Principality and Decidable Type Inference for Finite-Rank Intersection Types

A. J. Kfoury*†
Boston University
Boston, MA 02215, U.S.A.
kfoury@cs.bu.edu
http://www.cs.bu.edu/~kfoury

J. B. Wells†‡
Heriot-Watt University
EDINBURGH, EH14 4AS, Scotland
jbw@cee.hw.ac.uk
http://www.cee.hw.ac.uk/~jbw

Abstract

Principality of typings is the property that for each typable term, there is a typing from which all other typings are obtained via some set of operations. Type inference is the problem of finding a typing for a given term, if possible. We define an intersection type system which has principal typings and types exactly the strongly normalizable λ-terms. More interestingly, every finite-rank restriction of this system (using Leivant's first notion of rank) has principal typings and also has decidable type inference. This is in contrast to System F where the finite rank restriction for every finite rank at 3 and above has neither principal typings nor decidable type inference. This is also in contrast to earlier presentations of intersection types where the status (decidable or undecidable) of these properties is unknown for the finite-rank restrictions at 3 and above. Furthermore, the notion of principal typings for our system involves only one operation, substitution, rather than several operations (not all substitution-based) as in earlier presentations of principality for intersection types (without rank restrictions). In our system the earlier notion of *expansion* is integrated in the form of *expansion variables*, which are subject to substitution as are ordinary variables. A unification-based type inference algorithm is presented using a new form of unification, β-unification.

1 Introduction

1.1 Background and Motivation

The Desire for Polymorphic Type Inference Programming language designers now generally recognize the benefits (as well as the costs!) of strong static typing. Languages such as Haskell [PJHH+93], Java [GJS96], and ML [MTHM90] were all designed with strong typing in mind. To avoid imposing an undue burden on the programmer, the compiler is expected to infer as much type information as possible. To avoid rejecting perfectly safe programs, the type inference algorithm should support as much *type polymorphism* as possible. The main options for polymorphism are universal types, written $\forall\alpha.\tau$, and intersection types, written $\sigma \wedge \tau$. (Their duals are the existential types, written $\exists\alpha.\tau$, and union types, written $\sigma \vee \tau$.)

The most popular type inference algorithm is algorithm $\mathcal{W}$ by Damas and Milner [DM82] for the type system commonly called Hindley/Milner which supports polymorphism with a restricted form of universal types. In practice this type system is somewhat inflexible, sometimes forcing the programmer into contortions to convince the compiler that their code is well typed. This has motivated a long search for more expressive type systems with decidable typability. In this search, there have been a great number of negative results, e.g., undecidability of System F [Wel94], finite rank restrictions of F above 3 [KW94], $F_{\leq}$ [Pie94], F_{ω} [Urz97], $F+\eta$ [Wel96], and unrestricted intersection types [Pot80]. Along the way, there have been a few positive results, some extensions of the Damas/Milner approach, but, perhaps more interestingly, some with intersection types.

What are Principal Typings? The various systems of intersection types have generally had a *principal typings* property, which differs from the *principal types* property of the Hindley/Milner system, as described by Jim [Jim96]:

Principal Types
Given: a term M typable in type environment A.
There exists: a type σ representing all possible types for M in A.

Principal Typings
Given: a typable term M.
There exists: a judgement $A \vdash M : \tau$ representing all possible typings for M.

A typing for a program is principal if all other typings for the same program can be derived from it by some set of operations. As explained by Jim, this kind of approach supports the possibility of true separate compilation as well as other benefits.

*Partly supported by NATO grant CRG 971607.
†Partly supported by NSF grant CCR–9417382.
‡Partly supported by EPSRC grant GR/L 36963.

Principal Typings with Intersection Types The first system of intersection types for which principal typings was proved (as far as we are aware) was presented by Coppo, Dezani, and Venneri [CDCV80] (a later version is [CDCV81]). Like many systems of intersection types, it is similar to ours in that "$\wedge$" can not appear to the right of "$\rightarrow$" and $\wedge$-elimination can only occur at λ-term variables. Like our system, this system is restricted so that the binding type of the bound variable of an abstraction must be an intersection of exactly the set of types at which it is used. However, this system differs from ours by allowing different occurrences of the bound variable to use the same member of an intersection. It also has a rule to assign the special type ω (representing the intersection of 0 types) to any term.

There is a general approach for an algorithm for finding principal typings that was followed by Coppo, Dezani, and Venneri for their type system as well as by Ronchi della Rocca and Venneri [RDRV84] and van Bakel [vB93] for other systems of intersection types. In this approach, the principal typing algorithm first finds a normal form (or *approximate* normal form) and then creates a typing for the normal form. A separate proof shows that any typing for the normal form is also a typing for the original term. The algorithms of this approach are intrinsically impractical, not only due to the expense of normalization but, more importantly, because there is no possibility of a short cut to normalization. The principality of the principal typing is shown using a technique of several different kinds of operations: *expansion* (sometimes called *duplication*), *lifting* (sometimes called *rise*), and *substitution*. The biggest difference with the approach we present in this paper is that we use *expansion variables* to formalize expansion in a much simpler way as part of substitution. This allows our approach to be based on both substitution and unification. This opens the possibility of more efficient algorithms by adding additional (unnecessary) constraints to the unification problem to shortcut the solution, an adaptation we leave to future work.

Sayag and Mauny [SM97, SM96] continue the earlier work cited above, and succeed in defining a simpler notion of principal typings for a system of intersection types. An important difference with our analysis is the continued use of an expansion operation, although considerably simplified from earlier formulations, in part because they restrict attention to λ-terms in normal form. Moreover, their approach is not substitution-based and it is not immediately clear how to extend it to arbitrary λ-terms not in normal form.

The first unification-based approach to principal typing with intersection types is by Ronchi della Rocca [RDR88]. Of course, the general method here will diverge for some terms in the full type system, but a decidable restriction is presented which bounds the height of types. Unfortunately, this approach uses the old, complicated approach to expansion and is very difficult to understand. It also has trouble with commutativity and associativity of "$\wedge$".

Subsequent unification-based approaches to principal typing with intersection types have focused on the *rank-2* restriction of intersection types, using Leivant's notion of rank [Lei83]. Van Bakel presents a unification algorithm for principal typing for a rank-2 system [vB93]. Later independent work by Jim also attacks the same problem, but with more emphasis on handling practical programming language issues such as recursive definitions, separate compilation, and accurate error messages [Jim96]. Successors to Jim's method include Banerjee's [Ban97], which integrates flow analysis, and Jensen's [Jen98], which integrates strictness analysis. Other approaches to principal typings and type inference with intersection types include [CG92] and [JMZ92].

1.2 Contributions of This Paper

The main contributions of this paper are the following:

- A fully substitution-based notion of principality for a system of intersection types (with or without a rank restriction on types). Expansion variables abstractly represent the possibility of multiple subderivations for the same term, supporting a substitution-based approach in place of the old notion of expansion.

  This contribution makes the technology of intersection types significantly more accessible to non-theorists. The notions of expansion in earlier literature are so complicated that few but the authors could understand them.

- A unification-based type inference algorithm for intersection types using a novel form of unification, β-unification. The algorithm always returns principal typings when it halts. The algorithm is terminating when restricted to finite-rank types.

  This algorithm is the first understandable type inference algorithm for intersection types beyond the rank-2 restriction which does not require that terms first be β-reduced to normal form. Although it may seem that there is quite a bit of material in this report, the vast majority of it exists only to prove properties of the algorithm. The actual algorithm is largely contained in definitions 2.11, 2.12, 2.13, 2.15, 3.1, 4.1, 4.2, 4.4, and 5.1, together with theorem 5.6.

- Decidability of type inference and principal typings for the restrictions to every finite rank.

  Ours is the first system of intersection types for which this has been shown. At rank 3, our system already types terms not typable in the very powerful system F_ω, e.g., the term $(\lambda x.z(x(\lambda f u.fu))(x(\lambda v g.gv)))(\lambda y.yyy)$, which was shown untypable in F_ω by Urzyczyn [Urz97].

1.3 Future Work

Using Intersection Types in Practice This work is carried out in the context of the Church Project,[1] a focused effort to explore the implications of intersection types for programming language design and implementation. The Church Project is actively implementing and evaluating intersection-type-based technology in an ML compiler. A number of practical concerns need to be addressed to finish the task of making the technology presented in this report usable in the overall project effort. In particular, the following tasks are important:

[1] ⟨URL:http://www.cs.bu.edu/groups/church/⟩.

1. Adapt the technology to type systems in which "$\wedge$" is associative, commutative, and idempotent. This will be vital for reducing the space and time complexity of our algorithm, because it will enable the expression of the rank restrictions without requiring an essentially linear flow analysis.

2. Add support for sum types, e.g., booleans and conditionals. This is highly likely to require the addition of union types.

3. Add support for recursive definitions, e.g., a fixpoint operator or letrec bindings. This will significantly complicate the analysis, because it will interfere with the invariant that λ-compatible constraint sets (definition 3.7) have constraints neatly divided into positive and negative types (definition 3.3). Also, polymorphic/polyvariant analysis of recursion is notoriously difficult.

4. Take advantage of the new notion of substitution developed in this report to devise efficient representations of polyvariant program analyses. This is particularly promising.

Theoretical Concerns The work presented here inspires the following possible tasks:

- Investigate the relationship between β-unification and other forms of unification – decidable and undecidable. In particular, investigate the relationship with second-order unification and semi-unification.

- Further develop the meta-theory of β-unification. In particular, investigate conditions under which β-unification (1) satisfies a principality property and (2) is decidable. Use this to develop more sophisticated type inference algorithms.

- Investigate the complexity of the decidable finite-rank restriction of β-unification introduced in section 6. Separately, investigate the complexity of the set of programs typable in the various finite-rank restrictions.

2 Intersection Types with Expansion Variables

This section defines a system of intersection types for the λ-calculus with the additional feature of expansion variables. The expansion variables do not affect what is typable; instead they support reasoning about principal typings via a notion of substitution.

Throughout the paper, the notation $\vec{X}^n$ is meta-notation standing for the notation $X_1, \ldots, X_n$. The notation $\vec{X}$ stands for $\vec{X}^n$ for some $n \geq 0$ which either does not matter or is clear from the context. A warning: Some authors use the notation $\bar{X}$ for similar purposes, but in this paper the bar mark on a symbol is *not* used to stand for a sequence.

2.1 The Type System

Definition 2.1 (λ-Terms). Let x and y range over λ-Var, the set of λ-term variables. We use the usual set of λ-terms

$$M, N \in \Lambda ::= x \mid \lambda x.M \mid MN\,,$$

quotiented by α-conversion as usual, and the usual notion of reduction

$$(\lambda x.M)N \to_\beta M[x := N]\,.$$

As usual, $\mathrm{FV}(M)$ denotes the set of free variables of M. □

The following definition gives a structure to type variable names that will be helpful later when we need to rename them distinctly.

Definition 2.2 (Type Variables). The set of *basic type variables* or *basic* T-*variables* is $\mathsf{TVar}_b = \{\, a_i \mid i \in \mathcal{N} \,\}$. The set of *basic expansion variables* or *basic* E-*variables* is $\mathsf{EVar}_b = \{\, \mathsf{F}_i \mid i \in \mathcal{N} \,\}$. We assume TVar_b and EVar_b are disjoint sets, and use Var_b to denote the union $\mathsf{EVar}_b \cup \mathsf{TVar}_b$.

We use binary strings in $\{0,1\}^*$, called *offset labels*, to name (and later to rename) variables. If $s, t \in \{0,1\}^*$, we write $s \cdot t$ for their concatenation. The statement $s \leq t$ holds iff $t = s \cdot s'$ for some $s' \in \{0,1\}^*$. The set of *type variables* or T-*variables* and and the set of *expansion variables* or E-*variables* are:

$$\mathsf{TVar} = \{\, a_i^s \mid i \in \mathcal{N},\ s \in \{0,1\}^* \,\}$$
$$\mathsf{EVar} = \{\, \mathsf{F}_i^s \mid i \in \mathcal{N},\ s \in \{0,1\}^* \,\}$$

Let TVar and EVar properly extend TVar_b and EVar_b by taking a_i to be a_i^ε and F_i to be F_i^ε, where ε denotes the empty string. Let $(a_i^s)^t$ denote $a_i^{s \cdot t}$ and let $(\mathsf{F}_i^s)^t$ denote $\mathsf{F}_i^{s \cdot t}$. Let α and β be metavariables ranging over TVar and let F (in italics) be a metavariable ranging over EVar. For example, if α denotes a_i^s, then α^t denotes $a_i^{s \cdot t}$. We use v (appropriately decorated) as a metavariable ranging over the disjoint union $\mathsf{Var} = \mathsf{EVar} \cup \mathsf{TVar}$. □

Definition 2.3 (Types). Let "$\to$" and "$\wedge$" be binary type constructors. The set $\mathbb{T}$ of *types* and its subset $\mathbb{T}^\to$ as well as metavariables over these sets are given as follows:

$$\bar{\tau} \in \mathbb{T}^\to ::= \alpha \mid (\tau \to \bar{\tau})$$
$$\tau \in \mathbb{T} \quad ::= \bar{\tau} \mid (\tau \wedge \tau') \mid (F\,\tau)$$

Note that $\bar{\tau}$ is only a metavariable over $\mathbb{T}^\to$. The letters ρ and σ will be used later to range over certain other subsets of $\mathbb{T}$. Observe in the tree representation of any $\tau \in \mathbb{T}$ that no "$\wedge$" and no E-variable can occur as the right child of "$\to$". We sometimes omit parentheses according to the rule that "$\to$" and "$\wedge$" associate to the right and the application of an expansion variable (e.g., $F\,\tau$) has higher precedence than "$\wedge$" which has higher precedence than "$\to$". For example, $F\,\tau_1 \wedge \tau_2 \to \tau_3 = ((F\,\tau_1) \wedge \tau_2) \to \tau_3$. □

Definition 2.4 (Type Environments). A *type environment* is a function A from λ-Var to $\mathbb{T}$ with finite domain of definition. A type environment may be written as a finite list of pairs, as in

$$x_1 : \tau_1,\ \ldots\ , x_k : \tau_k$$

for some distinct $x_1, \ldots, x_k \in \lambda\text{-Var}$, some $\tau_1, \ldots, \tau_k \in \mathbb{T}$ and some $k \geq 0$. If A is a type environment, then $A[x \mapsto \tau]$ is the type environment such that $(A[x \mapsto \tau])(x) = \tau$ and $(A[x \mapsto \tau])(y) = A(y)$ if $y \neq x$ and $A \backslash x = A - \{x \mapsto A(x)\}$. If A and B are type environments, then $A \wedge B$ is a new type environment given by:

$$(A \wedge B)(x) = \begin{cases} A(x) \wedge B(x) & \text{if both } A(x) \text{ and } B(x) \text{ defined,} \\ A(x) & \text{if only } A(x) \text{ defined,} \\ B(x) & \text{if only } B(x) \text{ defined,} \\ \text{undefined} & \text{if both } A(x) \text{ and } B(x) \text{ undefined.} \end{cases}$$

If $F \in \mathsf{EVar}$ is an E-variable and A is a type environment, then $F\,A$ is the type environment such that $(F\,A)(x) = F(A(x))$. □

Definition 2.5 (Skeletons and Derivations). The sets of *judgements*, *rule names*, and *pre-skeletons* are determined by the following grammar:

$$\begin{aligned} J \in \mathsf{Judg} &::= A \vdash M : \tau \mid A \vdash_{\mathrm{e}} M : \tau \\ R \in \mathsf{Rule} &::= \mathrm{VAR} \mid \text{ABS-K} \mid \text{ABS-I} \mid \mathrm{APP} \mid \wedge \mid F \\ Q \in \mathsf{PSkel} &::= \langle R, J, \vec{Q} \rangle \end{aligned}$$

Judgements formed with the $\vdash_{\mathrm{e}}$ symbol will be used to restrict the $\wedge$ and F rules so these rules are used only for subterms which are the arguments of an application. Observe that a pre-skeleton $\mathcal{S}$ is a rule name, a final judgement, and zero or more subskeletons. The order of the subskeletons is significant. Note that F is a rule name for every $F \in \mathsf{EVar}$.

A *skeleton* $\mathcal{S}$ of System $\mathbb{I}$ is a pre-skeleton Q such that, for every sub-pre-skeleton $Q' = \langle R, J, \vec{Q} \rangle$ occurring in Q, it holds that the judgement J is obtained from the end judgements of the pre-skeletons $\vec{Q}$ (whose order is significant) by rule R and rule R is one of the rules for skeletons of System $\mathbb{I}$ in figure 1. The rule named "$(\mathcal{S})$ APP" is used in skeletons for the case of APP. The order of the pre-skeletons $\vec{Q}$ determines the order in which their end judgements must match the premises of the rule R. A skeleton $\langle R, J, Q_1 \ldots Q_n \rangle$ may be written instead as:

$$\frac{Q_1 \quad \ldots \quad Q_n}{J}\, R$$

A *derivation* $\mathcal{D}$ of System $\mathbb{I}$ is a skeleton $\mathcal{S}$ such that every use of the rule named "$(\mathcal{S})$ APP" also qualifies as a use of the more restrictive rule named "$(\mathcal{D})$ APP" in figure 1. The "$(\mathcal{D})$ APP" rule differs from the "$(\mathcal{S})$ APP" rule in requiring the type of the term in function position to be a function type, requiring the domain portion of the function type to match the type of the term in argument position, and requiring the codomain portion of the function type to match the result type. In interpreting the rules in figure 1, the pattern $A \vdash_? M : \tau$ can refer to either $A \vdash M : \tau$ or $A \vdash_{\mathrm{e}} M : \tau$. Henceforth, all skeletons and derivations belong to System $\mathbb{I}$.

Observe that the set Deriv of derivations is a proper subset of the set Skel of skeletons. Henceforth, we do not consider pre-skeletons that are not also skeletons.

Let the statement $A \vdash_{\mathbb{I}} M : \tau$ hold iff there exists a derivation $\mathcal{D}$ of System $\mathbb{I}$ whose final judgement is $A \vdash M : \tau$. When this holds, we say that $\mathcal{D}$ is a *typing* for M. A term M is *typable* in System $\mathbb{I}$ iff $A \vdash_{\mathbb{I}} M : \tau$ holds for some A and τ.

Observe that the rule ABS-K is not a special case of the rule ABS-I. This is because there is no rule or other provision for "weakening" (adding redundant type assumptions to a type environment) in our system and therefore, if there is a proof for the judgement $A \vdash M : \tau$ where $x \notin \mathrm{FV}(M)$, then $A(x)$ is not defined. □

The following result is merely what anyone would expect from a system of intersection types formulated without a rule for ω.

Theorem 2.6 (Strong Normalization). *A λ-term M is strongly normalizable (i.e., there is no infinite β-reduction sequence starting from M) if and only if M is typable in System $\mathbb{I}$.* □

Corollary 2.7 (Undecidability of Typability). *It is undecidable whether an arbitrarily chosen λ-term M is typable in System $\mathbb{I}$.* □

Later in the paper, we will show certain restrictions of System $\mathbb{I}$ to have decidable typability.

REMARK 2.8. System $\mathbb{I}$ does not have the subject reduction property. For example,

$$z : \alpha_2 \to \alpha_1 \to \alpha_3, w : \alpha_1 \wedge \alpha_2 \vdash_{\mathbb{I}} (\lambda x.(\lambda y.zyx)x)w : \alpha_3\,,$$

but

$$z : \alpha_2 \to \alpha_1 \to \alpha_3, w : \alpha_1 \wedge \alpha_2 \nvdash_{\mathbb{I}} (\lambda x.zxx)w : \alpha_3\,.$$

By theorem 2.6, typability is preserved, so for example:

$$z : \alpha_2 \to \alpha_1 \to \alpha_3, w : \alpha_2 \wedge \alpha_1 \vdash_{\mathbb{I}} (\lambda x.zxx)w : \alpha_3\,.$$

The reason for the lack of subject reduction is that (1) "$\wedge$" is neither associative, commutative, nor idempotent, (2) the implicit $\wedge$-elimination done by the VAR rule and the way type environments are built together fix the component of an intersection type associated with a particular variable, and (3) there is no provision for weakening (i.e., introducing unneeded type assumptions). If subject reduction is viewed as a means to achieve other goals rather than as a goal by itself, then the lack of subject reduction is not a problem, because derivations of System $\mathbb{I}$ can be easily translated into derivations of more permissive systems of intersection types (see [vB93] for a survey) for which numerous desirable properties have already been verified. The features of System $\mathbb{I}$ which prevent subject reduction make the later analysis of principal typings and type inference much easier. □

2.2 Substitution

The notion of substitution defined here will be used later in unification for type inference and in establishing a principal typing property for System $\mathbb{I}$.

Definition 2.9 (Type Contexts). The symbol $\Box$ denotes a "hole". The set $\mathbb{T}_\Box$ of *type contexts* and its subset $\mathbb{T}_\Box^{\to}$ as well as metavariables over these sets are given as follows:

$$\begin{aligned} \bar{\varphi} \in \mathbb{T}_\Box^{\to} &::= \Box \mid \alpha \mid (\varphi \to \bar{\varphi}) \\ \varphi \in \mathbb{T}_\Box &::= \bar{\varphi} \mid (\varphi \wedge \varphi') \mid (F\,\varphi) \end{aligned}$$

Inference Rules for both Skeletons and Derivations

VAR $\quad x : \bar{\tau} \vdash x : \bar{\tau}$

ABS-I $\quad \dfrac{A[x \mapsto \tau] \vdash M : \bar{\tau}}{A \vdash (\lambda x.M) : (\tau \to \bar{\tau})}$

ABS-K $\quad \dfrac{A \vdash M : \bar{\tau}}{A \vdash (\lambda x.M) : (\bar{\tau}' \to \bar{\tau})}$ if $x \notin \mathrm{FV}(M)$

$\wedge \quad \dfrac{A_1 \vdash_? M : \tau_1; \qquad A_2 \vdash_? M : \tau_2}{A_1 \wedge A_2 \vdash_e M : \tau_1 \wedge \tau_2}$

$F \quad \dfrac{A \vdash_? M : \tau}{F\,A \vdash_e M : F\,\tau}$

Remember that $\bar{\tau}, \bar{\tau}' \in \mathbb{T}^{\to}$ and $F \in \mathsf{EVar}$.

Inference Rule for Skeletons Only

$(\mathcal{S})$ APP $\quad \dfrac{A_1 \vdash M : \bar{\tau}; \qquad A_2 \vdash_? N : \tau}{A_1 \wedge A_2 \vdash M\,N : \bar{\tau}'}$

Inference Rule for Derivations Only

$(\mathcal{D})$ APP $\quad \dfrac{A_1 \vdash M : \tau \to \bar{\tau}; \qquad A_2 \vdash_? N : \tau}{A_1 \wedge A_2 \vdash M\,N : \bar{\tau}}$

Figure 1: Inference Rules of System $\mathbb{I}$.

Note that $\bar{\varphi}$ is only a metavariable over $\mathbb{T}^{\to}_{\Box}$. If φ has n holes, we write $\#_{\Box}(\varphi) = n$ and use $\Box^{(1)}, \ldots, \Box^{(n)}$ to denote these n holes in their order of occurrence in φ from left to right. Care must be applied when inserting $\tau_1, \ldots, \tau_n \in \mathbb{T}$ in the holes of φ in order to obtain a type $\tau = \varphi[\tau_1, \ldots, \tau_n]$ which is valid according to Definition 2.3. Specifically, if hole $\Box^{(i)}$ in φ is to the immediate right of "$\to$", then τ_i must be restricted to the subset $\mathbb{T}^{\to}$. □

Definition 2.10 (Expansions). The set $\mathbb{E}$ of *expansions* is a proper subset of $\mathbb{T}_{\Box}$, defined by the following grammar:

$$e \in \mathbb{E} ::= \Box \mid (e \wedge e') \mid (Fe)$$

In words, an expansion is a type context which mentions no T-variable and no "$\to$". □

Definition 2.11 (Paths in Type Contexts). We define path as a partial function which determines the position of $\Box^{(i)}$ in φ as a string in $\{\,\mathsf{L}, \mathsf{R}, 0, 1\,\}^*$. The definition goes as follows, using a "value" of $\bot$ to indicate that the function is undefined on that input:

$$\mathrm{path}(\Box^{(i)}, \Box) = \begin{cases} \varepsilon & \text{if } i = 1, \\ \bot & \text{otherwise.} \end{cases}$$

$$\mathrm{path}(\Box^{(i)}, \alpha) = \bot$$

$$\mathrm{path}(\Box^{(i)}, \varphi \to \bar{\varphi}) = \begin{cases} \mathsf{L} \cdot p & \text{if } p = \mathrm{path}(\Box^{(i)}, \varphi) \neq \bot, \\ \mathsf{R} \cdot q & \text{if } q = \mathrm{path}(\Box^{(i-\#_{\Box}(\varphi))}, \bar{\varphi}) \\ & \quad \text{and } q \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

$$\mathrm{path}(\Box^{(i)}, \varphi \wedge \varphi') = \begin{cases} 0 \cdot p & \text{if } p = \mathrm{path}(\Box^{(i)}, \varphi) \neq \bot, \\ 1 \cdot q & \text{if } q = \mathrm{path}(\Box^{(i-\#_{\Box}(\varphi))}, \varphi') \\ & \quad \text{and } q \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

$$\mathrm{path}(\Box^{(i)}, F\varphi) = \mathrm{path}(\Box^{(i)}, \varphi)$$

Let $\mathrm{paths}(\varphi) = \{\,\mathrm{path}(\Box^{(i)}, \varphi) \mid 1 \leq i \leq \#_{\Box}(\varphi)\,\}$. Because an expansion $e \in \mathbb{E}$ is a type context that does not mention "$\to$", a path in e is a string in $\{0,1\}^*$ rather than in $\{\,\mathsf{L}, \mathsf{R}, 0, 1\,\}^*$. We thus use binary strings in $\{0,1\}^*$ for a dual purpose: as paths in expansions and as offset labels to rename variables (see definition 2.2). The coincidence between the two is by design. □

Definition 2.12 (Variable Renaming). For every $t \in \{0,1\}^*$ (we do not need to consider the larger set $\{\,\mathsf{L}, \mathsf{R}, 0, 1\,\}$), we define a variable renaming $\langle\ \rangle^t$ from $\mathbb{T}$ to $\mathbb{T}$, by induction:

1. $\langle \alpha_i^s \rangle^t = \alpha_i^{s \cdot t}$ for every $\alpha_i^s \in \mathsf{TVar}$.
2. $\langle \tau \to \bar{\tau} \rangle^t = \langle \tau \rangle^t \to \langle \bar{\tau} \rangle^t$.
3. $\langle \tau_1 \wedge \tau_2 \rangle^t = \langle \tau_1 \rangle^t \wedge \langle \tau_2 \rangle^t$.
4. $\langle F_i^s\ \tau \rangle^t = F_i^{s \cdot t}\ \langle \tau \rangle^t$ for every $F_i^s \in \mathsf{EVar}$.

In words, $\langle \tau \rangle^t$ is obtained from τ by appending t to every offset that is part of a variable name in τ. □

Definition 2.13 (Substitution on Types). A *substitution* is a total function $\mathbf{S} : \mathsf{Var} \to (\mathbb{E} \cup \mathbb{T}^{\to})$ which respects "sorts", i.e., $\mathbf{S}F \in \mathbb{E}$ for every $F \in \mathsf{EVar}$ and $\mathbf{S}\alpha \in \mathbb{T}^{\to}$ for every $\alpha \in \mathsf{TVar}$. Note that $\mathbf{S}(\alpha)$ can not have an E-variable or "$\wedge$" in outermost position. We write $\mathbf{S}F$ instead of $\mathbf{S}(F)$ and $\mathbf{S}\alpha$ instead of $\mathbf{S}(\alpha)$, as long as no ambiguity is introduced. We lift a substitution $\mathbf{S}$ to a function $\overline{\mathbf{S}}$ from $\mathbb{T}$ to $\mathbb{T}$ by induction on $\mathbb{T}$:

1. $\overline{\mathbf{S}}\alpha = \mathbf{S}\alpha$.
2. $\overline{\mathbf{S}}(\tau \to \bar{\tau}) = (\overline{\mathbf{S}}\tau) \to (\overline{\mathbf{S}}\bar{\tau})$.
3. $\overline{\mathbf{S}}(\tau_1 \wedge \tau_2) = (\overline{\mathbf{S}}\tau_1) \wedge (\overline{\mathbf{S}}\tau_2)$.
4. $\overline{\mathbf{S}}(F\tau) = e[\overline{\mathbf{S}}(\langle \tau \rangle^{s_1}), \ldots, \overline{\mathbf{S}}(\langle \tau \rangle^{s_n})]$, where $\mathbf{S}F = e$ and $s_i = \mathrm{path}(\Box^{(i)}, e)$ for $1 \leq i \leq \#_{\Box}(e) = n$.

When no ambiguity is possible, we write $\mathbf{S}$ for $\overline{\mathbf{S}}$ and $\mathbf{S}\,\tau$ for $\overline{\mathbf{S}}(\tau)$. □

Definition 2.14 (Support of Substitutions). Let $\mathbf{S} : \mathsf{Var} \to (\mathbb{E} \cup \mathbb{T}^{\to})$ be a substitution. The *non-trivial E-domain* and *non-trivial T-domain* of $\mathbf{S}$ are $\mathsf{EDom}(\mathbf{S}) = \{\,F \in \mathsf{EVar} \mid \mathbf{S}F \neq F\Box\,\}$ and $\mathsf{TDom}(\mathbf{S}) = \{\,\alpha \in \mathsf{TVar} \mid \mathbf{S}\alpha \neq \alpha\,\}$, respectively. The *non-trivial domain* of $\mathbf{S}$ (or the *support* of $\mathbf{S}$) is $\mathsf{Dom}(\mathbf{S}) = \mathsf{EDom}(\mathbf{S}) \cup \mathsf{TDom}(\mathbf{S})$. The notation

$$\{\!|F_1 := e_1, \ldots, F_m := e_m, \alpha_1 := \bar{\tau}_1, \ldots, \alpha_n := \bar{\tau}_n|\!\}$$

denotes a substitution $\mathbf{S}$ with the indicated mappings where $\mathsf{EDom}(\mathbf{S}) = \{F_1, \ldots, F_m\}$ and $\mathsf{TDom}(\mathbf{S}) = \{\alpha_1, \ldots, \alpha_n\}$. □

Definition 2.15 (Operations on Judgements and Skeletons). The notion of renaming from definition 2.12 is lifted to type environments, judgements, rule names, and skeletons in the obvious way.

The $\wedge$ operator and the operation of applying an E-variable are lifted to skeletons as follows:

1. $\mathcal{S} \wedge \mathcal{S}' = \langle \wedge, A_1 \wedge A_2 \vdash_{\mathrm{e}} M : \tau_1 \wedge \tau_2, \mathcal{S}, \mathcal{S}' \rangle$ if $\mathcal{S} = \langle R_1, A_1 \vdash_? M : \tau_1, \vec{\mathcal{S}} \rangle$ and $\mathcal{S}' = \langle R_2, A_2 \vdash_? M : \tau_2, \vec{\mathcal{S}'} \rangle$.
2. $F\,\mathcal{S} = \langle F, F\,A \vdash_{\mathrm{e}} M : F\,\tau, \mathcal{S} \rangle$ if $\mathcal{S} = \langle R, A \vdash_? M : \tau, \vec{\mathcal{S}} \rangle$.

Using the preceding, the notation for "expansion filling" is lifted to skeletons as follows:

1. $(e \wedge e')[\mathcal{S}_1, \ldots, \mathcal{S}_n] = \mathcal{S} \wedge \mathcal{S}'$ if $\#_{\Box}(e) = j$, $e[\mathcal{S}_1, \ldots, \mathcal{S}_j] = \mathcal{S}$, and $e'[\mathcal{S}_{j+1}, \ldots, \mathcal{S}_n] = \mathcal{S}'$.
2. $(F\,e)[\mathcal{S}_1, \ldots, \mathcal{S}_n] = F\,(e[\mathcal{S}_1, \ldots, \mathcal{S}_n])$.
3. $\Box[\mathcal{S}] = \mathcal{S}$.

The notion of substitution is lifted to type environments so that $\mathbf{S}\,A$ is the function such that $(\mathbf{S}\,A)(x) = \mathbf{S}(A(x))$. Substitution is lifted to judgements so that $\mathbf{S}(A \vdash_? M : \tau) = \mathbf{S}(A) \vdash_? M : \mathbf{S}(\tau)$. Substitution is lifted to skeletons as follows:

1. $\mathbf{S}\langle R, J, \mathcal{S}_1 \ldots \mathcal{S}_n \rangle = \langle R, \mathbf{S}\,J, (\mathbf{S}\,\mathcal{S}_1) \ldots (\mathbf{S}\,\mathcal{S}_n) \rangle$ if $R \notin \mathsf{EVar}$.
2. $\mathbf{S}\langle F, J, \mathcal{S} \rangle = e[\mathbf{S}(\langle \mathcal{S} \rangle^{s_1}), \ldots, \mathbf{S}(\langle \mathcal{S} \rangle^{s_n})]$, where $\mathbf{S}F = e$ and $s_i = \mathrm{path}(\Box^{(i)}, e)$ for $1 \leq i \leq \#_{\Box}(e) = n$. □

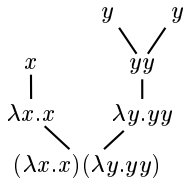
Lemma 2.16 (Substitution Preserves Skeletons and Derivations). *Let* $\mathbf{S}$ *be any substitution.*

1. *If* $\mathcal{S}$ *is a skeleton, then* $\mathbf{S}(\mathcal{S})$ *is a skeleton.*
2. *If* $\mathcal{D}$ *is a derivation, then* $\mathbf{S}\,\mathcal{D}$ *is a derivation.* □

Definition 2.17 (Principal Typings). A derivation $\mathcal{D}$ is a *principal typing* for λ-term M iff $\mathcal{D}$ is a typing for M and for every other typing $\mathcal{D}'$ for M, there is a substitution $\mathbf{S}$ such that $\mathcal{D}' = \mathbf{S}(\mathcal{D})$. The *principality property* for typings is the existence of a principal typing for every typable λ-term. □

Subsequent sections will establish that System $\mathbb{I}$ has the principality property. We next give two simple examples to illustrate how notions introduced so far are used, in particular, the key concepts of "skeleton", "derivation" and "substitution" in the presence of expansion variables.

Example 2.18 (Principal Typing for $(\lambda x.x)(\lambda y.yy)$). Let M_1 denote the λ-term $(\lambda x.x)(\lambda y.yy)$. Depicted in figure 2 is a skeleton $\mathcal{S}_1$ for M_1. The skeleton $\mathcal{S}_1$ is a particular one, produced from M_1 by the Skel algorithm of section 5. It is just a decorated version of the parse tree of M_1 and its size is therefore "small", i.e., proportional to the size of M_1:



Applying an arbitrary substitution to $\mathcal{S}_1$, we can obtain another skeleton for M_1. Thus, $\mathcal{S}_1$ is a scheme for infinitely many skeletons for M_1.

Associated with $\mathcal{S}_1$ is a constraint set

$$\Delta_1 = \{\ \alpha_1 \to \alpha_1 \doteq G(\alpha_2 \wedge F\alpha_3 \to \beta_1) \to \beta_2,\quad G\alpha_2 \doteq G(F\alpha_3 \to \beta_1)\ \}$$

produced from M_1 by the Γ algorithm of section 5. ("Constraint sets" and restrictions on them are defined precisely in section 3.) Note that there is one constraint in Δ_1 for each use of the APP rule in $\mathcal{S}_1$. A particular substitution $\mathbf{S}_1$, obtained from Δ_1 using the Unify algorithm of section 4, is given by:

$$\begin{array}{l}\mathbf{S}_1F = F\Box,\quad \mathbf{S}_1G = \Box,\\ \mathbf{S}_1\alpha_1 = \mathbf{S}_1\beta_2 = ((F\alpha \to \beta) \wedge F\alpha) \to \beta,\\ \mathbf{S}_1\alpha_2 = F\alpha \to \beta,\quad \mathbf{S}_1\alpha_3 = \alpha,\quad \mathbf{S}_1\beta_1 = \beta.\end{array}$$

Applying substitution $\mathbf{S}_1$ to skeleton $\mathcal{S}_1$, we obtain another skeleton which is now a derivation $\mathcal{D}_1 = \mathbf{S}_1(\mathcal{S}_1)$, depicted in figure 2.

A consequence of the analysis in sections 4 and 5 is that $\mathcal{D}_1$ is a principal typing for M_1, i.e., every typing $\mathcal{D}'$ for M_1 is of the form $\mathcal{D}' = \mathbf{S}'(\mathcal{D}_1)$ for some substitution $\mathbf{S}'$. □

Example 2.19 (A Principal Typing for $(\lambda x.\lambda y.xy)(\lambda z.zz)$). Let M_2 denote the λ-term $(\lambda x.\lambda y.xy)(\lambda z.zz)$. Depicted in figure 3 is a skeleton $\mathcal{S}_2$ for M_2.

As in example 2.18, the skeleton $\mathcal{S}_2$ is a particular one, produced from N_2 by the Skel algorithm. Associated with $\mathcal{S}_2$ is the following constraint set, produced from M_2 by the Γ algorithm of section 5:

$$\begin{array}{l}\Delta_2 = \{\ \alpha_1 \doteq F\alpha_2 \to \beta_1,\\ \qquad \alpha_1 \to F\alpha_2 \to \beta_1 \doteq H(\alpha_3 \wedge G\alpha_4 \to \beta_2) \to \beta_3,\\ \qquad H\alpha_3 \doteq H(G\alpha_4 \to \beta_2)\ \}.\end{array}$$

A particular substitution $\mathbf{S}_2$, obtained from Δ_2 using the Unify algorithm of section 4, is given by:

$$\begin{array}{l}\mathbf{S}_2F = \Box \wedge G\Box,\quad \mathbf{S}_2G = G\Box,\quad \mathbf{S}_2H = \Box,\\ \mathbf{S}_2\alpha_1 = \mathbf{S}_2\beta_3 = ((G\alpha \to \beta) \wedge G\alpha) \to \beta,\\ \mathbf{S}_2\alpha_2^0 = \mathbf{S}_2\alpha_3 = G\alpha \to \beta,\quad \mathbf{S}_2\alpha_2^1 = \mathbf{S}_2\alpha_4 = \alpha,\\ \mathbf{S}_2\beta_1 = \mathbf{S}_2\beta_2 = \beta.\end{array}$$

Observe that, in this example, $\mathbf{S}_2$ assigns values to the offsprings α_2^0 and α_2^1 of α_2, but does not need to assign any particular value to α_2 itself. This follows from the way substitutions are applied "outside-in", and becomes clear when we consider the action of $\mathbf{S}_2$ on $F\alpha_2$:

$$\begin{array}{l}\mathbf{S}_2(F\alpha_2) = (\Box \wedge G\Box)[\mathbf{S}\langle\alpha_2\rangle^0,\ \mathbf{S}\langle\alpha_2\rangle^1] =\\ (\Box \wedge G\Box)[\mathbf{S}\alpha_2^0,\ \mathbf{S}\alpha_2^1] = \mathbf{S}\alpha_2^0 \wedge G(\mathbf{S}\alpha_2^1) =\\ (G\alpha \to \beta) \wedge G\alpha\ .\end{array}$$

Applying substitution $\mathbf{S}_2$ to skeleton $\mathcal{S}_2$, we obtain a new skeleton which is also a derivation $\mathcal{D}_2 = \mathbf{S}_2(\mathcal{S}_2)$, as depicted in figure 3.

A consequence of the analysis in sections 4 and 5 is that $\mathcal{D}_2$ is a principal typing for M_2, i.e., every typing $\mathcal{D}'$ for M_2 is of the form $\mathcal{D}' = \mathbf{S}'(\mathcal{D}_2)$ for some substitution $\mathbf{S}'$. □

The skeleton $\mathcal{S}_1 = \mathrm{Skel}(M_1)$ is:

$$\dfrac{\dfrac{}{x:\alpha_1 \vdash x:\alpha_1}\,\mathrm{VAR}}{\vdash \lambda x.x:\alpha_1\to\alpha_1}\,\text{ABS-I} \qquad \dfrac{\dfrac{\dfrac{}{y:\alpha_2 \vdash y:\alpha_2}\,\mathrm{VAR} \quad \dfrac{\dfrac{}{y:\alpha_3 \vdash y:\alpha_3}\,\mathrm{VAR}}{y:F\alpha_3 \vdash_{\mathrm{e}} y:F\alpha_3}\,F}{y:\alpha_2\wedge F\alpha_3 \vdash yy:\beta_1}\,\text{APP}}{\vdash \lambda y.yy:\alpha_2\wedge F\alpha_3\to\beta_1}\,\text{ABS-I}}{\vdash_{\mathrm{e}} \lambda y.yy:G(\alpha_2\wedge F\alpha_3\to\beta_1)}\,G$$
$$\dfrac{\qquad\qquad\qquad\qquad\qquad\qquad}{\vdash (\lambda x.x)(\lambda y.yy):\beta_2}\,\text{APP}$$

Letting $\Delta_1 = \Gamma(M_1)$ and $\mathbf{S}_1 = \mathrm{Unify}(\Delta_1)$, the derivation $\mathcal{D}_1 = \mathbf{S}_1(\mathcal{S}_1)$ is:

$$\dfrac{\dfrac{\dfrac{}{x:\tau \vdash x:\tau}\,\mathrm{VAR}}{\vdash \lambda x.x:\tau\to\tau}\,\text{ABS-I} \qquad \dfrac{\dfrac{\dfrac{}{y:F\alpha\to\beta \vdash y:F\alpha\to\beta}\,\mathrm{VAR} \quad \dfrac{\dfrac{}{y:\alpha \vdash y:\alpha}\,\mathrm{VAR}}{y:F\alpha \vdash_{\mathrm{e}} y:F\alpha}\,F}{y:(F\alpha\to\beta)\wedge F\alpha \vdash yy:\beta}\,\text{APP}}{\vdash \lambda y.yy:\tau}\,\text{ABS-I}}{\vdash (\lambda x.x)(\lambda y.yy):\tau}\,\text{APP}$$

where $\tau = ((F\alpha\to\beta)\wedge F\alpha)\to\beta$.

Figure 2: Skeleton $\mathcal{S}_1$ and derivation $\mathcal{D}_1$ for $M_1 = (\lambda x.x)(\lambda y.yy)$.

The skeleton $\mathcal{S}_2 = \mathrm{Skel}(M_2)$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\alpha_1 \vdash x:\alpha_1}\,\mathrm{VAR} \quad \dfrac{\dfrac{}{y:\alpha_2 \vdash y:\alpha_2}\,\mathrm{VAR}}{y:F\alpha_2 \vdash_{\mathrm{e}} y:F\alpha_2}\,F}{x:\alpha_1, y:F\alpha_2 \vdash xy:\beta_1}\,\text{APP}}{x:\alpha_1 \vdash \lambda y.xy:F\alpha_2\to\beta_1}\,\text{ABS-I}}{\vdash \lambda x.\lambda y.xy:\alpha_1\to F\alpha_2\to\beta_1}\,\text{ABS-I} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{z:\alpha_3 \vdash z:\alpha_3}\,\mathrm{VAR} \quad \dfrac{\dfrac{}{z:\alpha_4 \vdash z:\alpha_4}\,\mathrm{VAR}}{z:G\alpha_4 \vdash_{\mathrm{e}} z:G\alpha_4}\,G}{z:\alpha_3\wedge G\alpha_4 \vdash zz:\beta_2}\,\text{APP}}{\vdash \lambda z.zz:\alpha_3\wedge G\alpha_4\to\beta_2}\,\text{ABS-I}}{\vdash_{\mathrm{e}} \lambda z.zz:H(\alpha_3\wedge G\alpha_4\to\beta_2)}\,H$$
$$\dfrac{\qquad\qquad\qquad\qquad\qquad\qquad}{\vdash (\lambda x.\lambda y.xy)(\lambda z.zz):\beta_3}\,\text{APP}$$

Letting $\Delta_2 = \Gamma(M_2)$ and $\mathbf{S}_2 = \mathrm{Unify}(\Delta_2)$, the derivation $\mathcal{D}_2 = \mathbf{S}_2(\mathcal{S}_2)$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\tau_3 \vdash x:\tau_3}\,\mathrm{VAR} \quad \dfrac{\dfrac{}{y:\tau_1 \vdash y:\tau_1}\,\mathrm{VAR} \quad \dfrac{\dfrac{}{y:\alpha \vdash y:\alpha}\,\mathrm{VAR}}{y:G\alpha \vdash_{\mathrm{e}} y:G\alpha}\,G}{y:\tau_2 \vdash_{\mathrm{e}} y:\tau_2}\,\wedge}{x:\tau_3, y:\tau_2 \vdash xy:\beta}\,\text{APP}}{x:\tau_3 \vdash \lambda y.xy:\tau_3}\,\text{ABS-I}}{\vdash \lambda x.\lambda y.xy:\tau_3\to\tau_3}\,\text{ABS-I} \qquad \dfrac{\dfrac{\dfrac{}{z:\tau_1 \vdash z:\tau_1}\,\mathrm{VAR} \quad \dfrac{\dfrac{}{z:\alpha \vdash z:\alpha}\,\mathrm{VAR}}{z:G\alpha \vdash_{\mathrm{e}} z:G\alpha}\,G}{z:\tau_2 \vdash zz:\beta}\,\text{APP}}{\vdash \lambda z.zz:\tau_3}\,\text{ABS-I}$$
$$\dfrac{\qquad\qquad\qquad\qquad\qquad\qquad}{\vdash (\lambda x.\lambda y.xy)(\lambda z.zz):\tau_3}\,\text{APP}$$

where τ_1, τ_2 and τ_3 abbreviate the following types:

$$\tau_1 = G\alpha\to\beta, \quad \tau_2 = \tau_1\wedge G\alpha = (G\alpha\to\beta)\wedge G\alpha, \quad \tau_3 = \tau_2\to\beta = (G\alpha\to\beta)\wedge G\alpha\to\beta.$$

Figure 3: Skeleton $\mathcal{S}_2$ and derivation $\mathcal{D}_2$ for $M_2 = (\lambda x.\lambda y.xy)(\lambda z.zz)$.

3 Lambda-Compatible Beta-Unification

The problem of β-unification was introduced and shown undecidable by Kfoury in [Kfo9X]. This section introduces λ-compatible β-unification, a restriction of β-unification, in order to develop a principality property and in preparation for a unification algorithm presented later.

Definition 3.1 (E-paths). The set EVar^* of all finite sequences of E-variables is also called the *set of E-paths*. We define a function E-path from $\mathsf{Var} \times \mathbb{T}$ to finite subsets of EVar^*. By induction:

1. $\text{E-path}(v, \alpha) = \begin{cases} \{\varepsilon\} & \text{if } v = \alpha, \\ \varnothing & \text{if } v \neq \alpha. \end{cases}$
2. $\text{E-path}(v, \tau \to \tau') = \text{E-path}(v, \tau) \cup \text{E-path}(v, \tau')$.
3. $\text{E-path}(v, \tau \wedge \tau') = \text{E-path}(v, \tau) \cup \text{E-path}(v, \tau')$.
4. $\text{E-path}(v, F\tau) = \begin{cases} \{F\vec{G} \mid \vec{G} \in \text{E-path}(v, \tau)\} & \text{if } v \neq F, \\ \{\varepsilon\} \cup \{F\vec{G} \mid \vec{G} \in \text{E-path}(v, \tau)\} & \text{if } v = F. \end{cases}$ □

Definition 3.2 (Well Named Types). If $\tau \in \mathbb{T}$, we write $\mathsf{EVar}(\tau)$ for the set of E-variables occurring in τ, $\mathsf{TVar}(\tau)$ for the set of T-variables occurring in τ, and $\mathsf{Var}(\tau)$ for the disjoint union $\mathsf{EVar}(\tau) \cup \mathsf{TVar}(\tau)$. We say that a type $\tau \in \mathbb{T}$ is *well named* iff both of the following statements hold:

1. For every $v \in \mathsf{Var}(\tau)$, it holds that $\text{E-path}(v, \tau) = \{\vec{G}\}$ (a singleton set) where v does not occur in $\vec{G}$.
2. For all $v^s, v^t \in \mathsf{Var}(\tau)$ with v basic and $s, t \in \{0,1\}^*$, if $s \leq t$ then $s = t$.

Informally, the first condition says that, for every (type or expansion) variable v, the sequence of E-variables encountered as we go from the root of τ (viewed as a tree) to any occurrence of v is always the same. Furthermore, E-variables do not nest themselves. If $\text{E-path}(v, \tau)$ is the singleton set $\{\vec{F}\}$, we can write $\text{E-path}(v, \tau) = \vec{F}$ without ambiguity.

The second condition says that if a variable v occurs in τ, then no proper offspring of v occurs in τ, where a variable $v^{s \cdot t}$ is called an *offspring* of v^s. Note that types that mention only basic variables automatically satisfy the second condition. □

Definition 3.3 (Positive and Negative Types). We identify two proper subsets $\mathbb{R}$ and $\mathbb{S}$ of $\mathbb{T}$, which we call the "positive types" and the "negative types", respectively. We first define $\mathbb{R}$ and $\mathbb{S}$ with polarities inserted, as $\pm\mathbb{R}$ and $\pm\mathbb{S}$, defined simultaneously with $\pm\mathbb{R}^{\to}$ and $\pm\mathbb{S}^{\to}$, together with metavariables over these sets, as follows:

$$\begin{aligned} \bar{\rho} \in \pm\mathbb{R}^{\to} &::= +\alpha \mid (\sigma \to \bar{\rho}) \\ \rho \in \pm\mathbb{R} &::= \bar{\rho} \mid (+F\bar{\rho}) \\ \bar{\sigma} \in \pm\mathbb{S}^{\to} &::= -\alpha \mid (+F\bar{\rho} \to \bar{\sigma}) \\ \sigma \in \pm\mathbb{S} &::= \bar{\sigma} \mid (\sigma \wedge \sigma') \mid (-F\,\sigma) \end{aligned}$$

We obtain $\mathbb{R}^{\to}$ and $\mathbb{R}$ from $\pm\mathbb{R}^{\to}$ and $\pm\mathbb{R}$, respectively, by omitting all polarities. Similarly we obtain $\mathbb{S}^{\to}$ and $\mathbb{S}$ from $\pm\mathbb{S}^{\to}$ and $\pm\mathbb{S}$.

Note that there is a restriction that exactly one E-variable occurs in each positive position to the left of "$\to$", and "$\wedge$" occurs only in negative positions. Note also that the metavariables $\bar{\rho}$ and $\bar{\sigma}$ are restricted to the subsets $\mathbb{R}^{\to}$ and $\mathbb{S}^{\to}$, respectively.

If $\rho \in \mathbb{R}$ (resp. $\sigma \in \mathbb{S}$), there is exactly one way of inserting polarities in ρ (resp. σ) so that the resulting type ρ' (resp. σ') with polarities is in $\pm\mathbb{R}$ (resp. $\pm\mathbb{S}$). Let $(\rho)^+ \in \pm\mathbb{R}$ (resp. $(\sigma)^- \in \pm\mathbb{S}$) be the uniquely defined expression obtained by inserting polarities in $\rho \in \mathbb{R}$ (resp. $\sigma \in \mathbb{S}$). We thus have two well-defined functions: $(\ )^+$ from $\mathbb{R}$ to $\pm\mathbb{R}$ and $(\ )^-$ from $\mathbb{S}$ to $\pm\mathbb{S}$. □

Definition 3.4 (Constraint Sets). A *constraint* is an equation of the form $\tau \doteq \tau'$ where $\tau, \tau' \in \mathbb{T}$. An *instance* Δ of β-unification is a finite set of constraints, i.e.,

$$\Delta = \{\tau_1 \doteq \tau_1',\ \tau_2 \doteq \tau_2',\ \ldots\ ,\ \tau_n \doteq \tau_n'\}$$

Given Δ of the above form, we write $\mathsf{EVar}(\Delta)$ for the set $\mathsf{EVar}(\tau_1 \wedge \cdots \wedge \tau_n')$, $\mathsf{TVar}(\Delta)$ for the set $\mathsf{TVar}(\tau_1 \wedge \cdots \wedge \tau_n')$ and $\mathsf{Var}(\Delta)$ for their disjoint union $\mathsf{EVar}(\Delta) \cup \mathsf{TVar}(\Delta)$. If Δ is a set of constraints and $F \in \mathsf{EVar}$, we write $F\Delta$ to denote the set of constraints:

$$F\Delta = \{\ F\tau \doteq F\tau' \mid \tau \doteq \tau' \text{ is a constraint in } \Delta\ \}$$ □

Definition 3.5 (Outer and Inner E-Variable Occurrences). Let $\vec{F} \in \mathsf{EVar}^*$, $\rho \in \mathbb{R}$ and $\sigma \in \mathbb{S}$, with $\mathsf{Var}(\rho) \cap \mathsf{Var}(\sigma) = \varnothing$. (All constraints generated will be of this form.) We say the expansion variable G has an *outer* (resp. *inner*) occurrence in the constraint $\vec{F}\rho \doteq \vec{F}\sigma$ iff $G \in \vec{F}$ (resp. iff $G \in \mathsf{EVar}(\rho \wedge \sigma)$). In words, an "outer" occurrence appears on both sides of the constraint and at the top level. Occurrences of T-variables are always inner; only occurrences of E-variables are differentiated between outer and inner.

Let Δ be a finite set of constraints, each of the form specified in the two preceding paragraphs. We say $G \in \mathsf{EVar}$ has an *outer* (resp. *inner*) occurrence in Δ if G has an outer (resp. inner) occurrence in a constraint in Δ.

The definition of "outer" and "inner" occurrences of E-variables carries over, in the obvious way, to the equation $\vec{F}(\rho)^+ \doteq \vec{F}(\sigma)^-$ after polarities are inserted. Only inner occurrences are said to be positive or negative. □

Definition 3.6 (Connected Constraint Sets). We say that a constraint set Δ is *connected* iff for all constraints $\tau \doteq \tau'$ and $\tilde{\tau} \doteq \tilde{\tau}'$ in Δ, there are constraints $\{\tau_1 \doteq \tau_1', \ldots, \tau_n \doteq \tau_n'\} \subseteq \Delta$ such that

$$\mathsf{Var}(\tau_i \doteq \tau_i') \ \cap\ \mathsf{Var}(\tau_{i+1} \doteq \tau_{i+1}') \ \neq\ \varnothing$$

for $1 \leq i \leq n-1$ where "$\tau \doteq \tau'$" is "$\tau_1 \doteq \tau_1'$" and "$\tilde{\tau} \doteq \tilde{\tau}'$" is "$\tau_n \doteq \tau_n'$". □

Definition 3.7 (λ-Compatibility). A constraint set Δ is *λ-compatible* iff Δ is of the form:

$$\Delta = \{\vec{F}_1\rho_1 \doteq \vec{F}_1\sigma_1, \ldots, \vec{F}_n\rho_n \doteq \vec{F}_n\sigma_n\}$$

where $\vec{F}_i \in \mathsf{EVar}^*$, and $\rho_i \in \mathbb{R}$ and $\sigma_i \in \mathbb{S}$ such that if $\rho_i \in \mathbb{R}^{\to}$ then $\sigma_i \in \mathbb{S}^{\to}$ for every $1 \leq i \leq n$, and moreover Δ satisfies all of the following conditions:

(A) Δ is *well named*, which we define to hold iff the type $\vec{F}_1\rho_1 \wedge \vec{F}_1\sigma_1 \wedge \cdots \wedge \vec{F}_n\rho_n \wedge \vec{F}_n\sigma_n$ is well named.

(B) Every expansion variable $F \in \mathsf{EVar}$ has at most one inner positive occurrence in Δ, i.e., $+F$ occurs at most once in $\pm\Delta$, where $\pm\Delta$ is obtained by inserting polarities in Δ:

$$\pm\Delta = \{\vec{F}_1(\rho_1)^+ \doteq \vec{F}_1(\sigma_1)^-, \ldots, \vec{F}_n(\rho_n)^+ \doteq \vec{F}_n(\sigma_n)^-\}$$

(C) Every type variable $\alpha \in \mathsf{TVar}$ occurs at most twice in Δ. And if it occurs twice, it occurs once positively as $+\alpha$ and once negatively as $-\alpha$ in the constraint set $\pm\Delta$.

(D) For every constraint $\vec{F}\rho \doteq \vec{F}\sigma$ in Δ, we have $\mathsf{Var}(\rho) \cap \mathsf{Var}(\sigma) = \varnothing$.

(E) For every connected $\Delta' \subset \Delta$ and constraint $\vec{F}\rho \doteq \vec{F}\sigma$ in $\Delta - \Delta'$, either $\mathsf{Var}(\Delta') \cap \mathsf{Var}(\rho) = \varnothing$ or $\mathsf{Var}(\Delta') \cap \mathsf{Var}(\sigma) = \varnothing$.

In words, the variables of a connected $\Delta' \subset \Delta$ can overlap with the inner variables of at most one side of a constraint $\vec{F}\rho \doteq \vec{F}\sigma$ in $\Delta - \Delta'$.

We use the name "λ-compatible" because, as shown in lemma 5.2, every constraint set induced by a λ-term satisfies the conditions above. □

Definition 3.8 (Solutions). Let $\mathbf{S} : \mathsf{Var} \to (\mathbb{E} \cup \mathbb{T}^{\to})$ be a substitution and let $\Delta = \{\tau_1 \doteq \tau_1', \ldots, \tau_n \doteq \tau_n'\}$ be a λ-compatible constraint set. We say $\mathbf{S}$ is a *solution* for Δ iff two conditions hold:

1. $\mathsf{Dom}(\mathbf{S}) \cap \mathsf{Var}(\mathbf{S}v) = \varnothing$ for every $v \in \mathsf{Var}$.
2. $\mathbf{S}\tau_i = \mathbf{S}\tau_i'$ for every $i \in \{1, \ldots, n\}$.

The first condition is a mild restriction. It can probably be eliminated, at the price of making these propositions and their proofs more complicated. □

Definition 3.9 (Principal Solutions). Let $\mathbf{S} : \mathsf{Var} \to (\mathbb{E} \cup \mathbb{T}^{\to})$ be a substitution and let Δ be a λ-compatible constraint set. The substitution $\mathbf{S}$ is a *principal solution* for Δ iff $\mathbf{S}$ is a solution for Δ and for every solution $\mathbf{S}'$ for Δ, there is a substitution $\mathbf{S}''$ such that $\mathbf{S}'\Delta = \mathbf{S}''(\mathbf{S}\Delta)$.[2] The *principality property* is the existence of a principal solution for every constraint set that has a solution. □

Example 3.10 (Uniqueness of Principal Solutions in a Weaker Sense). A peculiarity resulting from the presence of expansion variables is illustrated by a simple example. Let Δ be the constraint set:

$$\Delta = \{FG\alpha \doteq (\bar{\sigma}_1 \wedge \bar{\sigma}_2) \wedge (\bar{\sigma}_3 \wedge \bar{\sigma}_4)\}$$

where $\bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}_3, \bar{\sigma}_4 \in \mathbb{S}^{\to}$ are arbitrary. Assume that Δ is λ-compatible. By inspection, it is not difficult to see that Δ has three distinct principal solutions, ignoring any principal solution obtained from one of these three by renaming variables in its range or by adding redundant mappings. The three principal solutions in question are:

$$\mathbf{S}_1 = \{\!\!\{ F := \Box,\ G := (\Box \wedge \Box) \wedge (\Box \wedge \Box)]\!\!\} \cup \mathbf{S}_0$$

$$\mathbf{S}_2 = \{\!\!\{ F := \Box \wedge \Box,\ G^0 := \Box \wedge \Box,\ G^1 := \Box \wedge \Box]\!\!\} \cup \mathbf{S}_0$$

$$\mathbf{S}_3 = \{\!\!\{ F := (\Box \wedge \Box) \wedge (\Box \wedge \Box),\ G^{00} := \Box,\ G^{01} := \Box,\ G^{10} := \Box,\ G^{11} := \Box]\!\!\} \cup \mathbf{S}_0$$

where $\mathbf{S}_0 = \{\!\!\{\alpha^{00} := \bar{\sigma}_1, \alpha^{01} := \bar{\sigma}_2, \alpha^{10} := \bar{\sigma}_3, \alpha^{11} := \bar{\sigma}_4]\!\!\}$. None of the three substitutions can be obtained from the other two by variable renaming, in contrast to principal solutions in first-order unification, semi-unification, and other forms of unification. If $\mathbf{S}_1$, $\mathbf{S}_2$, and $\mathbf{S}_3$ are principal as claimed, then we must have $\mathbf{S}_i\Delta = \mathbf{S}(\mathbf{S}_j\Delta)$ for some substitution $\mathbf{S}$, where $i, j \in \{1, 2, 3\}$. This is indeed the case, by taking $\mathbf{S} = \{\!\!\{\ ]\!\!\}$, the identity substitution. Hence, after all, $\mathbf{S}_1\Delta = \mathbf{S}_2\Delta = \mathbf{S}_3\Delta$, and the uniqueness of principal solutions is recovered in a weaker sense. However, there is no substitution $\mathbf{S}$ such that $\mathbf{S}_i = \mathbf{S} \circ \mathbf{S}_j$, where $i, j \in \{1, 2, 3\}$ and $i \neq j$. It is worth noting that algorithm Unify in section 4 on input Δ returns $\mathbf{S}_3$, because Unify works in "top-down" fashion, i.e., it expands F before G. □

4 Algorithm for Lambda-Compatible Beta-Unification

We design a non-deterministic algorithm Unify which takes a λ-compatible constraint set Δ as input, such that if Δ has a solution then every evaluation of Unify(Δ) terminates returning a principal solution for Δ, and if Δ has no solution then every evaluation of Unify(Δ) diverges.

The presentation of algorithm Unify in figure 4 is largely self-contained — except for two parts in the "mode of operation", namely, the definition of E-env(Δ) and the evaluation of $\mathbf{S}_1 = \mathbf{S} \otimes_{\mathcal{E}} \mathbf{S}_0$, which we now explain. In general, the standard composition of two substitutions using "$\circ$" does not produce a substitution, i.e., for substitutions S_1 and S_2, there does *not* necessarily exist a substitution S_3 such that $\overline{S_3} = (\overline{S_2} \circ \overline{S_1})$. To work around this difficulty, we use "$\otimes_{\mathcal{E}}$" to denote a new binary operation on substitutions, which we will call "safe composition relative to $\mathcal{E}$", where $\mathcal{E}$ is an environment expressing certain naming constraints.

Definition 4.1 (E-Path Environment). Given a well-named type context φ, we form its *E-path environment* as follows:

$$(\text{E-env}(\varphi))(v) = \begin{cases} \vec{F} & \text{if E-path}(v, \varphi) = \{\vec{F}\}, \\ \text{undefined} & \text{if E-path}(v, \varphi) = \varnothing. \end{cases}$$

An E-path environment is a partial function $\mathcal{E} : \mathsf{Var} \to \mathsf{EVar}^*$ that is the result of applying the E-env function to a well-named type context. Let $\mathcal{E}$ be a metavariable over E-path environments. If $\Delta = \{\tau_1 \doteq \tau_1', \ldots, \tau_n \doteq \tau_n'\}$ is a well-named constraint set, then E-env(Δ) = E-env($\tau_1 \wedge \tau_1' \wedge \cdots \wedge \tau_n \wedge \tau_n'$). □

Definition 4.2 (Safe Composition). We first define a function Φ which, given a pair $(e, \mathbf{S})$ consisting of an expansion e and a substitution $\mathbf{S}$, returns a finite set

[2] We purposely write $\mathbf{S}'\Delta = \mathbf{S}''(\mathbf{S}\Delta)$ instead of $\mathbf{S}' = \mathbf{S}'' \circ \mathbf{S}$ in order to avoid pitfalls associated with the composition of substitutions in β-unification.

Metavariable conventions:

- $\bar{\rho} \in \mathbb{R}^{\rightarrow}$, $\rho \in \mathbb{R}$, $\bar{\sigma} \in \mathbb{S}^{\rightarrow}$, $\sigma, \sigma' \in \mathbb{S}$, $\tau, \tau', \tau_i, \tau_i' \in \mathbb{T}$, $\alpha \in \mathsf{TVar}$.
- $F, G \in \mathsf{EVar}$ and $H \in \mathsf{EVar}_b$, with F and G distinct and H fresh in **rule 4**.

Mode of operation:

- Initial call: $\mathrm{Unify}(\Delta) \Rightarrow \mathrm{Unify}(\mathrm{simplify}(\Delta), \{\!\{\,\}\!\}, \text{E-env}(\Delta))$.
- $\mathrm{Unify}(\varnothing, \mathbf{S}, \mathcal{E}) \Rightarrow \mathbf{S}$.
- $\mathrm{Unify}(\Delta_0, \mathbf{S}_0, \mathcal{E}) \Rightarrow \mathrm{Unify}(\Delta_1, \mathbf{S}_1, \mathcal{E})$, where $\Delta_0 = \Delta \cup \vec{F}\{\rho \doteq \sigma\}$, $\Delta_1 = \mathbf{S}\Delta_0$ and $\mathbf{S}_1 = \mathbf{S} \otimes_{\mathcal{E}} \mathbf{S}_0$, provided $\rho \doteq \sigma \Rightarrow \mathbf{S}$ is an instance of one of the rewrite rules.

Rewrite rules:

$$\begin{array}{llll} \alpha \doteq \bar{\sigma} \Rightarrow \{\!\{\alpha := \bar{\sigma}\}\!\} & \textbf{(rule 1)} & F\bar{\rho} \doteq \bar{\sigma} \Rightarrow \{\!\{F := \Box\}\!\} & \textbf{(rule 3)} \\ \bar{\rho} \doteq \alpha \Rightarrow \{\!\{\alpha := \bar{\rho}\}\!\} & \textbf{(rule 2)} & F\bar{\rho} \doteq G\sigma \Rightarrow \{\!\{F := GH\Box\}\!\} & \textbf{(rule 4)} \\ & & F\bar{\rho} \doteq \sigma \wedge \sigma' \Rightarrow \{\!\{F := F^0\Box \wedge F^1\Box\}\!\} & \textbf{(rule 5)} \end{array}$$

Applying substitutions to constraint sets:

- $\mathbf{S}(\{\tau \doteq \tau'\} \cup \Delta) = \mathrm{simplify}((\mathbf{S}\tau) \doteq (\mathbf{S}\tau')) \cup \mathbf{S}\Delta$.
- $\mathrm{simplify}(\{\tau \doteq \tau'\} \cup \Delta) = \mathrm{simplify}(\tau \doteq \tau') \cup \mathrm{simplify}(\Delta)$.
- $$\mathrm{simplify}(\tau \doteq \tau') = \begin{cases} F\,\mathrm{simplify}(\tau_1 \doteq \tau_1') & \text{if } \tau = F\tau_1 \text{ and } \tau' = F\tau_1', \\ \mathrm{simplify}(\tau_1' \doteq \tau_1) \cup \mathrm{simplify}(\tau_2 \doteq \tau_2') & \text{if } \tau = \tau_1 \to \tau_2 \text{ and } \tau' = \tau_1' \to \tau_2', \\ \mathrm{simplify}(\tau_1 \doteq \tau_1') \cup \mathrm{simplify}(\tau_2 \doteq \tau_2') & \text{if } \tau = \tau_1 \wedge \tau_2 \text{ and } \tau' = \tau_1' \wedge \tau_2', \\ \varnothing & \text{if } \tau = \tau', \\ \{\tau \doteq \tau'\} & \text{otherwise.} \end{cases}$$

Figure 4: Algorithm Unify.

$\Phi(e, \mathbf{S})$ of triples in $\{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*$. The definition of Φ is by induction on e, for a fixed $\mathbf{S}$ throughout:

$$\begin{array}{l} \Phi(\Box, \mathbf{S}) \\ \quad = \{(\varepsilon, \varepsilon, \varepsilon)\} \\ \Phi(e_0 \wedge e_1, \mathbf{S}) \\ \quad = \{\, (i \cdot p, i \cdot q, r) \mid (p, q, r) \in \Phi(e_i, \mathbf{S}), i \in \{0,1\} \,\} \\ \Phi(Fe, \mathbf{S}) \\ \quad = \{\, (s_i \cdot p, q, s_i \cdot r) \mid (p, q, r) \in \Phi(\langle e \rangle^{s_i}, \mathbf{S}), 1 \le i \le n \,\} \\ \quad \text{where } \mathbf{S}\,F = e',\ \#_{\Box}(e') = n, \text{ and} \\ \qquad s_i = \mathrm{path}(\Box^{(i)}, e') \text{ for } 1 \le i \le n \end{array}$$

The intention of Φ is that if $(p, q, r) \in \Phi(e, \mathbf{S})$, then the path p identifies hole $\Box^{(i)}$ in $\mathbf{S}\,e$, the path q identifies hole $\Box^{(j)}$ in e of which $\Box^{(i)}$ in $\mathbf{S}\,e$ is a copy, and r is the subsequence of the path p contributed by the substitution $\mathbf{S}$.

Given substitutions $\mathbf{S}_1$ and $\mathbf{S}_2$, together with an E-path environment $\mathcal{E}$, the *safe composition of* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *relative to* $\mathcal{E}$ is a new substitution defined by:

$$(\mathbf{S}_2 \otimes_{\mathcal{E}} \mathbf{S}_1)(v) = \begin{cases} \mathbf{S}_2\langle \mathbf{S}_1(v_0^q) \rangle^r & \text{if } v = v_0^p,\ \mathcal{E}(v_0) = \vec{F}, \\ & \mathbf{S}_1(\vec{F}\Box) = e, \text{ and} \\ & (p, q, r) \in \Phi(e, \mathbf{S}_2), \\ \{\!\{\,\}\!\}\,(v) & \text{otherwise.} \end{cases}$$

□

In lemma 4.3, we state a condition and prove that it is sufficient to guarantee that the action of the substitution $\mathbf{S}_2 \otimes_{\mathcal{E}} \mathbf{S}_1$ on a type τ is equal to the action of the composition $\mathbf{S}_2 \circ \mathbf{S}_1$ on τ.

Lemma 4.3 (Sufficient Condition for Safe Composition). *Let* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *be substitutions, let* τ *be a type, and let* $\mathbf{S}' = \mathbf{S}_2 \otimes_{\mathcal{E}} \mathbf{S}_1$ *for some E-path environment* $\mathcal{E}$*. If* τ *and* $\mathbf{S}_1\tau$ *are well-named types and* $\mathcal{E} \supseteq$ *E-env*(τ)*, then* $\mathbf{S}'(\tau) = \mathbf{S}_2(\mathbf{S}_1(\tau))$. □

Definition 4.4 (Unification Algorithm). The operation of Unify is based on the rewrite rules shown in figure 4. Because Unify is non-deterministic, there are in general many evaluations of Unify for the same input Δ. We exhibit the consecutive rewrite steps of a particular evaluation of $\mathrm{Unify}(\Delta)$ by writing: $\mathrm{Unify}(\Delta_0, \{\!\{\,\}\!\}, \mathcal{E}) \Rightarrow \mathrm{Unify}(\Delta_1, \mathbf{S}_1, \mathcal{E}) \Rightarrow \cdots \Rightarrow \mathrm{Unify}(\Delta_i, \mathbf{S}_i, \mathcal{E}) \Rightarrow \cdots$ where $\Delta_0 = \mathrm{simplify}(\Delta)$ and $\mathcal{E} = \text{E-env}(\Delta)$. To indicate that the evaluation of $\mathrm{Unify}(\Delta)$ makes $i \ge 1$ calls to Unify (beyond the initial call), and that the arguments of the ith call are the constraint set Δ_i, the substitution $\mathbf{S}_i$, and the E-path environment $\mathcal{E}$, we write:

$$\mathrm{Unify}(\Delta_0, \{\!\{\,\}\!\}, \mathcal{E}) \stackrel{i}{\Rightarrow} \mathrm{Unify}(\Delta_i, \mathbf{S}_i, \mathcal{E})\ .$$

We write $\mathrm{Unify}(\Delta_0, \{\!\{\,\}\!\}, \mathcal{E}) \stackrel{*}{\Rightarrow} \mathrm{Unify}(\Delta, \mathbf{S}, \mathcal{E})$ to mean that either $(\Delta_0, \{\!\{\,\}\!\}, \mathcal{E}) = (\Delta, \mathbf{S}, \mathcal{E})$ or there is an evaluation of $\mathrm{Unify}(\Delta)$ such that $\mathrm{Unify}(\Delta_0, \{\!\{\,\}\!\}, \mathcal{E}) \stackrel{i}{\Rightarrow} \mathrm{Unify}(\Delta, \mathbf{S}, \mathcal{E})$ for some $i \ge 1$.

Whenever we say Δ and Δ' are λ-compatible constraint sets such that $\mathrm{Unify}(\Delta, \mathbf{S}, \mathcal{E}) \Rightarrow \mathrm{Unify}(\Delta', \mathbf{S}', \mathcal{E})$ for some substitutions $\mathbf{S}$ and $\mathbf{S}'$ and some E-path environment $\mathcal{E}$, we assume that $\mathrm{simplify}(\Delta) = \Delta$. The way Unify is defined guarantees that $\mathrm{simplify}(\Delta') = \Delta'$ again. □

Lemma 4.5 (λ-Compatibility Is Invariant). *Let* Δ_0 *and* Δ_1 *be constraint sets such that*

$$\mathrm{Unify}(\Delta_0, \mathbf{S}_0, \mathcal{E}) \Rightarrow \mathrm{Unify}(\Delta_1, \mathbf{S}_1, \mathcal{E})$$

for some substitutions $\mathbf{S}_0$ *and* $\mathbf{S}_1$ *and some E-path environment* $\mathcal{E}$ *(which do not matter here). If* Δ_0 *is* λ*-compatible, then so is* Δ_1*. In words, the properties listed in definition 3.7 are invariant relative to the rewrite rules of* Unify. □

Lemma 4.6 (Progress). *If Δ_0 is a non-empty λ-compatible constraint set such that* $\mathrm{simplify}(\Delta_0) = \Delta_0$, *together with a substitution $\mathbf{S}_0$ and an E-path environment $\mathcal{E}$, then there is a constraint set Δ_1 such that*

$$\mathrm{Unify}(\Delta_0, \mathbf{S}_0, \mathcal{E}) \Rightarrow \mathrm{Unify}(\Delta_1, \mathbf{S}_1, \mathcal{E})$$

for some substitution $\mathbf{S}_1$ (which does not matter here). In words, Δ_0 always contains a constraint that can be processed by one of the rewrite rules of Unify. □

Lemma 4.7 (Solutions with Finite Support Suffice). *Let Δ be a λ-compatible constraint set and let $\mathbf{S} : \mathsf{Var} \to (\mathbb{E} \cup \mathbb{T}^{\rightarrow})$ be a substitution. If $\mathbf{S}$ is a solution for Δ, then we can construct a substitution $\mathbf{S}'$ from $\mathbf{S}$ such that:*

1. $\mathsf{Dom}(\mathbf{S}')$ *is finite.*
2. $\mathbf{S}'$ *is a solution for* Δ. □

Definition 4.8 (Size). Given a type $\tau \in \mathbb{T}$, the function size( ) applied to τ returns an integer $\mathrm{size}(\tau) \geq 1$ which is the number of symbols in τ excluding all occurrences of E-variables and all parentheses. We extend size( ) to expansions $e \in \mathbb{E}$ in the obvious way. Formal definitions, by induction on τ and e respectively, are omitted. □

Definition 4.9 (Degree). Let Δ be a λ-compatible constraint set, with $\mathsf{EVar}(\Delta) = \{F_1, \ldots, F_k\}$. Let $\mathbf{S}$ be a solution for Δ with finite $\mathsf{Dom}(\mathbf{S})$, say:

$$\mathbf{S} = \{\!|G_1 := e_1, \ldots, G_m := e_m, \alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n|\!\}\,.$$

If $\vec{F} \in \mathsf{EVar}^*$, let $|\vec{F}|$ denote its length, i.e., the number of variables in the sequence. We define a measure $\mathrm{degree}(\mathbf{S}, \Delta) = (p, q, r)$ on the pair $(\mathbf{S}, \Delta)$, where $p, q, r \in \mathcal{N}$, as follows:

- $p = \sum_{1 \leq i \leq m} (\mathrm{size}(e_i) - 1)$
- $q = \mathrm{size}(\Delta)$
- $r = \sum_{1 \leq i \leq k} (k - |\text{E-path}(F_i, \Delta)|)$

Given two triples (p, q, r) and (p', q', r') of natural numbers, we write $(p, q, r) < (p', q', r')$ iff either $p < p'$, or $p = p'$ and $q < q'$, or $p = p'$, $q = q'$ and $r < r'$. This is the so-called "lexicographic ordering" on triples of natural numbers, and it is easy to see that it is well-founded. □

Lemma 4.10 (Decreasing Degree when Solvable). *Let Δ_0 be a λ-compatible constraint set, let Δ_1 be a constraint set, and let*

$$\mathrm{Unify}(\Delta_0, \mathbf{S}_0, \mathcal{E}) \Rightarrow \mathrm{Unify}(\Delta_1, \mathbf{S}_1, \mathcal{E})$$

for some substitutions $\mathbf{S}_0$ and $\mathbf{S}_1$ and some E-path environment $\mathcal{E}$ (which do not matter here). If there is a solution $\mathbf{S}'_0$ for Δ_0 with $\mathsf{Dom}(\mathbf{S}'_0)$ finite, then there is a solution $\mathbf{S}'_1$ for Δ_1 with $\mathsf{Dom}(\mathbf{S}'_1)$ finite such that $\mathrm{degree}(\mathbf{S}'_1, \Delta_1) < \mathrm{degree}(\mathbf{S}'_0, \Delta_0)$. □

Lemma 4.11 (Principal Solution Constructed). *Let Δ_0 be a λ-compatible constraint set with $\mathcal{E} =$ E-env(Δ_0), let Δ_1 be a constraint set, and let*

$$\mathrm{Unify}(\Delta_0, \mathbf{S}, \mathcal{E}) \stackrel{*}{\Rightarrow} \mathrm{Unify}(\Delta_1, \tilde{\mathbf{S}} \otimes_{\mathcal{E}} \mathbf{S}, \mathcal{E})$$

for some substitutions $\mathbf{S}$ and $\tilde{\mathbf{S}}$. If $\mathbf{S}_1$ is a principal solution for Δ_1, then $\mathbf{S}_1 \otimes_{\mathcal{E}} \tilde{\mathbf{S}}$ is a principal solution for Δ_0. □

The following theorem shows that the algorithm is sound (i.e., the substitutions Unify produces when it terminates are in fact solutions) and complete (i.e., Unify produces a solution if there is one), as well as showing it produces principal solutions.

Theorem 4.12 (Soundness, Completeness, & Principality). *Let Δ be a λ-compatible constraint set with $\mathcal{E} =$ E-env(Δ).*

1. *Δ has a solution if and only if* $\mathrm{Unify}(\mathrm{simplify}(\Delta), \{\!|\ |\!\}, \mathcal{E}) \stackrel{*}{\Rightarrow} \mathrm{Unify}(\varnothing, \mathbf{S}, \mathcal{E})$ *for some substitution $\mathbf{S}$.*
2. *If* $\mathrm{Unify}(\mathrm{simplify}(\Delta), \{\!|\ |\!\}, \mathcal{E}) \stackrel{*}{\Rightarrow} \mathrm{Unify}(\varnothing, \mathbf{S}, \mathcal{E})$ *for some $\mathbf{S}$, then $\mathbf{S}$ is a principal solution for Δ.* □

Note that Unify diverges exactly when there is no solution. The evaluation strategy does not matter, because lemma 4.10 implies termination when there is a solution and lemma 4.11 implies divergence when no solution exists.

5 Type Inference Algorithm

This section defines a procedure which, given a λ-term M, generates a finite set $\Gamma(M)$ of constraints, the solvability of which is equivalent to the typability of the term M. We use this to prove the principality property for System $\mathbb{I}$ and to define a complete type inference algorithm.

Definition 5.1 (Algorithm Generating Constraints and Skeleton). For every λ-term M, figure 5 gives an inductive definition of a set of constraints $\Gamma(M)$ and a derivation skeleton $\mathrm{Skel}(M)$, defined simultaneously with a type $\mathrm{Typ}(M)$ and a type environment $\mathrm{Env}(M)$. In this definition, for a given subterm occurrence N, when a fresh variable is chosen, the same fresh variable must be used in $\mathrm{Env}(N)$, $\mathrm{Typ}(N)$, $\Gamma(N)$, and $\mathrm{Skel}(N)$. The process of going from M to $\Gamma(M)$ and $\mathrm{Skel}(M)$ is uniquely determined up to the choice of expansion variables and type variables. □

Lemma 5.2 (Constraint Set is λ-Compatible). *Let M be an arbitrary λ-term. The constraint set $\Gamma(M)$ induced by M is λ-compatible.* □

Lemma 5.3 (All Derivations Instances of $\mathrm{Skel}(M)$). *If $\mathcal{D}$ is a derivation of System $\mathbb{I}$ with final judgement $A \vdash M : \tau$, then there exists some substitution $\mathbf{S}$ such that $\mathcal{D} = \mathbf{S}(\mathrm{Skel}(M))$.* □

Theorem 5.4 (Constraint Set and Skeleton Equivalent). *Given λ-term M, a substitution $\mathbf{S}$ is a solution for $\Gamma(M)$ if and only if $\mathbf{S}(\mathrm{Skel}(M))$ is a derivation of System $\mathbb{I}$. Thus, $\Gamma(M)$ is solvable if and only if M is typable in System $\mathbb{I}$.* □

Corollary 5.5. *It is undecidable whether an arbitrarily chosen λ-compatible constraint set Δ has a solution.* □

From the principality property for λ-compatible β-unification, we can derive the following.

If $M = x$, for fresh $\alpha \in \mathsf{TVar}_b$:	$\mathrm{Typ}(M) = \alpha$, $\mathrm{Env}(M) = \{x \mapsto \alpha\}$, $\Gamma(M) = \varnothing$, $\mathrm{Skel}(M) = \langle \mathrm{VAR}, \mathrm{Env}(M) \vdash M : \alpha \rangle$.
If $M = (N_1 N_2)$, for fresh $F \in \mathsf{EVar}_b$, $\beta \in \mathsf{TVar}_b$:	$\mathrm{Typ}(M) = \beta$, $\mathrm{Env}(M) = \mathrm{Env}(N_1) \wedge F\,\mathrm{Env}(N_2)$, $\Gamma(M) = \Gamma(N_1) \cup F\,\Gamma(N_2) \cup \{\mathrm{Typ}(N_1) \doteq F\,\mathrm{Typ}(N_2) \to \beta\}$, $\mathrm{Skel}(M) = \langle \mathrm{APP}, \mathrm{Env}(M) \vdash M : \beta, \mathrm{Skel}(N_1)\ (F\,\mathrm{Skel}(N_2)) \rangle$.
If $M = (\lambda x.N)$, for fresh $\alpha \in \mathsf{TVar}_b$:	$\mathrm{Typ}(M) = \begin{cases} \mathrm{Env}(N)(x) \to \mathrm{Typ}(N) & \text{if } \mathrm{Env}(N)(x) \text{ defined,} \\ \alpha \to \mathrm{Typ}(N) & \text{otherwise,} \end{cases}$ $\mathrm{Env}(M) = \mathrm{Env}(N) \backslash x$, $\Gamma(M) = \Gamma(N)$, $\mathrm{Skel}(M) = \langle R, \mathrm{Env}(M) \vdash M : \mathrm{Typ}(M), \mathrm{Skel}(N) \rangle$ where if $x \in \mathrm{FV}(N)$, then $R = \text{ABS-I}$, else $R = \text{ABS-K}$.

Figure 5: Definition of $\Gamma(M)$, $\mathrm{Skel}(M)$, $\mathrm{Typ}(M)$, and $\mathrm{Env}(M)$.

Theorem 5.6 (Principal Typings and Completeness of Type Inference). *Let* PT *be the algorithm such that*

$$\mathrm{PT}(M) = (\mathrm{Unify}(\Gamma(M)))(\mathrm{Skel}(M))$$

If M is typable in System $\mathbb{I}$, then $\mathrm{PT}(M)$ *returns a principal typing for M, else* $\mathrm{PT}(M)$ *diverges. Thus, System $\mathbb{I}$ has the principality property and* PT *is a complete type inference algorithm for System $\mathbb{I}$.* □

6 Termination and Decidability at Finite Ranks

This section defines UnifyFR, an adaptation of algorithm Unify which produces a solution $\mathbf{S}$ with bounded *rank* k for a λ-compatible constraint set Δ. The definition of UnifyFR differs from Unify only in the "mode of operation" as presented in figure 6. The invocation of UnifyFR on Δ at rank k produces a solution $\mathbf{S}$ if $\mathrm{Unify}(\Delta)$ produces $\mathbf{S}$ and the rank of $\mathbf{S}$ is bounded by k. Otherwise UnifyFR halts indicating failure, unlike Unify which diverges if it can not find a solution.

Note that the principality of solutions produced by UnifyFR follows from the principality of solutions produced by Unify.

Definition 6.1 (Rank of Types). For every $s \in \{\mathsf{L}, \mathsf{R}, 0, 1\}^*$, let $\#_{\mathsf{L}}(s)$ denote the number of occurrences of L in s. Let $\tau \in \mathbb{T}$. There is a smallest (and, therefore, unique) $\varphi \in \mathbb{T}_\Box$ with $n \geq 1$ holes such that

1. $\tau = \varphi[\tau_1, \ldots, \tau_n]$ for some $\tau_1, \ldots, \tau_n \in \mathbb{T}$.
2. None of the types in $\{\tau_1, \ldots, \tau_n\}$ contains an occurrence of "$\wedge$".

The *rank of hole* $\Box^{(i)}$ *in* φ is given by $\text{hole-rank}(\Box^{(i)}, \varphi) = \#_{\mathsf{L}}(\mathrm{path}(\Box^{(i)}, \varphi))$. If $\varphi = \Box$, i.e., τ does not mention any "$\wedge$", we define $\mathrm{rank}(\tau) = 0$. If $\varphi \neq \Box$, we define $\mathrm{rank}(\tau)$ by:

$$\mathrm{rank}(\tau) = 1 + \max\{\text{hole-rank}(\Box^{(i)}, \varphi) \mid 1 \leq i \leq \#_\Box(\varphi)\}\ .$$

This definition of $\mathrm{rank}(\tau)$ is equivalent to others found in the literature.

If $\Delta = \{\tau_1 \doteq \tau_1', \ldots, \tau_n \doteq \tau_n'\}$ is a λ-compatible constraint set, we define $\mathrm{rank}(\Delta)$ by:

$$\mathrm{rank}(\Delta) = \max\{\mathrm{rank}(\tau_1), \mathrm{rank}(\tau_1'), \ldots, \mathrm{rank}(\tau_n), \mathrm{rank}(\tau_n')\}$$

Definition 6.2 (Rank-k System of Intersection Types). Let $k \geq 1$. If $\mathcal{S}$ is a skeleton of System $\mathbb{I}$ where every environment type has rank $\leq k-1$ and every derived type has rank $\leq k$, we write $\mathrm{rank}(\mathcal{S}) \leq k$ and say that $\mathcal{S}$ is a rank-k skeleton.

We define the restriction $\mathbb{I}_k$ of System $\mathbb{I}$ as follows. A skeleton $\mathcal{S}$ of $\mathbb{I}$ is a skeleton of $\mathbb{I}_k$ iff $\mathrm{rank}(\mathcal{S}) \leq k$. A particular susbet of the rank-k skeletons are the rank-k derivations. □

Definition 6.3 (Rank-k Solution). Let Δ be a λ-compatible constraint set, $A \subseteq \mathsf{TVar}$, and $k \geq 1$. We say that a substitution $\mathbf{S} : \mathsf{Var} \to (\mathbb{E} \cup \mathbb{T}^{\to})$ is *a rank-k solution for Δ relative to A* provided:

1. $\mathbf{S}$ is a solution for Δ.
2. $\mathrm{rank}(\mathbf{S}\alpha) \leq k-1$, for every $\alpha \in A$.
3. $\mathrm{rank}(\mathbf{S}\alpha) \leq k$, for every $\alpha \notin A$.

The set A discriminates between T-variables corresponding to environment types and T-variables corresponding to derived types in a typing; for a rank-k typing, the first must have rank $\leq k-1$, and the second must have rank $\leq k$. □

Lemma 6.4. *Let M be a λ-term and $\Delta = \Gamma(M)$. If $\mathbf{S}$ is a rank-k solution of Δ, then there is a derivation of M is System $\mathbb{I}_{k+2}$.* □

To show that for a fixed $A \subseteq \mathsf{TVar}$ and fixed $k \geq 1$, an evaluation of $\mathrm{UnifyFR}(\Delta, A, k)$ always terminates, we need to reason about the *rank of a constraint* in a constraint set. The following definitions support this.

Definition 6.5 (λ-Compatible Pairs). Let (τ, τ') be a pair of types. We define its *constraint decomposition sequence* $d(\tau, \tau') = (\varphi, \hat{\tau}_1, \ldots, \hat{\tau}_n, \hat{\tau}_1', \ldots, \hat{\tau}_n')$ with $1+2n$ entries, where $\varphi \in \mathbb{T}_\Box$ with $n \geq 0$ holes is the largest (and therefore unique) type context such that

$$\begin{array}{ll} \tau = \varphi[\tau_1, \ldots, \tau_n], & \\ \tau' = \varphi[\tau_1', \ldots, \tau_n'], & \\ \{\tau_i, \tau_i'\} = \{\hat{\tau}_i, \hat{\tau}_i'\} & \text{for } 1 \leq i \leq n, \\ (\tau_i, \tau_i') = (\hat{\tau}_i, \hat{\tau}_i') & \text{if hole-rank}(\Box^{(i)}, \varphi) \text{ is even,} \\ (\tau_i, \tau_i') = (\hat{\tau}_i', \hat{\tau}_i) & \text{if hole-rank}(\Box^{(i)}, \varphi) \text{ is odd.} \end{array}$$

Mode of operation:

- Initial call: $\mathrm{UnifyFR}(\Delta, A, k) \Rightarrow \mathrm{UnifyFR}(\mathrm{simplify}(\Delta), \{\![\,]\!\}, \text{E-env}(\Delta), A, k)$ for $A \subseteq \mathsf{TVar}$ and $k \geq 1$.
- $\mathrm{UnifyFR}(\varnothing, \mathbf{S}, \mathcal{E}, A, k) \Rightarrow \mathbf{S}$.
- $\mathrm{UnifyFR}(\Delta_0, \mathbf{S}_0, \mathcal{E}, A, k) \Rightarrow \mathrm{UnifyFR}(\Delta_1, \mathbf{S}_1, \mathcal{E}, A, k)$ if $\mathrm{Unify}(\Delta_0, \mathbf{S}_0, \mathcal{E}) \Rightarrow \mathrm{Unify}(\Delta_1, \mathbf{S}_1, \mathcal{E})$ and also:
  - $\mathrm{rank}(\mathbf{S}_1\alpha) \leq k-1$ for every $\alpha \in A$.
  - $\mathrm{rank}(\mathbf{S}_1\alpha) \leq k$ for every $\alpha \notin A$.

Figure 6: Algorithm UnifyFR (refer to figure 4 for missing parts).

If $\vec{G}_i = \text{E-path}(\square^{(i)}, \varphi)$ for $1 \leq i \leq n$, then its *constraint set decomposition* is:

$$\Delta(\tau, \tau') = \{\vec{G}_1\hat{\tau}_1 \doteq \vec{G}_1\hat{\tau}'_1, \ldots, \vec{G}_n\hat{\tau}_n \doteq \vec{G}_n\hat{\tau}'_n\}.$$

If $\Delta(\tau, \tau')$ is a λ-compatible constraint set, then (τ, τ') is a *λ-compatible pair*. In this case, $\hat{\tau}_i \in \mathbb{R}$ and $\hat{\tau}'_i \in \mathbb{S}$ for $1 \leq i \leq n$, so we can let $\hat{\tau}_i = \rho_i$ and $\hat{\tau}'_i = \sigma_i$ for $1 \leq i \leq n$ and write $\Delta(\tau, \tau')$ in the form:

$$\Delta(\tau, \tau') = \{\vec{G}_1\rho_1 \doteq \vec{G}_1\sigma_1, \ldots, \vec{G}_n\rho_n \doteq \vec{G}_n\sigma_n\}.$$

Note that $\mathrm{simplify}(\Delta(\tau, \tau')) = \Delta(\tau, \tau')$, because $d(\tau, \tau')$ chooses the largest φ with the stated property. We define the *rank of constraint $\vec{G}_i\rho_i \doteq \vec{G}_i\sigma_i$ in (τ, τ')*:

$$\mathrm{rank}(\vec{G}_i\rho_i \doteq \vec{G}_i\sigma_i, (\tau, \tau')) = \text{hole-rank}(\square^{(i)}, \varphi).$$

We also define $h(\tau, \tau')$:

$$h(\tau, \tau') = \min\{\,\text{hole-rank}(\square^{(i)}, \varphi) \mid 1 \leq i \leq n\,\},$$

i.e., $h(\tau, \tau')$ is a lower bound on the $\llcorner$-distance of all the holes in φ from its root (viewed as a binary tree). If $\tau = \tau' = \varphi$, i.e., φ has 0 holes, we leave $h(\tau, \tau')$ undefined. □

Definition 6.6 (Evaluating λ-Compatible Pairs). Let (τ_0, τ'_0) and (τ_1, τ'_1) be λ-compatible pairs. Let **rule a** be one of the 5 rules listed in figure 4. We write

$$(\tau_0, \tau'_0) \underset{\mathbf{a}}{\Rrightarrow} (\tau_1, \tau'_1)$$

iff $d(\tau_0, \tau'_0) = (\varphi, \rho_1, \ldots, \rho_n, \sigma_1, \ldots, \sigma_n)$ and there is $i \in \{1, \ldots, n\}$ such that:

1. $\rho_i \doteq \sigma_i \Rightarrow \mathbf{S}$ is an instance of **rule a**.
2. $(\tau_1, \tau'_1) = (\mathbf{S}\tau_0, \mathbf{S}\tau'_0)$.

In such a case, we say that (τ_0, τ'_0) is *evaluated to* (τ_1, τ'_1) by **rule a**. Moreover, if $\text{hole-rank}(\square^{(i)}, \varphi) = k$, we say that the constraint $\rho_i \doteq \sigma_i$ is *at rank k* and that the evaluation from (τ_0, τ'_0) to (τ_1, τ'_1) is also *at rank k*, indicated by writing

$$(\tau_0, \tau'_0) \underset{(\mathbf{a},k)}{\Longrightarrow} (\tau_1, \tau'_1).$$

We write $(\tau_0, \tau'_0) \Rightarrow (\tau_1, \tau'_1)$ in case $(\tau_0, \tau'_0) \underset{\mathbf{a}}{\Rrightarrow} (\tau_1, \tau'_1)$ for some **rule a**, and $\overset{*}{\Rightarrow}$ for the reflexive transitive closure of $\Rightarrow$.

Let $\mathcal{R} \subseteq \{\mathbf{1}, \ldots, \mathbf{5}\}$. Let $(\tau_0, \tau'_0) \Rightarrow \cdots \Rightarrow (\tau_n, \tau'_n)$ be an evaluation sequence with $n \geq 1$ steps. We write $(\tau_0, \tau'_0) \overset{n}{\underset{\mathcal{R}}{\Rrightarrow}} (\tau_n, \tau'_n)$ to indicate that the evaluation has n steps, and that each step is carried out using **rule a** for some $\mathbf{a} \in \mathcal{R}$.

Finally, if there is no pair (τ_1, τ'_1) such that $(\tau_0, \tau'_0) \underset{\mathcal{R}}{\Rrightarrow} (\tau_1, \tau'_1)$, then we say that the pair (τ_0, τ'_0) is in *$\mathcal{R}$-normal form*. □

Lemma 6.7 (Evaluating without Rule 5). *Let $\mathcal{R} = \{\mathbf{1}, \ldots, \mathbf{4}\}$, i.e., $\mathcal{R}$ is the set of all rewrite rules without* **rule 5**. *If (τ_0, τ'_0) is a λ-compatible pair, there is a bound $M(\tau_0, \tau'_0)$ solely depending on (τ_0, τ'_0) such that for every evaluation with $\mathcal{R}$: $(\tau_0, \tau'_0) \overset{n}{\underset{\mathcal{R}}{\Rrightarrow}} (\tau_1, \tau'_1)$, we have $n \leq M(\tau_0, \tau'_0)$. In words, a non-terminating evaluation of (τ_0, τ'_0) must use infinitely many times* **rule 5**. □

Lemma 6.8 (Evaluating with Rule 5 at a Fixed Rank). *Let $\mathcal{R}$ be the set of all rewrite rules without* **rule 5**, *as in lemma 6.7.*

Hypothesis: *Let (τ_0, τ'_0) be a λ-compatible pair in $\mathcal{R}$-normal form, with $k = h(\tau_0, \tau'_0)$, and consider an arbitrary evaluation with the rules in $\mathcal{R}$ (with no rank restriction) and* **rule 5** *restricted to rank k:*

$$(\tau_0, \tau'_0) \underset{\mathbf{a}_1}{\Longrightarrow} (\tau_1, \tau'_1) \underset{\mathbf{a}_2}{\Longrightarrow} (\tau_2, \tau'_2) \underset{\mathbf{a}_3}{\Longrightarrow} \cdots \underset{\mathbf{a}_n}{\Longrightarrow} (\tau_n, \tau'_n)$$

where for every $1 \leq i \leq n$, either $\mathbf{a}_i \in \mathcal{R}$ or $\mathbf{a}_i = \mathbf{5}$ and the step $(\tau_{i-1}, \tau'_{i-1}) \underset{\mathbf{5}}{\Rrightarrow} (\tau_i, \tau'_i)$ is at rank k.

Conclusion: *There is a uniform bound $N(\tau_0, \tau'_0)$ solely depending on (τ_0, τ'_0) such that $n \leq N(\tau_0, \tau'_0)$. Moreover, if (τ_n, τ'_n) cannot be evaluated further, i.e., if (τ_n, τ'_n) is in $\mathcal{R}$-normal form and in $(\mathbf{5}, k)$-normal form (see definition 6.6), then either $\tau_n = \tau'_n$ or $h(\tau_n, \tau'_n) > k$.* □

Definition 6.9 (Increasing-Rank Evaluations). Let $\mathcal{R}$ be the set of all rewrite rules without **rule 5**, as in lemma 6.8. Let (τ_0, τ'_0) be a λ-compatible pair. An *increasing-rank* evaluation of (τ_0, τ'_0) is of the form:

$$(\tau_0, \tau'_0) \overset{*}{\underset{\mathcal{R}}{\Rrightarrow}} (\tau_1, \tau'_1) \overset{*}{\underset{\mathcal{R}_1}{\Rrightarrow}} (\tau_2, \tau'_2) \overset{*}{\underset{\mathcal{R}_2}{\Rrightarrow}} \cdots$$

where (τ_1, τ'_1) is in $\mathcal{R}$-normal form and, for every $i \geq 1$, if $\tau_i \neq \tau'_i$ then $\mathcal{R}_i = \mathcal{R} \cup \{(\mathbf{5}, k_i)\}$ where $k_i = h(\tau_i, \tau'_i)$ and $(\tau_{i+1}, \tau'_{i+1})$ is in $\mathcal{R}_i$-normal form. □

Definition 6.10 (Coding λ-Compatible Constraint Sets as λ-Compatible Pairs). The coding $\ulcorner\Delta\urcorner$ of a λ-compatible constraint set $\Delta = \{\tau_1 \doteq \tau'_1, \ldots, \tau_n \doteq \tau'_n\}$ is the pair (τ, τ') given by:

$$\tau = (\tau_1 \wedge \cdots \wedge (\tau_{n-1} \wedge \tau_n))$$
$$\tau' = (\tau'_1 \wedge \cdots \wedge (\tau'_{n-1} \wedge \tau'_n))$$

It is clear that $\ulcorner\Delta\urcorner$ is a λ-compatible pair (definition 6.5). □

Lemma 6.11 (Increasing-Rank Evaluations Complete). *Let Δ be a λ-compatible constraint set, and let $(\tau, \tau') = \ulcorner\Delta\urcorner$. If an increasing-rank evaluation of (τ, τ') does not terminate, then Δ has no solution.* □

Theorem 6.12 (Decidability of Finite-Rank β-Unification). *Let Δ be a λ-compatible constraint set, A a set of T-variables, and $k \geq 1$.*

1. *Δ has a rank-k solution relative to A iff there is a successful evaluation* $\mathrm{UnifyFR}(\Delta, A, k) \stackrel{*}{\Rightarrow} \mathbf{S}$.
2. *There are no infinite (diverging) evaluations starting from* $\mathrm{UnifyFR}(\Delta, A, k)$.
3. *It is decidable whether Δ has a rank-k solution relative to A.* □

References

[vB93] S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Catholic University of Nijmegen, 1993.

[Ban97] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*, 1997.

[CDCV80] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In Seldin and Hindley [SH80], pp. 535–560.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Z. Math. Log. Grund. Math.*, 27:45–58, 1981.

[CG92] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *17th Colloq. Trees in Algebra and Programming*, vol. 581 of *LNCS*, pp. 102–123. Springer-Verlag, 1992.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Principles of Programming Languages*, pp. 207–212, 1982.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[Jen98] T. Jensen. Inference of polymorphic and conditional strictness properties. In *Conf. Rec. POPL '98: 25th ACM Symp. Principles of Prog. Languages*, 1998.

[Jim96] T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Principles of Prog. Languages*, 1996.

[JMZ92] B. Jacobs, I. Margaria, and M. Zacchi. Filter models with polymorphic types. *Theor. Comp. Sc.*, 95:143–158, 1992.

[Kfo9X] A. J. Kfoury. Beta-reduction as unification. In D. Niwinski, ed., *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996).* Springer-Verlag, 199X.

[KW94] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, 1994.

[Lei83] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Principles of Programming Languages*, pp. 88–98, 1983.

[MTHM90] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1990.

[Pie94] B. Pierce. Bounded quantification is undecidable. *Inf. & Comput.*, 112:131–165, 1994.

[PJHH+93] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: A technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conf.*, 1993.

[Pot80] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In Seldin and Hindley [SH80], pp. 561–577.

[RDR88] S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theor. Comp. Sc.*, 59:181–209, 1988.

[RDRV84] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theor. Comp. Sc.*, 28:151–169, 1984.

[SH80] J. P. Seldin and J. R. Hindley, eds. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[SM96] É. Sayag and M. Mauny. A new presentation of the intersection type discipline through principal typings of normal forms. Technical Report RR-2998, INRIA, Oct. 16, 1996.

[SM97] É. Sayag and M. Mauny. Structural properties of intersection types. In *Proceedings of the 8th International Conference on Logic and Computer Science – Theoretical Foundations of Computing (LIRA)*, pp. 167–175, Novi Sad, Yugoslavia, Sept. 1997.

[Urz97] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Struc. in Comp. Sc.*, 7(4):329–358, 1997.

[Wel94] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. 9th Ann. IEEE Symp. Logic in Computer Sci.*, 1994. Superseded by [Wel9X].

[Wel96] J. B. Wells. Typability is undecidable for F+eta. Tech. Rep. 96-022, Comp. Sci. Dept., Boston Univ., Mar. 1996.

[Wel9X] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure & Appl. Logic*, 199X. To appear. Supersedes [Wel94].