eta \\
\llbracket \Lambda X{:}K.T \rrbracket_\eta & = & \lambda P {\in} \llbracket K \rrbracket . \, \llbracket T \rrbracket_{\eta[X \leftarrow P]}
\end{array}$$

We say that an environment η *satisfies* a context Γ, written $\eta \models \Gamma$, if $\mathrm{dom}(\eta) = \mathrm{dom}(\Gamma)$ and

1. $\Gamma \equiv \emptyset$; or

2. $\Gamma \equiv \Gamma_1,\, x{:}T$, where $\eta \backslash x$ satisfies Γ_1 and either $\llbracket T \rrbracket_{\eta\backslash x}\uparrow$ or $\eta(x) \in \mathrm{dom}(\llbracket T \rrbracket_{\eta\backslash x})$; or

3. $\Gamma \equiv \Gamma_1,\, X{\leq}T{:}K$, where $\eta\backslash X$ satisfies Γ_1 and either $\llbracket T \rrbracket_{\eta\backslash X}\uparrow$ or $\eta(X) \subseteq^K \llbracket T \rrbracket_{\eta\backslash X}$.

Iterating the definition immediately yields that either $\llbracket T \rrbracket_{\eta\backslash\Gamma_2\backslash X}\uparrow$ or $\llbracket T \rrbracket_{\eta\backslash\Gamma_2\backslash X} \in \llbracket K \rrbracket$, whenever $\eta \models \Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2$. Also, note that if η' extends η and $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\eta)$, then either $\llbracket T \rrbracket_\eta\uparrow$ and $\llbracket T \rrbracket_{\eta'}\uparrow$, or else both are defined and $\llbracket T \rrbracket_\eta = \llbracket T \rrbracket_{\eta'}$.

Lemma 5.4.1 Let T be a type, Γ a context, and η an environment such that $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\eta)$ and $\eta \models \Gamma$. If $\Gamma \vdash T \in K$, then $\llbracket T \rrbracket_\eta\downarrow$ and $\llbracket T \rrbracket_\eta \in \llbracket K \rrbracket$.

Proof: We need to check the desired result together with an additional fact:

1. If $\Gamma \vdash T \in K$, then $\llbracket T \rrbracket_\eta$ is defined and $\llbracket T \rrbracket_\eta \in \llbracket K \rrbracket$.

2. If $\Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2 \vdash \mathrm{ok}$, then $\llbracket T \rrbracket_{\eta\backslash\Gamma_2\backslash X}\downarrow$ and $\llbracket T \rrbracket_{\eta\backslash\Gamma_2\backslash X} \in \llbracket K \rrbracket$.

The two are proved by simultaneous induction on derivations. We give only the interesting cases; the rest follow by straightforward use of the induction hypothesis and simple properties of HOPERs.

1. K-TVar We are given that $\Gamma \equiv \Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2$ and that $\Gamma \vdash \mathrm{ok}$. By part 2 of the induction hypothesis, $\llbracket T \rrbracket_{\eta\backslash\Gamma_2\backslash X}\downarrow$. By the definition of satisfaction, $(\eta\backslash\Gamma_2)(X) \subseteq^K \llbracket T \rrbracket_{\eta\backslash\Gamma_2\backslash X}$, which implies in particular that $\eta(X) \in \llbracket K \rrbracket$. By the definition of interpretation, $\llbracket X \rrbracket_\eta = \eta(X) \in \llbracket K \rrbracket$.

K-All We are given that $K \equiv \star$ and $T \equiv \forall X{\leq}T_1{:}K_1.T_2$, and that $\Gamma,\, X{\leq}T_1{:}K_1 \vdash T_2 \in \star$. By lemma 2.4.1, there exists a shorter derivation of $\Gamma,\, X{\leq}T_1{:}K_1 \vdash \mathrm{ok}$, and, by part 2 of the induction hypothesis, we have that $[\![T_1]\!]_\eta{\downarrow}$ and $[\![T_1]\!]_\eta \in [\![K_1]\!]$. Now, suppose $P \subseteq^{K_1} [\![T_1]\!]_\eta$. Then the definition of satisfaction yields $\eta[X{\leftarrow}P] \models \Gamma,\, X{\leq}T_1{:}K_1$. By part 1 of the induction hypothesis, $[\![T_2]\!]_{\eta[X\leftarrow P]}{\downarrow}$ and $[\![T_2]\!]_{\eta[X\leftarrow P]} \in [\![\,\star\,]\!]$. Since PER is closed under intersections, $[\![T]\!]_\eta = \bigcap^{\star}_{P\subseteq^{K_1}[\![T_1]\!]_\eta} [\![T_2]\!]_{\eta[X\leftarrow P]} \in [\![\,\star\,]\!]$.

K-OAbs Similar.

2. C-Var We are given that $\Gamma \equiv \Gamma',\, x{:}S$, where $\Gamma' \equiv \Gamma_1,\, X{\leq}T{:}K,\, \Gamma'_2$ and that $\Gamma' \vdash S \in \star$. By lemma 2.4.1, $\Gamma' \vdash \mathrm{ok}$. By the definition of satisfaction, $\eta\backslash x \models \Gamma'$. By the induction hypothesis, the result follows.

C-TVar The case where $\Gamma_2 \not\equiv \emptyset$ is similar to the previous case. When $\Gamma_2 \equiv \emptyset$, we are given that $\Gamma \equiv \Gamma_1,\, X{\leq}T{:}K$ and that $\Gamma_1 \vdash T \in K$. By the definition of satisfaction, $\eta\backslash X \models \Gamma'$. By the induction hypothesis, the result follows. □

In order to prove the soundness of subtyping we need some technical results about substitution and β-conversion.

Lemma 5.4.2

Let η be an environment with $X \notin \mathrm{dom}(\eta)$ and such that $\mathrm{FTV}(S[X{\leftarrow}T]) \subseteq \mathrm{dom}(\eta)$ and $[\![S]\!]_{\eta[X\leftarrow[\![T]\!]_\eta]}{\downarrow}$. Then $[\![S[X{\leftarrow}T]]\!]_\eta = [\![S]\!]_{\eta[X\leftarrow[\![T]\!]_\eta]}$.

Proof: By induction on the structure of S. □

Lemma 5.4.3 Let η be an environment such that $\mathrm{FTV}(S) \subseteq \mathrm{dom}(\eta)$. If $S =_{\beta\wedge} T$ and $[\![S]\!]_\eta{\downarrow}$, then $[\![S]\!]_\eta = [\![T]\!]_\eta$.

Proof: By induction on the definition of $\beta\wedge$-conversion, it is easy to see that it suffices to show the statement for a one-step reduction $S \rightarrow_{\beta\wedge} T$. This is proved by induction on the structure of S. The only interesting cases are when S is a $\beta\wedge$-redex and T its reduct. Let $S \equiv (\Lambda X{:}K.T_2 T_1$ and $T \equiv T_1[X{\leftarrow}T_2]$. Then

$$\begin{array}{lcll}
[\![(\Lambda X{:}K.T_2 T_1]\!]_\eta & = & (\lambda P{\in}[\![K]\!].\, [\![T_2]\!]_{\eta[X\leftarrow P]})[\![T_1]\!]_\eta & \text{by definition of } [\![\,-\,]\!]_\eta \\
 & = & [\![T_2]\!]_{\eta[X\leftarrow[\![T_1]\!]_\eta]} & \\
 & = & [\![T_2[X{\leftarrow}T_1]]\!]_\eta & \text{by lemma 5.4.2.}
\end{array}$$

For the other redexes the result follows from lemma 5.3.1 given that for items 4 through 8 the inclusions are equalities. □

Our main semantic results are the soundness of subtyping and typing.

Theorem 5.4.4 (*Soundness of subtyping*) If $\Gamma \vdash S \leq T$ and $\Gamma \vdash S \in K$, and if $\eta \models \Gamma$, then $[\![S]\!]_\eta \subseteq^K [\![T]\!]_\eta$.

Proof: The proof proceeds by induction on the structure of a derivation of $\Gamma \vdash S \leq T$. For the sake of readability, we often make implicit use of the fact that if $\Gamma \vdash S \leq T$ and $\Gamma \vdash S \in K$, then, by proposition 2.4.19 and lemmas 2.4.7 and 5.4.1, $[\![S]\!]_\eta$ and $[\![T]\!]_\eta$ are defined and belong to $[\![K]\!]$ whenever $\eta \models \Gamma$.

S-Conv We are given that $S =_{\beta\wedge} T$. Then, by lemma 5.4.3, $[\![S]\!]_\eta = [\![T]\!]_\eta$

S-Trans By the induction hypothesis and the transitivity of $\subseteq^K$ (lemma 5.3.1).

S-TVar We are given $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \mathrm{ok}$. Now, $[\![X]\!]_\eta = \eta(X) \subseteq^K [\![T]\!]_{\eta\backslash X}$, because $\eta \models \Gamma$ and, as η extends $\eta\backslash X$, $[\![T]\!]_{\eta\backslash X} = [\![T]\!]_\eta$.

S-Arrow We are given $\Gamma \vdash T_1 \leq S_1$ and $\Gamma \vdash S_2 \leq T_2$, with $\Gamma \vdash S_1 {\rightarrow} S_2 \in \star$. By the uniqueness of kinds (lemma 2.4.7), $K \equiv \star$. Now by the well-kindedness of subtyping (proposition 2.4.19) and syntax directedness of kinds (proposition 2.4.6) we have $\Gamma \vdash S_2, T_1 \in \star$. By the induction hypothesis, $[\![S_2]\!]_\eta \subseteq^\star [\![T_2]\!]_\eta$ and $[\![T_1]\!]_\eta \subseteq^\star [\![S_1]\!]_\eta$. Hence, by the covariance on the right and contravariance on the left of the function space constructor on PERs (easily verified from its definition), $[\![S_1]\!]_\eta {\rightarrow} [\![S_2]\!]_\eta \subseteq^\star [\![T_1]\!]_\eta {\rightarrow} [\![T_2]\!]_\eta$, i.e. $[\![S_1 {\rightarrow} S_2]\!]_\eta \subseteq^\star [\![T_1 {\rightarrow} T_2]\!]_\eta$.

S-All We are given that

$$\begin{array}{l} \Gamma,\ X{\leq}U{:}K_1 \vdash S_2 \leq T_2 \quad \text{and} \\ \Gamma \vdash \forall X{\leq}U{:}K_1.S_2 \in \star. \end{array}$$

By proposition 2.4.6, $\Gamma,\ X{\leq}U{:}K_1 \vdash S_2 \in \star$. Let $P \in [\![U]\!]_\eta$. Then, by the definition of satisfaction, $\eta[X{\leftarrow}P] \models \Gamma,\ X{\leq}U{:}K_1$. Now, by the induction hypothesis, it follows that $[\![S_2]\!]_{\eta[X\leftarrow P]} \subseteq^\star [\![T_2]\!]_{\eta[X\leftarrow P]}$. Hence, $\bigcap^\star_{P \subseteq^{K_1} [\![U]\!]_\eta} [\![S_2]\!]_{\eta[X\leftarrow P]} \subseteq^\star \bigcap^\star_{P \subseteq^{K_1} [\![U]\!]_\eta} [\![T_2]\!]_{\eta[X\leftarrow P]}$. Consequently, $[\![\forall X{\leq}U{:}K_1.S_2]\!]_\eta \subseteq^\star [\![\forall X{\leq}U{:}K_1.T_2]\!]_\eta$.

S-OAbs We are given $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash S \leq T$ and $\Gamma \vdash \Lambda X{:}K_1.S \in K$. By the syntax-directedness of kinding, $K \equiv K_1 {\rightarrow} K_2$ and $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash S \in K_2$ for some K_2. If $P \in [\![K_1]\!]_\eta$ then $\eta[X{\leftarrow}P] \models \Gamma,\ X{\leq}\top^{K_1}{:}K_1$. Now by the induction hypothesis, $[\![S]\!]_{\eta[X\leftarrow P]} \subseteq^{K_2} [\![T]\!]_{\eta[X\leftarrow P]}$. Then $\lambda P \in [\![K_1]\!].\, [\![S]\!]_{\eta[X\leftarrow P]} \subseteq^{K_1\rightarrow K_2} \lambda P {\in} [\![K_1]\!].\, [\![T]\!]_{\eta[X\leftarrow P]}$. Consequently, $[\![\Lambda X{:}K_1.S]\!]_\eta \subseteq^{K_1\rightarrow K_2} [\![\Lambda X{:}K_1.T]\!]_\eta$.

S-OApp By induction hypothesis, using the syntax-directedness of kinding.

S-Meet-G By the induction hypothesis and lemma 5.3.1(3).

S-Meet-LB By lemma 5.3.1(4). □

The *type context* Γ/TV *obtained from a context* Γ is defined in the obvious way:

$$\begin{array}{rcl} \emptyset/TV & = & \emptyset, \\ (\Gamma,\ x{:}T)/TV & = & \Gamma/TV, \\ (\Gamma,\ X{\leq}T{:}K)/TV & = & \Gamma/TV,\ X{\leq}T{:}K. \end{array}$$

THEOREM 5.4.5 Let $\eta_1 \models \Gamma$ and $\eta_2 \models \Gamma$, such that $\eta_1/TV = \eta_2/TV$ and, for all $x{\in}\mathrm{dom}(\Gamma)$, $\eta_1(x)\ \{\llbracket \Gamma(x) \rrbracket_{\eta}\}\ \eta_2(x)$, where $\eta = \eta_1/TV$. Then $\llbracket e \rrbracket_{\eta_1}\ \{\llbracket T \rrbracket_{\eta}\}\ \llbracket e \rrbracket_{\eta_2}$.

PROOF: From $\Gamma \vdash e \in T$ it follows by the well-kindedness of typing (proposition 2.4.20) that $\Gamma \vdash T \in \star$. Note that $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\Gamma/TV)$; then, by strengthening (lemma 2.4.5), $\Gamma/TV \vdash T \in \star$. Note also that $\eta \models \Gamma/TV$; by lemma 5.4.1, $\llbracket T \rrbracket_{\eta}$ is defined and in $\llbracket \star \rrbracket$. We often use this fact implicitly in the following. The proof now proceeds by induction on a derivation of $\Gamma \vdash e \in T$.

T-VAR From the assumption that for every x in $\mathrm{dom}(\Gamma)$. $\eta_1(x)\ \{\llbracket \Gamma(x) \rrbracket_{\eta}\}\ \eta_2(x)$.

T-ABS We are given that $\Gamma,\, x{:}T_1 \vdash e \in T_2$. Suppose that $p \in D$ and $q \in D$ are such that $p\ \{\llbracket T_1 \rrbracket_{\eta}\}\ q$. Then $\eta_1[x{\leftarrow}p] \models \Gamma,\, x{:}T_1$ and $\eta_2[x{\leftarrow}q] \models \Gamma,\, x{:}T_1$. By the induction hypothesis, $\llbracket e \rrbracket_{\eta_1[x\leftarrow p]}\ \{\llbracket T_2 \rrbracket_{\eta}\}\ \llbracket e \rrbracket_{\eta_2[x\leftarrow q]}$, that is, $\llbracket |\mathrm{erase}(e)| \rrbracket_{\eta_1[x\leftarrow p]}\ \{\llbracket T_2 \rrbracket_{\eta}\}\ \llbracket |\mathrm{erase}(e)| \rrbracket_{\eta_2[x\leftarrow q]}$. From lemma 5.2.1(2) it follows that $\llbracket \mathrm{fun}^{\star}\,(x)\ |\mathrm{erase}(e)| \rrbracket_{\eta_1} \cdot p\ \{\llbracket T_2 \rrbracket_{\eta}\}\ \llbracket \mathrm{fun}^{\star}\,(x)\ |\mathrm{erase}(e)| \rrbracket_{\eta_2} \cdot q$, that is, $\llbracket \lambda x{:}T_1.e \rrbracket_{\eta_1} \cdot p\ \{\llbracket T_2 \rrbracket_{\eta}\}\ \llbracket \lambda x{:}T_1.e \rrbracket_{\eta_2} \cdot q$. Since p and q were chosen freely, we have that$\llbracket \lambda x{:}T_1.e \rrbracket_{\eta_1}\ \{\llbracket T_1 \rrbracket_{\eta} {\rightarrow} \llbracket T_2 \rrbracket_{\eta}\}\ \llbracket \lambda x{:}T_1.e \rrbracket_{\eta_2}$, in other words $\llbracket \lambda x{:}T_1.e \rrbracket_{\eta_1}\ \{\llbracket T_1 {\rightarrow} T_2 \rrbracket_{\eta}\}\ \llbracket \lambda x{:}T_1.e \rrbracket_{\eta_2}$.

T-APP By the induction hypothesis, using the fact that $\llbracket f a \rrbracket_{\eta} = \llbracket f \rrbracket_{\eta} \llbracket a \rrbracket_{\eta}$.

T-TABS We are given that $\Gamma,\, X{\leq}T_1{:}K_1 \vdash e \in T_2$. Suppose that $P \subseteq^{K_1} \llbracket T_1 \rrbracket_{\eta}$. Then it follows that $\eta_1[X{\leftarrow}P] \models \Gamma,\, X{\leq}T_1{:}K_1$ and that $\eta_2[X{\leftarrow}P] \models \Gamma,\, X{\leq}T_1{:}K_1$. Since $\llbracket e \rrbracket_{\eta_1} = \llbracket e \rrbracket_{\eta_1[X\leftarrow P]}$, we have by the induction hypothesis that $\llbracket e \rrbracket_{\eta_1}\ \{\llbracket T_2 \rrbracket_{\eta[X\leftarrow P]}\}\ \llbracket e \rrbracket_{\eta_2}$. Since P was chosen freely, we have that $\llbracket e \rrbracket_{\eta_1}\ \{\bigcap^{\star}_{P \subseteq^{K_1} \llbracket T_1 \rrbracket_{\eta}} \llbracket T_2 \rrbracket_{\eta[X\leftarrow P]}\}\ \llbracket e \rrbracket_{\eta_2}$. Now the result follows from the definition $\llbracket - \rrbracket_{\eta}$ and the fact that $\llbracket \lambda X{\leq}T_1{:}K_1.e \rrbracket_{\eta} = \llbracket e \rrbracket_{\eta} = \llbracket e \rrbracket_{\eta[X\leftarrow P]}$.

T-TAPP We are given that $\Gamma \vdash f \in \forall X{\leq}T_1{:}K_1.T_2$ and that $\Gamma \vdash S \leq T_1$. By the induction hypothesis, $\llbracket f \rrbracket_{\eta_1}\ \{\llbracket \forall X{\leq}T_1{:}K_1. \rrbracket_{\eta}\}\ \llbracket f \rrbracket_{\eta_2}$, which means that, $\llbracket f \rrbracket_{\eta_1}\ \{\bigcap^{\star}_{P \subseteq^{K_1}} \llbracket T_2 \rrbracket_{\eta[X\leftarrow P]}\}\ \llbracket f \rrbracket_{\eta_2}$. By the well-kindedness of typing, syntax directedness of kinds, and well-kindedness of subtyping, $\Gamma \vdash S \in K_1$, and, by soundness of subtyping, $\llbracket S \rrbracket_{\eta} \subseteq^{K_1} \llbracket T_1 \rrbracket_{\eta}$. So we have $\llbracket f \rrbracket_{\eta_1}\ \{\llbracket T_2 \rrbracket_{\eta[X\leftarrow \llbracket S \rrbracket_{\eta}]}\}\ \llbracket f \rrbracket_{\eta_2}$, which, by lemma 5.4.2 and the fact that $\llbracket f \rrbracket_{\eta} = \llbracket f S \rrbracket_{\eta}$ for any η, is $\llbracket f S \rrbracket_{\eta_1}\ \{\llbracket T_2[X{\leftarrow}S] \rrbracket_{\eta}\}\ \llbracket f S \rrbracket_{\eta_2}$.

T-FOR Immediate from the induction hypothesis, because $\mathrm{erase}(e[X{\leftarrow}S]) = \mathrm{erase}(e)$.

T-MEET We are given that $\Gamma \vdash \mathrm{ok}$ and also $\Gamma \vdash e \in T_i$ for each i in $\{1\,..\,n\}$. The result follows by the induction hypothesis and the definitions of $\bigcap^{\star}$ and $\llbracket - \rrbracket_{\eta}$.

T-SUB We are given that $\Gamma \vdash e \in S$ and that $\Gamma \vdash S \leq T$. By the induction hypothesis, $\llbracket e \rrbracket_{\eta_1}\ \{\llbracket S \rrbracket_{\eta}\}\ \llbracket e \rrbracket_{\eta_2}$. By well-kindedness of typing $\Gamma \vdash S \in \star$, and by the soundness of subtyping $\llbracket S \rrbracket_{\eta} \subseteq^{\star} \llbracket T \rrbracket_{\eta}$. Hence, $\llbracket e \rrbracket_{\eta_1}\ \{\llbracket T \rrbracket_{\eta}\}\ \llbracket e \rrbracket_{\eta_2}$. □

Corollary 5.4.6 (*Soundness of typing*) Let η be an environment such that $\eta \models \Gamma$. Then $\Gamma \vdash e \in T$ implies $[\![e]\!]_\eta \in \mathrm{dom}([\![T]\!]_\eta)$.

Proof: Take $\eta_1 = \eta_2 = \eta$. □

Chapter 6

Multiple Inheritance

6.1 Introduction

This chapter is extracted from [CP93]. The reader is invited to read [Bru94, FM94, PT94] for a complete account on the foundations of object-oriented programming. Here we intend to illustrate how the concept of multiple inheritance is captured by intersection types. We use records and record types which are not part of the syntax of $F^{\omega}_{\wedge}$. The reader is referred to [Car92] for an implementation of records in a typed lambda calculus.

Informally, in class based object-oriented programming there are entities called objects which are organised in classes, and each class of objects is associated with a set of functions or methods. This set of methods is known as the interface of the objects of a class. The use of a method of the interface to access an object is called message passing. A suitable type theory for an object-oriented programming discipline should prevent access to objects other than through the corresponding interface. This protection against illegal access is known as encapsulation.

Existing classes may be used to create new ones. In this way, classes are organised in a hierarchy or genealogy, where ancestors are super-classes and descendants are subclasses. The mechanism through which a subclass of objects use methods of a superclass is known as inheritance. So far we used the word class as a synonym of collection. In the sequel the word class is used formally to refer to a term of the $F^{\omega}_{\wedge}$ language.

The common goal of studies in this area is to prove the safety of a type system describing a set of high-level syntactic constructs for object encapsulation, message passing, and inheritance. Our approach consists of translating the high-level syntax into a more conventional λ-calculus, whose own type-safety is established separately; the soundness of the typing rules for the object features then follows from the soundness of the target system. So, for example, the keyword `new`, which by itself does not represent any entity in an object-oriented programming language, is interpreted as a term which given suitable arguments creates an object. One advantage of this style is that we can verify type safety automatically using a type checker for the underlying λ-calculus.

The object model of Pierce and Turner [PT94], on which the present study is based, encodes objects as expressions of $F^{\omega}_{\leq}$, an extension of Girard's System F^{ω} [Gir72] with bounded quantification. Given a description `M` of a public interface — the names and types of a set of methods — the type `Object(M)` denotes the type of objects satisfying this description. Technically, object interfaces are type operators of kind $\star{\rightarrow}\star$, maps from types to types, and `Object`$\in (\star{\rightarrow}\star) \rightarrow \star$ is a higher-order constructor. See sections 6.2 and 6.3 for details and [PT94] for a longer discussion. For example, if `PointM` describes the interface of one-dimensional point objects responding to the messages `setX` and `getX` then `Object(PointM)` is the type of points. Associated with each such collection of objects is a group of functions for sending messages, with types like:

```
Point'setX  :  All(M < PointM)
                         Object(M) -> Int -> Object(M)
```

The bounded quantifier `All(M<PointM)` expresses the fact that the message `setX` can actually be sent to any object whose interface refines the interface of points. Given such an object and an integer representing its new x-coordinate, `Point'setX` returns a new object with an appropriately updated position. The foregoing accounts for the fundamental features of object encapsulation and interface refinement (and correctly handles their interaction; this is the difficult part). But it omits some characteristic features of popular object-oriented languages, notably inheritance.

In general terms, inheritance is a mechanism allowing the implementations of different sorts of objects that share some of their behavior to be factored so that the common behavior is written just once.

In our framework a class is a term containing the type of the internal representation of objects its objects, a default value for the private data and an implementation of the methods. Therefore, a class can be used in two ways: as a template for creating new objects, because it has a default value and the set of methods which is all that is needed to create an object, and as the basis for defining subclasses by incremental extension of the set of methods. If `M` is an object interface, then `Class(M)` is the type of classes that can be used to create objects of type `Object(M)`. The polymorphic function `new` maps a class into a new object:

```
new : All(M:*->*) Class(M) -> Object(M)
```

Another function, `extend`, takes an existing class and a description of some new methods and constructs a new class combining the old and new behaviors:

```
extend : All(SuperM:*->*)
         All(SelfM<SuperM)
           Class(SuperM) -> ... -> Class(SelfM)
```

(The details hidden here by ... are revealed in the following section and section 6.3.) The bounded quantifier `All(SelfM<SuperM)` ensures that the interface of the new class refines that of the old: `Object(SelfM)` is a subtype of `Object(SuperM)`.

To handle multiple inheritance in this setting, we must enrich the `extend` function to take two or more superclasses as arguments. Consider the case where the new class inherits methods from two superclasses. This version — call it `extend2` (there will be an analogous one for each n) — should have a type like:

```
extend2 : All(SuperM1:*->*)    % the interface of one superclass
          All(SuperM2:*->*)    % the interface of the other superclass
          All(SelfM<???)       % the interface of the class being built
               Class(SuperM1)  % the first superclass
            -> Class(SuperM2)  % the second superclass
            -> ...             % (how to build the new class)
            -> Class(SelfM)    % the new class itself
```

But the upper bound of `SelfM` presents a problem: we must ensure that `SelfM` is a subtype of *both* `SuperM1` and `SuperM2`, which falls outside the expressive scope of our target λ-calculus $F^\omega_\leq$.

Intuitively, what we want to write is

```
extend2 : All(SuperM1:*->*) All(SuperM2:*->*)
          All(SelfM < SuperM1 "and" SuperM2)
               Class(SuperM1)
            -> Class(SuperM2)
            -> ...
            -> Class(SelfM)
```

where, informally, `"and"` forms the conjunction of the two superclass specifications. Fortunately, a type constructor with exactly this meaning has already appeared in the literature. First-order type systems with intersection types have been investigated by the group in Torino [CDC78, BCD83] and elsewhere. (See [CC90] for background and further references.) A second-order λ-calculus with intersection types was studied by Pierce [Pie91]. The calculus needed here is the ω-order extension of this system.

A type system combining intersection types with a powerful form of polymorphism is of independent interest. Reynolds [Rey88] has argued that intersection types can form the basis of elegant language designs. But his Forsythe language has only a first-order type system, and thus lacks some of the expressive possibilities of polymorphic languages like ML. Our work represents a step toward a synthesis of these styles of language design.

The following section shows some examples of multiple inheritance using a simple high-level syntax, and section 6.3 develops an implementation of inheritance in this setting.

6.2 An example of multiple inheritance

We begin by recalling the encodings of some basic concepts of object-oriented programming in $F^\omega_\leq$ and showing a simple example of multiple inheritance in this setting.

In this setting, an object *interface specification* is modelled as a function from types to types, describing the behaviors of a collection of methods as transformations on the object's internal state. For example, the interface of one-dimensional point objects supporting the messages `getX`, `setX`, and `bump` is captured by the type operator

```
# PointM = Fun(Rep){| setX: Rep->Int->Rep,
#                     getX: Rep->Int,
#                     bump: Rep->Rep |};
PointM : *->*
```

which expresses the fact that the `getX` method of a point interrogates its internal state and returns an integer, that the `set` method transforms the internal state and a new position into an updated internal state, and that `bump`, which increases the position by one, maps one internal state to another. (The # in the left-hand margin indicates that this expression has been checked by our implementation; the typechecker's response follows.) The abstraction over the type `Rep` of the internal state hides the actual internal state from outside view. Concretely, a point whose internal state type is `{|x:Int|}` — a one-field record containing an integer — will contain a record of methods with types

```
   {|  setX: {|x:Int|} -> Int -> {|x:Int|},
       getX: {|x:Int|} -> Int,
       bump: {|x:Int|} -> {|x:Int|}         |}
```

while a point whose internal state type is richer, say `{|x:Int,y:Int|}`, will have correspondingly richer concrete types for its methods:

```
   {|  setX: {|x:Int,y:Int|} -> Int -> {|x:Int,y:Int|},
       getX: {|x:Int,y:Int|} -> Int,
       bump: {|x:Int,y:Int|} -> {|x:Int,y:Int|}          |}
```

Externally, we expect the difference between these two to be invisible; thus, the public interface to the methods, `PointM`, abstracts away from any particular representation type. Both point objects are elements of the type `Object(PointM)`. (For present purposes, it is not important how `Object` itself is defined. C.f. [PT94, HP95].)

New objects are created by applying the polymorphic function `new` to a *class*. Given an interface `M` and a class for this interface — that is, a class whose *instances* are objects with interface `M` — `new` creates and returns such an object. New classes, in turn, are created by applying the polymorphic function `extend` to an existing class along with a specification of an incremental change to its behavior:

```
extend = <val>
       : All(SuperM)
         All(SelfM<SuperM)
         All(SelfDiffR)
              (Class SuperM)
           -> SelfDiffR
```

```
          -> (All(FinalR)
                (Extractor FinalR SelfDiffR)
              ->(SuperM FinalR)
              ->(SelfM FinalR)
              ->(SelfM FinalR))
          -> (Class SelfM)
```

In detail, the arguments expected by `extend` comprise:

- The interface `SuperM` of the existing class.
- The interface `SelfM` of the new class that will be returned by `extend`.
- The type `SelfDiffR`, which describes the difference between the representation of the superclass (whatever it may be) and the representation of the new class. In conventional terminology, this is the set of new instance variables introduced by the subclass.
- The superclass itself — an element of `Class(SuperM)` (our typechecker prints it as `Class SuperM`).
- An initial value — an element of `SelfDiffR` — for the new part of the state.
- A polymorphic "method builder" function.

Given all these, `extend` returns a class for the interface `SelfM`.

The method builder function, which does the work of constructing the vector of methods to be used in instances of the new class, must itself take several parameters:

- The "final" representation type `FinalR`, which is fixed at the moment when `new` is applied to a class.
- An "extractor," which provides a mapping back and forth between the final representation type and the local representation type, allowing the local methods to access the part of the state that interests them.
- The "super methods" of the existing class.
- The "self methods" of the new class, which are used to model the characteristic object-oriented feature of "sending a message to `self`."

Given these, the method builder must return a collection of methods for the new object.

For uniformity, let us assume that there is just one base class — the class of "things," whose instances are objects with no behavior at all:

```
# ThingM = Fun(Rep) {| |};
ThingM : *->*
thingClass = <val> : Class ThingM
```

To build a class of points extending `thingClass`, we first choose the "local" part of the representation of points.

```
# PointDiffR = {| x:Int |};
PointDiffR : *
```

Now we create `pointClass` by applying `extend` as follows (see section 6.3 for more details).

```
# pointClass =
#  extend
#   ThingM                            % superclass interface
#   PointM                            % interface for new class
#   PointDiffR                        % local state type
#   thingClass                        % the superclass itself
#   { x = 0 }                         % initial value for local state
#   (fun(FinalR)                      % "method builder" function...
#    fun(e: Extractor FinalR PointDiffR)  % mapping a "state extractor"
#    fun(super: ThingM FinalR)             % and the "super methods"
#    fun(self: PointM FinalR)              % and the "self methods"
#      {getX = fun(s:FinalR)               % to a getX method
#                 (e.get s).x,             %   that returns
#                                          %   the local x field
#       setX = fun(s:FinalR)               % and a setX method
#                 fun(i:Int) e.put s {x=i},%   that overwrites
#                                          %   the x field
#       bump = fun(s:FinalR)               % and a bump method
#                 self.setX s              %   that calls setX on self
#                   (plus (self.getX s) 1) %   to set x to one more
#      });                                 %   than self.getX
pointClass = <val> : Class PointM
```

Of course, this definition of `pointClass` is quite verbose. It is not hard to design higher-level syntax for objects, message passing, and class extension that looks like ordinary object-oriented source code, but since we are building a foundational model here, we prefer the low-level notation.

Similarly, we can define the interface for "colored objects" — objects supporting the messages `setC` and `getC` — as follows:

```
# ColoredM = Fun(Rep) {| setC: Rep->Color->Rep, getC: Rep->Color |};
ColoredM : *->*
```

Again, one instance variable suffices to represent the color of a colored object:

```
# ColoredDiffR = {| c:Color |};
ColoredDiffR : *
```

A class of colored objects can now be created by extending `thingClass` as we did to build `pointClass`:

```
# coloredClass =
#   extend ThingM ColoredM ColoredDiffR thingClass
#   { c = black }
#   (fun(FinalR)
#    fun(e: Extractor FinalR ColoredDiffR)
#    fun(super: ThingM FinalR)
#    fun(self: ColoredM FinalR)
#      {getC = fun(s:FinalR) (e.get s).c,
#       setC = fun(s:FinalR) fun(newc:Color) e.put s {c=newc}
#      });
coloredClass = <val> : Class ColoredM
```

Now we have reached the point where we can use *multiple* inheritance to combine the classes of point objects and colored objects, yielding a new class of colored points. The interface of colored points contains all the messages of both superclasses:

```
# CPointM = Fun(Rep) {| setX: Rep->Int->Rep,
#                       getX: Rep->Int,
#                       bump: Rep->Rep,
#                       setC: Rep->Color->Rep,
#                       getC: Rep->Color |};
CPointM : *->*
```

For this simple implementation, no additional instance variables are needed: we can set `CPointDiffR = {| |}`.

To make the example more interesting, we take the methods `getX`, `setC`, and `getC` unchanged from the superclasses, while *overriding* the definition of `setX` so that, in addition to setting the `x` coordinate as usual, it also sets the color to, say, blue:

```
# cpointClass =
#   extend2 PointM ColoredM CPointM
#     CPointDiffR pointClass coloredClass { }
#     (fun(FinalR)
#      fun(e: Extractor FinalR CPointDiffR)
#      fun(super1: PointM FinalR)
#      fun(super2: ColoredM FinalR)
#      fun(self: CPointM FinalR)
#       {setX = fun(s:FinalR) fun(i:Int)% the new setX method:
#         let s1 = super1.setX s i in   % use pointClass's setX
#                                       % to set position
#         let s2 = super2.setC s1 blue  % and coloredClass's setC
#                                       % to set color
#         in s2   end end,
#        getX = super1.getX,            % copy all the remaining
#        bump = super1.bump,            % methods from the
#        setC = super2.setC,            % appropriate superclass
#        getC = super2.getC
#       });
cpointClass = <val> : Class CPointM
```

Here, the low level at which we are working is reflected in the fact that the old methods `getX`, `bump`, `setC`, and `getC` must be copied explicitly from the superclasses to the new class. Introducing high-level syntax for multiple inheritance would, of course, raise all the usual questions (must each inherited method appear in only one of the superclasses? if it appears in more than one, which should be copied to the subclass? etc.), for which the usual solutions will apply.

To test what we have done, let's build a colored point and send it some messages:

```
# p = new CPointM cpointClass;
p = <val> : Object CPointM
# Colored'getC CPointM p;
black : Color
# p1 = Point'bump CPointM p;
p1 = <val> : Object CPointM
# Point'getX CPointM p1;
1 : Int
# Colored'getC CPointM p1;
blue : Color
```

Note that sending our colored point the `bump` method has the effect of changing its color to blue: the overridden behavior of the `setX` method is observable in the behavior of `bump` method, even though `bump` was not redefined in the subclass.

6.3 Encoding multiple inheritance

We close with a full implementation of the `extend2` function, generalising the `extend` function in section 7 of [PT94]. As we suggested in the introduction, an intersection type must be used at one point (marked `***` in the definition of `extend`) to obtain a sound typing; the rest is straightforward.

This implementation of classes and inheritance makes the local state of each class inaccessible both to clients of objects and to methods defined in subclasses. Other variations are possible; we chose this one to simplify the presentation of section 6.2.

If `M` is an object interface — an operator of kind `*->*` — then `Class(M)` is the set of classes whose instances have type `Object(M)`. Each such class consists of a local representation type `MyR` (whose identity is hidden by an existential quantifier), an element `initstate`∈`MyR` that is used as the initial value of the state in new objects created from this class, and a function `buildM` that can be used to construct the methods of the new objects:

```
# Class =
#   Fun(M:*->*)
#     Some(R)
#       {| initstate: R,
#          buildM: ClassMethods M R |};
Class : (*->*)->*
```

To cope with different representations of local state in subclasses, the method-building function is abstracted on two parameters: a type `FinalR` representing the "full" state of an eventual subclass, and an "extractor" giving access to the components of interest to the methods being built. The method builder is also abstracted on a collection of `self`-methods of the same types as its own methods. Given these, it yields a concrete collection of methods specialized to work properly in an object with representation type `FinalR`:

```
# ClassMethods =
#   Fun(MyM:*->*)
#   Fun(MyR)
#     All(FinalR)
#     (Extractor FinalR MyR) ->
#     (MyM(FinalR)) ->
#     (MyM(FinalR));
ClassMethods : (*->*)->*->*
```

Finally, an extractor is just a pair of maps, `get` and `put`.

```
# Extractor = Fun(SS) Fun(TT) {| get: SS->TT, put: SS->TT->SS |};
Extractor : *->*->*
```

Intuitively, `get` extracts the "superclass part" of an element of a subclass's state, while `put` overwrites the superclass part, yielding a new subclass state.

For example, the point class of section 6.2 can be defined directly (rather than as an extension of `thingClass`) as follows:

```
# pointClass =
#   < {|x:Int|},
#     {initstate = {x=0},
#      buildM = fun(FinalR)
#               fun(e: Extractor FinalR {|x:Int|})
#               fun(self: PointM FinalR)
#                 {getX = fun(s:FinalR) (e.get s).x,
#                  setX = fun(s:FinalR) fun(i:Int) e.put s {x=i},
#                  bump = fun(s:FinalR) e.put s {x = plus 1 (e.get s).x}
#                 }}
#    > : Class PointM;
pointClass = <val> : Class PointM
```

(We use the ascii syntax "`<R,b>:T`" for introducing elements of existential types: `R` is the hidden witness type, `b` is the body, and `T` is the existential type where the result is to live. The corresponding elimination form is written "`open e as <R,x> in b`.")

A class with two superclasses generates objects whose internal states have three parts: one for each superclass and one for the new components local to the class itself. For example, an instance of cpointClass contains a point state of type {|x:Int|}, a colored-object state of type {|c:Color|}, and an empty local state. The extend2 function takes two classes, an initial local state, and a function for incrementally building a collection of new methods from the old ones, and constructs a subclass of this form.

```
# extend2 =                             %Arguments:
#  fun(SuperM1: *->*)                       %first superclass interface
#  fun(SuperM2: *->*)                       %second superclass interface
#  fun(MyM < SuperM1/\SuperM2)     %***     %new subclass interface
#  fun(MyLocalR: *)                         %local state type
#  fun(superClass1: Class SuperM1)          %first superclass
#  fun(superClass2: Class SuperM2)          %second superclass
#  fun(myinitstate: MyLocalR)               %initial local state
#  fun(mymethods:                           %incr method extension fun
#                                           %with arguments
#         All(FinalR)                         %a final rep type
#           (Extractor FinalR MyLocalR) ->    %an extractor
#                                             %for the local state
#           (SuperM1(FinalR)) ->              %the first superclass's
#                                             %methods
#           (SuperM2(FinalR)) ->              %the second superclass's
#                                             %methods
#           (MyM(FinalR)) ->                  %the self-methods
#           (MyM(FinalR)))                    %returning the new methods
#  open superClass1
#       as <SuperR1,superData1> in          %open the first superclass
#  open superClass2
#       as <SuperR2,superData2> in          %open the second superclass
#  let MyR =                                %define the new state type
#       Triple SuperR1 SuperR2 MyLocalR in  %(a triple)
#   <                                    %Result: a new class
#    MyR,                                   %with state type MyR
#    {initstate =                           %and initial state
#         triple SuperR1 SuperR2 MyLocalR   %a triple of
#                    (superData1.initstate)   %first super's
#                                             %initial state
#                    (superData2.initstate)   %second super's
#                                             %initial state
#                    myinitstate,             %local initial state
#     buildM =                              %and method-builder,
#                                           %a fun with args
#      fun(FinalR)                            %a final rep type
#      fun(e: Extractor FinalR MyR)           %an extractor
#      fun(self: MyM(FinalR))                 %and a collection
#        let eself =                          %of self-methods...
#          composeExtractors
#             FinalR MyR MyLocalR e
#             (extract3of3 SuperR1 SuperR2 MyLocalR) in
#        let esuper1 =
#          composeExtractors
#             FinalR MyR SuperR1 e
#             (extract1of3 SuperR1 SuperR2 MyLocalR) in
#        let esuper2 =
#          composeExtractors
#             FinalR MyR SuperR2 e
#             (extract2of3 SuperR1 SuperR2 MyLocalR) in
#        mymethods FinalR eself             %returning
#                                           %methods built by mymethods
#         (superData1.buildM FinalR esuper1 self)%when applied to
#                                                %concrete methods
#         (superData2.buildM FinalR esuper2 self)%of the superclasses
#          self                                  %and the self-methods
#        end end end}
#    > : Class MyM
#   end end end;
extend2 = <val>
        : All(SuperM1)
          All(SuperM2)
          All(MyM<SuperM1/\SuperM2)
          All(MyLocalR)
               (Class SuperM1)
            -> (Class SuperM2)
            -> MyLocalR
            -> (All(FinalR)
                    (Extractor FinalR MyLocalR)
                  ->(SuperM1 FinalR)
                  ->(SuperM2 FinalR)
                  ->(MyM FinalR)
                  ->(MyM FinalR))
            -> (Class MyM)
```

This definition uses a utility function for composing extractors in the obvious way:

```
# composeExtractors =
#   fun(T1) fun(T2) fun(T3)
#     fun(e1: Extractor T1 T2)
#       fun(e2: Extractor T2 T3)
#         {get = fun(t1:T1) e2.get (e1.get t1),
#          put = fun(t1:T1) fun(t3:T3)
#                  e1.put t1 (e2.put (e1.get t1) t3)};
composeExtractors = <val>
                  : All(T1)
                    All(T2)
                    All(T3)
                         (Extractor T1 T2)
                      -> (Extractor T2 T3)
                    -> {|get:T1->T3, put:T1->T3->T1|}
```

For forming triples, we use the type abbreviation

```
# Triple = Fun(T1) Fun(T2) Fun(T3) {| fst:T1, snd:T2, thd:T3 |};
Triple : *->*->*->*
```

with the constructor

```
# triple =
#   fun(T1) fun(T2) fun(T3)
#   fun(t1:T1) fun(t2:T2) fun(t3:T3)
#     {fst=t1, snd=t2, thd=t3};
triple = <val>
       : All(T1)
         All(T2)
         All(T3)
           T1 -> T2 -> T3 -> {|fst:T1, snd:T2, thd:T3|}
```

and the projections

```
# extract1of3 =
#  fun(T1) fun(T2) fun(T3)
#    {get = fun(p: Triple T1 T2 T3) p.fst,
#     put = fun(p: Triple T1 T2 T3)
#             fun(t:T1)
#                {fst=t, snd=p.snd, thd=p.thd} };
extract1of3 = <val>
            : All(T1)
              All(T2)
              All(T3)
              {|get: (Triple T1 T2 T3)->T1,
                put: (Triple T1 T2 T3)->T1->{|fst:T1, snd:T2, thd:T3|}|}
```

and `extract2of3` and `extract3of3`, which are defined similarly.

A slightly different formulation of the `extend2` function provides an alternative perspective on its behavior. The original `extend2` is parametric on three class interfaces, `SuperM1`, `SuperM2`, and `MyM`, where `MyM` is constrained to refine both `SuperM1` and `SuperM2`. The type of the following function `extend2'` emphasizes the fact that `MyM` is typically formed by *adding* some new methods to those given by `SuperM1` and `SuperM2`: it is parameterized on `SuperM1`, `SuperM2`, and a "partial interface" `MyOwnM`, which is conjoined with the other two to form `MyM`:

```
# extend2' =
#  fun(SuperM1: *->*)                     % first superclass interface
#  fun(SuperM2: *->*)                     % second superclass interface
#  fun(MyOwnM: *->*)                      % new methods specification
#  let MyM = SuperM1/\SuperM2/\MyOwnM in  % new class interface
#     % ...(the rest, as before)...
extend2' = <val>
```

```
: All(SuperM1)
  All(SuperM2)
  All(MyOwnM)
  All(MyLocalR)
       (Class SuperM1)
    -> (Class SuperM2)
    -> MyLocalR
    -> (All(FinalR)
            (Extractor FinalR MyLocalR)
          ->(SuperM1 FinalR)
          ->(SuperM2 FinalR)
          ->(SuperM1/\(SuperM2/\MyOwnM) FinalR)
          ->(SuperM1/\(SuperM2/\MyOwnM) FinalR))
    -> (Class (SuperM1/\(SuperM2/\MyOwnM)))
```

Note that all of the quantifiers in this version are unbounded: bounded quantification has been replaced by unbounded quantification and intersection.

Part II

First-Order Subtyping

Chapter 7

Implicit and Explicit Subtyping

7.1 Introduction

In the analysis of λ-calculi we can distinguish between two main groups of systems, namely, explicitly typed systems, usually called *à la Church* and implicitly typed systems also called *à la Curry.* In the implicitly typed systems type free lambda terms are assigned a type and this is why these calculi *à la Curry* are sometimes called systems of type assignment. On the other hand, in the explicitly typed systems the terms are not terms of the type-free λ-calculus but terms themselves containing type information. To illustrate the difference we write the canonical example of typing the corresponding identity term in both styles.

$$\vdash_{Curry} \lambda x.x \in \sigma{\rightarrow}\sigma$$
$$\vdash_{Church} \lambda x{:}\sigma.x \in \sigma{\rightarrow}\sigma.$$

Observe that the Church style term has extra typing information, namely '$:\sigma$'. This explicit mention of types in a term makes it easier to decide whether a term has a certain type. For some systems à la Curry this question is undecidable. See [Bar92] for some examples. In these systems the problem of finding a type for a given term involves solving sets of equations. (See [Wan87] for an elegant and concise algorithm of type inference for simply typed λ-calculus à la Curry).

The idea of subtype appears quite naturally in programming languages. If we think of types as sets, we can easily picture what a subtype could be. Informally, we can say that a type σ is a subtype of τ ($\sigma \leq \tau$) if any element of σ can be seen as an element of τ. We say *can be seen as* and not directly *is* because the act of considering an element of type σ as an element of type τ might hide some transformation. Consider for example the types *Int* and *Real* of integers and real numbers respectively. Usually, on a computer, integer numbers are represented in a different way than real numbers are; even if we might think of the integers as a subset of the real numbers, there is a translation going on. The act of considering an element of type σ as an element of type τ will be called *coercion.* In other words, we say that an element of type σ is coerced into an element of type τ. Somehow an element of type σ has enough information to be seen as an element of type τ.

While dealing with coercions we can again distinguish between an explicit style and an implicit style. A style with explicit coercions means that coercions are explicitly indicated and in an implicit style, as the name suggests, coercions are left implicit. In systems including subtyping there is usually a rule for typing coerced terms. Then, in an explicit style the coercion rule might look as follows.

$$\frac{\Gamma \vdash M \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash c_{\sigma\tau}{<}M{>} \in \tau} \qquad (\textsc{Coercion})$$

Similarly, in an implicit style the corresponding rule is as follows.

$$\frac{\Gamma \vdash M \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash M \in \tau} \qquad (\textsc{Subsumption})$$

From the previous discussion it follows that we can split subtyping systems into four main groups combining implicit or explicit typing with implicit or explicit coercions. Explicit coercions have been used as a way of giving semantics to systems with implicit coercions in [CG92]. In [CL91], PER models for Quest, a higher order lambda calculus with subsumption, and Quest$_C$, a higher order lambda calculus with coercion, are studied.

An implicit coercion is motivated by the fact that the same term can be considered as belonging to two different types without performing any change in the term, as for example is the case when one of the types is included in the other (with the intuitive idea of set inclusion), while an explicit coercion wishes to state explicitly that there is a transformation going on. We can think, for example, of a function f with the real numbers as domain, and a (sub)set A of real numbers. If x is a variable of type A, then we would like to use f on x as well, without performing any extra calculation to apply f to x.

The system $\lambda_{\leqslant}$ (lambda sub), an extension of the simply typed λ-calculus *à la Church* with subtyping, is presented in section 7.4. The extension consists of adding the previously mentioned SUBSUMPTION rule, in other words, coercions are left implicit. The subtyping relation mentioned in the rule is based on a finite set of subtyping axioms, closed under reflexivity and transitivity, and extended to arrow types in the standard way. We show that $\lambda_{\leqslant}$ satisfies the minimal types property, and we exhibit an algorithm to compute minimal types ($Alg\lambda_{\leqslant}$). Moreover, we show that type checking and type inference are decidable.

The subtyping relation is studied in section 7.2, where a method to establish whether two types are in the subtype relation is given and proven sound and complete with respect to the definition of the subtyping relation. The decidability of the predicate $\sigma \leq \tau$ was already stated in [Mit84]. Types in the subtype relation are looked at through the magnifying glass to establish the relation between the structure of σ and τ when σ is less than or equal to τ.

In section 7.5, λ_C (lambda coerce), another extension of the simply typed lambda calculus with subtyping, is introduced. This time the rule added is the previously mentioned COERCION rule. Basic properties of this system are established, and, in section 7.6 we show that the "invisible" coercions in a $\lambda_{\leqslant}$ typing statement can be uniformly reconstructed producing a legal statement of λ_C. The fact that we translate typing statements instead of typing derivations as in [CG92] and [BCGS91], avoids coherence problems.

Finally, in section 7.7, a translation of λ_C into the simply typed lambda calculus $\lambda\!\!\to$ is developed. This means that a system with two different kinds of judgements, typing judgements and subtyping judgements, is translated into a system without subtyping. The idea is to mimic λ_C inside $\lambda\!\!\to$. The translation of the typing system is straightforward; the Coercion rule is omitted. The translation of the subtyping statements is as follows: the subtyping axioms are collected as a so called environment (like a signature in ELF [AH87]), and the subtyping rules are perfectly captured by computational properties of the λ-calculus. A proof of a subtyping statement is then a $\lambda\!\!\to$-term containing constants of the environment. This translation together with the translation from $\lambda_{\leqslant}$ into λ_C, imply that subtyping can be coded into a system without subtyping.

The $\lambda\!\!\to$ that we define in section 7.7 is not exactly the one presented in [Bar92]. We prefer a formulation in which constants are syntactically different from variables, the rules prevent abstraction over constants, and there is a typing rule for constants, so that nothing that is ilegal can be derived from the rules without extra proviso in the metalanguage.

Convention 7.1.1 Throughout this chapter the metavariables α, β, γ and δ will range over type variables, σ, τ, and ρ will range over types, M, N, and P will range over terms, x, y, and z over term variables, Γ will range over contexts, and Σ over environments.

7.2 The subtyping relation

The relation $\leqslant_C$

In the present section we define the subtyping relation, $\leqslant_C$, and an algorithm, *Subtype*, to check whether two types are in the subtyping relation. In proposition 7.2.7, we prove the correctness of the algorithm *Subtype* with respect to the definition of $\leqslant_C$.

Let $\mathbb{V}$ be a set of type variables, $\mathbb{T}$ a set of types defined by

$$\mathbb{T}::=\mathbb{V} \mid \mathbb{T}\to\mathbb{T},$$

and let $C \subset \mathbb{V} \times \mathbb{V}$ a finite set of subtyping axioms, where if $(\alpha, \beta) \in C$ then α, β are different variables.

We will restrict our attention to the particular case when $C \subset \mathbb{V} \times \mathbb{V}$, given that the more general case when $C \subset \mathbb{T} \times \mathbb{T}$ could allow typing non-terminating terms like for example $(\lambda x{:}\sigma.xx)(\lambda x{:}\sigma.xx)$.

Definition 7.2.1 (*Subtyping*) The relation $\leqslant_C \subset \mathbb{T} \times \mathbb{T}$ is the smallest relation closed under the following rules.

$$\begin{array}{rclr} (\alpha, \beta)\in C & \Rightarrow & \alpha \leqslant_C \beta & \text{S-Incl}, \\ & \sigma \leqslant_C \sigma & & \text{S-Refl}, \\ \sigma \leqslant_C \tau,\ \tau \leqslant_C \rho & \Rightarrow & \sigma \leqslant_C \rho & \text{S-Trans}, \\ \sigma \leqslant_C \sigma',\ \tau' \leqslant_C \tau & \Rightarrow & \sigma'\to\tau' \leqslant_C \sigma\to\tau & \text{S-Arrow}. \end{array}$$

The S-Arrow rule deserves a close look. If we consider the relation $\leqslant_C$ as an ordering, then $\to$ is monotonic in the second argument and antimonotonic in the first argument. Intuitively, if every value of type σ can be treated as a value of type σ', then every function which maps σ' to τ also maps σ to τ.

In what follows we define the algorithm *Subtype*, which is a decision procedure for the $\leqslant_C$ relation; as it is shown in proposition 7.2.7. But first we need the following definition.

Definition 7.2.2 (*Transitive Closure of C*)

1. $(\alpha,\beta)\in C \Rightarrow (\alpha,\beta)\in \mathit{Trans}(C)$,
   (α,β) and $(\beta,\gamma)\in \mathit{Trans}(C) \Rightarrow (\alpha,\gamma)\in \mathit{Trans}(C)$.
2. $\mathit{trans}(\sigma,\tau,C) = \mathsf{true}$ if and only if (σ,τ) belongs to $\mathit{Trans}(C)$.

We can now write down the algorithm.

Definition 7.2.3 (*Subtype*) $\mathit{Subtype} : \mathbb{T}\times\mathbb{T}\times 2^{\mathbb{V}\times\mathbb{V}} \to \mathit{Bool}$

```
Subtype(σ, τ, C) =
          if $\sigma \equiv \tau$
          then true
          else if $\sigma$ and $\tau$ are variables
               then $trans(\sigma,\tau,C)$
               else if $\sigma \equiv \sigma_1\to\sigma_2$ and $\tau \equiv \tau_1\to\tau_2$
                    then $Subtype(\tau_1,\sigma_1,C)$ and $Subtype(\sigma_2,\tau_2,C)$
                    else false
```

Where $\equiv$ is the syntactic equality.

Since C is finite, $\mathit{Trans}(C)$ is also finite. Consequently, $\mathit{trans}(\sigma,\tau,C)$ is decidable. Moreover, the recursive calls have arguments of strictly smaller size. Hence, the algorithm *Subtype* always terminates.

Definition 7.2.4 The *Shadow* of a type is defined as follows.

$$\begin{array}{rcll} \mathit{Shadow}(\alpha) & = & \bullet & \text{if } \alpha\in\mathbb{V} \\ \mathit{Shadow}(\sigma\to\tau) & = & \begin{array}{c} \to \\ / \qquad \backslash \\ \mathit{Shadow}(\sigma) \qquad \mathit{Shadow}(\tau) \end{array} & \end{array}$$

The difference between the usual underlying tree structure and the shadow of a type expression is that different type variables have different underlying trees but the same shadow. Then two type expressions that only differ in their atomic subexpressions (type variables), have the same shadow.

Lemma 7.2.5 Let $\alpha,\beta\in\mathbb{V}$ and $\sigma,\tau\in\mathbb{T}$. Then,

1. If $\mathit{trans}(\alpha, \beta, C)$, then $\alpha \leqslant_C \beta$.
2. If $\sigma \leqslant_C \tau$, then $\mathit{Shadow}(\sigma) = \mathit{Shadow}(\tau)$.
3. If $\alpha \leqslant_C \beta$, then $\mathit{trans}(\alpha, \beta, C)$ or $\alpha \equiv \beta$.

PROOF:

1. By induction on the definition of $\mathit{Trans}(C)$.
2. By induction on the derivation of $\sigma \leqslant_C \tau$.
3. By induction on the derivation of $\alpha \leqslant_C \beta$. □

LEMMA 7.2.6 Let $\sigma_1, \sigma_2, \tau_1$, and $\tau_2 \in \mathbb{T}$. Then

$$\sigma_1 \rightarrow \sigma_2 \ \leqslant_C \ \tau_1 \rightarrow \tau_2 \ \text{ if and only if } \ \tau_1 \leqslant_C \sigma_1 \text{ and } \sigma_2 \leqslant_C \tau_2.$$

PROOF: From right to left, it is just the S-ARROW rule. From left to right, the proof follows by induction on the derivation of $\sigma_1 \rightarrow \sigma_2 \ \leqslant_C \ \tau_1 \rightarrow \tau_2$. Since C only contains pairs of type variables, the S-INCL rule could not have been the last rule of the derivation.

S-REFL $\sigma_1 \rightarrow \sigma_2 \ \equiv \ \tau_1 \rightarrow \tau_2$ and this means that $\tau_1 \equiv \sigma_1$ and $\sigma_2 \equiv \tau_2$. Hence, by S-REFL, it follows that $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$.

S-TRANS We are in the case that for some ρ, $\sigma_1 \rightarrow \sigma_2 \ \leqslant_C \ \rho$ and $\rho \leqslant_C \tau_1 \rightarrow \tau_2$. By lemma 7.2.5(2), ρ is of the form $\rho_1 \rightarrow \rho_2$. Then, by the induction hypothesis, $\rho_1 \leqslant_C \sigma_1$, $\sigma_2 \leqslant_C \rho_2$, $\tau_1 \leqslant_C \rho_1$, and $\rho_2 \leqslant_C \tau_2$. Then, by S-TRANS, we conclude that $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$.

S-ARROW If the last rule was S-ARROW, then the only possibility for the hypothesis is $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$. □

We can now show that the algorithm *Subtype* is correct with respect to definition 7.2.1. The correctness is split into two parts, usually called soundness and completeness. Soundness means that if $\mathit{Subtype}(\sigma, \tau, C) = \mathsf{true}$, then it is the case that $\sigma \leqslant_C \tau$. Conversely, completeness means that if $\sigma \leqslant_C \tau$, then the algorithm outputs true when called with arguments σ, τ and C.

PROPOSITION 7.2.7 (*Soundness and completeness of Subtype*)

$$\mathit{Subtype}(\sigma, \tau, C) = \mathsf{true} \text{ if and only if } \sigma \leqslant_C \tau.$$

PROOF:

$\Rightarrow$) By induction on the complexity of σ and τ.

Case 1. $\sigma, \tau \in \mathbb{V}$. Then we have to consider the following two cases.

Case 1a. If $\sigma \equiv \tau$, then, by S-REFL, $\sigma \leqslant_C \tau$.

Case 1b. If $\mathit{trans}(\sigma,\tau,C) = \mathsf{true}$, then, by lemma 7.2.5(1), we know that $\sigma \leqslant_C \tau$.

Case 2. If either σ or τ is a variable and the other one is not, then the algorithm never yields true.

Case 3. Neither σ nor τ is a variable. Again we have to consider two cases.

$\sigma \equiv \tau$. Then, by S-Refl, $\sigma \leqslant_C \tau$.

$\sigma \not\equiv \tau$. Say $\sigma \equiv \sigma_1 \to \sigma_2$ and $\tau \equiv \tau_1 \to \tau_2$. Then it is the case that $\mathit{Subtype}(\tau_1,\sigma_1,C) = \mathsf{true}$ and $\mathit{Subtype}(\sigma_2,\tau_2,C) = \mathsf{true}$.
By the induction hypothesis, $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$. Hence, due to the S-Arrow rule, $\sigma_1 \to \sigma_2 \leqslant_C \tau_1 \to \tau_2$.

$\Leftarrow$) By induction on the complexity of σ.

$\sigma \in \mathbb{V}$. Then, by lemma 7.2.5(2), τ is also a variable. By lemma 7.2.5(3), it follows that $\mathit{trans}(\sigma,\tau,C) = \mathsf{true}$ or $\sigma \equiv \tau$, and in both cases we have that $\mathit{Subtype}(\sigma,\tau,C) = \mathsf{true}$.

$\sigma \equiv \sigma_1 \to \sigma_2$. Then, by lemma 7.2.5(2), τ is of the form $\tau_1 \to \tau_2$, and, because of lemma 7.2.6, we know that $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$. Then $\mathit{Subtype}(\tau_1,\sigma_1,C) = \mathsf{true}$ and $\mathit{Subtype}(\sigma_2,\tau_2,C) = \mathsf{true}$, by the induction hypothesis. Hence, $\mathit{Subtype}(\sigma,\tau,C) = \mathsf{true}$. □

A closer look at the algorithm uncovers some proof theoretic properties of the subtyping relation $\leqslant_C$. Observe that in the algorithm the S-Trans rule is considered only at the level of variables, in other words, the S-Trans rule is never used as the last rule of a proof of a statement of the form $\sigma_1 \to \sigma_2 \leqslant_C \tau_1 \to \tau_2$. The corollary is then, that, if there exists a proof of $\sigma \leqslant_C \tau$, then there exists also a proof of $\sigma \leqslant_C \tau$ in which the applications of the S-Trans rule are only on statements of the form $\alpha \leqslant_C \beta$ and $\beta \leqslant_C \gamma$, where α, β, and γ are type variables. This fact can be read as follows: the system in which the S-Trans rule is replaced by

S-Trans' $\alpha \leqslant_C \beta,\ \beta \leqslant_C \gamma \quad \Rightarrow \quad \alpha \leqslant_C \gamma$, where $\alpha, \beta, \gamma \in \mathbb{V}$.

can prove the same subtyping statements as the original system defined in 7.2.1.

From the proof theoretic point of view there is another possible refinement that consists of restricting the application of the S-Refl rule to type variables. In other words, we could replace S-Refl by

S-Refl' $\alpha \leqslant_C \alpha$ for all $\alpha \in \mathbb{V}$.

But, from an algorithmic point of view, this is not a very satisfactory choice, because the proofs with the S-Refl rule can be shorter. The use of the S-Refl rule instead of the S-Refl' rule avoids superfluous recursive calls. For example, to prove $\alpha \to (\beta \to \alpha) \leqslant_C \alpha \to (\beta \to \alpha)$ requires two applications of the S-Arrow rule and three applications of the S-Refl' rule, while it can be proved in one step with the original S-Refl rule.

About the sets $\{\tau{\in}\mathbb{T} \mid \tau \leqslant_C \sigma\}$ and $\{\tau{\in}\mathbb{T} \mid \sigma \leqslant_C \tau\}$

In this section we focus our attention on the sets of types which are smaller and bigger than a given type with respect to $\leqslant_C$. We define simultaneously the functions *after* and *before* that, given a type, retrieve the set of bigger and smaller types respectively, as we prove in lemma 7.2.9(3).

Definition 7.2.8 *after*,*before* $:\mathbb{T} \to 2^{\mathbb{T}}$.

$$\begin{array}{rcll} \mathit{before}(\alpha) & = & \{\alpha\} \cup \{\beta{\in}\mathbb{V} \mid \mathit{trans}(\beta,\alpha,C) = \mathsf{true}\}, & \text{if } \alpha{\in}\mathbb{V}. \\ \mathit{after}(\alpha) & = & \{\alpha\} \cup \{\beta{\in}\mathbb{V} \mid \mathit{trans}(\alpha,\beta,C) = \mathsf{true}\}, & \text{if } \alpha{\in}\mathbb{V}. \\ \mathit{before}(\sigma{\to}\tau) & = & \{\sigma'{\to}\tau'{\in}\mathbb{T} \mid \sigma'{\in}\mathit{after}(\sigma) \text{ and } \tau'{\in}\mathit{before}(\tau)\}. & \\ \mathit{after}(\sigma{\to}\tau) & = & \{\sigma'{\to}\tau'{\in}\mathbb{T} \mid \sigma'{\in}\mathit{before}(\sigma) \text{ and } \tau'{\in}\mathit{after}(\tau)\}. & \end{array}$$

Lemma 7.2.9 Let $\sigma, \tau{\in}\mathbb{T}$.

1. $\sigma{\in}\mathit{before}(\sigma)$ and $\sigma{\in}\mathit{after}(\sigma)$.
2. $\sigma{\in}\mathit{before}(\tau) \Leftrightarrow \tau{\in}\mathit{after}(\sigma)$
3. $\sigma \leqslant_C \tau \Leftrightarrow \sigma{\in}\mathit{before}(\tau)$.
4. $\{\tau{\in}\mathbb{T} \mid \tau \leqslant_C \sigma\}$ and $\{\tau{\in}\mathbb{T} \mid \sigma \leqslant_C \tau\}$ are finite sets.

Proof:

1. Straightforward.
2. By induction on the structure of σ
3. $\Rightarrow$) By induction on the structure of σ using proposition 7.2.7 and 1.

   $\Leftarrow$) By induction on the structure of σ.
4. By items 2 and 3, we know that

   $$\begin{array}{l} \{\tau{\in}\mathbb{T} \mid \tau \leqslant_C \sigma\} = \mathit{before}(\sigma), \text{ and} \\ \{\tau{\in}\mathbb{T} \mid \sigma \leqslant_C \tau\} = \mathit{after}(\sigma), \end{array}$$

   Since C is finite, *before*(σ) and *after*(σ) are finite sets. □.

About the form of types in the $\leqslant_C$ relation

In order to study types in the subtyping relation it is useful fo have a language which enables us to refer to a specific subexpresion of a given type. Having in mind the underlying tree structure of a type, say σ, we define the notion of *binary code*. Each binary code uniquely determines a subtree of the underlying tree of σ, which in its turn, is linked to a subexpresion of σ.

Definition 7.2.10

1. A *binary code* is a possibly empty sequence of zeros and ones.
2. A *positive binary code* is a binary code with an even number of ones.
3. A *negative binary code* is a binary code with an odd number of ones.

Definition 7.2.11 The subexpression of code b in the type expression σ, notation $\mathit{Sub}(b,\sigma)$, is defined as follows. $\mathit{Sub} : \{0,1\}^*{\times}\mathbb{T} \to \mathbb{T}$

$$\begin{array}{rcl} \mathit{Sub}([],\sigma) & = & \sigma \\ \mathit{Sub}(1b,\sigma{\to}\tau) & = & \mathit{Sub}(b,\sigma) \\ \mathit{Sub}(0b,\sigma{\to}\tau) & = & \mathit{Sub}(b,\tau) \end{array}$$

Observe that *Sub* is a partial function; not every binary code indicates a subexpression of a given type. For example, $\mathit{Sub}(10,\alpha)$ with $\alpha{\in}\mathbb{V}$ is undefined.

Notation 7.2.12 we will write $b(\sigma)$ instead of $\mathit{Sub}(b,\sigma)$. We frequently use *code* instead of *binary code.*

Definition 7.2.13

1. b is called a *code in* σ if $b(\sigma)$ is defined.
2. b is called a *binary leaf code in* σ if $b(\sigma){\in}\mathbb{V}$.

Intuitively, a *binary code in* a type σ is a path starting from the root of the underlying tree of σ, where left is indicated with 1, and right with 0. Note that each code in σ uniquely determines a subexpression of σ. Then we can say that a subexpression is positive if it has a positive code, and negative otherwise. Note that then in the path from the root of the underlying tree of σ to the root of a positive (respectively negative) subexpression we have chosen an even (respectively odd) number of times the left branch of an arrow node.

Lemma 7.2.14 $\mathit{Shadow}(\sigma) = \mathit{Shadow}(\tau)$ if and only if every leaf code of σ is a leaf code of τ.

Proof: From left to right, the result follows by straightforward induction on the structure of σ. From right to left. By induction on the structure of σ.

$\sigma{\in}\mathbb{V}$. [] is the only (leaf) code of σ. Since [] is also a leaf code of τ, it follows that τ is a variable. Hence, $\mathit{Shadow}(\sigma) = \mathit{Shadow}(\tau)$.

$\sigma \equiv \sigma_1{\to}\sigma_2$. The leaf codes of σ are of the form $1b_1$ and $0b_2$, for every leaf code b_1 of σ_1 and for every leaf code b_2 of σ_2. Since $1b_1$ and $0b_2$ are also leaf codes of τ, τ is of the form $\tau_1{\to}\tau_2$. Then b_1 is a leaf code of τ_1 and b_2 of τ_2. Then by the induction hypothesis and the definition of *Shadow*, it follows that $\mathit{Shadow}(\sigma) = \mathit{Shadow}(\tau)$.

PROPOSITION 7.2.15

$$\begin{array}{r}\text{If for every positive leaf code } b \text{ in } \sigma,\quad b(\sigma) \leqslant_C b(\tau),\\ \text{and for every negative leaf code } b \text{ in } \sigma,\quad b(\tau) \leqslant_C b(\sigma),\quad \text{then } \sigma \leqslant_C \tau.\end{array}$$

PROOF: By induction on the complexity of σ.

$\sigma \in \mathbb{V}$. Then the only code in σ is the empty code, and as the empty code is positive, we have that $[](\sigma) \leqslant_C [](\tau)$, but $[](\sigma)$ is σ and $[](\tau)$ is τ.

$\sigma \equiv \sigma_1 \to \sigma_2$. Then the codes in σ are of the form $1b_1$ and $0b_2$, where b_1 is a code in σ_1 and b_2 is a code in σ_2.

Since b is a leaf code of σ, $b(\sigma)$ is a variable, and since $b(\sigma) \leqslant_C b(\tau)$, by the correctness of the algorithm *Subtype*, $b(\tau)$ is also a variable. Hence, b is a leaf code of τ. By lemma 7.2.14, it follows that *Shadow*(σ) = *Shadow*(τ). Then we know that τ is of the form $\tau_1 \to \tau_2$. Our goal is to prove that $\sigma_1 \to \sigma_2 \leqslant_C \tau_1 \to \tau_2$. For that, it is enough to show that $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$.

- Let b_1 be a negative leaf code in σ_1. By the definition of code, $b_1(\sigma_1) = 1b_1(\sigma)$, and by assumption, since $1b_1$ is a positive code, also $1b_1(\sigma) \leqslant_C 1b_1(\tau)$, and, as $1b_1(\tau) = b_1(\tau_1)$, we conclude $b_1(\sigma_1) \leqslant_C b_1(\tau_1)$.
- Let b_1 be a positive leaf code in σ_1. By definition of code, $b_1(\sigma_1) = 1b_1(\sigma)$, then as $1b_1$ is a negative code in σ, $1b_1(\tau) \leqslant_C 1b_1(\sigma)$, and, as $1b_1(\tau) = b_1(\tau_1)$, we have that $b_1(\tau_1) \leqslant_C b_1(\sigma_1)$.

We conclude that $\tau_1 \leqslant_C \sigma_1$ by the induction hypothesis. Similarly, it follows that $\sigma_2 \leqslant_C \tau_2$. □

LEMMA 7.2.16 Suppose $\sigma \leqslant_C \tau$. Then,

b is a code in σ if and only if b is a code in τ.

PROOF: By induction on the complexity of σ.

$\sigma \in \mathbb{V}$. Then, by lemma 7.2.5(2), τ is also a variable. Then the only possible code is the empty code, and $[]$ is a code in every element of $\mathbb{T}$.

$\sigma \equiv \sigma_1 \to \sigma_2$. Then, by lemma 7.2.5(2), τ is of the form $\tau_1 \to \tau_2$. By lemma 7.2.6, $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$.

$\Rightarrow$) Let b be a code in σ, then the following cases have to be considered.

Case 1. $b \equiv 0b'$ with b' a code in σ_2. Then, by the induction hypothesis, b' is also a code in τ_2, but this means that $0b'$ is a code in τ.

Case 2. $b \equiv 1b'$ with b' a code in σ_1. By the induction hypothesis, b' is a code in τ_1. Hence, $1b'$ is a code in τ.

$\Leftarrow$) Similar to the proof of $\Rightarrow$), interchanging the roles of σ and τ. □

PROPOSITION 7.2.17 Suppose $\sigma \leqslant_C \tau$. Then, for every leaf code b in σ

if b is positive, then $b(\sigma) \leqslant_C b(\tau)$ and
if b is negative, then $b(\tau) \leqslant_C b(\sigma)$.

PROOF: By induction on the complexity of σ.

$\sigma \in \mathbb{V}$. This means that the only code in σ is the empty code. Hence $b(\sigma)$ is σ and $b(\tau)$ is τ. The empty code is positive and, by hypothesis, $\sigma \leqslant_C \tau$, so there is nothing else to prove.

$\sigma \equiv \sigma_1 \rightarrow \sigma_2$. Then, by lemma 7.2.5(2), τ is of the form $\tau_1 \rightarrow \tau_2$. By lemma 7.2.6, $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$. We are in the case that $b \equiv db'$ with $d = 1$ and b' a code in σ_1, or $d = 0$ and b' a code in σ_2. According to the possible codes in σ we have to consider the following two cases.

- b is positive.

  Case 1. $d \equiv 0$ and b' positive. As b' is a code in σ_2, by the induction hypothesis, $b'(\sigma_2) \leqslant_C b'(\tau_2)$, what, by definition of code, can be read as $0b'(\sigma) \leqslant_C 0b'(\tau)$.

  Case 2. $d \equiv 1$ and b' negative. By lemma 7.2.16, b' is also a code in τ_1 and, by the induction hypothesis, $b'(\sigma_1) \leqslant_C b'(\tau_1)$, what, by definition of code, is $1b'(\sigma) \leqslant_C 1b'(\tau)$.

- b is negative.

  Case 1. $d \equiv 0$ and b' negative. By the induction hypothesis, $b'(\tau_2) \leqslant_C b'(\sigma_2)$, what means that $0b'(\tau) \leqslant_C 0b'(\sigma)$.

  Case 2. $d \equiv 1$ and b' positive. Lemma 7.2.16 implies that b' is also a code in τ_1. Then, by the induction hypothesis, $b'(\tau_1) \leqslant_C b'(\sigma_1)$. Finally, by the definition of code, it follows that $1b'(\tau) \leqslant_C 1b'(\sigma)$. □

THEOREM 7.2.18

$\sigma \leqslant_C \tau$ if and only if for every leaf code b in σ
if b is positive, then $b(\sigma) \leqslant_C b(\tau)$ and
if b is negative, then $b(\tau) \leqslant_C b(\sigma)$.

PROOF: By propositions 7.2.15 and 7.2.17. □

This theorem suggests yet another way to check whether $\sigma \leqslant_C \tau$, by only looking at the leaves of the underlying trees of σ and τ.

7.3 Simply typed λ-calculus

We use a slightly different version of $\lambda{\to}$ than the one in [Bar92], the difference being that our version contains constants as pseudo-terms that are syntactically different from variables. Constants are assigned a type in an *environment* as in [AH87], and there is a rule for typing constants.

DEFINITION 7.3.1 The typed λ-calculus, $\lambda{\to}$, is defined as follows.

1. The set of pseudo-terms $\Lambda = \Lambda(\lambda{\to})$ is defined by the following syntax.

$$\Lambda ::= V \mid K \mid \lambda V{:}\mathbb{T}.\Lambda \mid \Lambda\Lambda,$$

   where V is a set of (term) variables and K is a set of constants such that V and K are disjoint sets.

2. An *environment* is a set of statements with only distinct constants as subjects. The symbol Σ is used for environments.

   The set of types $\mathbb{T}$ and the concepts of statement, typing assumption, subject, context, derivable statement and legal term are as in definition 7.4.1.

DEFINITION 7.3.2 (*Typing rules*)

$$\Gamma \vdash_\Sigma k \in \sigma, \qquad \text{if } k{:}\sigma{\in}\Sigma \tag{T-CONS}$$

$$\Gamma \vdash_\Sigma x \in \sigma, \qquad \text{if } x{:}\sigma{\in}\Gamma \tag{T-VAR}$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma M \in \tau}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.M \in \sigma{\to}\tau} \tag{T-ABS}$$

$$\frac{\Gamma \vdash_\Sigma M \in \sigma{\to}\tau \qquad \Gamma \vdash_\Sigma N \in \sigma}{\Gamma \vdash_\Sigma MN \in \tau} \tag{T-APP}$$

Basic properties of $\lambda{\to}$

Let us mention the following properties of $\lambda{\to}$ without giving their proofs. The interested reader can find more about these results in [Bar92]. There the $\lambda{\to}$-system presented does not have constants, but the proofs of the propositions below are straightforward extensions of the proofs given in [Bar92]; nevertheless the strong normalization property deserves a more careful examination. Let us use $\vdash$ for derivability in the $\lambda{\to}$-system presented in [Bar92] where we consider $K \cup V$ as the set of variables. Note that if $\Gamma \vdash_\Sigma M \in \sigma$ then $\Sigma, \Gamma \vdash M \in \sigma$. Then the strong normalization property for the system in which our constants are treated as free variables implies the corresponding result for the system with constants.

Let $FV(M)$ denote, as usual, the set of free variables of M and $Dom(\Gamma)$ the domain of Γ, i.e. the set of subjects of Γ.

LEMMA 7.3.3

1. (Free Variable) If $\Gamma \vdash_\Sigma M \in \sigma$, then $FV(M) \subset Dom(\Gamma)$.

2. (Weakening for $\lambda\!\rightarrow$)
   Let Γ and Γ' be contexts such that $\Gamma\subseteq\Gamma'$. Then if $\Gamma\vdash_\Sigma M\in\sigma$, then $\Gamma'\vdash_\Sigma M\in\sigma$.

Proposition 7.3.4

1. (Generation for $\lambda\!\rightarrow$)
   (a) If $\Gamma\vdash_\Sigma k\in\sigma$, then $k{:}\sigma\in\Sigma$.
   (b) If $\Gamma\vdash_\Sigma x\in\sigma$, then $x{:}\sigma\in\Gamma$.
   (c) If $\Gamma\vdash_\Sigma MN\in\sigma$, then there exists τ such that $\Gamma\vdash_\Sigma M\in\tau\rightarrow\sigma$ and $\Gamma\vdash_\Sigma N\in\tau$.
   (d) If $\Gamma\vdash_\Sigma \lambda x{:}\sigma.M\in\rho$, then there exists τ such that $\rho\equiv\sigma\rightarrow\tau$ and $\Gamma,\, x{:}\sigma\vdash_\Sigma M\in\tau$.
2. (Unicity of types for $\lambda\!\rightarrow$) If $\Gamma\vdash_\Sigma M\in\sigma$ and $\Gamma\vdash_\Sigma M\in\sigma'$, then $\sigma\equiv\sigma'$.
3. (Strong normalization for $\lambda\!\rightarrow$) If $\Gamma\vdash_\Sigma M\in\sigma$, then M is strongly normalizing.

7.4 $\lambda_\leqslant$, a system with implicit coercions

We define the system $\lambda_\leqslant$ (lambda sub), an extension of the simply typed λ-calculus with subtyping. The difference between the simply typed λ-calculus and $\lambda_\leqslant$ is the following rule.

$$\frac{\Gamma\vdash M\in\sigma \qquad \sigma\leqslant_C\tau}{\Gamma\vdash M\in\tau} \qquad \text{(T-Subsumption)}$$

An immediate consequence of the addition of this rule is the loss of the unicity of types property. Fortunately, the system has instead the minimal type property. Namely, if $\Gamma\vdash M\in\sigma$, then there exists τ such that $\Gamma\vdash M\in\tau$ and $\tau\leqslant_C\sigma$. That property is relevant in the design of type checking and type inference algorithms. Consider the case of type checking. Knowing that if a term is typeable, then it has a minimal type, we try to identify a fragment of the system in which every typeable term is assigned a minimal type. In our case, we define a subsystem of $\lambda_\leqslant$, $Alg\lambda_\leqslant$, which has the unicity of types property and is syntax directed, in such a way that that $Alg\lambda_\leqslant$ defines an algorithm to compute minimal types in $\lambda_\leqslant$.

We give now the formal definition of a simply typed λ-calculus with implicit coercions, $\lambda_\leqslant$.

Definition 7.4.1 The typed λ-calculus with implicit coercions, $\lambda_\leqslant$, is defined as follows.

1. The set of types $\mathbb{T}=\text{Type}(\lambda_\leqslant)$ is defined by

$$\mathbb{T} ::= \mathbb{V} \mid \mathbb{T}\rightarrow\mathbb{T},$$

   where $\mathbb{V}$ is a set of (type) variables.

2. The set of pseudo-terms $\Lambda = \Lambda(\lambda_{\leqslant})$ is defined as follows.

   $$\Lambda ::= V \mid \lambda V{:}\mathbb{T}.\Lambda \mid \Lambda\Lambda,$$

   where V is a set of (term) variables.

3. A *statement* is of the form $M \in \sigma$ (M is of type σ) with $M{\in}\Lambda$ and $\sigma{\in}\mathbb{T}$. The term M is called the *subject* of the statement. A typing assumption is an expression of the form $x{:}\sigma$. The variable x is called the subject of the typing assumption.

4. A *context* is a set of typing assumptions with distinct variables as subjects.

Definition 7.4.2 (*Typing rules*)

$$\Gamma \vdash_{\lambda_{\leqslant}} x \in \sigma \quad \text{if } x{:}\sigma{\in}\Gamma \tag{T-Var}$$

$$\frac{\Gamma, x{:}\sigma \vdash_{\lambda_{\leqslant}} M \in \tau}{\Gamma \vdash_{\lambda_{\leqslant}} \lambda x{:}\sigma.M \in \sigma{\to}\tau} \tag{T-Abs}$$

$$\frac{\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma{\to}\tau \qquad \Gamma \vdash_{\lambda_{\leqslant}} N \in \sigma}{\Gamma \vdash_{\lambda_{\leqslant}} MN \in \tau} \tag{T-App}$$

$$\frac{\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma \qquad \sigma \leqslant_C \tau}{\Gamma \vdash_{\lambda_{\leqslant}} M \in \tau} \tag{T-Subsumption}$$

Definition 7.4.3 A statement $M \in \sigma$ is *derivable* from the context Γ, we write $\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma$ – Γ yields M of type σ –, if $\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma$ can be obtained using the rules T-Var, T-Abs, T-App, and T-Subsumption in definition 7.4.2. A $\lambda_{\leqslant}$-term M is *legal*, if there exist Γ and σ such that $\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma$, In other words, a legal term is the subject of a derivable statements.

The typing rules in definition 7.4.2, are not syntax directed. In order to describe a type inference algorithm, we need an alternative presentation of the typing rules in which the term to be typed uniquely determines the last rule of the derivation of its typing statement. In the next section, we define $Alg\lambda_{\leqslant}$, an algorithmic presentation of $\lambda_{\leqslant}$. This presentation has the property of finding a minimal type for $\lambda_{\leqslant}$.

Definition 7.4.4 (*A type inference algorithm*)

$$\Gamma \vdash_{Alg\lambda_{\leqslant}} x \in \sigma \quad \text{if } x{:}\sigma{\in}\Gamma \tag{T-Var}$$

$$\frac{\Gamma, x{:}\sigma \vdash_{Alg\lambda_{\leqslant}} M \in \tau}{\Gamma \vdash_{Alg\lambda_{\leqslant}} \lambda x{:}\sigma.M \in \sigma{\to}\tau} \tag{T-Abs}$$

$$\frac{\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma{\to}\tau \qquad \Gamma \vdash_{Alg\lambda_{\leqslant}} N \in \rho \qquad \rho \leqslant_C \sigma}{\Gamma \vdash_{Alg\lambda_{\leqslant}} MN \in \tau} \tag{T-App$_{\leqslant}$}$$

Proposition 7.4.5 (*Generation for* Alg$\lambda_{\leqslant}$)

1. If $\Gamma \vdash_{Alg\lambda_{\leqslant}} x \in \sigma$, then $x{:}\sigma{\in}\Gamma$.
2. If $\Gamma \vdash_{Alg\lambda_{\leqslant}} \lambda x{:}\tau.M \in \sigma$, then there exists ρ such that $\Gamma, x{:}\tau \vdash_{Alg\lambda_{\leqslant}} M \in \rho$ and $\sigma \equiv \tau{\to}\rho$.
3. If $\Gamma \vdash_{Alg\lambda_{\leqslant}} MN \in \sigma$, then there exist ρ and ρ' such that $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \rho{\to}\sigma$, $\rho' \leqslant_C \rho$, and $\Gamma \vdash_{Alg\lambda_{\leqslant}} N \in \rho'$.

Proof: Since the system is syntax directed, the form of each judgement uniquely determines the last rule of its derivation. □

Proposition 7.4.6 (*Unicity of types for* Alg$\lambda_{\leqslant}$) If $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$ and $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma'$, then $\sigma \equiv \sigma'$.

Proof: By induction on the complexity of M, using that the system is syntax directed. □

Lemma 7.4.7 (*Well-foundedness of* Alg$\lambda_{\leqslant}$) The *Alg*$\lambda_{\leqslant}$ rules (7.4.4) define a terminating algorithm.

Proof: Since Γ is finite T-Var cannot cause non-termination. In rules T-App$_{\leqslant}$ and T-Abs, the size of the subject in each of the premises is strictly smaller than the size of the subject in the corresponding conclusion. Moreover, the relation $\leqslant_C$ is decidable because the algorithm *Subtype*, which is sound and complete with respect to $\leqslant_C$, always terminates. □

Proposition 7.4.8 (*Decidability of type inference for* Alg$\lambda_{\leqslant}$) For any Γ and M it is decidable whether there exists σ such that $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$.

Proof: By the well-foundedness of *Alg*$\lambda_{\leqslant}$.

Proposition 7.4.9 (*Decidability of type checking for* Alg$\lambda_{\leqslant}$)
Given Γ, M, and σ, it is decidable whether $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$.

Proof: Because of the unicity of types property of *Alg*$\lambda_{\leqslant}$, the decidability of type inference (7.4.8) implies the decidability of type checking.

To prove the strong normalization property for *Alg*$\lambda_{\leqslant}$ we use the corresponding result for $\lambda{\to}$ in section 7.3. We first provide some definitions that allow us to relate *Alg*$\lambda_{\leqslant}$-terms to $\lambda{\to}$-terms in such a way that β-reductions are preserved.

Definition 7.4.10 Let $\delta_0{\in}\mathbb{T}(\lambda{\to})$. $\_^{\delta_0} : \mathbb{T}(Alg\lambda_{\leqslant}){\to}\mathbb{T}(\lambda{\to})$.

$$\begin{array}{rcll} \alpha^{\delta_0} & = & \delta_0 & \text{if } \alpha{\in}\mathbb{V}. \\ (\sigma{\to}\tau)^{\delta_0} & = & \sigma^{\delta_0}{\to}\tau^{\delta_0} & \end{array}$$

The homomorphic extension to contexts is defined as follows.

$$\begin{array}{rcl} \{\}^{\delta_0} & = & \{\} \\ (\Gamma \cup \{x{:}\sigma\})^{\delta_0} & = & \Gamma^{\delta_0} \cup \{x{:}\sigma^{\delta_0}\} \end{array}$$

Definition 7.4.11 $\_\!\downarrow : \Lambda(Alg\lambda_{\leqslant}) \to \Lambda(\lambda\!\to)$

$$\begin{array}{rcl} x\downarrow & = & x \\ (\lambda x{:}\sigma.M)\downarrow & = & \lambda x{:}\sigma^{\delta_0}.M\downarrow \\ (MN)\downarrow & = & M\downarrow N\downarrow \end{array}$$

The choice of δ_0 is irrelevant, the essential feature is that it is fixed.

Lemma 7.4.12 Let $\sigma, \tau \in \mathbb{T}$. If $Shadow(\sigma) = Shadow(\tau)$, then $\sigma^{\delta_0} \equiv \tau^{\delta_0}$.

Lemma 7.4.13 Let $M \in \Lambda(Alg\lambda_{\leqslant})$. If $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$, then $\Gamma^{\delta_0} \vdash_{\Sigma} M\downarrow \in \sigma^{\delta_0}$.

Proof: By induction on the derivation of $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$. Consider the case of the T-App$_{\leqslant}$ rule. Then the situation is as follows.

$$\frac{\Gamma \vdash_{Alg\lambda_{\leqslant}} P \in \tau\to\sigma \quad \Gamma \vdash_{Alg\lambda_{\leqslant}} N \in \rho \quad \rho \leqslant_C \tau}{\Gamma \vdash_{Alg\lambda_{\leqslant}} PN \in \sigma,}$$

where $M \equiv PN$. By the induction hypothesis, the definition of $\_^{\delta_0}$, and lemmas 7.2.5(2) and 7.4.12, we have that

$\Gamma^{\delta_0} \vdash_{\Sigma} P\downarrow \in \tau^{\delta_0}\to\sigma^{\delta_0}$ and $\Gamma^{\delta_0} \vdash_{\Sigma} N\downarrow \in \tau^{\delta_0}$.

Finally, by T-App and the definition of $\_\!\downarrow$, it follows that $\Gamma^{\delta_0} \vdash_{\Sigma} PN\downarrow \in \sigma^{\delta_0}$. □

Lemma 7.4.14 (*Substitution*) Let M, N in $\Lambda(Alg\lambda_{\leqslant})$. Then,

$(M[x{:=}N])\downarrow \equiv M\downarrow[x{:=}N\downarrow]$.

Proof: By induction on the complexity of M. □

Lemma 7.4.15 (*Reduction preservation*)
Let M, N in $\Lambda(Alg\lambda_{\leqslant})$. If $M \to_{\beta} N$, then $M\downarrow \to_{\beta} N\downarrow$.

Proof: The proof is by straightforward induction on the complexity of M. In particular, the case when M is a redex is a consequence of lemma 7.4.14. □

Proposition 7.4.16 (*Strong normalization for* Alg$\lambda_{\leqslant}$) If $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$, then M is strongly normalizing.

Proof: The result follows from lemmas 7.4.13 and 7.4.15, using a similar argument as in theorem 7.7.12. □

The system $Alg\lambda_{\leqslant}$ does not satisfy the subject reduction property. Consider the following simple example.
Let $\alpha \leqslant_C \gamma$, then $z{:}\alpha \vdash_{Alg\lambda_{\leqslant}} (\lambda x{:}\gamma.x)z \in \gamma$ and $(\lambda x{:}\gamma.x)z \to_{\beta} z$, but $z{:}\alpha \nvdash_{Alg\lambda_{\leqslant}} z{:}\gamma$. This example illustrates the fact that a step of β-reduction of a term may reduce its original type. The reason being that, in order to type $(\lambda x{:}\gamma.x)z$, $\alpha \leqslant_C \gamma$ is used, and such information can only be used by the application rule, T-App$_{\leqslant}$. The β-reduction step "erases" the application. Therefore the subtyping information cannot be used any longer. Nevertheless, the following monotonic subject reduction property holds. The following result is necessary to prove proposition 7.4.18.

Lemma 7.4.17 If $\Gamma, x{:}\sigma \vdash_{Alg\lambda_\leqslant} M \in \tau$, $\Gamma \vdash_{Alg\lambda_\leqslant} N \in \sigma'$, and $\sigma' \leqslant_C \sigma$, then there exists τ' such that $\Gamma \vdash_{Alg\lambda_\leqslant} M[x{:=}N] \in \tau'$ and $\tau' \leqslant_C \tau$.

Proof: By induction on the complexity of M. □

Proposition 7.4.18 (*Monotonic subject reduction*)

If $\Gamma \vdash_{Alg\lambda_\leqslant} M \in \sigma$ and $M \to_\beta M'$, then there exists τ such that $\Gamma \vdash_{Alg\lambda_\leqslant} M' \in \tau$ and $\tau \leqslant_C \sigma$.

Proof: By straightforward induction on the complexity of M, using lemma 7.4.17. Let us consider the case when M is a redex and M' its reduct, that is the only case that demands some work. Then the situation is as follows.

$$\Gamma \vdash_{Alg\lambda_\leqslant} (\lambda x{:}\rho.M_1)M_2 \in \sigma.$$

By generation (proposition 7.4.5), we have that

$$\Gamma, x{:}\rho \vdash_{Alg\lambda_\leqslant} M_1 \in \sigma,\ \Gamma \vdash_{Alg\lambda_\leqslant} M_2 \in \rho' \text{ and } \rho' \leqslant_C \rho.$$

By lemma 7.4.17, there exists σ' such that $\Gamma \vdash_{Alg\lambda_\leqslant} M_1[x{:=}M_2] \in \sigma'$ and $\sigma' \leqslant_C \sigma$. □

Observe that because of the unicity of types property M' cannot have type σ as well.

Here we show that $Alg\lambda_\leqslant$ describes an effective procedure to compute minimal types in $\lambda_\leqslant$.

Proposition 7.4.19

1. (Soundness) If $\Gamma \vdash_{Alg\lambda_\leqslant} M \in \sigma$, then $\Gamma \vdash_{\lambda_\leqslant} M \in \sigma$.
2. (Completeness and minimal typing) If $\Gamma \vdash_{\lambda_\leqslant} M \in \tau$, then there exists σ such that $\sigma \leqslant_C \tau$ and $\Gamma \vdash_{Alg\lambda_\leqslant} M \in \sigma$.

Proof:

1. By induction on the derivation of $\Gamma \vdash_{Alg\lambda_\leqslant} M \in \sigma$.
2. By induction on the derivation of $\Gamma \vdash_{\lambda_\leqslant} M \in \sigma$. We consider the case for T-App, the other cases follow with similar or simpler arguments. We are given that

$$\begin{array}{l} M \equiv M_1\, M_2, \\ \Gamma \vdash_{\lambda_\leqslant} M_1 \in \tau{\to}\sigma, \text{ and} \\ \Gamma \vdash_{\lambda_\leqslant} M_2 \in \tau. \end{array}$$

By the induction hypothesis, there exist ρ_1 and ρ_2 such that

$$\begin{array}{l} \Gamma \vdash_{Alg\lambda_\leqslant} M_1 \in \rho_1 \text{ where } \rho_1 \leqslant_C \tau{\to}\sigma, \text{ and} \\ \Gamma \vdash_{Alg\lambda_\leqslant} M_2 \in \rho_2 \text{ where } \rho_2 \leqslant_C \tau. \end{array}$$

By the correctness of the algorithm *Subtype* (7.2.3), we know that $\rho_1 \equiv \tau_1{\to}\sigma_1$, and by lemma 7.2.6, we have that $\tau \leqslant_C \tau_1$ and $\sigma_1 \leqslant_C \sigma$. By S-Trans, $\rho_2 \leqslant_C \tau_1$. Finally, by T-App$_\leqslant$, we have that $\Gamma \vdash_{Alg\lambda_\leqslant} M_1\, M_2 \in \sigma_1$. □

The last proposition says that $\lambda_\leqslant$ and $Alg\lambda_\leqslant$ type the same set of terms, in other words, the set of legal terms of $\lambda_\leqslant$ is equal to that of $Alg\lambda_\leqslant$. The difference is that they may assign different types to the same term. Note that item 1 says that $Alg\lambda_\leqslant$-typing statements are also $\lambda_\leqslant$-typing statements, the converse is not true. Consider the following simple example.

Let $C = \{(\alpha, \beta)\}$. Then $x{:}\alpha \vdash_{\lambda_\leqslant} x \in \beta$, but $x{:}\alpha \vdash_{Alg\lambda_\leqslant} x \in \beta$ is not a derivable statement.

Observe that the typing information present in the $\lambda_\leqslant$ terms is essential for the minimal type property. Consider the following example in the system presented in [Mit84], i.e. a simply typed λ-calculus à la Curry with implicit coercions to see that it might be the case that neither the unicity of types property nor the minimal type property are satisfied.

Let $C = \{(\alpha, \beta)\}$ and let α, β, and γ be different type variables. Then the identity function, $\lambda x.x$, has as type scheme $\sigma{\to}\tau$ where $\sigma \leqslant_C \tau$. In particular we have that

$$\Gamma \vdash \lambda x.x{:}\alpha{\to}\beta \text{ and } \Gamma \vdash \lambda x.x{:}\gamma{\to}\gamma,$$

but there is no type ρ such that

$$\Gamma \vdash \lambda x.x{:}\rho,\ \ \rho \leqslant_C \alpha{\to}\beta,\ \text{ and } \rho \leqslant_C \gamma{\to}\gamma.$$

Unlike the system $Alg\lambda_\leqslant$ the system $\lambda_\leqslant$ satisfies the subject reduction property.

Proposition 7.4.20 (*Subject reduction for* $\lambda_\leqslant$)
If $\Gamma \vdash_{\lambda_\leqslant} M \in \sigma$ and $M \to_\beta M'$, then $\Gamma \vdash_{\lambda_\leqslant} M' \in \sigma$.

Proof: By lemma 7.4.19(2), we know that there exists τ such that $\Gamma \vdash_{Alg\lambda_\leqslant} M \in \tau$ and $\tau \leqslant_C \sigma$. By monotonic subject reduction (7.4.18), $\Gamma \vdash_{Alg\lambda_\leqslant} M \in \tau'$, for some $\tau' \leqslant_C \tau$. Then, by lemma 7.4.19(1), it follows that $\Gamma \vdash_{\lambda_\leqslant} M \in \tau'$. Finally, since $\tau' \leqslant_C \sigma$, by T-Subsumption, it follows that $\Gamma \vdash_{\lambda_\leqslant} M' \in \sigma$. □

7.5 λ_C, a system with explicit coercions

Definition 7.5.1 The typed λ-calculus with explicit coercions, λ_C, is defined as follows. The set of pseudo-terms $\Lambda = \Lambda(\lambda_C)$ is defined by the following grammar.

$$\Lambda{::=}V \mid \lambda V{:}\mathbb{T}.\Lambda \mid \Lambda\Lambda \mid Q{<}\Lambda{>},$$

where V is a set of (term) variables and Q is a set of constants with $c_{\sigma,\tau}{\in}Q$ if and only if $\sigma \leqslant_C \tau$.

The set of types $\mathbb{T}$ and the concepts of statement, typing assumption, subject, context, derivable statement, and legal term are as in definition 7.4.1.

Definition 7.5.2

$$\Gamma \vdash_{\lambda_C} x \in \sigma, \quad \text{if } x{:}\sigma{\in}\Gamma \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}\sigma \vdash_{\lambda_C} M \in \tau}{\Gamma \vdash_{\lambda_C} \lambda x{:}\sigma.M \in \sigma{\to}\tau} \qquad (\text{T-Abs})$$

$$\frac{\Gamma \vdash_{\lambda_C} M \in \sigma{\to}\tau \qquad \Gamma \vdash_{\lambda_C} N \in \sigma}{\Gamma \vdash_{\lambda_C} MN \in \tau} \qquad (\text{T-App})$$

$$\frac{\Gamma \vdash_{\lambda_C} M \in \sigma \qquad \sigma \leqslant_C \tau}{\Gamma \vdash_{\lambda_C} c_{\sigma,\tau}{<}M{>} \in \tau} \qquad (\text{T-Coerce})$$

The standard notions of reduction and substitution are extended with the following rules.

1. If $M \to_\beta M'$, then $c_{\sigma,\tau}{<}M{>} \to_\beta c_{\sigma,\tau}{<}M'{>}$.

2. $c_{\sigma,\tau}{<}N{>}[x{:=}M] = c_{\sigma,\tau}{<}N[x{:=}M]{>}$.

Lemma 7.5.3 (*Weakening for λ_C*) Let Γ and Γ' be contexts such that $\Gamma \subset \Gamma'$. Then, if $\Gamma \vdash_{\lambda_C} M \in \sigma$, then $\Gamma' \vdash_{\lambda_C} M \in \sigma$.

Proof: By induction on the derivation of $\Gamma \vdash_{\lambda_C} M \in \sigma$. □

Proposition 7.5.4 (*Generation for λ_C*)

1. If $\Gamma \vdash_{\lambda_C} x \in \sigma$, $x{:}\sigma{\in}\Gamma$.

2. If $\Gamma \vdash_{\lambda_C} MN \in \sigma$, there exists τ such that $\Gamma \vdash_{\lambda_C} M \in \tau{\to}\sigma$ and $\Gamma \vdash_{\lambda_C} N \in \tau$.

3. If $\Gamma \vdash_{\lambda_C} \lambda x{:}\rho.M \in \sigma$, there exists τ such that $\sigma \equiv \rho{\to}\tau$ and $\Gamma, x{:}\sigma \vdash_{\lambda_C} M \in \tau$.

4. If $\Gamma \vdash_{\lambda_C} c_{\rho,\tau}{<}M{>} \in \sigma$, $\Gamma \vdash_{\lambda_C} M \in \rho$, $\tau \equiv \sigma$, and $\rho \leqslant_C \tau$.

Proof: Since the system is syntax directed, the form of the subject uniquely determines the last rule of the derivation of the typing statement. □

Lemma 7.5.5 Let $M{\in}\Lambda(\lambda_C)$.
If Γ, $x{:}\tau \vdash_{\lambda_C} M \in \sigma$ and $\Gamma \vdash_{\lambda_C} N \in \tau$, then $\Gamma \vdash_{\lambda_C} M[x{:=}N] \in \sigma$.

Proof: By induction on the complexity of M. □

Proposition 7.5.6 (*Subject reduction for λ_C*)

If $\Gamma \vdash_{\lambda_C} M \in \sigma$ and $M \to_\beta M'$, then $\Gamma \vdash_{\lambda_C} M' \in \sigma$.

Proof: By induction on the complexity of M. Let us consider the case when M is an application and in particular a redex. Then, $M \equiv (\lambda x{:}\tau.P)N$ and $M' \equiv P[x{:=}N]$. By generation, we have that $\Gamma, x{:}\tau \vdash_{\lambda_C} P \in \sigma$ and $\Gamma \vdash_{\lambda_C} N \in \tau$. Finally, due to lemma 7.5.5, we get $\Gamma \vdash_{\lambda_C} P[x{:=}N] \in \sigma$. □

It is instructive to compare these last two results with lemma 7.4.17 and proposition 7.4.18, the corresponding results for $Alg\lambda_{\leqslant}$.

7.6 The relation between $\lambda_\leqslant$ and λ_C

If we go back to the discussion in the introduction, the T-Coerce rule is more than what we actually need to be able to apply a function to an argument of a type smaller than its domain type. On the other hand, the T-App$_\leqslant$ rule fits perfectly in our requirements. In the introduction we mention the difference between implicit and explicit coercions; in this framework $Alg\lambda_\leqslant$ has implicit coercions and furthermore, given a (legal) term M in $\Lambda(Alg\lambda_\leqslant)$, there is a uniform way to find a term M' in $\Lambda(\lambda_C)$ that is the explicitly coerced version of M, as will be defined in definition 7.6.3. This can be read as: there is no need to write the coercions because they can be automatically recovered.

Definition 7.6.1 (*The implicitly coerced version of* M)
$|\_| : \Lambda(\lambda_C) \to \Lambda(Alg\lambda_\leqslant)$.

$$\begin{array}{rcl}
|x| & = & x, \\
|\lambda x{:}\sigma.M| & = & \lambda x{:}\sigma.|M|, \\
|MN| & = & |M||N|, \\
|c_{\sigma,\tau}{<}M{>}| & = & |M|.
\end{array}$$

Lemma 7.6.2 If $\Gamma \vdash_{\lambda_C} M \in \sigma$, then there exists τ such that $\Gamma \vdash_{Alg\lambda_\leqslant} |M| \in \tau$ and $\tau \leqslant_C \sigma$.

Proof: By induction on the derivation of $\Gamma \vdash_{\lambda_C} M \in \sigma$. □

This lemma says that the implicitly coerced version of a (legal) λ_C-term is a (legal) $Alg\lambda_\leqslant$-term.

Definition 7.6.3 dec : Context$\times\Lambda(Alg\lambda_\leqslant)\to\Lambda(\lambda_C)$

$$\begin{array}{rcl}
dec_\Gamma(x) & = & x \\
dec_\Gamma(\lambda x{:}\sigma.M) & = & \lambda x{:}\sigma.dec_{\Gamma,x:\sigma}(M) \\
dec_\Gamma(MN) & = & dec_\Gamma(M)c_{\rho,\sigma}{<}dec_\Gamma(N){>} \\
 & & \text{where } \Gamma \vdash_{Alg\lambda_\leqslant} N \in \rho \\
 & & \text{and } \Gamma \vdash_{Alg\lambda_\leqslant} M \in \sigma\to\tau
\end{array}$$

dec stands for *decoration.*

Observe that *dec* is a partial mapping given that there may not be such ρ or $\sigma\to\tau$ in the application case, and that it may not be the case that $\rho \leqslant_C \sigma$, in which case there is no $c_{\rho,\sigma}$ constant. On the other hand, the next result shows that *dec* is total on the subset of legal $Alg\lambda_\leqslant$-terms.

Lemma 7.6.4 Let $M\in\Lambda(Alg\lambda_\leqslant)$. then
$\Gamma \vdash_{Alg\lambda_\leqslant} M \in \sigma$ if and only if $\Gamma \vdash_{\lambda_C} dec_\Gamma(M) \in \sigma$.

Proof:

$\Leftarrow$) By induction on the complexity of M.

$\Rightarrow$) By induction on the derivation of $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$. Let us consider the case of the T-App$_{\leqslant}$ rule. Then, the situation is as follows.

$$\frac{\Gamma \vdash_{Alg\lambda_{\leqslant}} P \in \tau{\rightarrow}\sigma \quad \Gamma \vdash_{Alg\lambda_{\leqslant}} N \in \rho \quad \rho \leqslant_C \tau}{\Gamma \vdash_{Alg\lambda_{\leqslant}} PN \in \sigma}$$

where $M \equiv PN$.

By the induction hypothesis $\Gamma \vdash_{\lambda_C} dec_\Gamma(P) \in \tau{\rightarrow}\sigma$ and $\Gamma \vdash_{\lambda_C} dec_\Gamma(N) \in \rho$ and by the T-Coerce rule $\Gamma \vdash_{\lambda_C} c_{\rho,\tau}{<}dec_\Gamma(N){>} \in \tau$. Finally, by the T-App rule, it follows that $\Gamma \vdash_{\lambda_C} dec_\Gamma(P)c_{\rho,\tau}{<}dec_\Gamma(N){>} \in \sigma$ and, by proposition 7.4.6, we conclude $dec_\Gamma(PN) \equiv dec_\Gamma(P)c_{\rho,\tau}{<}dec_\Gamma(N){>}$. □

Lemma 7.6.5
$\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma$ if and only if there exists τ, such that $\Gamma \vdash_{\lambda_C} c_{\tau,\sigma}{<}dec_\Gamma(M){>} \in \sigma$.

Proof:

1. If $\Gamma \vdash_{\lambda_C} c_{\tau,\sigma}{<}dec_\Gamma(M){>} \in \sigma$, by generation for λ_C (proposition 7.5.4),

   $\Gamma \vdash_{\lambda_C} dec_\Gamma(M) \in \tau$ and $\tau \leqslant_C \sigma$.

   By lemma 7.6.4, $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \tau$, and, by the soundness of $Alg\lambda_{\leqslant}$ (7.4.19), $\Gamma \vdash_{\lambda_{\leqslant}} M \in \tau$. Finally, the result follows by T-Subsumption. □

2. If $\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma$, then, by the completeness of $Alg\lambda_{\leqslant}$ (7.4.19), there exists τ such that

   $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \tau$, and
   $\tau \leqslant_C \sigma$.

   Then, by lemma 7.6.4, it follows that $\Gamma \vdash_{\lambda_C} dec_\Gamma(M) \in \tau$. Finally, the result follows by T-Coerce. □

This last lemma says that $\lambda_{\leqslant}$ can be translated into λ_C. Moreover, together with theorem 7.7.9, implies that $\lambda_{\leqslant}$ can be translated into $\lambda{\rightarrow}$.

We could compare the systems $Alg\lambda_{\leqslant}$ and λ_C using an auxiliary system, call it $\lambda_C{}^-$, obtained by replacing the rules T-App and T-Coerce of λ_C by the following rule.

$$\frac{\Gamma \vdash_{\lambda_C{}^-} M \in \sigma{\rightarrow}\tau \qquad \Gamma \vdash_{\lambda_C{}^-} N \in \rho \qquad \rho \leqslant_C \sigma}{\Gamma \vdash_{\lambda_C{}^-} M\, c_{\rho,\sigma}{<}N{>} \in \tau} \qquad (\text{T-App}_C)$$

It is easy to check that $\lambda_C{}^-$ satisfies the unicity of types property. Unfortunately, the system $\lambda_C{}^-$ is not sufficiently well-behaved to be of independent interest. For example, it does not satisfy the subject reduction property. The failure is such that β-reductions on well typed terms can yield illegal terms. For example, if $C = \{(\beta, \alpha)\}$, then $y{:}\beta \vdash_{\lambda_C{}^-} (\lambda x{:}\alpha.x)c_{\beta,\alpha}{<}y{>} \in \alpha$, but $y{:}\beta \vdash_{\lambda_C{}^-} c_{\beta,\alpha}{<}y{>} \in \alpha$ is not a derivable statement.

The following lemma shows that $\lambda_C{}^-$ is an intermediate step between $Alg\lambda_{\leqslant}$ and λ_C.

Lemma 7.6.6 1. If $\Gamma \vdash_{\lambda_C{}^-} M \in \sigma$ then $\Gamma \vdash_{\lambda_C} M \in \sigma$.

2. If $\Gamma \vdash_{\lambda_C} M \in \sigma$ then there exists τ such that $\tau \leqslant_C \sigma$ and $\Gamma \vdash_{\lambda_C{}^-} M \in \tau$.

3. If $\Gamma \vdash_{\lambda_C{}^-} M \in \sigma$ then $\Gamma \vdash_{Alg\lambda_{\leqslant}} |M| \in \sigma$.

4. If $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$ then $\Gamma \vdash_{\lambda_C{}^-} dec_\Gamma(M) \in \sigma$.

5. If $\Gamma \vdash_{\lambda_C{}^-} M \in \sigma$ then $dec_\Gamma(|M|) = M$.

6. If $\Gamma \vdash_{Alg\lambda_{\leqslant}} M \in \sigma$ then $|dec_\Gamma(M)| = M$.

Observe that lemma 7.6.2 is now a consequence of items 2 and 3 of the last lemma; and the last four items say that $Alg\lambda_{\leqslant}$ and ${\lambda_C}^-$ are equivalent in the sense that every term that can be typed in $Alg\lambda_{\leqslant}$ has its explicitly coerced version in ${\lambda_C}^-$, and every term that can be typed in ${\lambda_C}^-$ has its implicitly coerced version in $Alg\lambda_{\leqslant}$. Furthermore, the translation between the two systems does not cause any loss of information as long as no reduction is involved.

7.7 Simply typed λ-calculus and λ_C

In this section we show how to translate λ_C into $\lambda{\rightarrow}$. The first step of this translation consists of representing subtyping statements as $\lambda{\rightarrow}$-terms. For this we define an environment in which there is a typing statement for each subtyping axiom of C as follows.

Definition 7.7.1 The environment Σ_C. Let $k_{\alpha_1,\beta_1}...k_{\alpha_n,\beta_n}$ be different constants of K. Then,

$$\Sigma_C = \{k_{\alpha_1,\beta_1}{:}\alpha_1{\rightarrow}\beta_1, ..., k_{\alpha_n,\beta_n}{:}\alpha_n{\rightarrow}\beta_n\},$$

such that $(\alpha_i, \beta_i){\in}C$ if and only if $k_{\alpha_i,\beta_i}{\in}\Sigma_C$.

Observe that k_{α_i,β_i} is just a mnemonic name for a constant. Next we define the function *find* that finds a term that performs the coercion from σ to τ if $\sigma \leqslant_C \tau$. For that we use the following auxiliary definition.

Definition 7.7.2 (*Typed composition*)

$$f \circ_\sigma g = \lambda x{:}\sigma.f(gx).$$

For the sake of readability we use simply $\circ$.

If we look at C as a directed graph, there may be more than one path between two variables of C, and that is the reason why a choice is involved. An example of such situation is the following.

Example 7.7.3 Suposse $C = \{(\alpha,\beta);(\beta,\gamma);(\alpha,\delta);(\delta,\gamma)\}$. Then there are two non-convertible terms that perform the coercion from α to γ, namely

$k_{\beta,\gamma} \circ k_{\alpha,\beta}$ and $k_{\delta,\gamma} \circ k_{\alpha,\delta}$.

Therefore, we assume a choice function *choose* so that *find* is well defined.

Definition 7.7.4 $\mathit{find} : \mathbb{T} \times \mathbb{T} \times 2^{\mathbb{V}\times\mathbb{V}} \to \Lambda(\lambda\to)$.

$$\begin{array}{rcl} \mathit{find}(\sigma,\sigma,C) & = & \lambda x{:}\sigma.x, \\ \mathit{find}(\alpha,\beta,C) & = & \mathit{choose}\{k_{\gamma_{n-1},\gamma_n} \circ \ldots \circ k_{\gamma_0,\gamma_1} | \gamma_0 \equiv \alpha, \\ & & \gamma_n \equiv \beta \; \mathit{and} \; (\gamma_i,\gamma_{i+1}) \in C, \text{ for every } i \in \{1..n\}\}, \\ \mathit{find}(\sigma_1\to\sigma_2,\tau_1\to\tau_2,C) & = & \lambda x{:}\sigma_1\to\sigma_2.\mathit{find}(\sigma_2,\tau_2,C) \circ x \circ \mathit{find}(\tau_1,\sigma_1,C). \end{array}$$

where $\alpha \neq \beta$, $\sigma_1 \neq \tau_1$ and $\sigma_2 \neq \tau_2$.

Observe that *find* is a partial mapping, given that *choose* may fail, and it is only defined when the first and second arguments are in the $\leqslant_C$ relation. Note that if we impose the restriction that C must be transitively closed, then we can simplify the function *find* and redefine the second clause as

$\mathit{find}(\alpha,\beta,C) = k_{\alpha,\beta}$.

The next lemma states that if $\sigma \leqslant_C \tau$, then $\mathit{find}(\sigma,\tau,C)$ is a codification of a subtyping statement in the simply typed lambda calculus without subtyping.

Lemma 7.7.5 If $\sigma \leqslant_C \tau$, then $\vdash_{\Sigma_C} find(\sigma,\tau,C) \in \sigma\to\tau$.

Proof: By induction on the complexity of σ.

Case 1. σ is a variable. Then, by lemma 7.2.5(2), τ is also a variable. According to lemma 7.2.5(3), we can distinguish two cases.

Case 1a. $\sigma \equiv \tau$. Then, by definition, $\mathit{find}(\sigma,\tau,C) \equiv \lambda x{:}\sigma.x$. Furthermore, it holds that $\vdash_{\Sigma_C} \lambda x{:}\sigma.x \in \sigma\to\sigma$.

Case 1b. $\mathit{trans}(\sigma,\tau,C)$. Then, as $\vdash_{\lambda_C} k_{\gamma_i,\gamma_j} \in \gamma_i\to\gamma_j$, by the definition of Σ_C, we have that $\vdash_{\Sigma_C} k_{\gamma_{n-1},\gamma_n} \circ \ldots \circ k_{\gamma_0,\gamma_1} \in \sigma\to\tau$.

Case 2. $\sigma \equiv \sigma_1\to\sigma_2$. Then, by lemma 7.2.5(2), $\tau \equiv \tau_1\to\tau_2$. If $\sigma \equiv \tau$, the result follows as in Case 1a. Otherwise, by lemma 7.2.6, $\tau_1 \leqslant_C \sigma_1$ and $\sigma_2 \leqslant_C \tau_2$. By the induction hypothesis, we have

$\vdash_{\Sigma_C} \mathit{find}(\tau_1,\sigma_1,C) \in \tau_1\to\sigma_1$ and
$\vdash_{\Sigma_C} \mathit{find}(\sigma_2,\tau_2,C) \in \sigma_2\to\tau_2$.

Hence, $\vdash_{\Sigma_C} \lambda x{:}\sigma_1\to\sigma_2.\mathit{find}(\sigma_2,\tau_2,C) \circ x \circ \mathit{find}(\tau_1,\sigma_1,C)\sigma\to\tau \in$. □

Observe that, actually, this result could be obtained as a corollary of these two facts:

- If $\sigma \leqslant_C \tau$, then $\mathit{find}(\sigma,\tau,C)$ is defined, and
- If $\mathit{find}(\sigma,\tau,C)$ is defined, then $\vdash_{\Sigma_C} \mathit{find}(\sigma,\tau,C) \in \sigma{\to}\tau$.

With the following definition we can relate the system λ_C with $\lambda{\to}$.

Definition 7.7.6 $M^{\to}$ (The $\lambda{\to}$ version of M). $\_^{\to}{:}\Lambda(\lambda_C) \to \Lambda(\lambda{\to})$.

$$\begin{array}{rcl} x^{\to} & = & x, \\ (\lambda x{:}\sigma.M)^{\to} & = & \lambda x{:}\sigma.M^{\to}, \\ (MN)^{\to} & = & M^{\to}N^{\to}, \\ (c_{\sigma,\tau}{<}M{>})^{\to} & = & \mathit{find}(\sigma,\tau,C)M^{\to}. \end{array}$$

Then, $M^{\to}$ is obtained from M by replacing the coercion constants by terms that will perform the corresponding coercion.

Note that $\_^{\to}$ is a total function because the existence of $c_{\sigma,\tau}$ implies $\sigma \leqslant_C \tau$.

Proposition 7.7.7 Let M in $\Lambda(\lambda_C)$. Then,
If $\Gamma \vdash_{\lambda_C} M \in \sigma$, then $\Gamma \vdash_{\Sigma_C} M^{\to} \in \sigma$.

Proof: By induction on the derivation of $\Gamma \vdash_{\lambda_C} M \in \sigma$.

T-Var Let $M \equiv x$. Then $x{:}\sigma{\in}\Gamma$. As $x^{\to} \equiv x$, using the T-Var rule in $\lambda{\to}$ we get $\Gamma \vdash_{\Sigma_C} x^{\to} \in \sigma$.

T-Abs Let $M \equiv \lambda x{:}\rho.N$ and $\sigma \equiv \rho{\to}\tau$. Then $\Gamma \vdash_{\lambda_C} \lambda x{:}\rho.N \in \rho{\to}\tau$ follows from $\Gamma, x{:}\rho \vdash_{\lambda_C} N \in \tau$. By the induction hypothesis, it follows that $\Gamma, x{:}\rho \vdash_{\Sigma_C} N^{\to} \in \tau$. By T-Abs, we conclude $\Gamma \vdash_{\Sigma_C} \lambda x{:}\rho.N^{\to} \in \rho{\to}\tau$, and, by the definition of $\_^{\to}$, $\Gamma \vdash_{\Sigma_C} (\lambda x{:}\rho.N)^{\to} \in \rho{\to}\tau$.

T-App $M \equiv NP$, and $\Gamma \vdash_{\lambda_C} NP \in \sigma$ follows from the statements $\Gamma \vdash_{\lambda_C} P \in \tau$ and $\Gamma \vdash_{\lambda_C} N \in \tau{\to}\sigma$ for some τ. By the induction hypothesis, the T-App rule of $\lambda{\to}$, and the definition of $\_^{\to}$, the result follows.

T-Coerce M is $c_{\tau,\sigma}{<}N{>}$ and $\Gamma \vdash_{\lambda_C} c_{\tau,\sigma}{<}N{>} \in \sigma$ follows from $\Gamma \vdash_{\lambda_C} N \in \tau$ and $\tau \leqslant_C \sigma$. By the induction hypothesis, $\Gamma \vdash_{\Sigma_C} N^{\to} \in \tau$, and due to lemma 7.7.5, $\vdash_{\Sigma_C} \mathit{find}(\tau,\sigma,C) \in \tau{\to}\sigma$. Finally, using the weakening lemma together with the T-App rule of $\lambda{\to}$, and the definition of $\_^{\to}$, we get $\Gamma \vdash_{\Sigma_C} (c_{\tau,\sigma}{<}N{>})^{\to} \in \sigma$. □

Proposition 7.7.8 Let M in $\Lambda(\lambda_C)$. Then

$$\Gamma \vdash_{\Sigma_C} M^{\to} \in \sigma \quad \Rightarrow \quad \Gamma \vdash_{\lambda_C} M \in \sigma.$$

Proof: By induction on the complexity of M.

Case 1. $M \equiv x{\in}V$. Then $M^{\to} \equiv x$. By the free variable lemma of $\lambda{\to}$, we can conclude that $x{:}\sigma{\in}\Gamma$, and, by T-Var $\Gamma \vdash_{\lambda_C} x \in \sigma$.

Case 2. $M \equiv PQ$. Then $M^{\rightarrow} \equiv P^{\rightarrow}Q^{\rightarrow}$. By generation, there exists τ such that

$$\Gamma \vdash_{\Sigma_C} P^{\rightarrow} \in \tau{\rightarrow}\sigma \text{ and } \Gamma \vdash_{\Sigma_C} Q^{\rightarrow} \in \tau.$$

By the induction hypothesis,

$$\Gamma \vdash_{\lambda_C} P \in \tau{\rightarrow}\sigma \text{ and } \Gamma \vdash_{\lambda_C} Q \in \tau.$$

By T-App, we get $\Gamma \vdash_{\lambda_C} PQ \in \sigma$.

Case 3. $M \equiv \lambda x{:}\tau.N$. Then $M^{\rightarrow} \equiv \lambda x{:}\tau.N^{\rightarrow}$. By generation, it follows that $\Gamma, x{:}\tau \vdash_{\Sigma_C} N^{\rightarrow} \in \rho$ and $\sigma \equiv \tau{\rightarrow}\rho$. We now can apply the induction hypothesis and we get $\Gamma, x{:}\tau \vdash_{\lambda_C} N \in \rho$, and to that we only have to apply the T-Abs rule to get $\Gamma \vdash_{\lambda_C} \lambda x{:}\tau.N \in \sigma$.

Case 4. $M \equiv c_{\tau,\sigma}{<}N{>}$. Then, $M^{\rightarrow} \equiv \mathit{find}(\tau, \sigma, C)N^{\rightarrow}$. Recall that the constant $c_{\tau,\sigma}$ exists if and only if $\tau \leqslant_C \sigma$. Then using lemma 7.7.5, we know that $\vdash_{\Sigma_C} \mathit{find}(\tau, \sigma, C) \in \tau{\rightarrow}\sigma$. By generation, the unicity of types lemma, and the weakening lemma, it follows that $\Gamma \vdash_{\Sigma_C} N^{\rightarrow} \in \tau$. By the induction hypothesis, we have $\Gamma \vdash_{\lambda_C} N \in \tau$. Finally, by applying T-Coerce, it follows that $\Gamma \vdash_{\lambda_C} c_{\tau,\sigma}{<}N{>} \in \sigma$. □

Putting together the last two propositions we can state the following theorem.

Theorem 7.7.9 Let M in $\Lambda(\lambda_C)$. Then,

$$\Gamma \vdash_{\lambda_C} M \in \sigma \quad \Leftrightarrow \quad \Gamma \vdash_{\Sigma_C} M^{\rightarrow} \in \sigma.$$

This last result can be read as follows. The simply typed λ-calculus without subtyping is an appropriate model for the simply typed λ-calculus with subtyping. We can also extract from it the conclusion that the simply typed λ-calculus with explicit subtyping is a conservative extension of the λ-calculus without subtyping, because if M is a $\lambda{\rightarrow}$ term then $M^{\rightarrow} \equiv M$.

Metatheory of λ_C

The system λ_C is of independent interest. The previous theorem and the fact that $M^{\rightarrow}$ is always defined, imply that the type checking and type inference problems are decidable given that the corresponding problems in $\lambda{\rightarrow}$ are (see [Bar92]). Recall that a pseudoterm M is called *strongly* normalizing if there is no infinite reduction chain starting from M. We reduce the strong normalization property of λ_C to the strong normalization result for $\lambda{\rightarrow}$ (see [Bar92]).

First, we prove some auxiliary results first.

Lemma 7.7.10 (*Substitution lemma*) Let M, N in $\Lambda(\lambda_C)$ and $x{\in}V$. Then

$$(M[x{:=}N])^{\rightarrow} = M^{\rightarrow}[x{:=}N^{\rightarrow}]$$

Proof: By induction on the complexity of M. We consider here only two cases, the missing ones are proven in a similar way.

- $M \equiv x$.

$$\begin{array}{rcll} (x[x{:=}N])^{\rightarrow} & = & N^{\rightarrow} & \text{by definition of } [:=]. \\ & = & x[x{:=}N^{\rightarrow}] & \text{by definition of } [:=]. \\ & = & x^{\rightarrow}[x{:=}N^{\rightarrow}] & \text{by definition of } \_^{\rightarrow}. \end{array}$$

- $M \equiv c_{\sigma,\tau}{<}P{>}$.

$$\begin{array}{rcll} (c_{\sigma,\tau}{<}P{>}[x{:=}N])^{\rightarrow} & = & c_{\sigma,\tau}{<}P[x{:=}N]{>}^{\rightarrow} & \text{by definition of } [:=]. \\ & = & \mathit{find}(\sigma,\tau,C)(P[x{:=}N])^{\rightarrow} & \text{by definition of } \_^{\rightarrow}. \\ & = & \mathit{find}(\sigma,\tau,C)(P^{\rightarrow}[x{:=}N^{\rightarrow}]) & \text{by the induction} \\ & & & \text{hypothesis.} \\ & = & (\mathit{find}(\sigma,\tau,C)P^{\rightarrow})[x{:=}N^{\rightarrow}] & \\ & = & (c_{\sigma,\tau}{<}P{>})^{\rightarrow}[x{:=}N^{\rightarrow}] & \text{by definition of } \_^{\rightarrow}. \end{array}$$

Observe that $\mathit{find}(\sigma,\tau,C)$ is a closed term. □

Lemma 7.7.11 (*Reduction preservation*) Let M, N in $\Lambda(\lambda_C)$. Then, if $M \rightarrow_\beta N$, then $M^{\rightarrow} \rightarrow_\beta N^{\rightarrow}$.

Proof: By induction on the complexity of M.

Case 1. If $M{\in}V$ the result is vacuously true.

Case 2. $M \equiv \lambda x{:}\sigma.M_1$. Then N is of the form $\lambda x{:}\sigma.N_1$, where $M_1 \rightarrow_\beta N_1$. By the induction hypothesis and the definition of $\_^{\rightarrow}$, the result holds.

Case 3. $M \equiv M_1 M_2$. Then we have the following three cases.

1. $N \equiv N_1 M_2$, where $M_1 \rightarrow_\beta N_1$,
2. $N \equiv M_1 N_2$, where $M_2 \rightarrow_\beta N_2$ and,
3. $M \equiv (\lambda x{:}\sigma.P)M_2$ and $N \equiv P[x{:=}M_2]$.

The first two cases follow from the induction hypothesis and the definition of $\_^{\rightarrow}$ and the third is a consequence of the substitution lemma.

Case 4. $M \equiv c_{\sigma,\tau}{<}M_1{>}$. The result follows from the induction hypothesis and the definition of $\_^{\rightarrow}$. □

Theorem 7.7.12 (*Strong normalization for λ_C*)
If $\Gamma \vdash_{\lambda_C} M \in \sigma$, then M is strongly normalizing.

Proof: Suppose, towards a contradiction, that M is not strongly normalizing. This means that there exists an infinite reduction chain starting from M. By the reduction preservation lemma (7.7.11), we know that there is also an infinite chain of reductions starting from $M^{\rightarrow}$, and using proposition 7.7.7, we know that

$$\Gamma \vdash_{\Sigma_C} M^{\rightarrow} \in \sigma.$$

But, we also know that $\lambda{\rightarrow}$ has the strong normalization property which yields a contradiction. Hence, M is strongly normalizing. □

7.8 Confluence

Using the results about confluence of orthogonal combinatory reduction systems (CRSs) in [vR92], we can state that the systems λ_C and $\lambda_{\leqslant}$ are confluent as a consequence of what we have already proven in this chapter. The sets of pseudo-terms $\Lambda(\lambda_C)$ with the β-reduction rule and $\Lambda(\lambda_{\leqslant})$ with the β-reduction rule are, according to the definitions of [vR92], two orthogonal CRSs. The subject reduction property of λ_C (7.5.6) and of $\lambda_{\leqslant}$ (7.4.20), imply that the corresponding sets of legal terms are two substructures of $\Lambda(\lambda_C)$ and $\Lambda(\lambda_{\leqslant})$ respectively. Since substructures of orthogonal CRSs are also orthogonal, it follows that the systems λ_C and $\lambda_{\leqslant}$ are confluent.

7.9 Conclusions

In this chapter we analyze two different styles of subtyping, subtyping with implicit coercions and subtyping with explicit coercions. We define and study two alternative presentations of subtyping for simply typed lambda calculus. The first one $\lambda_{\leqslant}$, a system with implicit coercions, and the second one λ_C, a system with explicit coercions. We show that the system $\lambda_{\leqslant}$ can be translated into λ_C, and, in its turn, λ_C can be translated into $\lambda{\rightarrow}$. In other words, both disciplines can be compiled into the simply typed lambda calculus without subtyping.

Chapter 8

Future research

The study of the meta-theory of a rich typed lambda-calculus such as $F^{\omega}_{\wedge}$ has drawn our attention towards open and challenging problems such as the ones listed below.

A normalizing fragment of $F^{\omega}_{\wedge}$

Although the reduction on types $\rightarrow_{\beta\wedge}$ is strongly normalizing on well-kinded types, the reductions on terms are not strongly normalizing for the simple reason that every closed term can be assigned a type. Therefore we would like to characterize a normalizing subset of the language of terms. A similar situation arises for the intersection type discipline à la Curry studied by the group in Torino. In their framework, a term e is strongly normalizing if and only if there exists a derivation of a typing statement with e as subject which does not contain the maximal type ω. In our case, this statement is not true. A very simple counterexample is the term

$$\lambda x{:}\top^{\star}.x,$$

which is in normal form and all whose derivations contain the maximal type $\top^{\star}$. This problem is subject of current research by Mariangiola Dezani and the author of this thesis.

Bounded operator abstraction

In $F^{\omega}_{\wedge}$ and also in $F^{\omega}_{\leq}$, abstraction on types is of the form $\Lambda X{:}K.T$, and its associated formation rule is

$$\frac{\Gamma,\, X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K_2}{\Gamma \vdash \Lambda X{:}K_1.T_2 \in K_1{\rightarrow}K_2.} \qquad (\textsc{K-OAbs})$$

A natural enrichment of the theory would replace $\Lambda X{:}K.T$ by $\Lambda X{\leq}S{:}K.T$, using the following formation rule.

$$\frac{\Gamma,\, X{\leq}S{:}K_1 \vdash T_2 \in K_2}{\Gamma \vdash \Lambda X{\leq}S{:}K.T \in \forall X{\leq}S{:}K_1.K_2.} \qquad (\textsc{K-Bounded-OAbs})$$

This means that we need to modify the language of kinds because S, the bound of X, is required in order to assign a kind to a type operator application. Then the formation rule is as follows.

$$\frac{\Gamma \vdash T \in \forall X{\leq}S{:}K_1.K_2 \qquad \Gamma \vdash S' \leq S}{\Gamma \vdash T\,S' \in K_2[X{\leftarrow}S']} \qquad \text{(K-Bounded-OApp)}$$

Consequently, the kind inference and kind checking algorithms become more complicated, since they involve checking a subtyping judgement. The meta-theory and applications of this extension are the subject of current research by Paula Severi and the author of this thesis.

Subtyping dependent types

The type-theoretic foundations of proof development systems include the Automath family of calculi [dB80], the Edinburgh Logical Framework [HHP92], the Calculus of Constructions [CH88], Extended Calculus of Constructions [Luo90], and Martin-Löf type theory [SNP90]. Implementations of these theories have been used in a number of proof development systems (Automath at Eindhoven, Coq at INRIA, Lego at the LFCS, and Alf at Göteborg). A common feature of these systems is their heavy reliance on dependent types, and the consequent difficulty of their meta-theoretic analysis.

The interaction of subtyping with dependent types seems to be the principal obstacle to its integration in proof development systems. Some work in this area has been started by Cardelli [Car87, Car88b], who gives basic definitions and some ideas about type checking algorithms, and by Pfenning [Pfe93], who has proposed variants of the Logical Framework and the Calculus of Constructions with refinement types, a simple form of intersection types whose interaction with type conversion and dependency is strictly controlled. Aside from these preliminary efforts, the area remains unexplored.

The system λP is an extension of the simply typed lambda calculus with dependent types [Bar92]. The meta-theory of $\lambda P_{\leq}$, an extension of λP with subtyping, is being developed by David Aspinall and the author of this thesis.

Appendix A

Summary of Definitions

A.1 $F^{\omega}_{\wedge}$

A.1.1 Reduction rules for types

1. $(\Lambda X{:}K.T_1)T_2 \to_{\beta\wedge} T_1[X{\leftarrow}T_2]$
2. $S \to \bigwedge^{\star}[T_1..T_n] \to_{\beta\wedge} \bigwedge^{\star}[S{\to}T_1 \mathrel{..} S{\to}T_n]$
3. $\forall X{\leq}S{:}K.\bigwedge^{\star}[T_1..T_n] \to_{\beta\wedge} \bigwedge^{\star}[\forall X{\leq}S{:}K.T_1 \mathrel{..} \forall X{\leq}S{:}K.T_n]$
4. $\Lambda X{:}K_1.\bigwedge^{K_2}[T_1..T_n] \to_{\beta\wedge} \bigwedge^{K_1\to K_2}[\Lambda X{:}K_1.T_1 \mathrel{..} \Lambda X{:}K_1.T_n]$
5. $\bigwedge^{K_1\to K_2}[T_1..T_n]\,U \to_{\beta\wedge} \bigwedge^{K_2}[T_1\,U \mathrel{..} T_n\,U]$
6. $\bigwedge^{K}[T_1 \mathrel{..} \bigwedge^{K}[S_1..S_n] \mathrel{..} T_m] \to_{\beta\wedge} \bigwedge^{K}[T_1 \mathrel{..} S_1..S_n \mathrel{..} T_m]$

A.1.2 Reduction rules for terms

1. $(\lambda x{:}T_1.e_1)e_2 \to_{\beta fors} e_1[x{\leftarrow}e_2]$
2. $(\lambda X{\leq}T_1{:}K_1.e)T \to_{\beta fors} e[X{\leftarrow}T]$
3. $(\mathrm{for}(X{\in}T_1..T_n)e_1)e_2 \to_{\beta fors} \mathrm{for}(X{\in}T_1..T_n)e_1\,e_2$
4. $\mathrm{for}(X{\in}T_1..T_n)e \to_{\beta fors} e$, if $X \notin \mathrm{FV}(e)$

A.1.3 Context formation rules

$$\emptyset \vdash \mathrm{ok} \qquad (\text{C-}\textsc{Empty})$$

$$\frac{\Gamma \vdash T \in \star \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma,\, x{:}T \vdash \mathrm{ok}} \qquad (\text{C-}\textsc{Var})$$

$$\frac{\Gamma \vdash T \in K \qquad X \notin \mathrm{dom}(\Gamma)}{\Gamma,\, X{\leq}T{:}K \vdash \mathrm{ok}} \qquad (\text{C-}\textsc{TVar})$$

A.1.4 Kinding rules

$$\frac{\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \text{ok}}{\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash X \in K} \tag{K-TVar}$$

$$\frac{\Gamma \vdash T_1 \in \star \qquad \Gamma \vdash T_2 \in \star}{\Gamma \vdash T_1{\rightarrow}T_2 \in \star} \tag{K-Arrow}$$

$$\frac{\Gamma,\ X{\leq}T_1{:}K_1 \vdash T_2 \in \star}{\Gamma \vdash \forall(X{\leq}T_1{:}K_1)T_2 \in \star} \tag{K-All}$$

$$\frac{\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K_2}{\Gamma \vdash \Lambda(X{:}K_1)T_2 \in K_1{\rightarrow}K_2} \tag{K-OAbs}$$

$$\frac{\Gamma \vdash S \in K_1{\rightarrow}K_2 \qquad \Gamma \vdash T \in K_1}{\Gamma \vdash S\,T \in K_2} \tag{K-OApp}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \text{for each } i,\ \Gamma \vdash T_i \in K}{\Gamma \vdash \bigwedge^K[T_1..T_n] \in K} \tag{K-Meet}$$

A.1.5 Subtyping rules

$$\frac{\Gamma \vdash S \in K \qquad \Gamma \vdash T \in K \qquad S =_{\beta\wedge} T}{\Gamma \vdash S \leq T} \tag{S-Conv}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U} \tag{S-Trans}$$

$$\frac{\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \text{ok}}{\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash X \leq T} \tag{S-TVar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma \vdash S_2 \leq T_2 \qquad \Gamma \vdash S_1{\rightarrow}S_2 \in \star}{\Gamma \vdash S_1{\rightarrow}S_2 \leq T_1{\rightarrow}T_2} \tag{S-Arrow}$$

$$\frac{\Gamma,\ X{\leq}U{:}K \vdash S \leq T \qquad \Gamma \vdash \forall X{\leq}U{:}K.S \in \star}{\Gamma \vdash \forall X{\leq}U{:}K.S \leq \forall X{\leq}U{:}K.T} \tag{S-All}$$

$$\frac{\Gamma,\ X{\leq}\top^{K}{:}K \vdash S \leq T}{\Gamma \vdash \Lambda X{:}K.S \leq \Lambda X{:}K.T} \tag{S-OAbs}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash S\,U \in K}{\Gamma \vdash S\,U \leq T\,U} \tag{S-OApp}$$

$$\frac{\text{for each } i,\ \Gamma \vdash S \leq T_i \qquad \Gamma \vdash S \in K}{\Gamma \vdash S \leq \bigwedge^K[T_1..T_n]} \tag{S-Meet-G}$$

$$\frac{\Gamma \vdash \bigwedge^K[T_1..T_n] \in K}{\Gamma \vdash \bigwedge^K[T_1..T_n] \leq T_i} \tag{S-Meet-LB}$$

A.1.6 Typing rules

$$\frac{\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash \text{ok}}{\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash x \in T} \qquad (\textsc{T-Var})$$

$$\frac{\Gamma,\, x{:}T_1 \vdash e \in T_2}{\Gamma \vdash \lambda x{:}T_1.e \in T_1{\to}T_2} \qquad (\textsc{T-Abs})$$

$$\frac{\Gamma \vdash f \in T_1{\to}T_2 \qquad \Gamma \vdash a \in T_1}{\Gamma \vdash f\, a \in T_2} \qquad (\textsc{T-App})$$

$$\frac{\Gamma,\, X{\le}T_1{:}K_1 \vdash e \in T_2}{\Gamma \vdash \lambda X{\le}T_1{:}K_1.e \in \forall X{\le}T_1{:}K_1.T_2} \qquad (\textsc{T-TAbs})$$

$$\frac{\Gamma \vdash f \in \forall X{\le}T_1{:}K_1.T_2 \qquad \Gamma \vdash S \le T_1}{\Gamma \vdash f\, S \in T_2[X{\leftarrow}S]} \qquad (\textsc{T-TApp})$$

$$\frac{\Gamma \vdash e[X{\leftarrow}S] \in T \qquad S \in \{S_1..S_n\}}{\Gamma \vdash \text{for}(X{\in}S_1..S_n)e \in T} \qquad (\textsc{T-For})$$

$$\frac{\Gamma \vdash \text{ok} \qquad \text{for each } i,\ \Gamma \vdash e \in T_i}{\Gamma \vdash e \in \bigwedge^{\star}[T_1..T_n]} \qquad (\textsc{T-Meet})$$

$$\frac{\Gamma \vdash e \in S \qquad \Gamma \vdash S \le T}{\Gamma \vdash e \in T} \qquad (\textsc{T-Subsumption})$$

A.1.7 Subtype checking, ***Alg***$F^{\omega}_{\wedge}$ subtyping rules

$$\frac{\Gamma \vdash X \in K}{\Gamma \vdash_{\mathit{Alg}} X \le X} \qquad (\textsc{AlgS-TVarRefl})$$

$$\frac{\Gamma \vdash T\, S \in K}{\Gamma \vdash_{\mathit{Alg}} T\, S \le T\, S} \qquad (\textsc{AlgS-OAppRefl})$$

$$\frac{\Gamma \vdash_{\mathit{Alg}} \Gamma(X) \le A \qquad X \not\equiv A}{\Gamma \vdash_{\mathit{Alg}} X \le A} \qquad (\textsc{AlgS-TVar})$$

$$\frac{\Gamma \vdash_{\mathit{Alg}} T_1 \le S_1 \qquad \Gamma \vdash_{\mathit{Alg}} S_2 \le T_2 \qquad \Gamma \vdash S_1{\to}S_2 \in \star}{\Gamma \vdash_{\mathit{Alg}} S_1{\to}S_2 \le T_1{\to}T_2} \qquad (\textsc{AlgS-Arrow})$$

$$\frac{\Gamma,\, X{\le}U{:}K \vdash_{\mathit{Alg}} S \le T \qquad \Gamma \vdash \forall X{\le}U{:}K.S \in \star}{\Gamma \vdash_{\mathit{Alg}} \forall X{\le}U{:}K.S \le \forall X{\le}U{:}K.T} \qquad (\textsc{AlgS-All})$$

$$\frac{\Gamma,\, X{\le}\top^{K}{:}K \vdash_{\mathit{Alg}} S \le T}{\Gamma \vdash_{\mathit{Alg}} \Lambda X{:}K.S \le \Lambda X{:}K.T} \qquad (\textsc{AlgS-OAbs})$$

$$\frac{\Gamma \vdash_{\mathit{Alg}} (\mathit{lub}_{\Gamma}(T\, S))^{\mathit{nf}} \le A \qquad \Gamma \vdash T\, S \in K \qquad T\, S \not\equiv A}{\Gamma \vdash_{\mathit{Alg}} T\, S \le A} \qquad (\textsc{AlgS-OApp})$$

$$\frac{\forall i{\in}\{1..m\}\, \Gamma \vdash_{\mathit{Alg}} A \le T_i \qquad \Gamma \vdash A \in K}{\Gamma \vdash_{\mathit{Alg}} A \le \bigwedge^{K}[T_1..T_m]} \qquad (\textsc{AlgS-}\forall)$$

$$\frac{\exists j{\in}\{1..n\}\, \Gamma \vdash_{\mathit{Alg}} S_j \le A \qquad \forall k{\in}\{1..n\}\, \Gamma \vdash S_k \in K}{\Gamma \vdash_{\mathit{Alg}} \bigwedge^{K}[S_1..S_n] \le A} \qquad (\textsc{AlgS-}\exists)$$

$$\frac{\forall i{\in}\{1..m\}\ \exists j{\in}\{1..n\}\ \Gamma \vdash_{\mathit{Alg}} S_j \leq T_i \qquad \forall k{\in}\{1..n\}\ \Gamma \vdash S_k \in K}{\Gamma \vdash_{\mathit{Alg}} \bigwedge^K[S_1..S_n] \leq \bigwedge^K[T_1..T_m]} \tag{AlgS-$\forall\exists$}$$

A.1.8 Type inference

Definition [Homomorphic extension of *lub* to intersections, lub^*]

$$\begin{aligned}
\mathit{lub}^*_\Gamma(X) &= \Gamma(X),\\
\mathit{lub}^*_\Gamma(S\,T) &= \mathit{lub}^*_\Gamma(S)\,T,\\
\mathit{lub}^*_\Gamma(\bigwedge^K[T_1..T_n]) &= \bigwedge^K[T'_1..T'_n], \qquad \text{if } \exists i{\in}\{1..n\} \text{ such that } \mathit{lub}^*_\Gamma(T_i){\downarrow},
\end{aligned}$$

where T'_i is $\mathit{lub}^*_\Gamma(T_i)$, if $\mathit{lub}^*_\Gamma(T_i){\downarrow}$, and T_i otherwise.

Definition [Functional Least Upper Bound] The functional least upper bound of a type T, in a context Γ, $\mathit{flub}_\Gamma(T)$ is defined as follows.

$$\mathit{flub}_\Gamma(T) = \begin{cases} \mathit{flub}_\Gamma(\mathit{lub}^*_\Gamma(T^{\mathit{nf}})), & \text{if } \mathit{lub}^*_\Gamma(T^{\mathit{nf}}){\downarrow};\\ T^{\mathit{nf}}, & \text{otherwise.[1]} \end{cases}$$

Definition [*arrows* and *alls*]

1. $$\begin{aligned}
\mathit{arrows}\,(T_1{\to}T_2) &= \{T_1{\to}T_2\},\\
\mathit{arrows}\,(\bigwedge^\star[T_1..T_n]) &= \cup_{i\in\{1..n\}}\,\mathit{arrows}\,(T_i),\\
\mathit{arrows}\,(T) &= \emptyset,\\
&\quad\ \text{if } T \not\equiv T_1{\to}T_2 \text{ and } T \not\equiv \bigwedge^\star[T_1..T_n].
\end{aligned}$$

2. $$\begin{aligned}
\mathit{alls}\,(\forall X{\leq}T_1{:}K.T_2) &= \{\forall X{\leq}T_1{:}K.T_2\},\\
\mathit{alls}\,(\bigwedge^\star[T_1..T_n]) &= \cup_{i\in\{1..n\}}\,\mathit{alls}\,(T_i),\\
\mathit{alls}\,(T) &= \emptyset,\\
&\quad\ \text{if } T \not\equiv \forall X{\leq}T_1{:}K.T_2 \text{ and } T \not\equiv \bigwedge^\star[T_1..T_n].
\end{aligned}$$

Type inference rules

$$\frac{\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash \text{ok}}{\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash_{\mathit{inf}} x \in T} \tag{AT-Var}$$

$$\frac{\Gamma,\ x{:}T_1 \vdash_{\mathit{inf}} e \in T_2}{\Gamma \vdash_{\mathit{inf}} \lambda x{:}T_1.e \in T_1{\to}T_2} \tag{AT-Abs}$$

$$\frac{\Gamma \vdash_{\mathit{inf}} f \in T \qquad \Gamma \vdash_{\mathit{inf}} a \in S}{\Gamma \vdash_{\mathit{inf}} f\ a \in \bigwedge^\star[T_i \mid S_i{\to}T_i \in \mathit{arrows}\,(\mathit{flub}_\Gamma(T)) \text{ and } \Gamma \vdash S \leq S_i]} \tag{AT-App}$$

$$\frac{\Gamma,\ X{\leq}T_1{:}K_1 \vdash_{\mathit{inf}} e \in T_2}{\Gamma \vdash_{\mathit{inf}} \lambda X{\leq}T_1{:}K_1.e \in \forall X{\leq}T_1{:}K_1.T_2} \tag{AT-TAbs}$$

$$\frac{\Gamma \vdash_{\mathit{inf}} f \in T}{\Gamma \vdash_{\mathit{inf}} f\ S \in \bigwedge^\star[T_i[X{\leftarrow}S] \mid \forall X{\leq}S_i{:}K.T_i \in \mathit{alls}\,(\mathit{flub}_\Gamma(T)) \text{ and } \Gamma \vdash S \leq S_i]} \tag{AT-TApp}$$

$$\frac{\text{for all } i{\in}\{1..n\} \quad \Gamma \vdash_{\mathit{inf}} e[X{\leftarrow}S_i]{\in}T_i}{\Gamma \vdash_{\mathit{inf}} \text{for}(X{\in}S_1..S_n)e \in \bigwedge^\star[T_1..T_n]} \tag{AT-For}$$

A.2 First order subtyping

A.2.1 $\lambda_{\leqslant}$

$$\Gamma \vdash_{\lambda_{\leqslant}} x \in \sigma \quad \text{if } x{:}\sigma{\in}\Gamma \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}\sigma \vdash_{\lambda_{\leqslant}} M \in \tau}{\Gamma \vdash_{\lambda_{\leqslant}} \lambda x{:}\sigma.M \in \sigma{\to}\tau} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma{\to}\tau \qquad \Gamma \vdash_{\lambda_{\leqslant}} N \in \sigma}{\Gamma \vdash_{\lambda_{\leqslant}} MN \in \tau} \qquad \text{(T-App)}$$

$$\frac{\Gamma \vdash_{\lambda_{\leqslant}} M \in \sigma \qquad \sigma \leqslant_C \tau}{\Gamma \vdash_{\lambda_{\leqslant}} M \in \tau} \qquad \text{(T-Subsumption)}$$

A.2.2 λ_C

$$\Gamma \vdash_{\lambda_C} x \in \sigma, \quad \text{if } x{:}\sigma{\in}\Gamma \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}\sigma \vdash_{\lambda_C} M \in \tau}{\Gamma \vdash_{\lambda_C} \lambda x{:}\sigma.M \in \sigma{\to}\tau} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash_{\lambda_C} M \in \sigma{\to}\tau \qquad \Gamma \vdash_{\lambda_C} N \in \sigma}{\Gamma \vdash_{\lambda_C} MN \in \tau} \qquad \text{(T-App)}$$

$$\frac{\Gamma \vdash_{\lambda_C} M \in \sigma \qquad \sigma \leqslant_C \tau}{\Gamma \vdash_{\lambda_C} c_{\sigma,\tau}{<}M{>} \in \tau} \qquad \text{(T-Coerce)}$$

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118), and as DEC Systems Research Center Research Report number 62, August 1990.

[AH87] A. Avron and I. Honsell, F.and Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, LFCS, University of Edinburgh, July 1987.

[Bar84] H. P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

[Bar90] Henk P. Barendregt. Functional programming and lambda calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 7, pages 323–363. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Bar92] H. Barendregt. Lambda calculi with types. In T. S. E. Maibaum S. Abramsky, Dov M. Gabbay, editor, *Handbook of Login in Computer Science*, volume 2, pages 117–309. Clarendon Press. Oxford, 1992.

[BCD83] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[BCGS91] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[BL90] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in [GM94]. An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

[BM92] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992.

[BMM90] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. In Huet [Hue90], pages 213–272. Also appeared in Information and Computation 84, 1 (January 1990).

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. A preliminary version appeared in POPL 1993 under the title “Safe Type Checking in a Statically Typed Object-Oriented Programming Language”.

[Car87] Luca Cardelli. Typechecking dependent types and subtypes. In *Proc. of the Workshop on Foundations of Logic and Functional Programming*, Trento, Italy, December 1987.

[Car88a] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Preliminary version in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

[Car88b] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[Car90] Luca Cardelli. Notes about $\mathrm{F}^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. Also in [GM94].

[CC90] Felice Cardone and Mario Coppo. Two extensions of Curry’s type inference system. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing, pages 19–76. Academic Press, 1990.

[CCH$^+$89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CD80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

[CDC78] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19:139–156, 1978.

[CDdL93] Felice Cardone, Mariangiola Dezani-Ciancaglini, and Ugo de’ Liguoro. Combining type disciplines, 1993. Manuscript.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $\mathrm{F}_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in [GM94].

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. Also in [GM94].

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *Amer. J. Math*, 58:354–363, 1936.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in [GM94], and available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[Com94] Adriana B. Compagnoni. Subtyping in $F^{\omega}_{\wedge}$ is decidable. Technical Report ECS-LFCS-94-281, LFCS, University of Edinburgh, January 1994. Presented at Computer Science Logic, September 1994, under the title "Decidability of Higher-Order Subtyping with Intersection Types".

[CP93] Adriana B. Compagnoni and Benjamin C. Pierce. Multiple inheritance via intersection types. Technical Report ECS-LFCS-93-275, LFCS, University of Edinburgh, August 1993. Also available as Catholic University Nijmegen computer science technical report 93-18.

[CP94] Giuseppe Castagna and Benjamin Pierce. Decidable bounded quantification. In *Proceedings of Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, OR.* ACM, January 1994.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[dB80] Nicolas G. 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 589–606. Academic Press, 1980.

[FM94] Kathleen Fisher and John Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software, Sendai, Japan*, pages 844–885. Springer-Verlag, April 1994. LNCS 789.

[FMRS90] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. In *Fifth Annual Symposium on Logic in Computer Science (Philadelphia, PA)*, pages 346–354. IEEE Computer Society Press, June 1990.

[Ghe90] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Università di Pisa, March 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

[Ghe94] Giorgio Ghelli, January 1994. Message to the `Types` mailing list.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

[HHP92] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 1995. To appear. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[Hue90] Gérard Huet, editor. *Logical Foundations of Functional Programming.* University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[Lan65] P. J. Landin. A correspondence between ALGOL–60 and Church's lambda notation. *Communications of the ACM*, 8:89–101, 1965.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[Mar73] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118, Amsterdam, 1973. North Holland.

[Mit84] J. C. Mitchell. *Lambda Calculus Models of Typed Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1984.

[Mit90a] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990. Also in in [GM94].

[Mit90b] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In Huet [Hue90], pages 195–212.

[Mit90c] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 315–328, May 1993.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Rey80] John Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science. Springer-Verlag, January 1980. Also in [GM94].

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Sco76] Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[SNP90] Jan Smith, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[SP94] Martin Steffen and Benjamin Pierce. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, June 1994. An earlier version appeared as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, February 1994.

[vR92] Femke. van Raamsdonk. A simple proof of confluence for weakly orthogonal combinatory reduction systems. Technical Report CS-R9234, CWI, Amsterdam, August 1992.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae X*, pages 115–122, 1987. North Holland.

Curriculum Vitæ

July 1984 - December 1986	Courses of the *Licenciatura en Ciencias Matemáticas* at Universidad de Buenos Aires, Argentina.
March 1987 - December 1989	*Licenciatura en Informática* at Escuela Superior Latino-Americana de Informática, Buenos Aires, Argentina.
January 1990	Undergraduate degree: *Licenciada en Informática* Escuela Superior Latino-Americana de Informática,
June 1990 - May 1994	PhD program at the Katholieke Universiteit Nijmegen, Nijmegen, The Netherlands. NWO-SION project *Typed lambda calculus*, 612-316-030.
October 1993 - April 1994	Visited the Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland.
From June 1994	Research Fellow at the Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland. EPSRC GRANT, GR/G 55792. Constructive logic as a basis for program development.

Samenvatting

Subtypering is een primitieve relatie waarmee op een uniforme wijze begrippen uit diverse gebieden van de informatica kunnen worden beschreven. In het geval dat S en T verzamelingen zijn, betekent $S \leq T$ (S *is een subtype van* T): elementen van S zijn ook elementen van T. Als S en T specificaties zijn, dan voldoen elementen die aan de specificatie S voldoen, ook aan T. Als S en T objectbeschrijvingen zijn in objectgeoriënteerd programmeren, dan betekent $S \leq T$ dat het op plaatsen waar een object met interface T wordt verwacht, ook een object met interface S gebruikt mag worden. Wanneer S en T module interfaces zijn in een software systeem, dan is een implementatie van S ook een implementatie van T. Als S en T stellingen zijn, dan is een bewijs van S ook een bewijs van T. Het begrijpen van de essentie, de subtiliteiten, en de algemene eigenschapen van subtypering, werpt licht op een omvangrijk gebied.

Dit proefschift bevat twee delen. Het eerste deel geeft een gedetailleerde analyse van de meta-theorie van een getypeerde lambda calculus waarin intersectietypes en hogereorde begrensde quantificatie worden gecombineerd. Ons onderzoek betreft syntactische, semantische en pragmatische aspecten.

- In hoofdstuk 2 definiëren we het systeem $F^\omega_\wedge$, en bewijzen we een aantal elementaire eigenschappen.
  - We definëren de getypeerde lambda calculus $F^\omega_\wedge$, een natuurlijke generalisatie van Girard's systeem F^ω met intersectietypes en begrensd polymorfisme. Een nieuw aspect van onze presentatie is het gebruik van termherschrijftechnieken om intersectietypes te definiëren, waardoor de computationele semantiek (reductieregels) duidelijk van de syntax (inferentieregels) van het systeem wordt gescheiden.
  - De reductieregels van $F^\omega_\wedge$ kunnen gesplitst worden in twee hoofdgroepen: reductie van types ($\rightarrow_{\beta\wedge}$) en reductie van termen ($\rightarrow_{\beta fors}$). Hoewel confluentie in het algemeen niet een modulaire eigenschap is, is het in ons geval wel mogelijk om een modulair bewijs te geven. In sectie 2.3, combineren we de onafhankelijke bewijzen van confluentie voor reductie van types en confluentie voor reductie van termen, tot een bewijs van confluentie van de reductierelatie van het gehele systeem.
  - We bewijzen de sterke normalisatie eigenschap van $\rightarrow_{\beta\wedge}$ op goedgevormde types.
- Hoofdstuk 3 bevat het meest belangrijke resultaat van dit proefschrift. Onze voornaamste bijdrage is het bewijs van het feit dat subtypering in $F^\omega_\wedge$ beslisbaar is. Dit resultaat heeft een oplossing tot gevolg voor het tot nu toe open probleem van de beslisbaarheid van subtypering in $F^\omega_\leq$, het intersectie-vrije deel van

$F^{\omega}_{\wedge}$, omdat het subtyperinssysteem van $F^{\omega}_{\wedge}$ een conservatieve uitbreiding van de subtyperingsrelatie van $F^{\omega}_{\leq}$ is. Verder is de beslisbaarheid van subtypering essentieel voor de beslisbaarheid van type checking en type inferentie. Een andere oorspronkelijke bijdrage is het gebruik van een keuzeoperator om het gedrag van variabelen tijdens subtype checking te modelleren. Het bewijs van de beslisbaarheid wordt opgesplitst in de volgende stappen.

  - We definiëren een algoritmische presentatie van de subtyperingsrelatie, waarbij we alleen types in normaalvorm beschouwen.
  - We bewijzen dat deze algoritmische presentatie sound en complete is met betrekking tot de definitie van subtypering, dat wil zeggen dat hij een deterministische procedure bepaalt voor het checken van subtypering in $F^{\omega}_{\wedge}$.
  - Tenslotte bewijzen we dat de door de algoritmische presentatie beschreven procedure termineert. Het bewijs van terminatie wordt herleid tot de sterke normalisatie-eigenschap van reductie op types, uitgebreid met een keuzereductie die het gedrag van variabelen tijdens het checken van subtypering modelleert.

- In hoofdstuk 4 bewijzen we dat $F^{\omega}_{\wedge}$ de minimale type-eigenschap heeft, en we beschrijven een algoritme voor het berekenen van de minimale types. Bovendien bewijzen we dat type inferentie en type checking in $F^{\omega}_{\wedge}$ beslisbaar zijn. De minimale type-eigenschap wordt gebruikt om te bewijzen dat $F^{\omega}_{\wedge}$ de subject reductie-eigenschap heeft.
- In hoofdstuk 5 definiëren we een model gebaseerd op partiële equivalentierelaties, en we bewijzen dat de subtyperingsrelatie sound is met betrekking tot dit model.
- Hoewel $F^{\omega}_{\wedge}$ gedefinieerd was om een model te creëren voor object georiënteerd programmeren met multiple inheritance, is het niet de bedoeling van dit proefschrift om de grondslagen van object georiënteerd programmeren te behandelen. In hoofdstuk 6, laten we zien hoe multiple inheritance met behulp van subtypering gemodelleerd kan worden. Dit is een voortzetting van het onderzoek naar de type-theoretische grondslagen van object georiënteerd programmeren door Pierce en Turner [PT94], die multiple inheritance buiten beschouwing laten.

In het tweede deel van dit proefschrift worden twee verschillende stijlen van subtypering bestudeerd: subtypering met impliciete coërcies en subtypering met expliciete coërcies. We definiëren en bestuderen twee alternatieve presentaties van subtypering voor de simpel getypeerde lambda calculus. De eerste, $\lambda_{\leqslant}$, is een systeem met impliciete coërcies, en de tweede, λ_C, is een systeem met expliciete coërcies. We laten zien dat het systeem $\lambda_{\leqslant}$ vertaald kan worden in λ_C, en dat λ_C vertaald kan worden in $\lambda{\rightarrow}$. Vanuit een pragmatische invalshoek betekent dat, dat impliciete of expliciete coërcies slechts een kwestie van smaak zijn, en dat beide benaderingen vertaald kunnen worden in de simpel getypeerde lambda calculus zonder subtypering.

Translated by Jan Kuper