Direct and Expressive Type Inference for the Rank 2 Fragment of System F

by

Bradley M. Lushman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Bradley M. Lushman

Abstract

This thesis develops a semiunification-based type inference procedure for the rank 2 fragment of System F, with an emphasis on practical considerations for the adoption of such a procedure into existing programming languages. Current semiunification-based rank 2 inference procedures (notably that of Kfoury and Wells) are limited in several ways, which hinder their use in real-world settings.

First of all, the translation from an instance of the type inference problem to an instance of the semiunification problem (SUP) is indirect; in particular, because of a series of source-level transformations that take place before translation, the translation is not syntax-directed. As a result, type errors discovered during the semiunification process cannot be cleanly translated back to specific subexpressions of the source program that caused the error.

Also, because the rank 2 fragment of System F lacks a principal types property, an inference procedure cannot output a single type that encompasses all of a given term's derivable types. The procedure must therefore either rely on user assistance to produce the right type, or simply choose arbitrarily one of the given term's possible types. The algorithm of Kfoury and Wells in particular makes degenerate type assumptions in the absence of user assistance, and consequently produces types that are of no practical use.

We build up our system in stages; we begin by improving the SUP translation. Whereas termination for the Kfoury-Wells rank 2 inference procedure is assured by translating terms into instances of the acyclic semiunification problem (a decidable subset of SUP, which is undecidable in general), we formulate and target the R-acyclic semiunification problem—a larger decidable subset of SUP that facilitates a more concise translation from source terms.

We next eliminate the source-level transformation of terms, in order to formulate a truly syntax-directed translation from a source term to a set of SUP-like constraints. In doing so, we find that even the full SUP itself is not sufficiently equipped to support such a translation. We formulate USUP, a superset of SUP that incorporates a new class of identifier we call an unknown. We formulate decidable subsets of USUP, and then formulate a truly syntax-directed translation from source terms into USUP, with guaranteed termination.

Finally, to address the principal types problem, we introduce a notation for types in which we allow a particular class of variable to stand for type constructors, rather than ordinary types (an idea based on the so-called third-order λ-calculus). We show that, with third-order types we can not only output large sets of useful types for a given term, without programmer assistance, but the types we output also generalize over more System F types than any type within System F itself

can do, and still be a valid type for the source term. Thus, our system increases opportunities for separate compilation and code reuse beyond any existing system of which we are aware. Our system is sound, though incomplete in certain well-characterized ways, despite which our system performs exactly as one would hope on a variety of examples, which we illustrate in this thesis.

Acknowledgements

I would like to thank my supervisor, Gordon Cormack, for all of the hard work he has put into getting me through to the completion and defence of this thesis. It is hard to overestimate the importance of a good supervisor on the overall graduate experience. Gord has always allowed me to choose my own research topics and set my own goals. He has never once suggested that I do anything other than the research that I found interesting. On the contrary, he always seen merit in my research and has done nothing but stand firmly behind me and my work through all adversity. His support has been unwavering, especially through the rough spots, and I truly appreciate all that he has done for me.

To the students of the Programming Languages Group at Waterloo, you have all contributed to a fantastic and unique working environment. I think we have all influenced each other in various ways, and I am grateful to everyone I have met and interacted with in my time here. I am especially grateful to Richard Bilson, Stefan Büttcher, Ashif Harji, Ayelet Israeli, Kelly Itakura, Maheedhar Kolla, Ben Korvemaker, Ghulam Lashari, Tom Lynam, Adam Richard, David Yeung, and Peter Yeung.

To Prabhakar Ragde, thanks for many interesting and detailed discussions on functional programming, logic, and teaching. To all the undergrads I have taught, teaching you has been a learning experience for me, as well as hopefully for you.

To my family, thanks for providing a support system outside of school that was second to none. I am grateful to you all, and especially to my mother, Pauline Lushman, my sister, Jill Lushman, and my grandparents, Gerry and Rita Larocque. Immortalizing your names on the printed page is only the least I can do in recognition of you.

To my examining committee: Gordon Cormack, Charlie Clarke, Ondrej Lhotak, Todd Veldhuizen, and Andrew Kennedy. Thanks for taking the time to read this thesis and offer your opinions and feedback, all of which I value.

Thanks to all others who, though not explicitly named, have played a role in bringing about this work, either directly or indirectly. Finally, I acknowledge the financial support I have received over the years. The research supporting this thesis has been supported at various times by a PGS B postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada, a GO-Bell departmental scholarship, and a David R. Cheriton graduate scholarship, also from the department. Your support is greatly appreciated.

Fewer than two weeks had passed since I began my doctoral studies, when my father died unexpectedly. This thesis was defended two days before the fifth anniversary of his passing, and is dedicated to his memory.

To John Lushman (July 18, 1945–September 19, 2002), whose contributions to this thesis over the past five years may not have been directly observable, but were no less real.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Polymorphic type inference, the process of automatically deriving a (possibly polymorphic) type for an unannotated program, has been studied for decades. Arguably the biggest success story in polymorphic type inference is the inference algorithm W of Damas and Milner [6], which forms the basis for the type systems of the ML family of programming languages. Since Algorithm W, there have been many efforts to apply similar techniques to richer kinds of static analyses, and to languages with more sophisticated type systems than that of ML. Some of these have been successful; others have turned out to be instances of undecidable problems.

Notable within the second category is the problem of type inference in Girard and Reynolds' System F [13, 54] (also called the "second-order λ-calculus"). This problem remained open for several years, until it was ultimately shown undecidable by Wells [59], via a reduction from the semiunification problem, which had recently been shown undecidable as well [25]. Within System F, however, Kfoury, Tiuryn, and Urzyczyn [26] identified a subset of System F, known as the rank 2 fragment, for which type inference is decidable. Their work formed the basis for an inference algorithm for the rank 2 fragment of System F, due to Kfoury and Wells [27]. Their algorithm is based on a reduction from type inference in rank 2 System F to the acyclic semiunification problem, a decidable subset of ordinary semiunification.

The Kfoury-Wells algorithm has been the standard reference algorithm for rank 2 type inference in System F since its introduction; however, it is not known to have been adopted into any real-world programming language implementation. A major obstacle to the algorithm's adop-

tion is that it is unable to output a useful type for a given source program without programmer assistance; for a term $\lambda x.E$, because the algorithm does not know what type to assume for the parameter x, it assigns x the type $\forall\alpha.\alpha$ (also written $\bot$) and types the body E under this type assignment. However, since no term in System F actually possesses the type $\bot$, the type returned by the algorithm for $\lambda x.E$ is of no practical use. As a result, the Kfoury-Wells algorithm operates essentially as a decision procedure for the typablility problem in the rank 2 fragment of System F.

A further weakness of the Kfoury-Wells algorithm is that it takes an indirect and unintuitive path towards the solution. In particular, the translation to acyclic semiunification is only defined for a very restricted class of input programs—all input programs must first be transformed, via a complicated set of rewriting rules, so that the result belongs to this class. As a result, the term upon which we perform type inference, though it possesses an equivalent type and normal form, does not necessarily resemble the input very closely. Therefore, any feedback from the result of the type inference process, especially type error messages, is difficult to trace back to a precise location in the source program.

Moreover, the translation procedure from the source program to acyclic semiunification is itself unintuitive and overly complicated, owing to the need to comply with the requirements of acyclicity. Though descriptional complexity is not really a severe enough problem to prohibit adoption of the algorithm, it does hinder a reader's understanding of the algorithm, and may thus stand in the way of our ability to reason about, and improve upon, the type inference process.

1.2 Goal of this Thesis

The goal of this thesis is to produce a type inference system for the rank 2 fragment of System F that repairs the deficiencies of the Kfoury-Wells algorithm. In particular, our system should have the following properties:

- it should return a large (i.e., infinite) set of practically useful types for a given expression, rather than a single useless type;
- it should facilitate the production of clear and descriptive error messages whenever the algorithm fails, by providing a clear correspondence between subexpressions of the input program and components of the semiunification instance to which it maps;
- the system should support link-time specializations of the types it produces, to facilitate well-typed separate compilation and linking; and

- it should be easy to state and understand.

We develop a system in this thesis that meets all of these criteria.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of System F, semiunification, and the Kfoury-Wells algorithm, and surveys related work. Chapter 3 discusses R-acyclicity as a generalization of the acyclicity condition for semiunification problems, and shows how R-acyclicity leads to a simplification of the Kfoury-Wells translation to semiunification. Chapter 4 generalizes semiunification to include a new class of identifiers, which we call "unknowns." Using unknowns, we can provide a syntax-directed translation from type-inference to semiunification. Chapter 5 introduces constructor variables into the semiunification process, which permit us to produce generalized, real-world types for rank 2 System F programs. Chapter 6 concludes and discusses possibilities for future work.

Chapter 2

Background and Related Work

2.1 System F and the Rank 2 Fragment

System F [13, 54] is a λ-calculus equipped with a type system for expressing parametric polymorphism. Syntax and type rules are given in Figure 2.1. Discussions of System F, and of typed λ-calculi in general, usually take either a *Church-style* or a *Curry-style* approach. Under a Church-style approach, the type of a term and the types of all of its subterms are an integral component of the term itself. Hence, terms are fully annotated with type information, making type derivations a purely syntax-directed endeavour. Conversely, under a Curry-style approach, terms exist independently of their types, and it becomes the task of the programmer (or some other external entity) to associate a term with a type in such a way that the type rules are satisfied. It is the Curry-style approach that gives rise to the notion of *type inference*, which is the problem of algorithmically deducing types for a given unannotated term such that the type rules of the system are satisfied; therefore, our presentation of System F follows the Curry style.

By far the best known and most successful type inference procedure is Damas and Milner's Algorithm W [6], which forms the basis for the type systems of several well-known higher-order, typed languages, including ML [36] and Haskell [46]. In the hope of building on the success of Algorithm W, extensions to the basic algorithm have been sought to accommodate richer language features, including the kind of polymorphism and polymorphic abstractions provided by System F. The problem of type inference for Curry-style System F was open for several years. Ultimately, it was shown to be undecidable [59]. Naturally, then, we consider subsets of System F for which type inference is decidable. In particular, we restrict the set of permissible types and

$$E ::= x \mid \lambda x.E \mid E\,E$$
$$\tau ::= \alpha \mid \tau \to \tau \mid \forall\alpha.\tau$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\text{var}]$$

$$\frac{\Gamma, x : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \lambda x.E : \tau_1 \to \tau_2}\ [\text{abs}]$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1 E_2 : \tau_2}\ [\text{app}]$$

$$\frac{\Gamma \vdash E : \forall\alpha.\tau_1}{\Gamma \vdash E : \tau_1[\tau_2/\alpha]}\ [\text{spec}]$$

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash E : \forall\alpha.\tau}\ (\alpha \text{ not free in } \Gamma)\ [\text{gen}]$$

Figure 2.1: Syntax and type rules for Curry-style System F.

type derivations of the system according to their *rank* [31]:

Definition 2.1 (Rank) *For $i \in \mathbb{N}$, let $\tau(i)$ denote the set of types of rank i. Then $\tau(i)$ is given by the following:*

$$\begin{aligned} \tau(0) &::= \alpha \mid \tau(0) \to \tau(0) \\ \tau(i) &::= \forall\alpha.\tau(i) \mid \tau(i-1) \to \tau(i) \mid \tau(j) \text{ for } j < i. \end{aligned}$$

In words, the rank n types are those types in which the $\forall$ symbol may be nested to the left of at most $n-1$ $\to$ symbols. For example, $\forall\alpha.\alpha \to \alpha$ is a rank 1 type, while $(\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$ is a rank 2 type.

For any $n \in \mathbb{N}$, the rank n fragment of System F consists of those terms whose type derivations contain only types of rank n.

The rank 1 fragment of System F corresponds to the simply-typed λ-calculus, for which type inference is well understood. On the other hand, type inference is known to be undecidable for all ranks above 2, since only rank-3 types are needed for the general undecidability proof for the full System F. The rank 2 fragment is the subset of System F addressed by the Kfoury-Wells inference algorithm, and therefore admits decidable type inference. The focus of this thesis is on the rank 2 fragment of System F.

2.2 Substitution, Unification and Semiunification

The Kfoury-Wells algorithm is based on a reduction from the typability problem for rank-2 System F to the *semiunification problem* (SUP), a generalization of the ordinary unification problem, which we present below:

Definition 2.2 (Signature, Term Algebra) *A signature is a set Σ of function symbols f, each of which has an associated arity,* $\text{arity}(f)$*. A term algebra is a set $\mathbb{T}$ of terms over Σ, employing a countably infinite set $\mathbb{X}$ of variables:*

- *if $x \in \mathbb{X}$, then $x \in \mathbb{T}$;*
- *for all $n \geq 0$, if* $\text{arity}(f) = n$ *and $\tau_1, \ldots, \tau_n \in \mathbb{T}$, then $f(\tau_1, \ldots, \tau_n) \in \mathbb{T}$.*

Definition 2.3 (Substitution) *Given a term algebra $\mathbb{T}$ comprising a signature Σ of functors with associated arities, and a set $\mathbb{X}$ of variables, a* substitution *is a map $\sigma : \mathbb{X} \to \mathbb{T}$ which is an*

identity map on all but finitely many variables. The domain *of a substitution σ, denoted* $\mathrm{dom}(\sigma)$*, is the set $\{x \in \mathbb{X} \mid \sigma(x) \neq x\}$ of variables on which σ is not an identity map. The notation $[\tau/\alpha]$ denotes a substitution σ for which $\mathrm{dom}(\sigma) = \{\alpha\}$ and $\sigma(\alpha) = \tau$. Substitutions are often written postfix, so that $\alpha\sigma$ has the same meaning as $\sigma(\alpha)$. Substitutions extend naturally to maps from terms to terms by the following rule:*

$$f(\tau_1, \ldots, \tau_n)\sigma \equiv f(\tau_1\sigma, \ldots, \tau_n\sigma) \ ,$$

where f is a functor. Given terms τ and μ, μ is called a substitution instance *of τ if there exists a substitution σ such that $\tau\sigma = \mu$.*

Definition 2.4 (Unification) *Within a given term algebra, an instance of the* unification problem *is a set $\Gamma = \{\tau_i = \mu_i\}_{i=1}^N$ of term equations. A substitution σ is a solution of the instance Γ if, for all i, $\tau_i\sigma = \mu_i\sigma$.*

In words, unification is the problem of finding a substitution that equates a pair of given terms (or each pair in a set of pairs, as stated in the definition). The unification problem was first formulated and solved by Robinson [55]. Linear time solutions have since been found [34, 45]. SUP, on the other hand, works over a set of term *inequalities* $\tau_i \leq \mu_i$. For a SUP instance to be solved, it suffices that each term on the right-hand side be simply a substitution instance of the corresponding term on the left-hand side, rather than match exactly. SUP is defined formally below:

Definition 2.5 (SUP) *An instance of SUP is a set $\{\tau_i \leq \mu_i\}_{i=1}^N$ of inequalities in some term algebra. A substitution σ is a solution of SUP if there exist substitutions $\sigma_1, \ldots, \sigma_N$ such that*

$$\begin{aligned} \tau_1\sigma\sigma_1 &= \mu_1\sigma \\ &\ldots \\ \tau_N\sigma\sigma_N &= \mu_N\sigma \end{aligned}$$

In particular, semiunification differs from ordinary unification in that we may perform additional substitutions on the left-hand sides of the inequalities in order to make them match the right-hand sides. In other words, σ is a solution of the instance iff, after applying σ throughout the instance, each right-hand side is a substitution instance of the corresponding left-hand side.

In the context of type systems, the functors in the term algebra over which SUP is defined are meant to represent the type constructors of the language. For pure System F, there is a

single, right-associative, binary constructor, $\rightarrow$, for functions, which by convention is written infix. Other constructors could be introduced to denote, for example, products or disjoint sums; we study SUP over an abstract set of functors in Chapter 3, and consider quantification over functors in Chapter 5.

In order to discuss procedures that operate over SUP instances, we need a formalism for speaking about subterms within a SUP instance. For this purpose, we use *paths*, defined below:

Definition 2.6 (Path) *For a term algebra comprising a set $\mathbb{F}$ of functors, a path (denoted by Σ) is a string over the set*

$$\{f_i | f \in \mathbb{F}, 1 \leq i \leq \text{arity}(f)\}$$

that acts as a partial function on terms as follows:

$$\begin{aligned} \epsilon(\tau) &= \tau \text{ for all } \tau \\ (\Sigma f_i)(f(\tau_1, \ldots, \tau_{\text{arity}(f)})) &= \Sigma(\tau_i) \ (1 \leq i \leq \text{arity}(f)), \end{aligned}$$

where τ ranges over terms and ϵ is the empty path.

Though it was widely believed to be decidable for years, SUP is now known to be undecidable [25]. This result has formed the basis for other undecidability results within SUP's application domains [59].

Though no full solution procedure for SUP can exist, Kfoury, Tiuryn, and Urzyczyn [25] present the following solution *semi-procedure* for SUP, which we call the *redex procedure* (see Kfoury, Tiuryn, and Urzyczyn [26], Baaz [1], and Henglein [19] for alternative solution semi-procedures):

Input: set $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^N$ of term inequalities
Output: substitution σ_Γ that solves Γ

1. Set $\sigma_0 = []$ and $k = 0$; go to step 2.
2. If $\mu_i\sigma_k$ is a substitution instance of $\tau_i\sigma_k$ for all i, set $\sigma_\Gamma = \sigma_k$ and terminate with success.
3. Perform one of the following steps:
   (a) (Redex-I reduction) Let Σ be a path and $1 \leq i \leq N$ be such that $\Sigma(\mu_i\sigma_k)$ is a variable and $\Sigma(\tau_i\sigma_k)$ is not a variable. Set $\sigma_{k+1} = [\Sigma(\tau_i\sigma_k)'/\Sigma(\mu_i\sigma_k)] \circ \sigma_k$, where $\Sigma(\tau_i\sigma_k)'$ is $\Sigma(\tau_i\sigma_k)$ with all variables renamed consistently to fresh variables. Add 1 to k, and go to step 2.

(b) (Redex-II reduction) Let Σ_1 and Σ_2 be paths, α a variable, and $1 \leq i \leq N$ be such that $\Sigma_1(\tau_i\sigma_k) = \Sigma_2(\tau_i\sigma_k) = \alpha$ and $\Sigma_1(\mu_i\sigma_k) \neq \Sigma_2(\mu_i\sigma_k)$. If $\Sigma_1(\mu_i\sigma_k)$ and $\Sigma_2(\mu_i\sigma_k)$ are not unifiable, terminate with failure. Else, let θ be the most general unifier of $\Sigma_1(\mu_i\sigma_k)$ and $\Sigma_2(\mu_i\sigma_k)$, as output by Robinson's unification algorithm, and set $\sigma_{k+1} = \theta \circ \sigma_k$. Add 1 to k, and go to step 2.

(c) If neither of steps 3a and 3b is possible, then there is a functor mismatch; terminate with failure.

The redex procedure has the property that it terminates with a correct answer on all SUP instances that possess a solution, and either returns an error or loops forever on SUP instances that do not possess a solution [25].

The substitution output by the redex procedure (when it terminates) is principal (or "most general") in a specific sense, as described below [15]:

Theorem 2.1 *Let Γ be a solvable SUP instance on which the redex procedure terminates, and σ_0 the substitution returned by the redex procedure. Let σ be any other substitution that solves Γ. Then there is a substitution σ' such that, for any variable α occurring in Γ, $\alpha\sigma = \alpha\sigma_0\sigma'$. In other words, when viewed as maps with domains restricted to variables in Γ, we have $\sigma = \sigma' \circ \sigma_0$.*

Proof See Henglein [15].

This result is slightly weaker than the corresponding result for unification, in which we would have $\sigma = \sigma' \circ \sigma_0$, without qualification [55].

2.3 Acyclic Semiunification

The subset of SUP known as the *acyclic* semiunification problem (ASUP) was first presented by Kfoury, Tiuryn, and Urzyczyn [26], and is defined as follows:

Definition 2.7 (LVars, RVars) *For an inequality $\tau \leq \mu$, define*

$$\mathrm{LVars}(\tau \leq \mu) = \mathrm{Vars}(\tau)$$
$$\mathrm{RVars}(\tau \leq \mu) = \mathrm{Vars}(\mu).$$

Definition 2.8 (Acyclic) *An instance* Γ *of SUP is* acyclic *if its inequalities can be arranged into* m *columns such that the sets* $V_0, \ldots, V_m$ *defined by*

$$\begin{aligned} V_0 &= \bigcup_{v \in \text{ col. } 1} \text{LVars}(v) \\ &\cdots \\ V_k &= \left(\bigcup_{v \in \text{ col. } k-1} \text{RVars}(v) \right) \bigcup \left(\bigcup_{v \in \text{ col. } k} \text{LVars}(v) \right) \\ &\cdots \\ V_m &= \bigcup_{v \in \text{ col. } m} \text{RVars}(v) \end{aligned}$$

are pairwise disjoint.

For example, the instance $\{\alpha \leq f(\beta),\ \beta \leq g(\gamma, \gamma),\ g(\alpha, \delta) \leq \epsilon\}$, where f is a unary functor and g is a binary functor, is acyclic—if we assign the first and third inequalities to column 1, and the second to column 2, then we have $V_0 = \{\alpha, \delta\}$, $V_1 = \{\beta, \epsilon\}$, and $V_2 = \{\gamma\}$, and these are pairwise disjoint.

Definition 2.9 (ASUP) *ASUP is the restriction of SUP to acyclic problem instances.*

A key property of ASUP is that the redex procedure is guaranteed to terminate on all instances of ASUP; that is, for ASUP, the redex procedure is a full solution procedure [27].

2.4 Algorithm KW

We present in this section a description of the rank 2 inference algorithm of Kfoury and Wells, which we refer to throughout this thesis as Algorithm KW. The algorithm proceeds in five steps:

1. Abstraction labelling
2. θ-reduction
3. Translation to ASUP
4. ASUP solution
5. Type recovery

We describe these steps in detail below.

2.4.1 Abstraction Labelling

We assume that all bound variables in the input program have distinct names from each other, and from all free variables. If this is not the case, we rename the bound variables such that this condition holds.

We begin by labelling each λ-abstraction in the program with either 1, 2, or 3, according to the following rules:

- We label with 1 every abstraction that is accompanied by an argument.
- We label with 3 every abstraction that is not accompanied by an argument, but appears as a subexpression of some function argument.
- We label all other abstractions with 2.

For example, the program $\lambda x.x((\lambda y.\lambda z.z)x)$ would be labelled as $\lambda^2 x.x((\lambda^1 y.\lambda^3 z.z)x)$. More precisely, a λ-term is labelled according to the following two definitions (which are taken from Kfoury and Wells [27]):

Definition 2.10 (Active Abstractions) *For a λ-term M, denote by* $\mathrm{act}(M)$ *the sequence of* active abstractions *in M:*

$$\begin{aligned} \mathrm{act}(x) &= \epsilon \text{ (the empty sequence)} \\ \mathrm{act}(\lambda x.M) &= x, \mathrm{act}(M) \\ \mathrm{act}(MN) &= \begin{cases} \epsilon & \text{if } \mathrm{act}(M) = \epsilon \\ x_2, \ldots, x_n & \text{if } \mathrm{act}(M) = x_1, \ldots, x_n \text{ for } n \geq 1 \end{cases} . \end{aligned}$$

Definition 2.11 (Abstraction Labelling) *Given a λ-term M, we label the λ-abstractions in M with 1, 2, or 3, as described above, by evaluating*

$$\mathit{label}(M, \mathrm{act}(M), 2) ,$$

where the function label is defined as follows:

$$\begin{aligned} \mathit{label}(x, X, i) &= x \\ \mathit{label}(\lambda x.M, X, i) &= \begin{cases} \lambda^i x.\mathit{label}(M, X, i) & \text{if } x \in X \\ \lambda^1 x.\mathit{label}(M, X, i) & \text{if } x \notin X \end{cases} \\ \mathit{label}(MN, X, i) &= \mathit{label}(M, X, i)\mathit{label}(N, \mathrm{act}(N), 3) . \end{aligned}$$

The purpose of abstraction labelling is to distinguish those abstractions that are permitted to demand a polymorphic argument from those that are not. The polymorphic abstractions are precisely the abstractions labelled with 1 or 2.

2.4.2 θ-Reduction

The labelled λ-term undergoes a source-level transformation known as *θ-reduction*, which consists of applying four reduction rules, as outlined below, until no reduction opportunities remain:

- (θ_1) $((\lambda^1 y.N)P)Q \to (\lambda^1 y.NQ)P$
- (θ_2) $\lambda^3 z.(\lambda^1 y.N)P \to (\lambda^1 v.\lambda^3 z.(N'))(\lambda^3 w.(P'))$, where $N' = N[vz/y]$, $P' = P[w/z]$, and v and w are fresh variables
- (θ_3) $N((\lambda^1 y.P)Q) \to (\lambda^1 y.NP)Q$
- (θ_4) $(\lambda^1 y.(\lambda^2 x.N))P \to \lambda^2 x.((\lambda^1 y.N)P)$

Rule θ_1 ensures that a λ^1-abstraction is applied to no more than a single argument. Intuitively, since the expression $(\lambda^1 y.N)PQ$ reduces to $N[P/y]Q$, which (by uniqueness of variable names) is equivalent to $NQ[P/y]$, and since $NQ[P/y]$ is the result of reducing $(\lambda^1 y.NQ)P$, the reduction preserves meaning.

Rule θ_3 ensures that a redex does not occur in the argument position of a function application. The expression $N((\lambda^1 y.P)Q)$ reduces to $N(P[Q/y])$, which is equivalent to $(NP)[Q/y]$, which expands to $(\lambda^1 y.NP)Q$. Hence, it preserves the meaning of the term.

Rule θ_4 ensures that λ^2-abstractions (i.e., unmatched polymorphic abstractions) outscope λ^1-abstractions. As above, the equivalence of the expressions before and after the reduction is straightforward.

In contrast to rule θ_4, rule θ_2 ensures that no λ^3-abstraction can outscope a λ^1-abstraction. The equivalence between the two expressions is more difficult to establish here. The expression $\lambda^3 z.(\lambda^1 y.N)P$ reduces to $\lambda^3 z.N[P/y]$. The expression $(\lambda^1 v.\lambda^3 z.N')(\lambda^3 w.P')$ reduces to $\lambda^3 z.N'[(\lambda^3 w.P')/v]$, which is equal to $\lambda^3 z.N[vz/y][(\lambda^3 w.P[w/z])/v]$. This expression, in turn, is equal to $\lambda^3 z.N[(\lambda^3 w.P[w/z])z/y]$, which is $\lambda^3 z.N[P/y]$, thus establishing the equivalence. As we observe in detail in Chapter 4, by the introduction of an additional λ^3-abstraction and additional variables, applications of rule θ_2 (even moreso than the other rules) mean that there is no one-to-one mapping between subexpressions of a given source term and particular subsets of the corresponding SUP instance.

Another way of viewing the θ-rules is by rephrasing them as operations on let-expressions, rather than on function applications:

- (θ_1) (let $y = P$ in N)$Q \rightarrow$ let $y = P$ in NQ
- (θ_2) $\lambda^3 z.$(let $y = P$ in N) $\rightarrow$ let $v = \lambda^3 w.P[w/z]$ in $\lambda^3 z.N[vz/y]$, where v and w are fresh variables
- (θ_3) N(let $y = Q$ in P) $\rightarrow$ let $y = Q$ in NP
- (θ_4) let $y = P$ in $\lambda^2 x.N \rightarrow \lambda^2 x.$let $y = P$ in N.

Here, we simply make use of the identity $(\lambda^1 y.N)P \equiv$ let $y = P$ in N. This presentation of the θ-rules may help to clarify for the reader the meaning behind them; However, we will use the original formulation of θ-reduction in the remainder of this thesis, as it is more convenient for our purposes.

At the end of θ-reduction, the term will have the following form (called a *θ-normal form*):

$$\lambda^2 x_1.\cdots\lambda^2 .x_m.(\lambda^1 y_1.(\cdots((\lambda^1 y_n.E_{n+1})E_n)\cdots)E_2)E_1,$$

with free variables $w_1, \ldots, w_p$, in which each E_i contains no λ^1- or λ^2-abstractions. Each λ^3-abstraction in subexpression E_i will bind a λ^3-variable $z_{i,j}$.

2.4.3 Translation to ASUP

A labelled λ-expression in θ-normal form is translated into an instance of ASUP with the following variables:

- for each x_j, the variable $\beta^x_{i,j}$ represents its type in subexpression E_i;
- for each y_j, the variable $\beta^y_{i,j}$ represents its type in subexpression E_i, for $i > j$;
- for each w_j, the variable $\beta^w_{i,j}$ represents its type in subexpression E_i;
- for each $z_{i,j}$, the variable $\gamma_{i,j}$ represents its type;
- for each subexpression M of each E_i, the variable δ_M represents its derived type (different occurrences of the same subexpression are assigned different δ-variables[1]).

[1]In the examples worked in the remainder of this thesis, when we need to disambiguate δ-variables for subexpressions that occur more than once, we shall add numeric subscripts to both the δ-variables, indicating which occurrence of the subexpression is meant. For example, in the term $\lambda^2 x.xx$, we use δ_{x_1} to refer to the first x in the body of the abstraction, and δ_{x_2} to refer to the second.

Then, for each subexpression M of each E_i, the ASUP instance contains the following inequalities:

- if $M = x_j$, the inequality $\beta^x_{i,j} \leq \delta_M$;
- if $M = y_j$, the inequality $\beta^y_{i,j} \leq \delta_M$;
- if $M = w_j$, the inequality $\beta^w_{i,j} \leq \delta_M$;
- if $M = z_{i,j}$, the inequality $\gamma_{i,j} = \delta_M$;
- if $M = \lambda^3 z_{i,j}.E$, the equality $\delta_M = \gamma_{i,j} \to \delta_E$;
- if $M = PQ$, the equality $\delta_P = \delta_Q \to \delta_M$,

where an equality $\tau = \mu$ is syntactic sugar for the inequality $\alpha \to \alpha \leq \tau \to \mu$, where α is a fresh variable. In addition, the instance contains the following inequalities:

- for each x_j, $1 \leq i \leq n$, the inequality $\beta^x_{i,j} \leq \beta^x_{i+1,j}$;
- for each y_j, $j < i \leq n$, the inequality $\beta^y_{i,j} \leq \beta^y_{i+1,j}$;
- for each w_j, $1 \leq i \leq n$, the inequality $\beta^w_{i,j} \leq \beta^w_{i+1,j}$.

Each redex $(\lambda^1 y_i.E_{i+1})E_i$ contributes the equality $\beta^y_{i,i+1} = \delta_{E_i}$. Finally, for the free variables w_j and the λ^2-variables x_j, we consult a type environment A (which may contain at most rank 1 bindings for these variables). If $A(w_j)$ yields a type τ, we add the equality $\beta^w_{1,j} = \tau$, and similarly for x_j; otherwise we add no new inequalities.

2.4.4 ASUP Solution and Type Recovery

The SUP instance generated from the labelled, θ-normal λ-expression in the previous step is guaranteed to belong to ASUP, and termination is therefore guaranteed for the redex procedure on this problem instance. The final steps of the algorithm consist of solving the ASUP instance and translating the solution back to a type for the entire expression, or a signal that none exists.

If the redex procedure fails, then we answer that the expression has no type; otherwise, let σ be the accumulated substitution performed by the redex procedure (i.e., the solution of the ASUP instance). Then, for a given type environment A, and for each λ^2-variable x_j, let

$$\tau_j = \begin{cases} \forall.A(x_j) & \text{if } A(x_j) \text{ exists,} \\ \bot & \text{otherwise} \end{cases},$$

where $\perp$ is shorthand for the degenerate type $\forall\alpha.\alpha$, and the notation $\forall.\tau$ denotes quantification over the free variables of τ. Then the type we return is given by

$$\forall.(\tau_1 \to \cdots \to \tau_m \to \delta_{E_{n+1}}\sigma) \ .$$

2.5 Worked Example

We illustrate the operation of Algorithm KW on the source term

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

After abstraction labelling, the term becomes

$$\lambda^2 x.(\lambda^1 v.v)(\lambda^3 z.(\lambda^1 y.y)z) \ .$$

After one step of θ_2 reduction, we obtain

$$\lambda^2 x.(\lambda^1 v.v)((\lambda^1 t.\lambda^3 u.tu)(\lambda^3 w.w)) \ .$$

A step of θ_3 reduction then yields

$$\lambda^2 x.((\lambda^1 t.(\lambda^1 v.v)\lambda^3 u.tu)(\lambda^3 w.w)) \ .$$

The term is now θ-normal, with $E_1 = \lambda^3 w.w$, $E_2 = \lambda^3 u.tu$, $E_3 = v$, $y_1 = t$, and $y_2 = v$. From this expression, we obtain the following inequalities:

$$\begin{aligned}
\delta_{\lambda^3 w.w} &= \gamma_w \to \delta_w \\
\gamma_w &= \delta_w \\
\delta_{\lambda^3 u.tu} &= \gamma_u \to \delta_{tu} \\
\delta_t &= \delta_u \to \delta_{tu} \\
\gamma_u &= \delta_u \\
\beta^y_{1,2} &\leq \delta_t \\
\beta^y_{2,3} &\leq \delta_v \\
\beta^1_{1,2} &= \delta_{\lambda^3 w.w} \\
\beta^1_{2,3} &= \delta_{\lambda^3 u.tu} \\
\beta^y_{1,2} &\leq \beta^y_{2,3} \\
\beta^x_{1,1} &\leq \beta^x_{1,2} \\
\beta^x_{1,2} &\leq \beta^x_{1,3}
\end{aligned}$$

To solve these inequalities, we begin by subsituting out all of the equalities (which is equivalent to redex-II reduction):

$$\begin{aligned}
\gamma_w \to \gamma_w &\leq \gamma_u \to \delta_{tu} \\
\gamma_u \to \delta_{tu} &\leq \delta_v \\
\gamma_w \to \gamma_w &\leq \beta^y_{2,3} \\
\beta^x_{1,1} &\leq \beta^x_{1,2} \\
\beta^x_{1,2} &\leq \beta^x_{1,3}
\end{aligned}$$

We now perform redex-II reduction on the first inequality, and redex-I reductions on the second and third inequalities:

$$\begin{aligned}
\gamma_w \to \gamma_w &\leq \gamma_u \to \gamma_u \\
\gamma_u \to \gamma_u &\leq \alpha \to \alpha \\
\gamma_w \to \gamma_w &\leq \beta \to \beta \\
\beta^x_{1,1} &\leq \beta^x_{1,2} \\
\beta^x_{1,2} &\leq \beta^x_{1,3}
\end{aligned}$$

The redex procedure now terminates, and we interpret the result type as the image of δ_{E_3}, i.e., δ_v. The image of this variable is the expression $\alpha \to \alpha$, and we obtain the type

$$\forall\alpha.\bot \to \alpha \to \alpha \ ,$$

where the $\bot$ arises from the assumed type of the unannotated λ^2-parameter x.

2.6 Notes on Algorithm KW

Algorithm KW has several characteristics worth noting, which we outline here. First, it relies on an environment A to supply types for the λ^2-variables x_j. These type assignments for the x_j variables would have been placed into the environment by user annotations of the original program code. Thus, KW relies on user annotations in producing a type for the expression. In the absence of any annotations, each x_j is assigned the type $\bot$, and the resulting type becomes

$$\forall.(\bot \to \cdots \to \bot \to \delta_{E_{n+1}}\sigma) \ .$$

Since no term in System F actually has type $\perp$, the parameter types in this type expression do not match the types of any real-world terms; hence the type we obtain, though correct, has no practical use.

Also worth noting is that the translation procedure from typability to ASUP has some unintuitive characteristics. Each x_j, y_j and w_j is represented by not one, but up to n distinct variables $\beta^x_{i,j}$, $\beta^y_{i,j}$, and $\beta^w_{i,j}$. These are then related by chains of inequalities:

$$\beta^x_{1,j} \leq \beta^x_{2,j} \leq \cdots \leq \beta^x_{n+1,j} \ ,$$

and so on. It is not at all clear why (as implied by these chains), for example, a parameter x_j should have a more specialized type in E_{i+1} than in E_i for each i.

In fact, the reason why these chains exist is to ensure that the translation to SUP actually results in an instance of ASUP. Each x_j, y_j, and w_j requires a separate type variable in each E_i because the ASUP column assignment groups inequalities according to the expression E_i with which they are associated—hence if a variable occurs in more than E_i, its type must be represented in each E_i by a different variable.

The necessity of introducing so many additional variables and inequalities into the instance, just to ensure conformity with the ASUP restriction, suggests that ASUP is not an appropriate model into which to map instances of rank-2 type inference—this is the subject of the next chapter, in which we present a more intuitive subset of SUP for which termination of the redex procedure is still guaranteed. Under the translation to this new subset, which we call the *R-acyclic semiunification problem* (R-ASUP), additional variables for each x_j, y_j, and w_j (and consequently also the chains linking them) are no longer needed.

A final noteworthy characteristic of KW is the extreme nature of the source-level transformations brought about by the θ-reduction step. Every expression maps under θ-reduction to a θ-normal form, which has a very regular structure, as illustrated in Section 2.4.2. A term's θ-normal form can be quite different from the original term itself; a sequence of θ-reduction steps can produce a result that not only has a very different structure from the original term, but may also include variables not present in the original term. Consequently, a θ-normal form might not be recognizable as being the reduced form of a given λ-term.

A consequence of the effects of θ-reduction is that it becomes difficult to translate analyses on a term's ASUP instance back into statements about the original term. For example, if the ASUP instance has no solution, the system should report a type error to the programmer—but tracing the error back through the θ-reduction to find the point in the original program at which

the error occurs is difficult, particularly if the error involves variables that were created during θ_2-reduction, and therefore do not appear in the original term. Furthermore, θ-reduction prevents us from performing any kind of analysis on a program's subexpressions, because a term and its θ-normal form do not necessarily share the same set of subexpressions. Finally, it would be difficult, after θ-reduction, to use the solution of the ASUP instance to produce a fully typed version of the original source term (i.e., one with each subexpression's type indicated). Instead, we would prefer a more direct type inference algorithm, in which θ-reduction is not necessary. We discuss the innovations needed for such a system in Chapter 4.

2.7 Related Work

As the results in this thesis relate to both semiunification and polymorphic type inference, we survey separately related results from each field.

2.7.1 Semiunification

Over the years, there have been several unsuccessful attempts to give a full solution procedure for SUP, many of which have yielded decidable subsets of varying complexity. We outline some of these here.

Henglein [15, 16, 17] did pioneering work on semiunification, establishing links between SUP and the type systems of languages like the Milner-Mycroft calculus [39]. As part of his work, he provided a solution procedure for the *linear* semiunification problem, in which all functors have arity one, and conjectured general solvability. Under the assumption that all functors are unary, an instance can have no redex-II's, and therefore ordinary graph acyclicity (rather than R-acyclicity) is sufficient to guarantee termination of the redex procedure.

Baaz [1] gave a semi-procedure for general SUP problems that is based on a reduction to ordinary unification. Baaz's algorithm is less direct than the redex procedure, relying on variable renamings and appeals to unification, rather than performing any explicit term substitutions. A study of the behaviour of Baaz's algorithm on instances of R-ASUP is beyond the scope of this work.

Kapur et al [23] showed that SUP is decidable in polynomial time when restricted to instances containing a single inequality (this is called *uniform* semiunification). Oliart and Snyder [43] give a solution procedure that runs in $O(n^2\alpha(n)^2)$ time in general, and $O(n^2 \log^2(n\alpha(n))\alpha(n)^2)$ time if principal unifiers are required, where α is the inverse Ackermann function. In the case of a

single inequality, the only possible non-zero path in the instance's graph is a self-loop, which only arises if a variable occurs on both sides of the lone inequality. Since a self-loop is a cycle, both ASUP and R-ASUP prohibit this possibility. SUP is known to be undecidable as soon as the number of inequalities in the instance is at least 2 [23, 25, 52].

Left-linear semiunification restricts the problem instance such that within each left-hand side, no variable occurs more than once. Left-linear semiunification was introduced and shown decidable by Kfoury, Tiuryn, and Urzyczyn [24]. Henglein [18] gives a cubic time solution procedure. A left-linear instance cannot contain redex-II's, and therefore, as with linear semiunification, ordinary graph acylicity suffices to guarantee termination.

Leiß [30] showed that semiunification is decidable when restricted to two variables. Strictly speaking, this subset, like the others presented in this section, is neither a superset nor a subset of R-ASUP; nevertheless, it seems clear that R-ASUP is the largest and most significant decidable subset of SUP among all of these.

The use of SUP as a vehicle for studying type inference is an instance of the more general idea of constraint-based typing, in which a translation is given from a type inference problem to a set of constraints, and then a procedure given to solve the constraints. Early work on constraint-based typing was done by Mitchell [37], and Fuh and Mishra [11]. Pottier and Rémy [51] describe HM(X), a parameterized, unification-based constraint system for ML-style type systems, first studied by Odersky, Sulzmann, and Wehr [41], and later studied further by several authors, including Sulzmann [58] and Pottier [50]. Compared to semiunification, the constraint system of HM(X) appears considerably more complicated. Pottier and Rémy note that semiunification-based constraint systems have not, to their knowledge, been adopted in any real programming language implementation[2]. The results we present in the remainder of this thesis may serve to make such an adoption more attractive.

2.7.2 Polymorphic Type Inference

As full type inference for System F is impossible [59], research in the field concentrates on finding approximate solutions, either by restricting the language (as with the rank-2 fragment of System F), or by enhancing the type system, or by introducing a scheme of programmer-supplied type annotations to help the inference engine along.

[2]Pottier and Rémy do mention, however, an experimental extension of SML/NJ by Emms and Leiß [9] that incorporates polymorphic recursion, and whose theory is based on semiunification.

Prominent type constructions that admit type inference over a wider class of terms include recursive types [38] and intersection types [5]. Under recursive types a fixed-point operator μ is added to the type system, so that the expression $\mu\alpha.f(\alpha)$ represents a type τ that satisfies $\tau = f(\tau)$. Under this system, we can define types that cannot otherwise be expressed in finite terms. For example, the type $\mu\alpha.\texttt{int} \times \alpha$ represents an infinite list of integers. By using suitably-defined μ-types, we can perform type inference on a larger class of expressions than simply the rank 2-typable expressions. On the other hand, the type expression $\mu\alpha.\alpha \to \alpha$ defines a type τ satisfying the equivalence $\tau = \tau \to \tau$. Using this construction we can type any term in the untyped λ-calculus, and all terms will have type τ. Hence, type inference becomes a useless exercise. It seems, therefore, that μ-types must be used judiciously in order to strike a balance between being able to type a large number of expressions, and being able to extract useful information from those expressions.

Intersection types arise from imposing a subtyping discipline on a type system. In the context of System F, we say that τ is a subtype of μ if μ is a substitution instance of τ. An intersection type of τ_1 and τ_2, then, written $\tau_1 \wedge \tau_2$ is defined as the largest type in the subtype hierarchy that is a subtype of both τ_1 and τ_2. Used in positive contexts, intersection types are useful for expressing finitary overloading. Used in negative contexts, they represent a *demand* for finitary overloading. By using an intersection type in a negative context, we can give a type to a function that uses its parameter in different contexts, so that the parameter must have multiple types in order for the function to be typable. For example, the rank-2 System F function $\lambda x.xx$ can be given the type $\forall\alpha.\forall\beta.(\alpha \wedge (\alpha \to \beta)) \to \beta$. System F augmented with intersection types has a principal types property—for every term there is a "most general" type of which all other types are instances. For this reason, Jim [21] uses rank-2 intersection types to study type inference in System F, augmented with a facility for recursive function definitions. The typable terms in System F with intersection types are precisely the strongly normalizing terms. Hence, type inference with intersection types is undecidable. On the other hand, rank n intersection types find principal types for the rank n fragment of System F, for all n, via a terminating procedure [28]. Since rank n type inference is undecidable in ordinary System F for $n > 3$, there can be no Turing-computable mapping between rank n intersection types and rank n System F types.

Approaches to type inference within System F that are not based on enriching the type system are typically based on aiding the type inference process via type annotations supplied by the programmer at particular points in the program. Type checking is a trivial exercise for fully-annotated (i.e., Church-style) terms. On the other hand, Wells' result on the undecidability

of type-checking for System F [59] shows that it is insufficient to simply supply the final type and verify that the type is derivable for the term. Further, Boehm showed [3] that the general problem of type inference for partially annotated System F terms is undecidable. Hence, systems for partial type inference must be conservative in terms of how much annotation can be omitted.

Odersky and Läufer [40] present a type system in which expressions can be annotated with types of arbitrary rank, but for which full inference is possible at rank 1. However, a type variable cannot be instantiated with a quantified (i.e., rank 1 or higher) type. For example, $\forall\alpha.\alpha \to \alpha$ cannot be instantiated to $(\forall\beta.\beta \to \beta) \to (\forall\beta.\beta \to \beta)$. This shortcoming is alleviated by providing a type declaration facility, which is much like ML's `datatype` construction, except that the declared type may be built from polymorphic (i.e., quantified) components. Odersky and Läufer then give a translation from the full System F to their language. The encoding critically depends on the ability to declare types, this being the facility by which polymorphic instantiations may occur. As a result, though some annotation effort is saved at rank 1, higher rank constructions seem to require considerable annotation effort, due to the need to explicitly declare types.

Pierce and Turner [48] observe that even though a large amount of type information must be supplied in order for type inference to be decidable, many of the annotations in a fully-annotated program are both "common" and "silly" (notions they define). They propose a system in which at least these annotations may be elided, based on "local synthesis of type arguments" and "bidirectional propagation of type information." Odersky, Zenger, and Zenger [42] generalize the idea of local type inference by allowing partial type information to be propagated to a local constraint solver; types are "coloured" to indicate the direction of propagation. This approach allows them to type some terms that are untypable in Pierce and Turner's system. The bidirectional approach of Pierce and Turner also forms the basis for later work [8] in which support for union, intersection, and dependent types is added via an additional third pass, yielding a so-called "tri-directional" approach.

The systems of Odersky and Läufer, and Pierce and Turner form the basis for the higher-rank type inference facility provided by the Glasgow Haskell Compiler (GHC), as described by Peyton Jones and Shields [47]. They claim that the annotation burden in GHC is more modest than in Odersky and Läufer's system, that required and optional annotations are precisely specified, and that the system as a whole is relatively simple.

Jones [22] presents a higher-order extension to ML, in which type variables produced during ordinary first-order type inference may be specialized to any type in System F_ω. This system is

used as the basis for a module system for ML with first-class structures. However, it cannot infer polymorphic abstractions, or other higher-rank constructions, without programmer assistance.

Le Botlan and Rémy's ML^F [29] is a conservative extension of the ML type system that supports arbitrary-rank types via user-supplied annotations. Every ML^F program has a principal type; however the principality is with respect to these annotations. KW possesses this same notion of principality (i.e., modulo user-supplied parameter types, the result type is principal), and our system does as well. Like Kfoury and Wells, Le Botlan and Rémy cite co-/contravariance problems as primary reasons why principal types cannot be achieved without annotations; it is precisely these problems that we address in Chapter 5 of this thesis. One weakness of ML^F is the complexity of its type system and inference algorithm, which have been described elsewhere as "pretty complicated" [47]. By contrast, the types we output in Chapter 5 are templates for ordinary System F types (subject to semiunifiability constraints), and may thus be easier for programmers to read and write than those of ML^F.

Chapter 3

R-Acyclicity

In this chapter, we examine SUP from a graph-theoretic perspective. We give a graph-theoretic characterization of the acyclicity condition that defines ASUP, and then present R-ASUP [33], a more general decidable subset of SUP. Finally, we show how translating the rank-2 typability problem into R-ASUP rather than ASUP saves a quadratic number of instance variables.

3.1 Graph-Theoretic Characterization of ASUP

Apart from its connections to polymorphic type inference, SUP is an interesting problem, itself worthy of study. Applications of SUP can be found in fields such as logic programming [4], computational linguistics [7], and program analysis [2, 10]; for this reason, we present the results in this chapter in an application-independent way, so that they may be readily adopted to other application domains.

To begin our development, we define some concepts related to paths and path lengths in directed graphs.

Definition 3.1 (Undirected path) *Given a directed graph G and vertices v_1 and v_2 in G, an* undirected path *from v_1 to v_2 is a path from v_1 to v_2, in which we are not required to follow the direction of the edges.*

In other words, an undirected path is one that exists when we pretend that the underlying directed graph is undirected. Ordinary (directed) paths may also be considered undirected. There are two notions of path length associated with undirected paths:

Definition 3.2 (Unsigned, signed path length) *Given a directed graph G, with an undirected path π joining vertices v_1 and v_2, the* unsigned path length *of π, denoted $||\pi||$, is the number of edges in π. The* signed path length *of π, denoted $|\pi|$, is the length of π, where each forward arrow (i.e., pointing away from v_1 and towards v_2) counts for $+1$, and each reverse arrow (i.e., pointing towards v_1 and away from v_2) counts for -1.*

The difference between unsigned and signed path length is analogous to the distinction between distance and displacement in physics.

We also introduce the following notation:

Definition 3.3 *For a directed graph G containing vertices v_1 and v_2,we write $v_1 \to v_2$ (resp. $v_1 \to_U v_2$) if there is a directed (resp. undirected) edge from v_1 to v_2. We write $v_1 \to^* v_2$ (resp. $v_1 \to_U^* v_2$) if there is a directed (resp. undirected) path from v_1 to v_2. We write $v_1 \to^+ v_2$ (resp. $v_1 \to_U^+ v_2$) if there is a directed (resp. undirected) path of nonzero length from v_1 to v_2. Finally, we write $\pi : v_1 \to^* v_2$ (and analogously for the other cases) to indicate that π is a directed path from v_1 to v_2.*

We define the *graph* of a SUP instance as follows:

Definition 3.4 (Graph of a SUP instance) *Let $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^N$ be an instance of SUP. Then the* graph of Γ*, denoted $G(\Gamma)$, is defined as follows:*

- *the inequalities $\tau_i \leq \mu_i$ are the vertices v_i in G;*
- *$v_i \to v_j$ iff $\mathrm{RVars}(v_i) \cap \mathrm{LVars}(v_j) \neq \emptyset$*

For example, suppose we have the following SUP instance Γ:

$$\alpha \leq f(\beta, \gamma) \qquad \beta \leq \delta \qquad \gamma \leq \epsilon \qquad \eta \leq \delta \qquad \zeta \leq f(\eta, \gamma) \qquad g(\delta) \leq \theta \ ,$$

where f is a binary functor and g is a unary functor. The graph $G(\Gamma)$ of the instance is given in Figure 3.1. Let the inequalities in the instance be labelled as vertices v_1, v_2, v_3, v_4, v_5, and v_6, respectively. Then we see that there are directed paths from v_1 to v_3 and v_6, and also from v_5 to v_3 and v_6. On the other hand, there are two *undirected* paths from v_3 to v_6: one going through v_1 and v_2, and the other going through v_5 and v_4. Both have unsigned length equal to 3, and signed length equal to 1.

The following theorem establishes graph-theoretic criteria that are necessary and sufficient for a SUP instance to be an instance of ASUP:

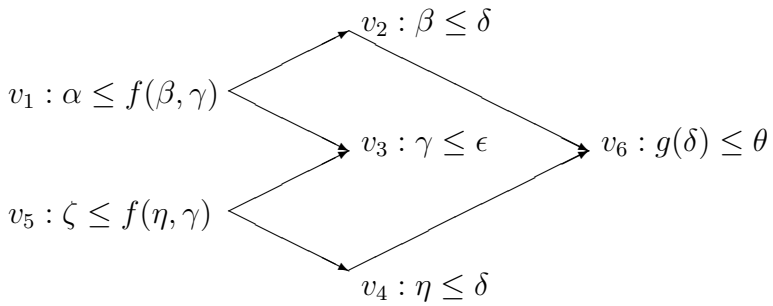

Figure 3.1: The graph of a SUP instance. The symbols f and g denote, respectively, a binary functor and a unary functor.

Theorem 3.1 *Let $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^N$ be an instance of SUP. Then Γ is an acyclic instance of SUP iff the following four symmetric conditions hold for $G(\Gamma)$:*

- *for any given variables α_1 and α_2, all paths $\pi : v_1 \rightarrow_U^* v_2$, such that $\alpha_1 \in \mathrm{LVars}(v_1)$ and $\alpha_2 \in \mathrm{LVars}(v_2)$, have the same signed length.*
- *for any given variables α_1 and α_2, all paths $\pi : v_1 \rightarrow_U^* v_2$, such that $\alpha_1 \in \mathrm{LVars}(v_1)$ and $\alpha_2 \in \mathrm{RVars}(v_2)$, have the same signed length.*
- *for any given variables α_1 and α_2, all paths $\pi : v_1 \rightarrow_U^* v_2$, such that $\alpha_1 \in \mathrm{RVars}(v_1)$ and $\alpha_2 \in \mathrm{LVars}(v_2)$, have the same signed length.*
- *for any given variables α_1 and α_2, all paths $\pi : v_1 \rightarrow_U^* v_2$, such that $\alpha_1 \in \mathrm{RVars}(v_1)$ and $\alpha_2 \in \mathrm{RVars}(v_2)$, have the same signed length.*

Note that our example SUP instance Γ, whose graph is presented in Figure 3.1, satisfies the conditions of Theorem 3.1, as Γ is indeed an instance of ASUP.

We begin with a brief outline of the proof:

Proof Sketch The forward direction is a proof by induction on $||\pi||$ that $|\pi|$ measures the difference in the column assignments of two inequalities. The reverse direction assigns columns to inequalities according to the constraints provided by the structure of the graph—namely that an inequality must have a column number one less than its successors in the graph and one more than its predecessors. We then show that this procedure and the conditions of the theorem together guarantee the disjointness of the resulting sets of variables. □

The full proof is as follows:

Proof of Theorem 3.1 We begin with the forward direction. Suppose Γ is an acyclic instance of SUP. Then there is an arrangement of the inequalities in Γ into m columns such that the following sets:

$$\begin{aligned} V_0 &= \bigcup_{v \in \text{col. } 1} \text{LVars}(v) \\ V_i &= \left(\bigcup_{v \in \text{col. } i} \text{RVars}(v) \right) \bigcup \left(\bigcup_{v \in \text{col. } i+1} \text{LVars}(v) \right) \\ V_m &= \bigcup_{v \in \text{col. } m} \text{RVars}(v) \end{aligned}$$

are pairwise disjoint. Now, consider any edge from $\mu_i \leq \tau_i$ to $\mu_j \leq \tau_j$ in $G(\Gamma)$. Then τ_i and μ_j share at least one variable in common. Thus, by the disjointness of the V_i's $\mu_i \leq \tau_i$ and $\mu_j \leq \tau_j$ must be in adjacent columns, say $\mu_i \leq \tau_i$ is in column k and $\mu_j \leq \tau_j$ is in column $k+1$ (so that the variables in τ_i and μ_j are in the set V_k). Hence the edge points from an inequality in column k to one in column $k+1$. (Since all edges point from a given column to the one immediately following it, it follows immediately that $G(\Gamma)$ is acyclic.) Now, let $\mu_1 \leq \tau_1$ and $\mu_2 \leq \tau_2$ be inequalities in Γ, i.e., vertices in $G(\Gamma)$, in columns k_1 and k_2, respectively, and suppose $\pi : \mu_1 \leq \tau_1 \rightarrow_U^* \mu_2 \leq \tau_2$ in $G(\Gamma)$. We prove by induction on $||\pi||$ that $|\pi| = k_2 - k_1$. If $||\pi|| = 0$, then the two vertices coincide, and the result is immediate. Otherwise, we can decompose π into a directed (though possibly reversed) path π_1 followed by an undirected path π_2 via a vertex $\mu_3 \leq \tau_3$, in column k_3. Since every edge joins consecutive columns, $|\pi_1|$ must be precisely $k_3 - k_1$. By induction, we claim that $|\pi_2| = k_2 - k_3$. Thus $|\pi| = k_2 - k_1$, independently of our choice of path. Hence, for any pair of vertices in $G(\Gamma)$, all undirected paths joining them have the same signed length. (Note that this implies that all *directed* paths joining two given vertices also have the same length.) Now, choose $i, j \in \{1, 2\}$. By the pairwise disjointness of $V_0, \ldots, V_m$, all inequalities $\tau_{11} \leq \tau_{12}$ such that $\alpha_1 \in \text{Vars}(\tau_{1i})$ are in some column k, and all inequalities $\tau_{21} \leq \tau_{22}$ such that $\alpha_2 \in \text{Vars}(\tau_{2j})$ are in some column k'. Hence all undirected paths joining such vertices must be of signed length precisely $k - k'$. This establishes the forward direction.

For the reverse direction, we begin with an acyclic digraph $G(\Gamma)$ satisfying our hypotheses, and arrange the inequalities into columns as follows:

- for each connected component of $G(\Gamma)$:

  - label any vertex v with any integer c
  - while there are unlabelled vertices:
    * choose a labelled vertex w, with label l_w
    * for all unlabelled vertices w' such that $w \to w'$, label w' with label $l_w + 1$
    * for all unlabelled vertices w' such that $w' \to w$, label w' with label $l_w - 1$

- while possible:
  - let G_1 and G_2 be connected components of $G(\Gamma)$, such that there are vertices $v_1 \in G_1$, $v_2 \in G_2$, with respective labels l_1 and l_2, such that $\mathrm{Vars}(v_1) \cap \mathrm{Vars}(v_2) \neq \emptyset$
  - subtract $l_2 - l_1$ from all labels in G_2
  - create a new vertex v_3, with no variables and label $l_1 + 1$, and edges from v_1 to v_3, and from v_2 to v_3, so that G_1 and G_2 are now connected
- let l_0 be the smallest label in $G(\Gamma)$ and subtract l_0 from all labels in $G(\Gamma)$
- erase all edges and vertices added to $G(\Gamma)$ in the second loop above; each vertex's label is its column

The following observation is immediate: if there is an assignment of the inequalities into columns, such that the V_i's are disjoint, then this algorithm will find it—every choice of label it makes is forced upon it by the edges of the graph, which constrain the possible column assignments. What we must show is that there is always such an assignment. In particular, after the algorithm is finished, if we form the V_i's, will these sets be pairwise disjoint?

First note that the edges of $G(\Gamma)$ actually used by the algorithm in assigning labels induce a spanning tree on each connected component of $G(\Gamma)$. So between any two vertices v_1 and v_2 (labelled l_1 and l_2, respectively) within a connected component of $G(\Gamma)$, there is a unique path along the spanning tree that joins them. Moreover the signed length of the path from v_1 to v_2 along the spanning tree is $l_2 - l_1$ (easy induction on path lengths).

Let each vertex's label be its column and form the sets $V_0, \ldots, V_m$. Suppose there are sets V_i and V_j with a variable ϕ such that $\phi \in V_i \cap V_j$. We first assume that the two corresponding occurrences of ϕ lie within the same connected component of $G(\Gamma)$. Then there are four cases, depending on whether ϕ is found on the left-hand sides or the right-hand sides of the inequalities involved. We consider one case in detail here—there are inequalities $\tau_1 \leq \mu_1$ and $\tau_2 \leq \mu_2$, in

columns i and j, respectively, such that $\phi \in \mathrm{Vars}(\mu_1) \cap \mathrm{Vars}(\mu_2)$. The signed path length between these two vertices is $j - i$. By hypothesis, all paths between two inequalities having ϕ on the right-hand side must then have this signed length. Consider now the distance from the vertex $\tau_1 \leq \mu_1$ to itself. It must also have value $i - j$, by hypothesis on $G(\Gamma)$, but of course the distance from a vertex to itself is 0. Hence $i - j = 0$, from which we obtain $i = j$. The remaining three cases are similarly easy.

We now suppose that $\tau_1 \leq \mu_1$ and $\tau_2 \leq \mu_2$ lie in *different* connected components of $G(\Gamma)$, so that there is no path joining them. There are then two possibilities:

- $\tau_1 \leq \mu_1$ and $\tau_2 \leq \mu_2$ were the vertices considered in the second part of the algorithm—then they were assigned the same label; hence $i = j$.
- otherwise two vertices v_1 and v_2, with a variable ψ in common, were used by the algorithm to temporarily join the connected components. Say v_1 and $\tau_1 \leq \mu_1$ are in the same connected component, as are v_2 and $\tau_2 \leq \mu_2$. By hypothesis on $G(\Gamma)$, the signed path length from v_1 to $\tau_1 \leq \mu_1$ is equal to the signed path length from v_2 to $\tau_2 \leq \mu_2$. Since the algorithm assigns v_1 and v_2 the same column, it follows again that $i = j$.

This completes the proof. □

A few characteristics of Theorem 3.1's formulation of the ASUP condition are worth noting. First, the disjointness of the sets $V_0, \ldots, V_m$ is modelled by a condition requiring constancy of path lengths. Second, although the constants mentioned in the four conditions are, of course, related to one another, we still need all four conditions—this is because a given variable might occur only on left-hand sides, or only on right-hand sides. In these cases, not all four constants may exist for a given choice of α_1 and α_2. Finally, although any directed graph satisfying the conditions of the theorem must be acyclic, there is no direct notion of acyclicity mentioned in the theorem. In Section 3.2, we generalize the condition for acyclicity, while maintaining decidability. The new condition clearly has an acyclic flavour.

3.2 R-ASUP

We now define a new acyclicity criterion for the graph G corresponding to a SUP instance Γ. We call this criterion *R-acyclicity*; we show that R-acyclicity is sufficient to guarantee termination of the redex procedure, and is more general than the original, column-based criterion.

Figure 3.2: The relation R and R-acyclicity. The notation $\mu(\alpha)$denotes an expression μ in which α occurs as a subexpression. Here, $\alpha R \beta$—indeed, $\alpha R' \beta$. Since also $\beta R \gamma R \alpha$, we have $\beta R^+ \alpha$; therefore, this graph is acyclic, but *not* R-acyclic.

Definition 3.5 (R-acyclic) *For a graph G of a SUP instance $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^N$, define relations R, R' on variables in G as follows: $\alpha R \beta$ (resp. $\alpha R' \beta$) if there exist vertices v_i and v_j with $\alpha \in \mathrm{RVars}(v_i)$, $\beta \in \mathrm{RVars}(v_j)$, and $v_i \rightarrow^* v_j$ (resp. $v_i \rightarrow^+ v_j$). G is said to be* R-acyclic *if whenever $\alpha_i R' \alpha_j$, we have $\neg(\alpha_j R^+ \alpha_i)$, where R^+ is the transitive closure of the relation R.*

The "R" in R-acyclic refers, of course, to the relation R in Definition 3.5. However, it also highlights the asymmetry in the definition between RVars and LVars—in particular, that we impose conditions on RVars, but not on LVars. Hence, "R-acyclic" may be read as "right-acyclic".

Although the statement of R-acyclicity is somewhat involved, R-acyclicity is not itself difficult to understand. For illustrative purposes, Figure 3.2 depicts a graph that is not R-acyclic. Note that R-acyclicity implies graph acyclicity in the ordinary sense. R-acyclicity may also be understood in terms of "broken graph cycles", which generalize ordinary graph cycles, in that there may be breaks in the cycle as long as the endpoints of any gap have a variable in common on the right-hand side (again, see Figure 3.2). The R-acyclic graphs are those that do not have broken cycles.

We wish to show that the redex procedure will terminate on R-acyclic instances. Our proof hinges on the observation that redex reduction preserves R-acyclicity:

Theorem 3.2 (Invariance of R-acyclicity) *Let Γ be a SUP instance, and let Γ' be the result of performing one iteration of the redex procedure on Γ (i.e., the result of reducing one redex in Γ). If $G(\Gamma)$ is R-acyclic, then $G(\Gamma')$ is R-acyclic.*

Proof Suppose a redex-I is reduced in Γ. Then all occurrences of some variable α are replaced with some expression τ, containing only fresh variables. Hence all vertices that contained α now contain the variables (if any) of τ, and no other vertices contain these variables because they are

all fresh—therefore no edges are created by this reduction. Hence, no R-cycles can be created, and G remains R-acyclic. Suppose now that a redex-II is reduced in Γ. If reduction causes G to lose R-acyclicity, then there are two possibilities:

- there is a variable replacement $[\tau/\alpha]$ that occurs during reduction, which induces, for some $\beta_1, \ldots \beta_n$, the relations $\beta_1 R \cdots R \beta_n$, and such that $\beta_n R' \beta_1$. Since $[\tau/\alpha]$ caused the violation, it created an edge that completed one of the paths from β_i to $\beta_{(i \bmod n)+1}$. For such an i, there is an edge from some $v_j \to v_k$ lying along this path, that was created by the substitution $[\tau/\alpha]$. Hence, one of $\mathrm{RVars}(v_j)$ and $\mathrm{LVars}(v_k)$ contains the variable α; the other contains a variable from τ, say γ. Now, for the redex $[\tau/\alpha]$ to exist, there must exist an inequality v_h that satisfies the conditions for this redex-II; hence α and τ are both in $\mathrm{RVars}(v_h)$. Since one of α and γ is in $\mathrm{LVars}(v_k)$, we have $v_h \to v_k$. Since either α or γ is in $\mathrm{RVars}(v_j)$, and both are in $\mathrm{RVars}(v_h)$, the transitive closure of R connects the path ending with v_j to the path beginning with v_h, and followed by v_k. Hence, the removal of the edge from v_j to $\tau_k \leq \mu_k$ does not restore R-acyclicity. Thus, removing edges introduced by redex-II reductions cannot convert graphs that are not R-acyclic to graphs that are.

- we had $\beta_i R \cdots R \beta_j$ and $\beta_k R \cdots R \beta_l$, for some β_i, β_j, β_k, and β_l, and the redex reduction unifies β_j and β_k, thus linking the two R-chains. In this case, if a redex-II reduction unifies β_j and β_k, then these two variables must occur together on the right-hand side of some inequality (the one in which the redex occurs). Hence $\beta_j R \beta_k$, and we already had $\beta_i R \cdots R \beta_j R \beta_k R \cdots R \beta_l$, i.e., the two R-chains were already linked. Again, the redex-II reduction can only result in a non-R-acyclic instance if the instance was non-R-acyclic to begin with.

In both cases, we see that redex-II reduction cannot reduce a graph that is R-acyclic to one that is not. In summary, then, the redex procedure preserves R-acyclicity. □

Corollary 1 *Let α and β be variables in an R-ASUP instance Γ, with $\alpha R' \beta$. Then no reduction σ of Γ will produce $\beta \sigma R \alpha \sigma$.*

Proof Consider the instance

$$\Gamma' := \Gamma \cup \{x_1 \leq f(\alpha, x_2),\ x_2 \leq \beta\} ,$$

where x_1 and x_2 are fresh variables. Then this extra inequality gives us $\alpha R' \beta$, which we already had, and $x_2 R' \beta$, which is of no consequence because x_2 does not occur anywhere else. Therefore,

this instance is R-acyclic iff Γ is. If Γ reduces such that we obtain $\beta\sigma R\alpha\sigma$, then in Γ', we have $\alpha\sigma R'\beta\sigma R\alpha\sigma$ (because no reduction is going to affect the two extra inequalities). Hence Γ' is now non-R-acyclic. But this contradicts the invariance of R-acyclicity. Therefore, Γ cannot reduce so as to produce $\beta\sigma R\alpha\sigma$.

Note that this argument presumes the existence of at least one binary functor, f. But without a binary functor, there can be no redex-II's, and redex-I reduction cannot create edges. Thus, the result follows either way. □

Corollary 2 *Let v_1 and v_2 be vertices in the graph of an R-ASUP instance Γ, such that v_1 precedes v_2 in the partial order induced by the graph. Suppose that after k iterations of the redex procedure (i.e., after reduction of k redexes), Γ reduces to an instance Γ_k. Let σ_k be the substitution that converts Γ to Γ_k (i.e., σ_k is the accumulated substitution encapsulating the k redex reductions). Then $v_2\sigma_k$ cannot precede $v_1\sigma_k$ in the graph of Γ_k.*

Proof Let $v_1 = \tau_1 \leq \mu_1$, $v_2 = \tau_2 \leq \mu_2$. Let $\alpha \in \text{Vars}(\mu_1)$, $\beta \in \text{Vars}(\mu_2)$. If v_1 precedes v_2 in the partial order induced by the graph, then we have $\alpha R'\beta$. If, after reduction, the graph has v_2 preceding v_1, then we would have $\beta R'\alpha$, contradicting the previous claim.

This argument presumes that μ_1 and μ_2 each contain at least one variable. We know that μ_1 must contain a variable; otherwise v_1 could not precede anything (it would have no out-edges). Further if μ_2 had no variables, then no reduction could make v_2 precede anything. Hence, the case where either inequality contains no variables on the right-hand side poses no difficulty. □

The following lemma, establishing the solvability, via the redex procedure, of singleton instances of SUP, will ultimately form the base case of our main result:

Lemma 3.1 *Every instance of SUP comprising a single inequality $\tau \leq \mu$, with $\text{Vars}(\tau) \cap \text{Vars}(\mu) = \emptyset$, is solvable by the redex procedure (that is, the redex procedure will terminate on such an input).*

Proof We bound the number of redex reductions that can be performed in $\tau \leq \mu$:

- *The number of redex-I reductions in $\tau \leq \mu$ is bounded by the number of leaf nodes in τ (i.e., by the number of variable occurrences in τ).* Every redex-I reduction causes at least one variable α in τ to be matched against a variable in μ. No further reduction will ever again cause this occurrence of α to be part of a redex-I. Hence there can be no more redex-I's than leaves in τ. (Note that, because $\text{Vars}(\tau) \cap \text{Vars}(\mu) = \emptyset$, redex reduction does not change τ.)

- *The number of redex-II reductions that can occur in $\tau \leq \mu$ before a redex-I reduction must occur is bounded by* $|\text{Vars}(\mu)|$. This is because each redex-II reduction replaces at least one variable in μ; hence it decreases $|\text{Vars}(\mu)|$ by at least 1.

Since the number of redex-II reductions that can occur between redex-I reductions is bounded, and the total number of redex-I reductions is bounded, the redex procedure must eventually terminate. □

Lemma 3.2 *Let $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^N$ be a SUP instance and suppose that for some i, a redex reduction σ in $v_i = (\tau_i \leq \mu_i)$ creates a redex in $v_j = (\tau_j \leq \mu_j)$ for some j. Then $v_i \to v_j$ in $G(\Gamma)$.*

Proof If a redex-I is created in v_j then for some path Σ, $\Sigma(\mu_j\sigma)$ is a variable, and $\Sigma(\tau_j\sigma)$ is not a variable. Since the redex-I did not exist previously, $\Sigma(\tau_j)$ must be a variable. Hence, σ contains the replacement $[\Sigma(\tau_j\sigma)/\Sigma(\tau_j)]$. Since redex reductions indicate replacements on the right-hand sides of inequalities in which they originate, it follows that $\Sigma(\tau_j) \in \text{Vars}(\mu_i)$. Hence there is an edge from v_i to v_j. If a redex-II is created in v_j, then for some paths Σ_1, Σ_2, $\Sigma_1(\tau_j\sigma) = \Sigma_2(\tau_j\sigma) = \alpha$ for some variable α and $\Sigma_1(\mu_j\sigma) \neq \Sigma_2(\mu_j\sigma)$. Since substitutions cannot "un-unify" two expressions, it follows that $\Sigma_1(\mu_j) \neq \Sigma_2(\mu_j)$, or at least one of these does not exist. In the former case, since the redex did not exist previously, we must have $\Sigma_1(\tau_j) \neq \Sigma_2(\tau_j)$. Hence, at least one of these was replaced during the redex reduction, and therefore occurs in μ_j. Hence, there is an edge from v_i to v_j. In the latter case, the non-existence of either $\Sigma_1(\mu_j)$ or $\Sigma_2(\mu_j)$ indicates that there is already at least one redex-I at that site, and reduction of this redex-I within v_j itself would create the redex-II anyway. Hence this redex-II does not arise strictly as a result of a reduction in v_i. In those cases in which it does, however, we always have $v_i \rightarrow v_j$. □

We now prove the redex procedure terminates on R-acyclic instances. The inductive step of the proof relies on the fact that every directed acyclic graph G creates a partial order $\sqsubseteq_G$ on its vertices, defined such that $v_1 \sqsubseteq_G v_2$ if there is a directed path from v_1 to v_2. The minimal elements in $\sqsubseteq_G$ are the source vertices, and the maximal elements are the sink vertices. Further, every directed acyclic graph has at least one source vertex and at least one sink vertex. Hence also, the relation $\sqsubseteq_G$ has at least one minimal element and at least one maximal element.

Theorem 3.3 *Let Γ be an instance of SUP and G be the graph induced by Γ according to Definition 3.4. If G is R-acyclic, then the redex procedure will terminate on Γ.*

Proof For each i, let Γ_i be the result of performing i reductions on the instance Γ, according to the redex procedure, and let $G_i = G(\Gamma_i)$. With each G_i is associated a partial order $\sqsubseteq_{G_i}$, induced by its edges, as described above. By Corollary 2 of Theorem 3.2, if, for vertices v_x and v_y, we have $v_x \sqsubseteq_{G_i} v_y$ for some i, then there is no j such that $v_y \sqsubseteq_{G_j} v_x$. Therefore the union of all of the partial orders, namely,

$$\sqsubseteq := \bigcup_i \sqsubseteq_{G_i} ,$$

is a partial order, respecting all of the partial orders associated with all reduced instances Γ_i. Let $\leq$ be any total order of the vertices of Γ consistent with $\sqsubseteq$, and number the vertices in Γ according to this order. Then any reduction of a redex in a vertex v_i can only induce redexes in vertices v_j for $i \leq j$. For each vertex v_i let n_i be the maximum number of redexes that can be reduced in v_i before it (considered in isolation) is solved (this number is finite, by Lemma 1). We then proceed by induction on the ordinal $(n_1, \ldots, n_N)$, under lexicographic ordering, which is a well-ordering of the N-tuple. Since reduction of a redex in any v_i reduces n_i, and can only increase n_j for $j > i$ (this by Lemma 3.2), and since the instance is solved when the ordinal is $(0, \ldots, 0)$, the result follows by induction. □

Corollary 3 *The set of SUP instances that have R-acyclic graphs forms a decidable subset of SUP.*

Definition 3.6 (*R*-ASUP) *R-ASUP is the restriction of SUP to R-acyclic problem instances.*

Theorem 3.3 establishes R-ASUP as a decidable subset of SUP, and moreover, one for which the redex procedure is a full solution procedure (that is, it is guaranteed to terminate on instances with no solution). It remains to establish the relationship between R-ASUP and the original ASUP.

Theorem 3.4 *R-ASUP is a strict superset of ASUP.*

Proof For variables α and β, if $\alpha R' \beta$ (where R' is as given in Definition 3.5), then α's column assignment is strictly smaller than β's. If also $\beta R^+ \alpha$, then β's column assignment would be less than or equal to α's, which is a contradiction. Hence $\neg(\beta R^+ \alpha)$, and therefore any instance that satisfies the column-based definition of acyclicity is R-acyclic. On the other hand, consider any

instance containing the following inequalities:

$$\begin{aligned} \alpha &\leq \beta \\ \beta &\leq \gamma \\ \alpha &\leq \gamma \end{aligned}$$

This instance does not satisfy the column-based criterion for acyclicity—for suppose that $\alpha \in V_i$. Then by the second inequality, $\gamma \in V_{i+2}$, but by the third inequality, $\gamma \in V_{i+1}$. On the other hand, it is easy to see that these inequalities are R-acyclic. Hence, indeed, R-acyclicity is strictly more general than the column-based criterion.

3.3 SUP Translation with Fewer Variables

In this section we show that, by targetting R-ASUP, rather than ASUP, the KW translation from typability to SUP can be done with quadratically fewer variables and associated inequalities. [32]

As we are considering a concrete application of R-ASUP in this section, we move from working in an abstract term algebra to the concrete algebra of functional types. In particular, we assume a countably infinite set of variables, and terms are built from variables, and the binary functor $\rightarrow$, which we write as an infix operator and consider to have right-associativity.

Recall that under the KW translation to ASUP (see Section 2.4.3), each variable x_j, y_j, and w_j in a θ-normal term E gets, respectively, $n+1$, $n-j+1$, and $n+1$ specializable type variables to track its type, one for each expression E_i in which the variable could possibly occur. These variables are called, respectively, $\beta^x_{1,j}, \ldots, \beta^x_{n+1,j}$, $\beta^y_{j+1,j}, \ldots, \beta^y_{n+1,j}$, and $\beta^w_{1,j}, \ldots, \beta^w_{n+1,j}$. Second, the variables associated with x_j, y_j, and w_j, are related to one another by chains of inequalities: $\beta^x_{1,j} \leq \cdots \leq \beta^x_{n+1,j}$, $\beta^y_{j+1,j} \leq \cdots \leq \beta^y_{n+1,j}$, and $\beta^w_{1,j} \leq \cdots \leq \beta^w_{n+1,j}$. Hence, any information we deduce for a variable "lower" in the chain will get propagated, via redex-I reduction, "up" the chain to the remaining variables. Collectively, we shall refer to the variables $\beta^x_{i,j}$, $\beta^y_{i,j}$, and $\beta^w_{i,j}$ as *β-variables*. Further, we write $\beta_{i,j}$ when the distinction among x, y, and w is unimportant. We use the notation $\beta_{\perp,j}$ to stand for $\beta^x_{1,j}$, $\beta^w_{1,j}$, or $\beta^y_{j+1,j}$.

Next, we observe a few properties of the ASUP solution procedure (which is the same as the R-ASUP solution procedure) in relation to the placement of variables:

Observation 3.1 *If a variable α is replaced during redex reduction, then it must occur as part of a redex—this occurrence is always on the right-hand side of some inequality.*

Observation 3.1 is immediate from the definition of the redex procedure.

Observation 3.2 *For $i > \bot$, $\beta_{\vec{i},j}$ only occurs on the left-hand sides of inequalities, except within the inequality $\beta_{\vec{i-1},j} \leq \beta_{\vec{i},j}$.*

Observation 3.2 is also immediate.

Since $\beta_{\vec{i},j}$, $i > 1$, only occurs on the right-hand side of an inequality when it occurs alone, it cannot occur as part of a redex-II. Hence, it can only be replaced as part of a redex-I reduction. This, in turn, can only take place if the variable $\beta_{\vec{i-1},j}$ is replaced during redex reduction. By induction, for all $i > 1$, the variable $\beta_{\vec{i},j}$ can only be replaced if the variable $\beta_{\vec{\bot},j}$ is replaced.

Each $\beta_{\vec{\bot},j}$ can be replaced in only one way:

- $\beta^x_{1,j}$ is replaced via the equality $\beta^x_{1,j} = \tau$ (i.e., the inequality $\alpha \to \alpha \leq \beta^x_{1,j} \to \tau$) if the parameter x_j is annotated with $\forall.\tau$, where τ is a rank 0 type.
- $\beta^w_{1,j}$ is replaced via the equality $\beta^w_{1,j} = \tau$ (i.e., the inequality $\alpha \to \alpha \leq \beta^w_{1,j} \to \sigma$) if the initial type environment associates the free variable w_j with the type $\forall.\tau$, where τ is a rank 0 type.
- $\beta^y_{j+1,j}$ is replaced via the equality $\beta^y_{j+1,j} = \delta_{E_i}$ (i.e., the inequality $\alpha \to \alpha \leq \beta^y_{j+1,j} \to \delta_{E_i}$) because of the redex $(\lambda y_j.P_j)E_j$.

These are the only inequalities in which the variables $\beta_{\vec{\bot},j}$ occur on the right-hand side.

Assuming these replacements occur (in the case of $\beta^y_{j+1,j}$, they will surely occur), and assuming they result in the replacement of $\beta_{\vec{\bot},j}$ by something other than just another variable, then there will be a redex-I in the inequality $\beta_{\vec{\bot},j} \leq \beta_{\vec{\bot}+1,j}$. Suppose that $\beta_{\vec{\bot},j}$ has been replaced by an expression $\tau_{\vec{j}}$.Then redex reduction replaces $\beta_{\vec{\bot}+1,j}$ by the expression $\tau_{\vec{j}}{}'$, which is $\tau_{\vec{j}}$, with all variables renamed consistently to fresh variables. Similarly, $\beta_{\vec{\bot}+2,j}$ is replaced by $\tau_{\vec{j}}{}''$, and so on. Therefore, we have proved the following:

Theorem 3.5 *Given j and the choice of x, y, of w, if $\beta_{\vec{\bot},j}$ is replaced by an expression $\tau_{\vec{j}}$, then all $\beta_{\vec{i},j}$ are replaced by expressions that differ from $\tau_{\vec{j}}$ by consistent renaming of variables to fresh ones. In particular, they are all structurally equivalent.*

The replacement of variables by fresh ones is important, because it keeps the problem instance within the realm of ASUP—for each i, the variable $\beta_{\vec{i},j}$ occurs within the set V_i and all sets V_i are declared by ASUP to be pairwise disjoint. Therefore, to remain within ASUP, the variables in each $\tau_{\vec{j}}{}^{(i)}$ must be pairwise disjoint sets.

We define the notion of a *solved inequality*:

Definition 3.7 (Solved) *An inequality $\tau \leq \mu$ is* solved of order 0 *if there is a substitution σ such that $\tau\sigma = \mu$, and the variables of τ do not occur on the right-hand side of any inequality in the instance. An inequality $\tau \leq \mu$ is* solved of order k *for $k > 0$ if there is a substitution σ such that $\tau\sigma = \mu$, and the variables of τ do not occur on the right-hand side of any inequality in the instance that is not solved of order j for some $j < k$. An inequality is* solved *if it is solved of order k for some $k \geq 0$.*

Informally, $\tau \leq \mu$ is solved if $\tau\sigma = \mu$ for some σ, and the variables of τ do not occur on the right-hand side of any unsolved inequality in the instance. The important property of solved inequalities is that they will never contain a redex, which is established by the following two lemmas:

Lemma 3.3 *If an inequality $\tau \leq \mu$ is solved of order 0, then the redex procedure will not find a redex in it.*

Proof If there is a substitution σ such that $\tau\sigma = \mu$, then the empty substitution semiunifies τ and μ. Thus, if the redex procedure produces a semiunifier σ_0, then by Theorem 2.1, there is a substitution σ' such that $\sigma' \circ \sigma_0 = []$. This is only possible if σ_0 is a renaming substitution, and the redex procedure does not produce renaming substitutions. □

Lemma 3.4 *If an inequality is solved, then the redex procedure will never find a redex in it.*

Proof If an inequality $\tau \leq \mu$ is solved, then it is solved of order k for some $k \geq 0$. For $k = 0$, the result follows from the previous lemma. Otherwise, there is a substitution σ such that $\tau\sigma = \mu$ and every occurrence of every variable in $\mathrm{Vars}(\tau)$ on a right-hand side is within an inequality that is solved of order $j < k$. By induction, no redex will be found in any such inequality, and therefore, no substitution performed by the procedure will ever replace a variable in τ. Since the empty substitution already semiunifies $\tau \leq \mu$, there are currently no redexes in $\tau \leq \mu$; since no substitution will ever replace a variable in τ, there will continue to be no redexes in $\tau \leq \mu$. Hence, by induction, the redex procedure will never find a redex in a solved inequality. □

Lemma 3.4 allows us to make the following observation about chains of β-variables:

Observation 3.3 *Once the redex-I's in the chains of β-variable inequalities have been reduced, the chains are solved.*

Therefore, once the variables $\beta_{i,j}$ have been replaced by $\tau_j^{(i)}$, we may assume that the chains of β-variable inequalities are no longer there. Therefore, for all practical purposes, the expressions $\tau_j^{(i)}$ only occur on left-hand sides.

Observation 3.4 *If a variable only occurs on left-hand sides of inequalities in a SUP instance, then it will never be replaced by the redex procedure.*

This is clear—since redex reductions arise from replacements on the right-hand side of an inequality, if a variable never occurs on a right-hand side, it will never be part of a redex.

Depending on the instance, we sometimes have a choice regarding whether a particular variable replacement is performed by one or more of the substitutions $\sigma_1, \ldots, \sigma_n$, or by the solution σ. We prefer solutions in which σ does as little work as possible; we formalize this notion below:

Definition 3.8 *Let $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^n$ be a SUP instance with solution σ. σ is called* canonical *if*

$$\mathrm{dom}(\sigma) \subseteq \bigcup_{i=1}^{n} \mathrm{Vars}(\mu_i) \ .$$

The following result shows that any SUP solution has a "canonical core" that is also a solution:

Theorem 3.6 (Canonical Solutions) *If a SUP instance $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^N$ has a solution σ, then σ can be written as $\sigma'_C \circ \sigma_C$, where σ_C is canonical,* $\mathrm{dom}\ \sigma_C \cap \mathrm{dom}\ \sigma'_C = \emptyset$*, and σ_C solves Γ. Moreover, for any $\alpha \in \mathrm{Vars}(\mu_i)$ for some i, $\alpha\sigma_C = \alpha\sigma$.*

Proof For each inequality $\tau_i \leq \mu_i$ in Γ, there is a substitution σ_i such that $\tau_i\sigma\sigma_i = \mu_i\sigma$. Let

$$V = \bigcup_{i=1}^{n} \mathrm{Vars}(\mu_i) \ .$$

Let $\sigma_C = \sigma|_V$ (the restriction of σ to variables in V) and $\sigma'_C = \sigma \setminus \sigma_C$ (i.e., the restriction of σ to variables not in V). Then σ_C is canonical by construction, $\sigma = \sigma'_C \circ \sigma_C$, and $\mathrm{dom}\ \sigma_C \cap \mathrm{dom}\ \sigma'_C = \emptyset$. It remains to show that σ_C solves Γ. We have

$$\tau_i\sigma\sigma_i = \mu_i\sigma \ .$$

Hence, since σ_C and σ'_C commute (their domains are disjoint),

$$\tau_i\sigma_C\sigma'_C\sigma_i = \mu_i\sigma'_C\sigma_C \ .$$

Since dom $\sigma'_C \cap V = \emptyset$, we have

$$\tau_i \sigma_C \sigma'_C \sigma_i = \mu_i \sigma_C \ .$$

Grouping σ'_C and σ_i together as $\sigma_i \circ \sigma'_C$, we see that σ_C solves each inequality in Γ; hence, it solves Γ. The final claim, $\alpha\sigma_C = \alpha\sigma$, follows immediately from the construction $S_C = \sigma|_V$ and the fact that $\alpha \in V$. □

From now on we implicitly assume that solutions of SUP instances are canonical. Note that any solution computed by the redex procedure arises from redexes whose reduction produces substitutions on at least one right-hand side; hence the redex procedure always computes canonical solutions.

Theorem 3.7 *Let $\tau \leq \mu$ be an inequality in a SUP instance Γ, such that the variables in τ do not occur on any right-hand sides. Let τ' be a consistent variable renaming of τ, such that the variables in τ' also do not occur on any right-hand sides. Let $\Gamma' = (\Gamma \setminus \{\tau \leq \mu\}) \cup \{\tau' \leq \mu\}$. Then Γ' has a solution if and only if Γ has a solution. Moreover, Γ and Γ' have the same solutions.*

Proof There exist renaming substitutions ρ and ρ' such that $\tau\rho = \tau'$ and $\tau'\rho' = \tau$. If S solves Γ, then there is a substitution S_1 such that $\tau S S_1 = \mu S$. But then $\tau'\rho' S S_1 = \mu S$. Taking $S' = \rho' \circ S$, and noting that the variables in the domain of ρ' do not intersect those in the domain of S (because S is assumed canonical), we have $\rho' \circ S = S \circ \rho'$. Thus, $\tau' S \rho' S_1 = \mu S$. Letting $S'_1 = S_1 \circ \rho'$, we have $\tau' S S'_1 = \mu S$. As S still solves the remaining inequalities in Γ, it follows that S solves Γ'. Similarly, any solution of Γ' also solves Γ. □

The expressions $\tau_j^{(i)}$ that replace the variables $\beta_{i,j}$ contain only variables that appear on the left-hand sides of inequalities (after the chains of β-variables, and other solved inequalities, have been removed). Hence, we may freely rename the variables in $\tau_j^{(i)}$, so long as the variable names we choose also do not occur on any right-hand sides. Thus we can choose to rename variables such that $\tau_j^{(i)} = \tau_j^{(k)}$ for all i, k, without affecting the solution. What we obtain is the same instance that would result if we used a single variable for each of x_j, y_j, w_j. Therefore, we have the following:

Theorem 3.8 *Consider an expression of the form*

$$\lambda x_1 \cdots x_m.(\lambda y_1.(\lambda y_2.(\cdots((\lambda y_n.E_{n+1})E_n)\cdots))E_2)E_1,$$

with free variables $w_1, \ldots w_p$, with all bound variables uniquely named, distinctly from all free variables, and in which each E_i may introduce variable bindings, called called $z_{i,1}, z_{i,2}, \ldots$. Let Γ be the ASUP instance derived for this expression, as described before. Let Γ' be the SUP instance produced by Algorithm KW, except that each term variable is represented by exactly one SUP variable, rather than a chain of such variables. Then Γ and Γ' have the same solutions.

The theorem establishes that the SUP instance obtained by our new translation procedure has the same solution as the corresponding ASUP instance. However, the theorem does not guarantee that, with the instance in this form, the redex procedure will be able to find the solution, as the output of this translation procedure will, in general, lie outside of ASUP. If, in order to find the solution, a procedure must fill in the variables and inequalities we just removed, then we are no further ahead. However, the following theorem shows that this is not necessary:

Theorem 3.9 *Every SUP instance produced by the procedure outlined in Theorem 3.8 is an instance of R-ASUP.*

Proof Let Γ be the ASUP instance obtained from some λ-term M. Since ASUP is contained in R-ASUP, Γ is an R-ASUP instance. Let Γ_1 be the result of removing all of the β-chains from Γ. Since removing nodes from a graph does not create any new paths, Γ_1 is an R-ASUP instance. Now, for each β-variable, there is at most one inequality $\tau_k \leq \mu_k$ that contains it on the right-hand side (moreover, since this inequality is phrased as an equality, it follows that $\tau_k = \alpha_k \to \alpha_k$, where α_k occurs in no other inequality). By merging all of the β-variables pertaining to a particular term variable (assume we rename all $\beta_{\bar{0},j}$ to $\beta_{\bar{j}}$), we create edges from $\tau_k \leq \mu_k$ to each inequality that had $\beta_{\bar{i},j}$ on the left-hand side. Can any of these new paths violate R-acyclicity? Since the original instance was an ASUP instance, any paths that violate R-acyclicity must contain these new edges. Since $\tau_k = \alpha_k \to \alpha_k$, and α_k is fresh, the vertex $\tau_k \leq \mu_k$ has no in-edges. So any path containing $\tau_k \leq \mu_k$ contains it as the first node in the path. Thus, for R-acyclicity to fail, we need to find a variable ϵ not in μ_k such that $\epsilon R \zeta$ for some variable ζ in μ_k. But since the variables in μ_k (i.e., $\beta_{\bar{j}}$ and the type expression with which it is equated) never occur anywhere else on a right-hand side, it follows that there is no variable ϵ not in μ_k such that $\epsilon R \zeta$ for any $\zeta \in \mathrm{Vars}(\mu_k)$, and therefore the instance is R-acyclic. □

As a consequence of Theorem 3.9, we can simply apply the redex procedure to the R-ASUP instance obtained via Theorem 3.8, and be guaranteed of termination with the same answer.

3.3.1 The New SUP Translation

By unifying all of the β-variables associated with a given term variable, and eliminating the associated chains of inequalities, the translation from typability to SUP (now R-ASUP, specifically) simplifies as shown below. For a term of the form

$$\lambda^2 x_1.\cdots\lambda^2.x_m.(\lambda^1 y_1.(\cdots((\lambda^1 y_n.E_{n+1})E_n)\cdots)E_2)E_1,$$

with free variables $w_1,\ldots,w_p$, in which each E_i contains no λ^1- or λ^2-abstractions, and in which each λ^3-abstraction in subexpression E_i will bind a λ^3-variable $z_{i,j}$, we translate to R-ASUP as follows:

- for each x_j, the variable β_{x_j} represents its type;
- for each y_j, the variable β_{y_j} represents its type;
- for each w_j, the variable β_{w_j} represents its type;
- for each $z_{i,j}$, the variable $\gamma_{i,j}$ represents its type;
- for each subexpression M of each E_i, the variable δ_M represents its derived type (different occurrences of the same subexpression are assigned different δ-variables).

Then, for each subexpression M of each E_i, the R-ASUP instance contains the following inequalities:

- if $M = x_j$, the inequality $\beta_{x_j} \leq \delta_M$;
- if $M = y_j$, the inequality $\beta_{y_j} \leq \delta_M$;
- if $M = w_j$, the inequality $\beta_{w_j} \leq \delta_M$;
- if $M = z_{i,j}$, the inequality $\gamma_{i,j} = \delta_M$;
- if $M = \lambda^3 z_{i,j}.E$, the equality $\delta_M = \gamma_{i,j} \to \delta_E$;
- if $M = PQ$, the equality $\delta_P = \delta_Q \to \delta_M$,

where an equality $\tau = \mu$ is syntactic sugar for the inequality $\alpha \to \alpha \leq \tau \to \mu$, where α is a fresh variable.

Each redex $(\lambda^1 y_i.E_{i+1})E_i$ contributes the equality $\beta_{y_i} = \delta_{E_i}$. Finally, for the free variables w_j and the λ^2-variables x_j, we consult a type environment A. If $A(w_j)$ yields a type τ, we add the equality $\beta_{w_j} = \tau$, and similarly for x_j; otherwise we add no new inequalities.

We obtain a translation that, by virtue of eliminating sets of variables (and their associated inequalities) with no intuitive purpose, and thereby more closely matching the form of the original term, is both more concise and easier to understand than the previous formulation. In particular, if we let m, n, and p represent, respectively, the number of λ^1-bound variables, λ^2-bound variables, and free variables, then we save

$$\left(\frac{m-1}{2} + n + p\right) m$$

variables, and as many inequalities. Hence, the resulting R-ASUP instance is quadratically smaller than the corresponding ASUP instance.

3.4 Chapter Summary

The notion of acyclicity lies at the heart of what makes ASUP decidable, while the full SUP is not. However, just what about ASUP makes it acyclic lies somewhat hidden behind a rather elaborate definition involving column assignments and disjoint sets. In fact, what the sets $V_0, \ldots V_m$ in the definition of ASUP express is information flow. Each column in the definition of ASUP is linked to its predecessor and successor (where applicable) by the variables it has on the left in common with its predecessor's right-hand side, and the variables it has on the right in common with its successor's left-hand side.

By moving from a column-based formulation to a graph-based formulation, the sets V_i re-emerge as edges between nodes, and the information flow they express becomes clear. The ASUP condition then becomes a requirement of constancy of path lengths, and it immediately becomes apparent that perhaps a less strict definition of acyclicity may offer the same guarantee of decidability as ASUP, while offering other benefits at the same time.

Plain graph acyclicity, as it can be shown, is insufficient, as there exist SUP instances with acyclic graphs that can be reduced to instances with cyclic graphs, and infinite reductions may ensue. Instead, we showed in this chapter that a somewhat stronger condition, which we have called R-acyclicity, is invariant under redex reduction, and ultimately gives rise to a decidable subset of SUP that is strictly more general than ASUP. In addition, it has the benefits of ease of description, including a clear notion of acyclicity in its definition.

Furthermore, when used as a target for the KW SUP translation from rank 2-typability, it renders certain variables and inequalities in the translation unnecessary, thereby giving rise to a quadratically smaller SUP instance after translation. In this way, the introduction of R-acyclicity makes progress towards our fourth goal (making the system easy to state and understand), as laid out in Chapter 1.

The notion of R-acyclicity plays a central role in the next chapter, when we generalize SUP to facilitate a fully syntax-directed translation from rank 2-typability to a SUP-like set of constraints. The core of our termination arguments will be an appeal to the invariance of R-acyclicity under redex reduction, for a suitably modified graph-theoretic characterization of problem instances.

Chapter 4

Unknowns

In this chapter, we build on the results from Chapter 3 by further simplifying the translation from rank 2 System F into SUP. In particular, our goal is to construct a translation that bypasses the θ-reduction step, with the goal of having a syntax-directed correspondence between the two systems. We see in this chapter that even the full SUP itself is not an appropriate target for a completely syntax-directed translation of rank 2-typability; rather, a more general problem (undecidable, but with decidable subsets analogous to acyclicity, R-acyclicity, etc.) provides a much more natural framework for translation.

4.1 Eliminating θ-Reduction

By targeting R-ASUP instead of ASUP, we showed in Chapter 3 that typability can be translated into a solvable SUP instance that contains quadratically fewer variables and inequalities than the instance associated with the standard KW translation. This innovation alone eases some of the cognitive burden associated with understanding the KW rank 2 inference algorithm.

Arguably, however, θ-reduction imposes a much greater barrier to our understanding of the translation between typability and SUP. Not only is it difficult to understand, but it also rearranges the source term to such an extent that identifying components of the resulting SUP instance with corresponding subexpressions of the original term, may be difficult, or even impossible.

One specific example of the difficulties imposed by θ-reduction is error-reporting. Type errors in the source term translate to SUP instances that have no solution. Once a SUP instance has

been found to have no solution, the offending inequality must somehow be translated back into a subexpression of the original term, so that the programmer knows what part of the source program caused the error. However, after θ-reduction, it can be difficult to trace the error back to a specific subexpression of the original program.

More generally, if we wish to do any kind of analysis on subprograms of the original program, then θ-reduction stands in our way, as there may be no corresponding subexpression in the θ-reduced form of the program, and therefore no corresponding subinstance of SUP on which to do the analysis.

For these reasons, it is our goal in this chapter to devise a translation from rank 2-typability to a SUP-like problem that can be performed on an untransformed source program, without requiring θ-reduction.

4.1.1 Syntax-Directed SUP Translation—First Attempt

Since we no longer allow ourselves to assume that our input term is θ-normal, as a first step towards our goal, we must formulate a basic translation to SUP that does not rely on the characteristics of θ-normal expressions. The translation will turn out to be unsound, and correcting the unsoundness is the major contribution of this chapter; however, before we can correct the unsoundness, we must pinpoint its source, which this translation will highlight.

Special provisions are made in the current translation procedure for λ^1- and λ^2-abstractions. In the former case, because the λ^1-abstraction is always paired with an argument, the translation procedure simply equates the type of the parameter with the type of the argument, and takes the type of the entire redex to be the type of the body of the λ^1-abstraction. Applied recursively, this second observation implies that the result type of the entire term is the type of the innermost λ^1-abstraction body. This observation is implicit in the type recovery phase of algorithm KW.

In the latter case, because the λ^2-abstractions in a θ-normal term outscope the entire term, there is one parameter type in the final computed type for each λ^2-abstraction; hence, rather than explicitly provide inequalities governing the types of the λ^2-abstractions, it suffices to simply prefix the computed result type (from the innermost λ^1-redex) with the type of each λ^2-bound variable x_i, in order.

These two provisions suggest that the following rules be added to the original KW translation procedure to accommodate λ^1- and λ^2-abstractions:

- for each $M = \lambda^1 y_i.E$, add the equality $\delta_M = \beta_{y_i} \to \delta_E$;

- for each $M = \lambda^2 x_i.E$, add the equality $\delta_M = \beta_{x_i} \rightarrow \delta_E$.

Furthermore, the rule for applications should now be extended to apply throughout the instance, rather than just within the subexpressions E_i (see Section 2.4.2), and the procedure should report as the final type the result of performing the accumulated substitution on δ_M (where M is the entire expression), rather than follow the previously-outlined type reconstruction procedure.

To see that these new rules operate identically to the previous formulation, when applied to θ-normal terms, consider the effect of applying the new translation on the redex $M = (\lambda^1 y_i.E_{i+1})E_i$. Our translation produces, in addition to the inequalities associated with E_i and E_{i+1}, the following two equalities, representing, respectively, the abstraction and the application:

$$\begin{aligned} \delta_{\lambda^1 y_i.E_{i+1}} &= \beta_{y_i} \rightarrow \delta_{E_{i+1}} \\ \delta_{\lambda^1 y_i.E_{i+1}} &= \delta_{E_i} \rightarrow \delta_M \ . \end{aligned}$$

By equating the two right-hand sides, we obtain

$$\beta_{y_i} \rightarrow \delta_{E_{i+1}} = \delta_{E_i} \rightarrow \delta_M \ .$$

Equating corresponding arguments of the $(\rightarrow)$ functor produces the following two equalities:

$$\begin{aligned} \beta_{y_i} &= \delta_{E_i} \\ \delta_M &= \delta_{E_{i+1}} \ . \end{aligned}$$

The first equality is identical to the equality produced by the previous formulation for the redex M. The second equality states that the type of the entire redex is equal to the type of the body of the abstraction part. Applied inductively to a θ-normal expression, the second equality states that the type of the large nested redex in the expression (i.e., the entire expression, with the λ^2-abstractions stripped off the front) is exactly the type of the body of the innermost λ^1-abstraction—this fact is reflected in the original procedure by reporting as the result type the type of the innermost λ^1-abstraction. Similarly, the inequalities introduced by λ^2-abstractions in our new translation are directly reflected in the original type-recovery procedure, as the type assigned to each λ^2-abstraction is prefixed onto the computed result type to form the final derived type.

As the above discussion shows, the generalized translation procedure we have presented in this section is equivalent to the original procedure, when applied to θ-normal terms. Our next step is to examine the behaviour of this translation on terms that are not θ-normal.

4.1.2 θ_1, θ_3, and θ_4

Now that we have a translation procedure that is, at least syntactically, applicable to arbitrary terms, we must investigate its soundness, or lack thereof, on non-θ-normal terms. Recall the four θ-reduction rules:

- (θ_1) $((\lambda^1 y.N)P)Q \to (\lambda^1 y.NQ)P$
- (θ_2) $\lambda^3 z.(\lambda^1 y.N)P \to (\lambda^1 v.\lambda^3 z.(N'))(\lambda^3 w.(P'))$, where $N' = N[vz/y]$, $P' = P[w/z]$, and v and w are fresh variables
- (θ_3) $N((\lambda^1 y.P)Q) \to (\lambda^1 y.NP)Q$
- (θ_4) $(\lambda^1 y.(\lambda^2 x.N))P \to \lambda^2 x.((\lambda^1 y.N)P)$

Examining these rules, we see that rules θ_1, θ_3, and θ_4 appear to be relatively straightforward rearrangements of terms, while rule θ_2 appears to be more complex. Consequently, in this section, we will consider the effect of our translation on terms that are removed from θ-normal form by a sequence of reductions involving only rules θ_1, θ_3, and θ_4. We will consider the effect of θ_2 in a later section.

Rule θ_1

We must show that our translation procedure produces the same, or an equivalent, SUP instance for a term, before and after one step of θ_1-reduction. Let $M = ((\lambda^1 y.N)P)Q$ and $M' = (\lambda^1 y.NQ)P$, under the assumption that bound variables are named distinctly from each other, and from all free variables. In addition to the inequalities derived from N, P, and Q themselves, the SUP instance for M contains the following inequalities:

$$\begin{aligned} \delta_{(\lambda^1 y.N)P} &= \delta_Q \to \delta_M \\ \delta_{\lambda^1 y.N} &= \delta_P \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \beta_y \to \delta_N \ . \end{aligned}$$

Meanwhile, the SUP instance for M' contains the following inequalities:

$$\begin{aligned} \delta_{\lambda^1 y.NQ} &= \delta_P \to \delta_{M'} \\ \delta_{\lambda^1 y.NQ} &= \beta_y \to \delta_{NQ} \\ \delta_N &= \delta_Q \to \delta_{NQ} \ . \end{aligned}$$

In the first instance, the variable $\delta_{\lambda^1 y.N}$ does not occur anywhere in the SUP instance other than in the two inequalities presented above. So we may substitute it out and produce an equivalent instance, and similarly for $\delta_{\lambda^1 y.NQ}$ in the second instance. Doing so produces the inequalities

$$\begin{aligned} \delta_{(\lambda^1 y.N)P} &= \delta_Q \to \delta_M \\ \beta_y \to \delta_N &= \delta_P \to \delta_{(\lambda^1 y.N)P} \end{aligned}$$

for M, and the inequalities

$$\begin{aligned} \delta_P \to \delta_{M'} &= \beta_y \to \delta_{NQ} \\ \delta_N &= \delta_Q \to \delta_{NQ} \end{aligned}$$

for M'. Equivalently, we may write these sets of inequalities as

$$\begin{aligned} \delta_{(\lambda^1 y.N)P} &= \delta_Q \to \delta_M \\ \beta_y &= \delta_P \\ \delta_N &= \delta_{(\lambda^1 y.N)P} \end{aligned}$$

and

$$\begin{aligned} \delta_P &= \beta_y \\ \delta_{M'} &= \delta_{NQ} \\ \delta_N &= \delta_Q \to \delta_{NQ} \ , \end{aligned}$$

respectively. Since the equivalence between β_y and δ_P is common to the two instances (and P lies within the same scope in both M and M'), we may omit it and focus on the remaining sets of inequalities:

$$\begin{aligned} \delta_{(\lambda^1 y.N)P} &= \delta_Q \to \delta_M \\ \delta_N &= \delta_{(\lambda^1 y.N)P} \end{aligned}$$

and

$$\begin{aligned} \delta_{M'} &= \delta_{NQ} \\ \delta_N &= \delta_Q \to \delta_{NQ} \ . \end{aligned}$$

As before, since $\delta_{\lambda^1 y.N)P}$ has no occurrences outside the first pair of inequalities, we may substitute it away, and similarly for δ_{NQ} in the second pair of inequalities. Doing so leaves us with

$$\delta_N = \delta_Q \to \delta_M$$

and

$$\delta_N = \delta_Q \to \delta_{M'} .$$

Since the inequalities associated with N and Q are necessarily the same (the fact that, after reduction, Q lies within the scope of y makes no difference because of our assumption of unique naming of bound variables; hence, no capture can occur), we conclude that the same value will be derived for $\delta_{M'}$ in the second instance as for δ_M in the first instance. Hence, our translation procedure is robust with respect to θ_1-reduction.

Rule θ_3

Let $M = N((\lambda^1 y.P)Q)$ and $M' = (\lambda^1 y.NP)Q$, under the assumption that bound variables are named distinctly from each other, and from all free variables. In addition to the inequalities derived from N, P, and Q themselves, the SUP instance for M contains the following inequalities:

$$\begin{aligned} \delta_N &= \delta_{(\lambda^1 y.P)Q} \to \delta_M \\ \delta_{\lambda^1 y.P} &= \delta_Q \to \delta_{(\lambda^1 y.P)Q} \\ \delta_{\lambda^1 y.P} &= \beta_y \to \delta_P . \end{aligned}$$

The SUP instance for M' contains the following inequalities:

$$\begin{aligned} \delta_{\lambda^1 y.NP} &= \delta_Q \to \delta_{M'} \\ \delta_{\lambda^1 y.NP} &= \beta_y \to \delta_{NP} \\ \delta_N &= \delta_P \to \delta_{NP} . \end{aligned}$$

Since $\delta_{\lambda^1 y.P}$ and $\delta_{\lambda^1 y.NP}$ only occur in the above sets of inequalities, we may substitute them out, obtaining

$$\begin{aligned} \delta_N &= \delta_{(\lambda^1 y.P)Q} \to \delta_M \\ \beta_y \to \delta_P &= \delta_Q \to \delta_{(\lambda^1 y.P)Q} \end{aligned}$$

and

$$\begin{aligned} \beta_y \to \delta_{NP} &= \delta_Q \to \delta_{M'} \\ \delta_N &= \delta_P \to \delta_{NP} \ , \end{aligned}$$

equivalently,

$$\begin{aligned} \delta_N &= \delta_{(\lambda^1 y.P)Q} \to \delta_M \\ \beta_y &= \delta_Q \\ \delta_P &= \delta_{(\lambda^1 y.P)Q} \end{aligned}$$

and

$$\begin{aligned} \beta_y &= \delta_Q \\ \delta_{NP} &= \delta_{M'} \\ \delta_N &= \delta_P \to \delta_{NP} \ . \end{aligned}$$

Because Q sits within the same scope in both M and M', the occurrences of the equality $\beta_y = \delta_Q$ in the two instances are equivalent, and we may therefore focus our attention solely on the remaining sets of inequalities:

$$\begin{aligned} \delta_N &= \delta_{(\lambda^1 y.P)Q} \to \delta_M \\ \delta_P &= \delta_{(\lambda^1 y.P)Q} \end{aligned}$$

and

$$\begin{aligned} \delta_{NP} &= \delta_{M'} \\ \delta_N &= \delta_P \to \delta_{NP} \ . \end{aligned}$$

Since the only occurrences of δ_{NP} and $\delta_{(\lambda^1 y.P)Q}$ are within these sets of inequalities, we may safely substitute them out, and obtain

$$\delta_N = \delta_P \to \delta_M$$

and

$$\delta_N = \delta_P \to \delta_{M'} \ .$$

Since the inequalities associated with N and P are necessarily the same (the fact that N is outside the scope of y in M and inside the scope of y in M' makes no difference because of the uniqueness of bound variable names), the same value will be derived for $\delta_{M'}$ in the second instance as for δ_M in the first instance. Hence, our translation procedure is robust with respect to θ_2-reduction.

Rule θ_4

Let $M = (\lambda^1 y.\lambda^2 x.N)P$ and $M' = \lambda^2 x.((\lambda^1 y.N)P)$, under the assumption that bound variables are named distinctly from each other, and from all free variables. In addition to the inequalities derived from N and P themselves, the SUP instance for M contains the following inequalities:

$$\begin{aligned} \delta_{\lambda^1 y.\lambda^2 x.N} &= \delta_P \to \delta_M \\ \delta_{\lambda^1 y.\lambda^2 x.N} &= \beta_y \to \delta_{\lambda^2 x.N} \\ \delta_{\lambda^2 x.N} &= \beta_x \to \delta_N \ , \end{aligned}$$

and the SUP instance for M' contains the following inequalities:

$$\begin{aligned} \delta_{M'} &= \beta_x \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \delta_P \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \beta_y \to \delta_N \ . \end{aligned}$$

Since $\delta_{\lambda^1 y.\lambda^2 x.N}$ and $\delta_{\lambda^1 y.N}$ only occur in the above sets of inequalities, we may substitute them out and obtain

$$\begin{aligned} \delta_P \to \delta_M &= \beta_y \to \delta_{\lambda^2 x.N} \\ \delta_{\lambda^2 x.N} &= \beta_x \to \delta_N \end{aligned}$$

and

$$\begin{aligned} \delta_{M'} &= \beta_x \to \delta_{(\lambda^1 y.N)P} \\ \delta_P \to \delta_{(\lambda^1 y.N)P} &= \beta_y \to \delta_N \ ; \end{aligned}$$

equivalently,

$$\begin{aligned} \delta_P &= \beta_y \\ \delta_M &= \delta_{\lambda^2 x.N} \\ \delta_{\lambda^2 x.N} &= \beta_x \to \delta_N \end{aligned}$$

and

$$\begin{aligned} \delta_{M'} &= \beta_x \rightarrow \delta_{(\lambda^1 y.N)P} \\ \delta_P &= \beta_y \\ \delta_{(\lambda^1 y.N)P} &= \delta_N \ . \end{aligned}$$

Since P does not lie within x's scope in M, we know that P contains no occurrences of x; hence the fact that P lies within x's scope in M' does not affect its type, and does not change the set of SUP inequalities associated with P. Hence, the two equalities $\delta_P = \beta_y$ above are equivalent, and we may focus our attention on the remaining sets of inequalities,

$$\begin{aligned} \delta_M &= \delta_{\lambda^2 x.N} \\ \delta_{\lambda^2 x.N} &= \beta_x \rightarrow \delta_N \end{aligned}$$

and

$$\begin{aligned} \delta_{M'} &= \beta_x \rightarrow \delta_{(\lambda^1 y.N)P} \\ \delta_{(\lambda^1 y.N)P} &= \delta_N \ . \end{aligned}$$

Finally, since the only occurrences of the variables $\delta_{\lambda^2 x.N}$ and $\delta_{(\lambda^1 y.N)P}$ are within the above two sets of inequalities, we may safely substitute them out and obtain

$$\delta_M = \beta_x \rightarrow \delta_N$$

and

$$\delta_{M'} = \beta_x \rightarrow \delta_N \ .$$

Since N occurs within the same scope in both expressions, the type derived for it will be the same in both cases. Further, the type for β_x (which will either come from a type environment or be defaulted to $\perp$) is also the same in both cases. Hence the same type is derived for both M and M', and our translation is robust with respect to θ_4-reduction.

4.1.3 The Trouble with θ_2

There is one very important difference between rule θ_2 and the other three rules: in the case of θ_1, θ_3, and θ_4, a subexpression previously not within scope of some variable is pushed inward, so

that in the reduced expression, it does fall within the variable's scope. Because of the assumption of unique naming of variables, moving the subexpression inward does not affect its type. In the case of θ_2, the subexpression P is being pulled *out* of the scope of the λ^3-variable z. Hence, the free occurrences of z in P must be bound to a new λ^3-variable w, and the association between w and z must be patched into the subexpression N via the new λ^1-variable v.

To see what, if any, effect this kind of transformation has on the SUP translation, we will attempt to repeat the argument we used for θ_1, θ_3, and θ_4. Let $M = \lambda^3 z.(\lambda^1 y.N)P$ and $M' = (\lambda^1 v.\lambda^3 z.N[vz/y])(\lambda^3 w.P[w/z])$. Then in addition to the inequalities derived for N and P themselves, we have

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \delta_P \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \beta_y \to \delta_N \end{aligned}$$

for M, and

$$\begin{aligned} \delta_{\lambda^1 v.\lambda^3 z.N[vz/y]} &= \delta_{\lambda^3 w.P[w/z]} \to \delta_{M'} \\ \delta_{\lambda^1 v.\lambda^3 z.N[vz/y]} &= \beta_v \to \delta_{\lambda^3 z.N[vz/y]} \\ \delta_{\lambda^3 z.N[vz/y]} &= \gamma_z \to \delta_{N[vz/y]} \\ \delta_{\lambda^3 w.P[w/z]} &= \gamma_w \to \delta_{P[w/z]} \end{aligned}$$

for M'. Since $\delta_{\lambda^1 y.N}$ and $\delta_{\lambda^1 v.\lambda^3 z.N[vz/y]}$ only occur in the above sets of inequalities, we may substitute them out and obtain

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{(\lambda^1 y.N)P} \\ \beta_y \to \delta_N &= \delta_P \to \delta_{(\lambda^1 y.N)P} \end{aligned}$$

for M, and

$$\begin{aligned} \delta_{\lambda^3 w.P[w/z]} \to \delta_{M'} &= \beta_v \to \delta_{\lambda^3 z.N[vz/y]} \\ \delta_{\lambda^3 z.N[vz/y]} &= \gamma_z \to \delta_{N[vz/y]} \\ \delta_{\lambda^3 w.P[w/z]} &= \gamma_w \to \delta_{P[w/z]} \end{aligned}$$

for M'. Splitting the equalities yields

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{(\lambda^1 y.N)P} \\ \beta_y &= \delta_P \\ \delta_N &= \delta_{(\lambda^1 y.N)P} \end{aligned}$$

and

$$\begin{aligned} \delta_{\lambda^3 w.P[w/z]} &= \beta_v \\ \delta_{M'} &= \delta_{\lambda^3 z.N[vz/y]} \\ \delta_{\lambda^3 z.N[vz/y]} &= \gamma_z \to \delta_{N[vz/y]} \\ \delta_{\lambda^3 w.P[w/z]} &= \gamma_w \to \delta_{P[w/z]} \ . \end{aligned}$$

Since $\delta_{(\lambda^1 y.N)P}$, $\delta_{\lambda^3 z.N[vz/y]}$, and $\delta_{\lambda^3 w.P[w/z]}$ only occur in the above sets of inequalities, we may substitute them out, leaving ourselves with

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_N \\ \beta_y &= \delta_P \end{aligned}$$

and

$$\begin{aligned} \delta_{M'} &= \gamma_z \to \delta_{N[vz/y]} \\ \beta_v &= \gamma_w \to \delta_{P[w/z]} \ . \end{aligned}$$

It is now our task to show that these two sets of inequalities are equivalent; specifically we must show that the same value is computed for δ_M as for $\delta_{M'}$. Naively, we could argue as follows: each occurrence of y in N in the first instance gives rise to an inequality of the form $\beta_y \leq \delta_{y_i}$, where i denotes the i-th occurrence of y. In the second instance y_i is replaced by the application $v_i z_i$, which gives rise to the inequalities

$$\begin{aligned} \beta_v &\leq \delta_{v_i} \\ \gamma_z &= \delta_{z_i} \\ \delta_{v_i} &= \delta_{z_i} \to \delta_{v_i z_i} \ . \end{aligned}$$

Now, since $P[w/z]$ is simply P under variable renaming, we have $\delta_{P[w/z]} = \delta_P$. Since $v_i z_i$ replaces each y_i in N in the second instance, we take $\delta_{v_i z_i} = \delta_{y_i}$ for each i. From $\delta_P = \beta_y$ in the first

instance, we obtain $\delta_P \leq \delta_{y_i}$ for each occurrence y_i of y (this comes from the inequality $\beta_y \leq \delta_{y_i}$, which is part of the translation procedure). Now, comparing the two equations $\delta_{v_i} = \delta_{z_i} \to \delta_{v_i z_i}$ and $\beta_v = \gamma_w \to \delta_{P[w/z]}$, since $\beta_v \leq \delta_{v_i}$, we have

$$\gamma_w \to \delta_{P[w/z]} \leq \delta_{z_i} \to \delta_{v_i z_i} \ ,$$

which implies that

$$\delta_{P[w/z]} \leq \delta_{v_i z_i} \ .$$

Identifying $\delta_{P[w/z]}$ with δ_P and $v_i z_i$ with y_i, we get

$$\delta_P \leq \delta_{y_i} \ ,$$

which is also implied by the inequalities

$$\begin{aligned} \beta_y &= \delta_P \\ \beta_y &\leq \delta_{y_i} \end{aligned}$$

in the first instance. Hence any solution of the second instance is also a solution of the first instance. However, the converse is not true—for it is *not* the case that

$$\delta_{P[w/z]} \leq \delta_{v_i z_i}$$

implies that

$$\gamma_w \to \delta_{P[w/z]} \leq \delta_{z_i} \to \delta_{v_i z_i} \ .$$

Hence, a solution of the first instance is not guaranteed (at least by this reasoning) to be a solution of the second instance, and therefore, the new translation may deem terms typable that are not typable under the original translation.

To see concretely that indeed the two translations are not equivalent under θ_2, consider the program[1]:

$$M = \lambda^3 z.(\lambda^1 y.yy)z \ .$$

Here, we have $P = z$ and $N = yy$. The θ_2-reduced version is

$$M' = (\lambda^1 v.\lambda^3 z.(vz)(vz))(\lambda^3 w.w) \ .$$

[1]In a self-contained program, the abstractions would not be labelled this way. In particular, the z would be a λ^2-variable. However, z can be made a λ^3-variable by prefixing the program with $(\lambda^1 x.x)$. We elide this prefix in the presentation here, because it is not important to the discussion.

For M', the previously-derived inequalities $\delta_{M'} = \gamma_z \to \delta_{N[vz/y]}$ and $\beta_v = \gamma_w \to \delta_{P[w/z]}$, combined with the inequalities for $N[vz/y]$ and $P[w/z]$, give rise to the following SUP instance:

$$\begin{aligned}
\delta_{M'} &= \gamma_z \to \delta_{(vz)(vz)} \\
\beta_v &= \gamma_w \to \delta_w \\
\delta_{v_1 z_1} &= \delta_{v_2 z_2} \to \delta_{(vz)(vz)} \\
\delta_{v_1} &= \delta_{z_1} \to \delta_{v_1 z_1} \\
\delta_{v_2} &= \delta_{z_2} \to \delta_{v_2 z_2} \\
\beta_v &\leq \delta_{v_1} \\
\beta_v &\leq \delta_{v_2} \\
\gamma_z &= \delta_{z_1} \\
\gamma_z &= \delta_{z_2} \\
\gamma_w &= \delta_w \ .
\end{aligned}$$

Substituting the last three equalities yields

$$\begin{aligned}
\delta_{M'} &= \gamma_z \to \delta_{(vz)(vz)} \\
\beta_v &= \gamma_w \to \gamma_w \\
\delta_{v_1 z_1} &= \delta_{v_2 z_2} \to \delta_{(vz)(vz)} \\
\delta_{v_1} &= \gamma_z \to \delta_{v_1 z_1} \\
\delta_{v_2} &= \gamma_z \to \delta_{v_2 z_2} \\
\beta_v &\leq \delta_{v_1} \\
\beta_v &\leq \delta_{v_2} \ .
\end{aligned}$$

Substitution on β_v, δ_{v_1}, and δ_{v_2} yields

$$\begin{aligned}
\delta_{M'} &= \gamma_z \to \delta_{(vz)(vz)} \\
\delta_{v_1 z_1} &= \delta_{v_2 z_2} \to \delta_{(vz)(vz)} \\
\gamma_w \to \gamma_w &\leq \gamma_z \to \delta_{v_1 z_1} \\
\gamma_w \to \gamma_w &\leq \gamma_z \to \delta_{v_2 z_2} \ .
\end{aligned}$$

From here we can see that the last two inequalities force the variables $\delta_{v_1 z_1}$ and $\delta_{v_2 z_2}$ to be equal to each other, by virtue of their both being unified with γ_z. But the second inequality implies

that $\delta_{v_1 z_1}$ and $\delta_{v_2 z_2}$ cannot be equal; hence, we have a contradiction. Therefore, the instance has no solution, and consequently, the program is deemed untypable.

For M, the previously-derived inequalities $\delta_M = \gamma_z \to \delta_N$ and $\beta_y = \delta_P$, combined with the inequalities for N and P, give rise to the following SUP instance:

$$\begin{aligned}
\delta_M &= \gamma_z \to \delta_{yy} \\
\beta_y &= \delta_z \\
\delta_{y_1} &= \delta_{y_2} \to \delta_{yy} \\
\beta_y &\leq \delta_{y_1} \\
\beta_y &\leq \delta_{y_2} \\
\gamma_z &= \delta_z \ .
\end{aligned}$$

Substituting out the second and last inequalities yields

$$\begin{aligned}
\delta_M &= \gamma_z \to \delta_{yy} \\
\delta_{y_1} &= \delta_{y_2} \to \delta_{yy} \\
\gamma_z &\leq \delta_{y_1} \\
\gamma_z &\leq \delta_{y_2} \ .
\end{aligned}$$

Then, substitution on the second inequality yields

$$\begin{aligned}
\delta_M &= \gamma_z \to \delta_{yy} \\
\gamma_z &\leq \delta_{y_2} \to \delta_{yy} \\
\gamma_z &\leq \delta_{y_2} \ .
\end{aligned}$$

We see now that this instance is solvable, via the substitution $[\gamma_z \to \delta_{yy}/\delta_M]$. Note that the auxiliary substitutions σ_2 and σ_3, for the second and third inequalities, are $[\delta_{y_2} \to \delta_{yy}/\gamma_z]$ and $[\delta_{y_2}/\gamma_z]$, respectively. In particular, they map γ_z to two different values, which will turn out to be an important observation.

Here, then, is a concrete example that shows that our direct SUP translation types more terms than the original translation. Hence, either our direct translation is unsound in this translation, or the original is incomplete. Given, however, that the term $(\lambda^3 z.(\lambda^1 y.yy)z)$ β-reduces to $\lambda^3 z.zz$, which is untypable because z is forced to be monomorphic, we are forced to conclude that the term should indeed be untypable, and therefore, that our syntax-directed translation procedure is unsound on terms that are not θ_2-normal.

4.1.4 What Went Wrong?

A key feature of terms that are not θ_2-normal is that they contain λ^3-abstractions that outscope λ^1-abstractions. These broadly-scoped monomorphic abstractions then allow for the possibility that a polymorphic variable, like y in our example, becomes bound, via β-reduction, to a monomorphic variable (z in our example). Thus y, being polymorphic, is used in the expression yy in a completely valid way, but by becoming bound to the monomorphic variable z, y effectively becomes monomorphic itself, thus rendering the expression yy invalid.

Having pinpointed the difficulty with syntax-directed translation from typability to SUP—namely monomorphic variables that outscope polymorphic abstractions—our task is now to find a way to express the monomorphism inherited by polymorphic variables like y above within a SUP (or SUP-like) setting. We develop a syntax-directed translation with this property in the next section.

4.2 Unknowns

Consider again the reduced SUP instance that comes from our syntax-directed translation of the term $M = (\lambda^3 z.(\lambda^1 y.yy)z)$:

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{yy} \\ \gamma_z &\leq \delta_{y_2} \to \delta_{yy} \\ \gamma_z &\leq \delta_{y_2} \ . \end{aligned}$$

This reduced instance has a solution because the definition of SUP allows us to supply a different substitution for γ_z between the second and third inequalities—but if γ_z represents the type of a monomorphic variable, then perhaps it would be better regarded as a constant. In the context of SUP, the notion of "constant" is modelled by nullary functors like *Int* and *Bool*, which are certainly monomorphic types. Indeed, if γ_z were treated as a nullary constructor, rather than as a variable, then we could immediately call the instance unsolvable, because the second inequality would contain a functor mismatch between γ_z and $\to$.

Ignoring this functor mismatch for the moment, another property of this instance (under the assumption that γ_z is a nullary functor) is noteworthy. In particular, if γ_z is a nullary functor, and therefore not a variable, then there is a redex-I in the last inequality, and the instance reduces

under redex-I reduction to

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{yy} \\ \gamma_z &\leq \gamma_z \to \delta_{yy} \\ \gamma_z &\leq \gamma_z \ . \end{aligned}$$

Here we have retained the last inequality, even though it is solved, for illustrative purposes. Notice how the redex-reduction has caused γ_z to propagate to *both* sides of both the second and third inequalities. Such an occurrence would normally disqualify any instance from being either acyclic or R-acyclic, and indeed might signal non-termination. Here, however, since γ_z is a functor, there is no problem; other functors, like *Int*, *Bool*, and $\to$, have no restrictions on where they may occur, and the same must now be true for γ_z. On the other hand, we now have *two* reasons to call this instance unsolvable: in addition to the functor mismatch we previously pointed out between γ_z and $\to$, we now also have an occurs-check violation in the second inequality, because a constant can certainly not become an expression containing itself after substitution, especially since substitutions have no effect on constants! Though this second disqualification of the instance—the occurs-check violation—may seem trivial, especially in light of the first disqualification, it will turn out to be the more important of the two.

Although treating γ_z as a constant correctly renders the SUP instance from our example unsolvable, our previous experience with SUP and type inference suggests that this measure solves one problem only to create another. Consider, for example, the expression

$$(\lambda^1 y.yy)(\lambda^3 z.z) \ .$$

If we worked through the corresponding SUP instance, we would find that the derived type of the subexpression $(\lambda^3 z.z)$ is $\gamma_z \to \gamma_z$. Equating β_y with $\gamma_z \to \gamma_z$ gives rise to the following partially-reduced instance:

$$\begin{aligned} \gamma_z \to \gamma_z &\leq \delta_{y_2} \to \delta_{yy} \\ \gamma_z \to \gamma_z &\leq \delta_{y_2} \ . \end{aligned}$$

Reducing the redex-I in the second inequality and the redex-II in the first inequality yields

$$\begin{aligned} \gamma_z \to \gamma_z &\leq (\alpha \to \alpha) \to (\alpha \to \alpha) \\ \gamma_z \to \gamma_z &\leq \alpha \to \alpha \ . \end{aligned}$$

This instance is solved, *provided that γ_z is treated as a variable*—the first inequality becomes an equation via the substitution $[\alpha \to \alpha/\gamma_z]$, while the second inequality becomes an equation via $[\alpha/\gamma_z]$. If, on the other hand, γ was a constant, then the second inequality would force $\alpha = \gamma_z$ and the first inequality would cause a functor mismatch between γ_z and $\to$ (and also an occurs-check violation).

We have, therefore, an analogous situation to our previous example: a term involving a λ^3-abstraction, whose SUP instance contains the monomorphic variable γ_z. In both cases, treating γ_z as a constant yielded a SUP instance with no solution, while treating γ_z as a variable rendered the instance solvable. In our previous example, we argued that treating γ_z as a constant yielded a correct answer (i.e., no solution). In this case however, even though z is a monomorphic entity represented by a monomorphic type variable γ_z, the *abstraction* $\lambda^3 z.z$ is a *polymorphic* entity, and its type, $\gamma_z \to \gamma_z$, is a polymorphic type. Further, if we look at the computation expressed by our example term, $(\lambda^1 y.yy)(\lambda^3 z.z)$, we see that we are simply applying an identity function, which is polymorphic, to itself. Hence, this example, unlike the previous example, should type-check.

In summary, we find ourselves in a situation where certain identifiers (the γ-variables) are sometimes more appropriately treated as constants (i.e., nullary functors) and sometimes more appropriately treated as variables. This observation itself strongly suggests that SUP may not be the right unification-like problem to serve as the target of a syntax-directed translation from typability. However, before we consider a generalization of SUP that fits our needs, we must examine more closely whether treating γ_z as a constant is truly the right solution, even for our first example.

Consider now the following source program[2]:

$$M = \lambda^3 z.\lambda^3 w.zw \ .$$

In this example, the types of the monomorphic term variables z and w are given by the monomorphic type variables γ_z and γ_w, respectively. This source program gives rise to the following

[2]Again, we elide a preceding $(\lambda^1 x.x)$ because it does not contribute to the example, except to force the remaining abstractions to be λ^3-abstractions.

unreduced SUP instance:

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{\lambda^3 w.zw} \\ \delta_{\lambda^3 w.zw} &= \gamma_w \to \delta_{zw} \\ \delta_z &= \delta_w \to \delta_{zw} \\ \gamma_z &= \delta_z \\ \gamma_w &= \delta_w \ . \end{aligned}$$

Here, γ_z and γ_w are used in reference to the variables z and w themselves; hence, consistency with our previous example demands that we treat γ_z and γ_w as constants. We then reduce the instance by first substituting out the variables δ_z and δ_w:

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{\lambda^3 w.zw} \\ \delta_{\lambda^3 w.zw} &= \gamma_w \to \delta_{zw} \\ \gamma_z &= \gamma_w \to \delta_{zw} \ . \end{aligned}$$

Our treatment of γ_z and γ_w as constants would now lead us to conclude that there is a functor mismatch in the third inequality between γ_z and $\to$, and therefore that the term is not typable. However, we know from experience with rank 1 type inference that this term must be typable, and indeed should have the type $\forall\alpha.\forall\beta.(\alpha \to \beta) \to (\alpha \to \beta)$. Further, the original ASUP translation procedure would map the source term in this example to a solvable ASUP instance that yields precisely this type. Hence, naively treating γ-variables as constants, even in contexts where they refer to the term variables themselves, rather than the enclosing abstractions, creates incompleteness in our translation procedure.

What we observe, then, is that even when γ-variables denote monomorphic entities, and therefore behave more as constants than as variables, their behaviour is not quite consistent with that of constants either—unlike constants, γ-variables must be eligible for substitution, at least in some limited fashion. We conclude that γ-variables, when acting as monomorphic entities, constitute a new class of identifier, more constant than a variable, and more variable than a constant. In reality their behaviour can be accurately described as constants whose identities are not yet known—the substitutions we applied in our last example serve to reveal, step-by-step, more and more about the identities of these constants, without ever requiring a constant to undergo mutually incompatible substitutions (as in our second example, where γ_z acts as a variable). For this reason, we call these identifiers *unknowns*.

4.3 SUP with Unknowns

Our investigation in the previous section culminates in the following extension of SUP to accommodate unknowns:

Definition 4.1 (USUP) *An instance of USUP (i.e., SUP with Unknowns) is a set* $\{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^{N}$ *of pairs, each consisting of an inequality* $\tau_i \leq \mu_i$ *and a (possibly empty) set* $\vec{\alpha_i}$ *of identifier names, in some term algebra. A substitution* σ *is a solution of USUP if there exist substitutions* $\sigma_1, \ldots, \sigma_N$ *such that*

$$\begin{aligned} \tau_1\sigma\sigma_1 &= \mu_1\sigma \\ &\ldots \\ \tau_N\sigma\sigma_N &= \mu_N\sigma \ , \end{aligned}$$

and for each i, $\mathrm{dom}(\sigma_i) \cap \mathrm{Vars}(\vec{\alpha_i}\sigma) = \emptyset$.

The sets $\vec{\alpha_i}$ in the definition represent the identifiers in each inequality that function as unknowns; those identifiers not contained in $\vec{\alpha_i}$ function as ordinary variables. These sets $\vec{\alpha_i}$ capture our observation that identifiers that function as unknowns (i.e., γ-variables) do so only within limited contexts; elsewhere, they are simply ordinary variables. Hence we are led to associate a set of unknowns to each inequality, rather than globally.

The final restriction on the domain of σ_i captures our previous discussion about what kinds of substitutions unknowns may undergo. As outlined by this restriction, the key feature of unknowns is that they may be replaced by the global solution substitution σ, but they may not be replaced by the auxiliary substitutions σ_i. Moreover, an unknown replaced by substitution can only be replaced with another unknown, or an expression with only unknowns at the leaves. In particular, an unknown cannot become a variable (or an expression containing a variable) as a result of substitution—this is because, if an unknown $\underline{\gamma}$ in $\tau_i \leq \mu_i$ is replaced by an expression τ, then all variables of τ become part of $\vec{\alpha_i}$, and therefore become unknowns.

Before we explore USUP in detail, we introduce a notational convention. The sets $\vec{\alpha_i}$ are useful for presenting USUP formally, but cumbersome when working with concrete USUP instances. For this reason, we adopt the convention that identifiers denoting unknowns in a particular inequality will be underlined. For example, in the inequality

$$\alpha \to \underline{\beta} \ \leq \ (\underline{\gamma} \to \underline{\gamma}) \to \delta \ ,$$

the identifiers β and γ denote unknowns. Of course, consistency demands that within a single inequality, either all occurrences of a given variable be underlined, or no occurrences be underlined. For example, we would not be permitted to underline only one of the occurrences of γ above. In the context of discussion, we will also adopt the convention (unless stated otherwise) that underlined identifiers denote unknowns.

4.3.1 Reduction Rules

In order to actually solve instances of USUP, we introduce two new reduction rules to augment the existing redex procedure. We first define some notation:

Definition 4.2 *Let μ be a term. Denote by μ^* the result of consistently replacing all variables in μ with fresh unknowns.*

Then the following two reductions constitute our addition to the redex procedure to accommodate unknowns (over an assumed USUP instance $\Gamma = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^N$, and recall that σ_k denotes the substitution performed on the k-th iteration of the procedure):

- (Redex-III reduction) Let $1 \leq i \leq N$ and Σ be a minimal path such that $\Sigma(\tau_i\sigma_k)$ contains only unknowns at the leaves (i.e., $\text{Vars}(\Sigma(\tau_i\sigma_k)) \subseteq \text{Vars}(\vec{\alpha_i}\sigma_k)$) and $\Sigma(\mu_i\sigma_k)$ exists, but is not equal to $\Sigma(\tau_i\sigma_k)$. If $\Sigma(\tau_i\sigma_k)$ and $\Sigma(\mu_i\sigma_k)$ are not unifiable, terminate with failure. Else, set $\sigma_{k+1} = \theta \circ \sigma_k$, where θ is the most general unifier of $\Sigma(\tau_i\sigma_k)$ and $\Sigma(\mu_i\sigma_k)$. Add 1 to k, and go to step 2.

- (Redex-IV reduction) Let Σ be a path and $1 \leq i \leq N$ be such that $\Sigma(\mu_i\sigma_k)$ is an unknown (i.e., $\Sigma(\mu_i\sigma_k) \in \text{Vars}(\vec{\alpha_i}\sigma_k)$) and $\Sigma(\tau_i\sigma_k)$ is not a variable (i.e., is either a functor application or an unknown). Set $\sigma_{k+1} = [\Sigma(\tau_i\sigma_k)^*/\Sigma(\mu_i\sigma_k)] \circ \sigma_k$. Add 1 to k, and go to step 2.

In words, anytime an unknown is matched against an expression on the other side of an inequality, it is replaced with a structural copy of the matching expression, but with all variables consistently renamed to fresh unknowns (the word "variable" here is specifically meant to exclude unknowns, although the expression $\text{Vars}(\tau)$ will denote the collection of all variables *and* unknowns in τ).

In addition, the existing rules must be reinterpreted under the presence of unknowns. In the case of Redex-I reduction, the phrase "not a variable" now includes not only functor applications, but also unknowns. In the case of Redex-II reduction, the issue lies with what it means to unify terms that might contain unknowns. For the purpose of unification, we will treat unknowns

simply as variables, so that no extensions to Robinson's algorithm are needed. Although this view of unknowns during unification may seem to allow an unknown to be replaced by a variable, it in fact does not—as we can see from the reduction rules above, when a substitution is applied to the inequalities in the SUP instance, it is also applied to the lists of unknowns. Hence, if an unknown $\underline{\gamma}$ is replaced during unification by a variable β, then in the sets $\vec{\alpha_i}$, occurrences of γ will also be replaced by β, thus turning β into an unknown, and producing an instance identical (up to renaming) to what would have resulted if we had insisted that $\underline{\gamma}$ replace β, rather than the reverse.

Finally, the introduction of unknowns to the problem means that we must expand our application of the occurs-check. Currently it only appears in the context of unification as part of redex-II reduction. Now, however, we must include the following check, after every redex reduction: if Σ and Π are paths, such that $\Sigma(\tau_i\sigma_{k+1}) = \Pi\Sigma(\mu_i\sigma_{k+1}) = \underline{\gamma}$ or $\Pi\Sigma(\tau_i\sigma_{k+1}) = \Sigma(\mu_i\sigma_{k+1}) = \underline{\gamma}$, for some unknown $\underline{\gamma}$, then the algorithm fails due to occurs-check violation.

4.3.2 Examples Recast

Having formalized USUP and its associated reduction rules, we can now revisit our examples from Section 4.2, this time treating the γ-variables as unknowns when they denote monomorphic entities. Our motivating example was $M = (\lambda^3 z.(\lambda^1 y.yy)z)$. Its USUP instance is

$$\begin{aligned}
\delta_M &= \gamma_z \to \delta_{yy} \\
\beta_y &= \delta_z \\
\delta_{y_1} &= \delta_{y_2} \to \delta_{yy} \\
\beta_y &\leq \delta_{y_1} \\
\beta_y &\leq \delta_{y_2} \\
\underline{\gamma_z} &= \delta_z \ .
\end{aligned}$$

Although $\underline{\gamma_z}$ only occurs within the last inequality, the identifier γ_z would be regarded as an unknown in all but the first inequality, since all of the last five inequalities arise from syntax lying strictly within the λ^3-abstraction, where γ_z can only refer to the monomorphic variable z.

Therefore, reduction produces the following reduced instance:

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{yy} \\ \underline{\gamma_z} &\leq \delta_{y_2} \to \delta_{yy} \\ \underline{\gamma_z} &\leq \delta_{y_2} \ . \end{aligned}$$

The above instance arises from simply reducing according to the redex-I and redex-II rules from before. Now, applying the redex-III rule to final inequality equates δ_{y_2} with $\underline{\gamma_z}$, thus producing the instance

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{yy} \\ \underline{\gamma_z} &\leq \underline{\gamma_z} \to \delta_{yy} \ . \end{aligned}$$

Notice that we can no longer conclude that the instance is unsolvable simply because there is a functor mismatch between $\underline{\gamma_z}$ on the left and $\to$ on the right, because these now simply indicate the presence of a redex-III. Instead, we conclude that the instance is unsolvable because an unknown is being compared with an expression involving itself, which is an occurs-check violation.

Also worth noting is that we could have reduced the second inequality before attempting to reduce the third inequality. In this case, we would obtain

$$\begin{aligned} \delta_M &= (\alpha_1 \to \alpha_2) \to \delta_{yy} \\ \underline{\alpha_1} \to \underline{\alpha_2} &\leq \delta_{y_2} \to \delta_{yy} \\ \underline{\alpha_1} \to \underline{\alpha_2} &\leq \delta_{y_2} \ . \end{aligned}$$

Redex-III reduction on the second inequality yields

$$\begin{aligned} \delta_M &= (\alpha_1 \to \alpha_2) \to \alpha_2 \\ \underline{\alpha_1} \to \underline{\alpha_2} &\leq \underline{\alpha_1} \to \underline{\alpha_2} \\ \underline{\alpha_1} \to \underline{\alpha_2} &\leq \underline{\alpha_1} \ , \end{aligned}$$

and now we have an occurs-check violation in the third inequality. Either way, we conclude that the term is untypable.

Our second example was $M = (\lambda^1 y.yy)(\lambda^3 z.z)$. The full USUP instance for this term is as

follows:

$$\begin{aligned}
\delta_{\lambda^1 y.yy} &= \delta_{\lambda^3 z.z} \to \delta_M \\
\delta_{\lambda^1 y.yy} &= \beta_y \to \delta_{yy} \\
\delta_{y_1} &= \delta_{y_2} \to \delta_{yy} \\
\beta_y &\leq \delta_{y_1} \\
\beta_y &\leq \delta_{y_2} \\
\delta_{\lambda^3 z.z} &= \gamma_z \to \delta_z \\
\underline{\gamma_z} &= \delta_z
\end{aligned}$$

In this case, γ_z is only regarded as an unknown in the last inequality. In all other inequalities, γ_z occurs in a context in which it refers to the abstraction $\lambda^3 z.z$, which is polymorphic; hence γ_z is simply a variable. Reduction of the last inequality replaces δ_z with γ_z throughout the instance; we are left with

$$\begin{aligned}
\delta_{\lambda^1 y.yy} &= \delta_{\lambda^3 z.z} \to \delta_M \\
\delta_{\lambda^1 y.yy} &= \beta_y \to \delta_{yy} \\
\delta_{y_1} &= \delta_{y_2} \to \delta_{yy} \\
\beta_y &\leq \delta_{y_1} \\
\beta_y &\leq \delta_{y_2} \\
\delta_{\lambda^3 z.z} &= \gamma_z \to \gamma_z \ .
\end{aligned}$$

Now the only inequality in which γ_z acts as an unknown is solved, and removed from the presentation. We are left with an instance that reduces, just as before, to

$$\begin{aligned}
\gamma_z \to \gamma_z &\leq (\alpha \to \alpha) \to (\alpha \to \alpha) \\
\gamma_z \to \gamma_z &\leq \alpha \to \alpha \ ,
\end{aligned}$$

which, as before, is solvable.

Our third example was $M = \lambda^3 z.\lambda^3 w.zw$. Its USUP instance is

$$\begin{aligned} \delta_M &= \gamma_z \to \delta_{\lambda^3 w.zw} \\ \delta_{\lambda^3 w.zw} &= \gamma_w \to \delta_{zw} \\ \delta_z &= \delta_w \to \delta_{zw} \\ \underline{\gamma_z} &= \delta_z \\ \underline{\gamma_w} &= \delta_w \ , \end{aligned}$$

in which γ_w is an unknown in all but the first two inequalities, and γ_z is an unknown in all but the first inequality. Ordinary redex-I and redex-II reduction yields

$$\begin{aligned} \delta_M &= \gamma_z \to (\gamma_w \to \delta_{zw}) \\ \underline{\gamma_z} &= \delta_w \to \delta_{zw} \\ \underline{\gamma_w} &= \delta_w \ . \end{aligned}$$

The last two inequalities now contain a redex-III's, whose reduction yields

$$\begin{aligned} \delta_M &= \gamma_z \to (\gamma_w \to \delta_{zw}) \\ \underline{\gamma_z} &= \underline{\gamma_w} \to \delta_{zw} \ , \end{aligned}$$

and then

$$\begin{aligned} \delta_M &= (\gamma_w \to \delta_{zw}) \to (\gamma_w \to \delta_{zw}) \\ \underline{\gamma_w} \to \underline{\delta_{zw}} &= \underline{\gamma_w} \to \underline{\delta_{zw}} \ . \end{aligned}$$

Reading off the first inequality gives us precisely the type we would have obtained from ordinary rank 1 or ML-style type inference.

4.4 Properties of USUP Reduction

In this section we prove several properties of USUP and the USUP reduction procedure, in order to establish USUP as a viable basis for translation from rank 2 typability.

Our first observation is straightforward:

Theorem 4.1 *USUP is undecidable.*

The result follows simply because SUP may be viewed as a subset of USUP in which there are no unknowns, or equivalently in which $\alpha_i = \emptyset$ for all i.

Before we pursue questions of termination and decidability any further, however, we first establish some basic correctness results about our reduction procedure.

4.4.1 Soundness and Completeness

There are two notions of soundness and completeness surrounding our presentation of USUP. The first is whether the USUP translation of a source term faithfully and completely captures the typability of the term. The second is whether the solutions output by our reduction procedure are consistent with the definition of a solution of USUP. We treat the latter in this section, and defer the former until Section 4.5.2, once we have formally specified the translation from typability to USUP.

Soundness of the Reduction Procedure

In order to establish soundness for our reduction procedure, we must show that it never outputs wrong answers. In other words, when it produces a solution, it must be a solution according Definition 4.1, and it must not produce a solution for an instance that does not have one.

Lemma 4.1 *Suppose an instance* Γ *of USUP is not solved. Then, in the absence of functor mismatches,* Γ *contains either a redex-I, redex-II, redex-III, or redex-IV.*

Proof If $\Gamma = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^{N}$ is not solved, then for some i, τ_i is not a substitution instance of μ_i via a substitution whose domain only contains variables. Let Σ be as large as possible such that $\Sigma(\tau_i)$ is not a substitution instance of $\Sigma(\mu_i)$ (again, considering only substitutions whose domains contain only variables). Let $\tau = \Sigma(\tau_i)$ and $\mu = \Sigma(\mu_i)$. Then τ is not a variable (otherwise, τ is trivially a substitution instance of μ). On the other hand, μ can be one of:

- a variable—then there is a redex-I reduced by $[\tau'/\mu]$.
- a functor application $f(\mu_{i_1}, \ldots, \mu_{i_n})$—then because there are no functor mismatches, if τ is also a functor application, then $\tau = f(\tau_{i_1}, \ldots, \tau_{i_n})$, for some $\tau_{i_1}, \ldots, \tau_{i_n}$. By maximality of Σ, each τ_{i_j} is a substitution instance of μ_{i_j} via a substitution σ_{i_j} whose domain only contains variables. Now, if all of the σ_{i_j} are compatible for $1 \leq j \leq n$, then τ itself is a substitution instance of μ, via, for example, σ_{i_1}, which is a contradiction. Hence at least

two of the σ_{i_j} are not compatible. For notational convenience, suppose σ_{i_1} and σ_{i_2} are incompatible. Then there exists a variable β such that $\beta\sigma_{i_1} \neq \beta\sigma_{i_2}$. Let Σ_1 and Σ_2 be paths that, when applied to τ, produce these two occurrence of β. If $\Sigma_1(\mu)$ and $\Sigma_2(\mu)$ both exist, then $\Sigma_1(\mu) \neq \Sigma_2(\mu)$, otherwise, $\beta\sigma_{i_1} = \beta\sigma_{i_2}$. Hence, there is a redex-II, reduced by $\mathrm{MGU}(\Sigma_1(\mu), \Sigma_2(\mu))$. If, for example, $\Sigma_1(\mu)$ does not exist, then let Π be the largest prefix of Σ_1 such that $\Pi(\mu)$ exists. Then $\Pi(\mu)$ is either a variable or an unknown, and $\Pi(\tau)$ is a compound expression. If $\Pi(\mu)$ is a variable, we have a redex-I, reduced by $[\Pi(\tau)'/\Pi(\mu)]$. If $\Pi(\mu)$ is an unknown, we have a redex-IV, reduced by $[\Pi(\tau)^*/\Pi(\mu)]$.

On the other hand, if τ is *not* a functor application, then τ must be an unknown. In this case, let Σ' be the smallest possible prefix of Σ such that $\Sigma'(\tau_i)$ contains only unknowns at the leaves (possibly Σ' is Σ itself). Then there is a redex-III at $\Sigma'(\tau_i)$, reduced by the most general unifier of $\Sigma'(\tau_i)$ and $\Sigma'(\mu_i)$.

- an unknown—if τ is a functor application, then there is a redex-IV, reduced by $[\tau^*/\mu]$. Otherwise, τ is an unknown. We cannot have $\tau = \mu$, by construction of Σ. Hence, $\tau \neq \mu$, and again there is a redex-IV, reduced by $[\tau^*/\mu]$, which is simply $[\tau/\mu]$.

As there are no other possibilities, the result follows—if an instance with no functor mismatches is not solved, it contains a redex. □

Theorem 4.2 *If, for a USUP instance* Γ*, the USUP redex procedure outputs a substitution* σ *on termination, then* σ *is a solution of* Γ*.*

Proof If the USUP instance terminates, then $\Gamma\sigma$, the result of applying the returned substitution σ throughout Γ, must not contain any redexes. Further, it contains no functor mismatches, as otherwise, the procedure, upon encountering no redexes, would have signalled a functor mismatch. Then by Lemma 4.1, $\Gamma\sigma$ must be a solved instance (i.e., it has the identity substitution as solution). Hence, σ is a solution of Γ. □

The following statement is immediate, but worth noting:

Corollary 4 *If a USUP instance is unsolvable, the redex procedure either loops forever, or outputs an error.*

Proof This is simply the contrapositive of Theorem 4.2—if the procedure does not loop or produce an error, then it must produce a solution. By Theorem 4.2, that solution solves the instance,

and in particular the instance is solvable. Since the instance is *not* solvable, the procedure must then either loop or output an error.

These results together establish soundness for our reduction procedure—it never reports a solution when one does not exist, and any substitution it produces is guaranteed to be a solution of the problem instance.

Completeness of the Reduction Procedure

In considering the notion of completeness for our procedure with respect to the definition of USUP, we must be careful, as we already know USUP to be undecidable, and therefore no solution procedure can be both sound and complete—that is, no solution procedure can solve all USUP instances without also producing false solutions. Since we have already shown our procedure to be sound, it cannot also be complete.

Instead, we restrict our attention to the set of USUP instances upon which the reduction procedure terminates (itself an undecidable set), and prove completeness on that set. Our completeness results are consequences of the following general theorem:

Theorem 4.3 *Let* $\Gamma = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^{N}$ *be a solvable USUP instance, upon which the USUP redex procedure terminates, and let* σ_0 *be a substitution performed by one iteration of the redex procedure. Then* $\Gamma\sigma_0 = \{\langle \tau_i\sigma_0 \leq \mu_i\sigma_0, \mathrm{Vars}(\alpha_i\vec{\sigma}_0)\rangle\}_{i=1}^{N}$ *is solvable. Moreover, if* σ *is any solution of the problem instance, then, when the domains are restricted to variables in* Γ*, we have* $\sigma = \sigma' \circ \sigma_0$ *for some substitution* σ'*. More precisely,* $\sigma|_{\mathrm{Vars}(\Gamma)} = (\sigma' \circ \sigma_0)|_{\mathrm{Vars}(\Gamma)}$.

Proof Let Γ be solvable. Then there is a substitution σ for which there are substitutions σ_i such that for all i,

$$\tau_i\sigma\sigma_i = \mu_i\sigma \text{ and } \mathrm{dom}(\sigma_i) \cap \mathrm{Vars}(\vec{\alpha_i}\sigma) = \emptyset \ .$$

There are four cases to consider:

- Redex-I reduction. For some i and path Σ, $\Sigma(\tau_i) = \tau$ for some expression τ, where τ is not a variable, and $\Sigma(\mu_i) = \alpha$ for some variable α. Since $\tau_i\sigma\sigma_i = \mu_i\sigma\sigma_i$, we have $\alpha\sigma = \tau\sigma\sigma_i$. $\alpha\sigma$ must be at least as big as $\tau\sigma$, otherwise there is no solution. Therefore, σ, restricted to Vars(Γ), can be written as $(\sigma' \circ [\tau'/\alpha])|_{Vars(\Gamma)}$ (where τ' is the consistent replacement of variables in τ by fresh ones) for some σ'. Since $[\tau'/\alpha]$ is the reduction σ_0 our procedure performs, the result follows for this case.

- Redex-II reduction. For some i and paths Σ_1 and Σ_2, we have $\Sigma_1(\tau_i) = \Sigma_2(\tau_i) = \alpha$, and $\Sigma_1(\mu_i) \neq \Sigma_2(\mu_i)$. Call these μ_{i_1} and μ_{i_2}, respectively. We have $\mu_1\sigma = \alpha\sigma\sigma_i = \mu_2\sigma$, so σ unifies μ_{i_1} and μ_{i_2}. Therefore, $\sigma = \sigma' \circ \sigma_U$, where $\sigma_U = \mathrm{MGU}(\mu_{i_1}, \mu_{i_2})$. Since σ_U is the reduction σ_0 we perform, the result holds for redex-II reduction.

- Redex-III reduction. For some i and minimal path Σ, we have $\Sigma(\tau_i) = \tau$ and $\Sigma(\mu_i) = \mu$, where τ is an expression containing only unknowns at the leaves and μ is any expression not equal to τ. Then $\tau\sigma\sigma_i = \mu\sigma$. Since σ cannot map an unknown to any expression containing variables, and σ_i cannot have unknowns in its domain, we have $\tau\sigma\sigma_i = \tau\sigma$. Therefore, σ unifies $\underline{\gamma}$ and μ. As a result, $\sigma = \sigma' \circ \sigma_U$ for some substitution σ', where $\sigma_U = \mathrm{MGU}(\tau, \mu)$. Since σ_U is the reduction σ_0 we perform, the result holds for redex-III reduction.

- Redex-IV reduction. For some i and path Σ, we have $\Sigma(\mu_i) = \underline{\gamma}$ and $\Sigma(\tau_i) = \tau$, where $\underline{\gamma}$ is an unknown and τ is any expression. Then $\tau\sigma\sigma_i = \underline{\gamma}\sigma$. $\underline{\gamma}\sigma$ must be at least as big as τ, and must be an expression involving only unknowns. Therefore σ, when restricted to $\mathrm{Vars}(\Gamma)$, can be written as some substitution composed with $\sigma_0 = [\tau^*/\gamma]$ (both sides restricted, as usual, to $\mathrm{Vars}(\Gamma)$), which is what the reduction rule prescribes. Hence, the result holds for redex-IV reduction.

Since this is the entire set of redex forms we reduce, our procedure does not render a solvable instance unsolvable. □

Corollary 5 *Let Γ be a solvable USUP instance, on which the USUP redex procedure terminates. Then the procedure produces a solution for Γ.*

Proof By repeated application of the theorem, each reduction results in a solvable instance. Therefore, when the procedure terminates upon finding no more redexes, the instance is solvable. Therefore, there will be no functor mismatches (and there can be no occurs-check violations in the absence of a redex). Then the only other possibility is that the redex procedure succeeds and returns a substitution on termination. Therefore, the procedure produces a solution for all solvable instances on which it terminates (correctness of this solution follows from our soundness result). □

Corollary 6 *Let Γ be a solvable USUP instance on which the redex procedure terminates and let σ be the solution produced by the procedure (whose existence is guaranteed by the previous corollary). Let σ' be any other solution of Γ. Then there exists a substitution σ'' such that $\sigma'|_{\mathrm{Vars}(\Gamma)} = (\sigma'' \circ \sigma)|_{\mathrm{Vars}(\Gamma)}$.*

To aid in the proof of the corollary, we introduce the following notation:

Definition 4.3 (Range of a Substitution) *Given a substitution σ define*

$$\mathrm{ran}(\sigma) = \bigcup_{\alpha \in \mathrm{dom}(\sigma)} \mathrm{Vars}(\alpha\sigma) \ .$$

Proof of Corollary 6 Given a set V of variables and a substitution σ such that $\mathrm{dom}(\sigma) \subseteq V$, define a function f on σ and V as follows:

$$f(\sigma, V) = \mathrm{ran}(\sigma) \cup (V \setminus \mathrm{dom}(\sigma)) \ .$$

The application $f(\sigma, V)$ denotes the set of values upon which a substitution σ' may act such that $(\sigma' \circ \sigma)|_V \neq \sigma$. Now, let σ_i denote the substitution performed by the i-th iteration of the redex procedure. Let $V_0 = \mathrm{Vars}(\Gamma)$. Then, from the theorem, we have

$$\sigma|_{V_0} = (\sigma' \circ \sigma_1)|_{V_0} \ ,$$

i.e.,

$$\sigma|_{V_0} = (\sigma'|_{f(\sigma_1, V_0)} \circ \sigma_1|_{V_0})|_{V_0} \ .$$

Now, for all $i > 0$, define $V_i = f(\sigma_i, V_{i-1})$, so that the equation becomes

$$\sigma|_{V_0} = (\sigma'|_{V_1} \circ \sigma_1|_{V_0})|_{V_0} \ .$$

Now, since σ solves Γ, so does $\sigma|_{V_0}$, and therefore, σ'_{V_1} solves $\Gamma\sigma_1$ (note that $\sigma_1|_{V_0} = \sigma_1$, by construction). Therefore, by a second application of the theorem,

$$\sigma'|_{V_1} = (\sigma''|_{V_2} \circ \sigma_2|_{V_1})|_{V_1} \ ,$$

for some substitution σ''. A third application of the theorem gives

$$\sigma''|_{V_2} = (\sigma'''|_{V_3} \circ \sigma_3|_{V_2})|_{V_2} \ ,$$

for some substitution σ'''. If the procedure performs n reductions before terminating, then the n-th application of the theorem gives

$$\sigma^{(n-1)}|_{V_{n-1}} = (\sigma^{(n)}|_{V_n} \circ \sigma_n|_{V_{n-1}})|_{V_{n-1}} \ ,$$

for some substitution $\sigma^{(n)}$. Substituting these equations into one another yields

$$\begin{aligned}
\sigma|_{V_0} &= (((\cdots(\sigma^{(n)}|_{V_n} \circ \sigma_n|_{V_{n-1}})|_{V_{n-1}} \circ \cdots \circ \sigma_3|_{V_2})|_{V_2} \circ \sigma_2|_{V_1})|_{V_1} \circ \sigma_1|_{V_0})|_{V_0} \\
&= (((\cdots(\sigma^{(n)} \circ \sigma_n)|_{V_{n-1}} \circ \cdots \circ \sigma_3|_{V_2})|_{V_2} \circ \sigma_2|_{V_1})|_{V_1} \circ \sigma_1|_{V_0})|_{V_0} \\
&= \cdots \\
&= (((\sigma^{(n)} \circ \sigma_n \circ \cdots \circ \sigma_3)|_{V_2} \circ \sigma_2|_{V_1})|_{V_1} \circ \sigma_1|_{V_0})|_{V_0} \\
&= (\sigma^{(n)} \circ \sigma_n \circ \cdots \circ \sigma_1)|_{V_0} \\
&= (\sigma^{(n)} \circ (\sigma_n \circ \cdots \circ \sigma_1))|_{V_0} \ .
\end{aligned}$$

Since $\sigma_n \circ \cdots \circ \sigma_1$ is the substitution returned by the procedure, we obtain the desired result. □

Thus, the USUP redex procedure is not only complete for the set of instances on which it terminates, but the solutions it outputs are most general semiunifiers (MGSU's) for the problem instance.

4.4.2 Termination

Having established soundness and completeness of the USUP redex procedure for instances on which it terminates, we now turn our attention to characterizing the set of instances on which the procedure terminates. As this set is not decidable, our goal will, of course, be to produce large, interesting, decidable subsets of the full set.

Our first termination result is that, in analogy to SUP, the redex procedure for USUP terminates on all USUP instances that possess a solution:

Theorem 4.4 *Let Γ be a USUP instance that possesses a solution. Then the USUP redex procedure terminates when applied to Γ.*

Proof Given two substitutions σ_1 and σ_2, and a USUP instance Γ, we define $\sigma_1 >_\Gamma \sigma_2$ iff

$$\sum_{\alpha \in \mathrm{Vars}(\Gamma)} |\alpha\sigma_1| > \sum_{\alpha \in \mathrm{Vars}(\Gamma)} |\alpha\sigma_2| \ ,$$

or

$$\sum_{\alpha \in \mathrm{Vars}(\Gamma)} |\alpha\sigma_1| = \sum_{\alpha \in \mathrm{Vars}(\Gamma)} |\alpha\sigma_2| \quad \text{and} \quad |\mathrm{Vars}(\Gamma\sigma_1)| < |\mathrm{Vars}(\Gamma\sigma_2)| \ ,$$

where, for an expression τ, $|\tau|$ denotes its size:

$$\begin{aligned} |\alpha| &= 1 \\ |f(\tau_1, \ldots, \tau_n)| &= 1 + \sum_{i=1}^{n} |\tau_i| \ . \end{aligned}$$

We show that the procedure must maintain the invariant that $\sigma \geq_\Gamma \sigma_k$, where σ is any solution of the instance Γ, and σ_k is the accumulated substitution after k iterations of the procedure, and moreover, $\sigma_k >_\Gamma \sigma_{k-1}$ for all $k > 1$. Since $>_\Gamma$ may be rephrased as simply a lexicographic ordering on the tuple

$$\left\langle \sum_{\alpha \in \mathrm{Vars}(\Gamma)} |\alpha\sigma_k|,\ |\mathrm{Vars}(\Gamma\sigma)| - |\mathrm{Vars}(\Gamma\sigma_k)| \right\rangle \ ,$$

it is a well-ordering, and therefore the above conditions are sufficient to guarantee termination.

Let $V = \mathrm{Vars}(\Gamma)$. For the first part of the claim, suppose σ is a solution of the instance and σ_k is the result of k iterations of the algorithm. Then by k-fold application of Theorem 4.3, we have that there exists a substitution σ' such that

$$\sigma|_V = (\sigma' \circ \sigma_k)|_V \ .$$

Suppose $\sigma \not\geq_\Gamma \sigma_k$. Then there are two possibilities:

- $\sum_{\alpha \in V} |\alpha\sigma| < \sum_{\alpha \in V} |\alpha\sigma_k|$. Since composing an additional substitution σ' onto σ_k can only replace identifiers at the leaves of $\alpha\sigma_k$, for each α, with other variables or with compound expressions (or unknowns), composing σ' onto σ_k cannot reduce σ_k's total size over variables in V; hence it cannot reduce σ_k's total size over V to that of σ, and therefore it cannot satisfy $\sigma|_V = (\sigma' \circ \sigma_k)|_V$, which contradicts our previous assertion.

- $\sum_{\alpha \in V} |\alpha\sigma| = \sum_{\alpha \in V} |\alpha\sigma_k|$, and $|\mathrm{Vars}(\Gamma\sigma)| > |\mathrm{Vars}(\Gamma\sigma_k)|$. Composing an additional substitution σ' onto σ_k replaces identifiers at the leaves of $\alpha\sigma_k$, for each α, with compound expressions or identifiers (i.e., variables or unknowns). The former is impossible, as it would increase the total size over V of σ_k beyond that of σ, which, as we have already seen is impossible. In the latter case, the total size of σ_k over V remains unchanged, as identifiers are simply replaced with other identifiers. In doing so, a substitution may unify two identifiers, thus reducing the total number of identifiers in the reduced instance, but it cannot split two occurrences of the same identifier into two different identifiers. Thus,

composing a substitution σ' onto σ_k cannot decrease the number of variables in the reduced instance, and therefore $|\text{Vars}(\Gamma\sigma_k\sigma')|$ cannot be equal to $|\text{Vars}(\Gamma\sigma)|$. Thus, σ' cannot satisfy $\sigma|_V = (\sigma' \circ \sigma_k)|_V$, which contradicts our previous assertion.

Thus, indeed, any solution σ of Γ satisfies $\sigma \geq_\Gamma \sigma_k$ for all k.

It remains to show that each redex reduction makes positive progress towards a solution σ, according to $>_\Gamma$, i.e., that for all $k > 1$, $\sigma_k >_\Gamma \sigma_{k-1}$, where σ_k and σ_{k-1} denote, respectively, the accumulated substitutions produced by k and $k-1$ iterations of the algorithm. We then have that $\sigma_k = \sigma_{k,k-1} \circ \sigma_{k-1}$, where $\sigma_{k,k-1}$ is the redex reduction performed by the k-th iteration of the procedure. The proof then proceeds according to the type of replacement performed by $\sigma_{k,k-1}$:

- $\sigma_{k,k-1}$ replaces a variable α with a compound expression. Then $\alpha \in V_k$, since the algorithm, on the k-th iteration, only replaces variables in V_k. Hence, there is a variable $\beta \in V$ such that $|\beta\sigma_k| > |\beta\sigma_{k-1}|$; hence $\sigma_k > \Gamma\sigma_{k-1}$.

- $\sigma_{k,k-1}$ replaces a variable α with an identifier (unknown or variable). Then the total size of σ_k over V is the same as that of σ_{k-1}. Note that $\sigma_{k,k-1}$ will not replace α with a fresh variable—the only reduction that generates fresh variables is redex-I reduction, and even within redex-I reduction, fresh variables are only generated when α is replaced by a functor application that contains variables. Moreover, the identifier with which $\sigma_{k,k-1}$ replaces α is not α itself—the redex procedure never outputs an identity substitution. The only remaining possibility is that $\alpha\sigma_{k,k-1}$ is an identifier $\beta \in V_k$. Thus $\sigma_{k,k-1}$ unifies the identifiers α and β, so that $|\text{Vars}(\Gamma\sigma_k)| = |\text{Vars}(\Gamma\sigma_{k-1})| - 1$. Therefore, $\sigma_k >_\Gamma \sigma_{k-1}$.

- $\sigma_{k,k-1}$ is an MGU from redex-II or redex-III reduction. Then $\sigma_{k,k-1}$ may be viewed as a sequence of substitutions (from successive iterations of Robinson's algorithm), each of which has one of the two forms above. Then the result follows by the above arguments.

Having exhausted the possible forms of $\sigma_{k,k-1}$, we conclude that in all cases, $\sigma_k >_\Gamma \sigma_{k-1}$. This, combined with the fact that any solution σ of Γ bounds all σ_k from above under $>_\Gamma$, and that the double induction is well-ordered, implies termination of the USUP redex procedure on all solvable instances. □

This last result implies that the USUP instances that give rise to non-termination all have no solution. Our task is now to prove that, even among the unsolvable instances, we still have termination for subsets of interest.

The problem of termination for unsolvable instances is considerably more subtle than for solvable instances—this is at least partly because USUP is undecidable, and therefore infinitely many unsolvable instances must cause the reduction procedure to loop forever.

We would ultimately like to find a graph-theoretic treatment of USUP instances, analogous to our treatment of SUP, upon which to base our termination argument. The principal difficulty in establishing such a formulation lies in deciding how to account for unknowns in the structure of the graph of a given USUP instance. While we might like, for example, to treat unknowns like ordinary variables, such a treatment would mean that inequalities like $\alpha \to \underline{\gamma} \leq \underline{\gamma} \to \beta$, in which a unknown $\underline{\gamma}$ appears on both sides, would contain self-loops, and therefore violate all notions of acyclicity heretofore considered. We would, however, like to be able to allow an unknown to appear on both sides of the same inequality—in fact, redex-I, redex-III, and redex-IV reduction, which by definition involve copying unknowns across an inequality, depend on this ability. At the same time, we must have a notion of acyclicity (one that is pertinent to our intended application, namely rank 2 type inference) to go with our formulation of USUP graphs; otherwise our graph-theoretic basis for proving termination for SUP becomes useless in the context of USUP.

One possible approach to treating unknowns in a graph-theoretic setting might be to exclude them from consideration—since they are not strictly variables, they would not be allowed to contribute edges to the graph. An intuitive justification for this approach might be that since unknowns are meant to model constants (i.e., nullary functors like *Int* and *Bool*), which do not contribute edges to the graph, unknowns should similarly be excluded. This approach is not completely satisfactory, however, because unlike true constants, unknowns can be replaced during reductions, and therefore information can flow from one vertex to another via the replacement of an unknown.

As an example of an unknown reduction triggering an infinite reduction sequence, consider the following instance:

$$\begin{aligned} \alpha \to \alpha &\leq \beta \\ \beta &\leq \underline{\alpha}\,. \end{aligned}$$

If unknowns are not permitted to contribute edges, then the USUP graph for this instance is simply a single edge pointing from the first inequality to the second inequality, and is clearly R-acyclic. The first inequality contains a redex-I, whose reduction yields

$$\begin{aligned} \alpha \to \alpha &\leq \gamma \to \gamma \\ \gamma \to \gamma &\leq \underline{\alpha}\,. \end{aligned}$$

There is now a redex-IV in the second inequality, whose reduction yields

$$\begin{aligned}(\delta \to \delta) \to (\delta \to \delta) &\leq \gamma \to \gamma \\ \gamma \to \gamma &\leq \underline{\delta} \to \underline{\delta}\ ,\end{aligned}$$

and the re-emergence of redex-I's in the first inequality makes it clear that this sequence of reductions will not terminate. On the other hand, if the unknown $\underline{\alpha}$ had been allowed to contribute an edge, then the graph would have exhibited a cycle between the two vertices, and the instance would have been rejected as a consequence.

A key feature of unknowns, not possessed by ordinary variables, is that they permit bidirectional information flow across an inequality. With ordinary variables, conditions on the left-hand side of an inequality can cause replacements on the right-hand side, but not vice versa. (We might consider this to be a left-to-right flow of information.) The same is possible for unknowns. Consider, for example, the following inequality:

$$\alpha \to \beta \ \leq\ \underline{\gamma}\ .$$

In this inequality, there is a redex-IV, whose reduction yields the following solved inequality:

$$\alpha \to \beta \ \leq\ \underline{\delta} \to \underline{\epsilon}\ ,$$

where $\underline{\delta}$ and $\underline{\epsilon}$ are fresh unknowns.

With unknowns, however, information flow in the reverse direction is also possible. Consider the same inequality, reversed:

$$\underline{\gamma}\ \leq\ \alpha \to \beta\ .$$

This inequality contains a redex-III, whose reduction yields

$$\underline{\alpha} \to \underline{\beta}\ \leq\ \underline{\alpha} \to \underline{\beta}\ .$$

This time, it is a condition on the *right-hand* side of the inequality that leads to a replacement on the *left-hand* side. Any graph-theoretic formulation of USUP, one would expect, then, must somehow be able to capture this kind of bi-directional information flow that can arise during unknown reduction.

One possible approach to constructing the graph of an instance containing unknowns might be to take each inequality $\tau \leq \mu$, in which an unknown appears, and add both the vertex $\tau \leq \mu$ and its reversal $\mu \leq \tau$ to the graph. In this way, since the inequality is present in both directions,

the bidirectional information flow of unknowns is definitely captured. However, this approach also implies a bidirectional information flow for ordinary variables, which is simply not the case. The resulting graph, therefore, would be overly pessimistic about non-termination. Furthermore, by including both a vertex and its reversal in the graph, cycles will surely be introduced into the graph structure, and these would somehow need to be explained away as harmless (if this is indeed the case).

Alternatively, we might consider mapping a USUP instance into several graphs, each of which contains either $\tau \leq \mu$ or $\mu \leq \tau$ as a vertex, for each inequality $\tau \leq \mu$ that features an unknown. This approach, however, leads to an exponentially large number of graphs for a USUP instance of even modest size. Though USUP graphs are primarily a theoretical tool to facilitate reasoning, it may be advantageous to some program analysis applications to actually construct the graphs, which would prove intractable with an exponentially large number of USUP graphs. Further, such a formulation is overly *optimistic*—within a single given inequality, unknown information may flow in *both* directions. Hence, we could not be certain of catching all non-terminating instances with an acyclicity argument.

Our approach, instead, is based on the following observation: because an unknown $\underline{\gamma}$ is only eligible for replacement within the solution substitution σ, and not within the auxiliary substitutions $\sigma_1, \ldots, \sigma_N$, any inequality in which γ appears on a left-hand side actually functions as an equality. For example, the inequality

$$\underline{\gamma} \ \leq \ \beta$$

behaves in precisely the same way as the equality

$$\underline{\gamma} \ = \ \beta \ ,$$

which is by definition identical to the inequality

$$\alpha \to \alpha \ \leq \ \underline{\gamma} \to \beta \ ,$$

for some variable α. For more complex inequalities, in which the unknown $\underline{\gamma}$ does not appear on its own on the left-hand side, this transformation is not possible; however, its spirit is still present in the behaviour of $\underline{\gamma}$ under replacement in any inequality in which it appears on the left. In general, any unknown, whether it occurs on the left-hand side or the right-hand side of an inequality, is really a citizen of the right-hand side. Motivated by this observation, then, we present the following graph-theoretic characterization of USUP instances:

Definition 4.4 (Graph of a USUP instance) *Let* $\Gamma = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^{N}$ *be a USUP instance. The* graph of Γ*, denoted* $G(\Gamma)$*, is defined as follows:*

- *the inequalities* $\tau_i \leq \mu_i$ *are the vertices* v_i *in* G*;*
- $v_i \rightarrow v_j$ *iff* $(\mathrm{RVars}(v_i) \cup (\mathrm{LVars}(v_i) \cap \vec{\alpha_i})) \cap (\mathrm{LVars}(v_j) \setminus \vec{\alpha_j}) \neq \emptyset$.

In the absence of unknowns (i.e., if each $\vec{\alpha_i} = \emptyset$), this definition is identical to the original for SUP (Definition 3.4). When unknowns are present, however, regardless of which side of the inequality actually contains them, they are treated as if they occur on the right-hand side.

Having defined the graph of a USUP instance, we now obtain definitions of acyclic semiunification with unknowns (AUSUP) and R-acyclic semiunification with unknowns (R-AUSUP) for free.

Under this characterization, the USUP instance in our first example,

$$\begin{aligned} \alpha \rightarrow \alpha &\leq \beta \\ \beta &\leq \underline{\alpha} \,, \end{aligned}$$

would have a graph in which each vertex had an edge leading to the other; hence, the graph would be cyclic, and therefore rejected. If we reversed the second inequality, thereby obtaining the instance

$$\begin{aligned} \alpha \rightarrow \alpha &\leq \beta \\ \underline{\alpha} &\leq \beta \,, \end{aligned}$$

reduction of the redex-III in the second inequality would immediately create an occurs-check violation, and terminate the redex procedure. What is noteworthy about this example is that if we had simply treated unknowns as ordinary variables for the purpose of constructing the graph, the resulting graph would have had no edges, and therefore would have been trivially R-acyclic. As it is, there is now an edge from the second inequality to the first (since the unknown $\underline{\alpha}$ is regarded as being on the right-hand side, but the variable α remains on the left). Since the variable β occurs on the right-hand sides of both inequalities, which are connected by an edge, the graph cannot be R-acyclic, and is therefore rejected before the occurs-check violation arises.

Under this graph-theoretic characterization of USUP, we can allow unknowns to occur on both sides of the same inequality without creating a self-loop in the graph—since the left-hand

side occurrences of the unknown are regarded as right-hand side occurrences for the purposes of the graph, there is no longer any basis for constructing an edge from the vertex to itself.

The most important property of R-acyclicity under USUP is that, like its counterpart under SUP, it is an invariant under redex reduction:

Theorem 4.5 (Invariance of R-acyclicity for USUP) *Let Γ be a USUP instance, and let Γ' be the result of performing one iteration of the USUP redex procedure on Γ (i.e., the result of reducing one redex in Γ). If $G(\Gamma)$ is R-acyclic, then $G(\Gamma')$ is R-acyclic.*

Proof The proof dispatches on the kind of redex reduced in Γ. Suppose the redex occurs in the inequality $\tau_i \leq \mu_i$. Then the argument proceeds as follows:

- redex-I. Then all occurrences of some variable α are replaced with some expression τ', containing only fresh variables, along with whatever unknowns are present in the corresponding expression τ on the left-hand side. If τ contains no unknowns, then all vertices that contained α now contain the variables (if any) of τ', and no other vertices contain these variables because they are all fresh—therefore no edges are created by this reduction. Hence, no R-cycles can be created, and G remains R-acyclic. If, on the other hand, τ contains an unknown $\underline{\gamma}$, then there are two possibilities:
  - the reduction $[\tau'/\alpha]$, where τ' contains the unknown $\underline{\gamma}$ induces, by the creation of edges, the relations $\beta_1 R \cdots R \beta_m$, and such that $\beta_m R' \beta_1$, for some $\beta_1, \ldots, \beta_m$. The introduction of $\underline{\gamma}$ in replacement of α can only create edges if there is an inequality $\tau_j \leq \mu_j$, in which γ appears on the left-hand side (i.e., within τ_j) as a simple variable. Then an edge is created from any vertex $\tau_k \leq \mu_k$ which contains α on the right-hand side, to $\tau_k \leq \mu_k$. The introduction of this edge produces $\delta R' \beta$ for any $\delta \in \mathrm{Vars}(\mu_k)$ and $\beta \in \mathrm{Vars}(\mu_j)$. However, in the inequality $\tau_i \leq \mu_i$, which contains the redex, we have $\underline{\gamma} R' \beta$, and since μ_i and μ_k both contain α, we have $\delta R \alpha R \gamma R' \beta$ before reduction. Hence the introduction of $\delta R' \beta$ cannot introduce an R-cycle that was not already there. Thus $G(\Gamma')$ is R-acyclic.
  - we had $\beta_i R \cdots R \beta_j$ and $\beta_k R \cdots R \beta_l$, for some β_i, β_j, β_k, and β_l, and the redex reduction unifies β_j and β_k, thus linking the two R-chains. In this case, if a redex-I reduction unifies β_j and β_k, then one of these two identifiers (say β_j) is the variable α, which occurs within μ_i. The other identifier (say β_k) is an unknown within τ' (without loss of generality, the unknown $\underline{\gamma}$), and occurs within τ_i. But since α occurs in μ_i, and $\underline{\gamma}$

is unknown and occurs in τ_i, we have $\alpha R \underline{\gamma}$, i.e., $\beta_j R \beta_k$. Thus, the variables β_k and β_k are R-related anyway, and any R-acyclicity violation involving these variables would have existed before they were unified. Thus, unifying these variables only results in a non-R-acyclic instance if the instance already was not R-acyclic.

- redex-II. If reduction causes G to lose R-acyclicity, then as in the case of redex-I reduction, there are two possibilities:
  - there is a variable replacement $[\tau/\alpha]$ that occurs during reduction, which induces, for some $\beta_1, \ldots \beta_n$, the relations $\beta_1 R \cdots R \beta_n$, and such that $\beta_n R' \beta_1$. Since $[\tau/\alpha]$ caused the violation, it created an edge that completed one of the paths from β_i to $\beta_{(i \bmod n)+1}$. For such an i, there is an edge from some $v_j \rightarrow v_k$ lying along this path, that was created by the substitution $[\tau/\alpha]$. Hence, one of RVars(v_j) and LVars(v_k) contains the identifier (variable or unknown) α; the other contains a variable or unknown from τ, say γ. Now, the redex $[\tau/\alpha]$ exists because of the inequality $v_i = (\tau_i \leq \mu_i)$ that satisfies the conditions for this redex-II; hence α and γ are both in RVars(v_i). Since one of α and γ is in LVars(v_k) (note that whichever of α or γ is in LVars(v_k) cannot be an unknown in v_k, otherwise the edge from v_j to v_k could not have existed in the first place), we have $v_i \rightarrow v_k$. Since either α or γ is in RVars(v_j), and both are in RVars(v_i), the transitive closure of R connects the path ending with v_j to the path beginning with v_i, and followed by v_k. Hence, the removal of the edge from v_j to $\tau_k \leq \mu_k$ does not restore R-acyclicity. Thus, removing edges introduced by redex-II reductions cannot convert graphs that are not R-acyclic to graphs that are.
  - we had $\beta_i R \cdots R \beta_j$ and $\beta_k R \cdots R \beta_l$, for some β_i, β_j, β_k, and β_l, and the redex reduction unifies β_j and β_k, thus linking the two R-chains. In this case, if a redex-II reduction unifies β_j and β_k, then these two variables (or unknowns) must occur together on the right-hand side of the inequality $v_i = (\tau_i \leq \mu_i)$, in which the redex occurs. Hence $\beta_j R \beta_k$, and we already had $\beta_i R \cdots R \beta_j R \beta_k R \cdots R \beta_l$, i.e., the two R-chains were already linked. Again, the redex-II reduction can only result in a non-R-acyclic instance if the instance was non-R-acyclic to begin with.

  In both cases, we see that redex-II reduction cannot reduce a graph that is R-acyclic to one that is not.

- redex-III. A redex-III reduction, which unifies an expression containing unknowns on the

left with an arbitrary expression on the right, may be viewed as a sequence of substitutions of the form $[\tau^*/\underline{\alpha}]$ for unknowns $\underline{\alpha}$ on the left, and redex-I reductions on the right. As we have already completed the argument for redex-I reductions, we can focus our attention on a single replacement $[\tau^*/\underline{\alpha}]$ on the left-hand side. Then, as before, there are two possibilities:

- Suppose a reduction $[\tau^*/\underline{\alpha}]$ in the vertex $v_i = (\tau_i \leq \mu_i)$ creates an edge from a vertex v_j to a vertex v_k. Let $\underline{\gamma}$ be an unknown in τ^* (if τ^* contains no unknowns, then it contains no identifiers and the replacement $[\tau^*/\alpha]$ cannot create edges). Then one of $\underline{\alpha}$ and $\underline{\gamma}$ is in RVars(v_j) (or unknown in v_j and in LVars(v_j)), and the other is in LVars(v_k) and is therefore not an unknown in v_k. On the other hand, $\underline{\alpha} \in$ LVars(v_i) and $\underline{\gamma} \in$ RVars(v_i). Thus, $v_i \rightarrow v_k$, and therefore either $\alpha R' \gamma$ or $\gamma R' \alpha$, depending on which is in LVars(v_k). But this gives, for any $\beta \in$ RVars(v_j), either $\beta R \alpha R' \gamma$ or $\beta R \gamma R' \alpha$. Hence, the edge created by the reduction is of no consequence in terms of creating non-vacuous R-cycles, and therefore, the reduced instance remains R-acyclic.
- Suppose instead that an R-acyclicity violation arises because we had $\beta_i R \cdots R \beta_j$ and $\beta_k R \cdots R \beta_l$, for some β_i, β_j, β_k, and β_l, and the redex reduction unifies β_j and β_k, thus linking the two R-chains. Then these two identifiers occur in corresponding positions on opposite sides of the inequality v_i, and both are unknown. Hence, $\beta_j R \beta_k$ anyway, and any R-acyclicity violation would therefore have already existed. Hence, the reduced instance is R-acyclic.

- redex-IV. The argument for redex-IV reduction is identical to the argument for redex-III reduction, except that τ^* (hence $\underline{\gamma}$) and $\underline{\alpha}$ now occur, respectively on the left-hand and right-hand sides of v_i. However, since unknowns are considered to reside on right-hand sides for the purpose of constructing USUP graphs, we still obtain either $\alpha R' \gamma$ or $\gamma R' \alpha$, depending on whether α or γ is in LVars(v_k).

In summary, then, the USUP redex procedure preserves R-acyclicity. □

To complete our proof of termination, we must show that an infinite redex reduction sequence cannot flow through an R-acyclic graph. Note that the two corollaries of Theorem 3.2 apply here as well, unchanged, with the same proofs. Our next task is to establish termination for single-inequality USUP instances. As a first step, we present the following result:

Definition 4.5 *We use the term* unknown reduction *to denote a redex-III or redex-IV reduction. We use the term* variable reduction *to denote a redex-I or redex-II reduction.*

Lemma 4.2 *Let* $\Gamma = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^{N}$ *be a USUP instance. Then the number of unknown reductions that can occur between successive variable reductions in* Γ *is finite.*

Proof Suppose an unknown reduction in Γ replaces an unknown $\underline{\gamma}$. The proof dispatches on the image of $\underline{\gamma}$ under the reduction:

- $\underline{\gamma}$ is replaced by a functor application τ^*. Then the total number of unknowns in the instance increases by $|\text{Vars}(\tau)| - 1$. This kind of reduction can only occur when there is a path Σ such that for some i, $\Sigma(\mu_i)$ is a functor application and $\Sigma(\tau_i)$ is an unknown (i.e., redex-III reduction) or vice versa (i.e., redex-IV reduction). Thus, the number of opportunities for this kind of reduction is bounded above by the number of paths Σ for which this condition holds, which is finite. Furthermore, reductions of this kind cannot create further opportunities for reduction, as no variables are replaced with larger expressions under this reduction (only unknowns are replaced). Thus, the paths Σ leading to unknowns grow in size, and consequently the size of corresponding variable expressions shrinks. Therefore, only finitely many of this kind of reduction can occur.

- $\underline{\gamma}$ is replaced by some other unknown $\underline{\gamma'}$. In this case, the total number of distinct unknowns in the instance decreases by one. As there are only finitely many unknowns at any given time, the number of replacements by unknowns that can occur between replacements by functor applications is finite. Further, replacing unknowns with unknowns cannot create an opportunity for further replacement by functor application because, as argued above, no variables are replaced. Therefore, the number of reductions of the first kind is finite in an absolute sense (assuming no variable reductions).

Since the number of reductions in the second category that can occur between reductions of the first kind is finite, and the total number of reductions in the first category is finite, we conclude that the total number of unknown reductions that can take place in the absence of variable reductions is finite. □

Before proceeding, it will be convenient to distinguish between two types of redex-I:

Definition 4.6 (Redex-I of the first and second kind) *A redex-I in an inequality* $\tau \leq \mu$*, given by a path* Σ *such that* $\Sigma(\mu)$ *is a variable and* $\Sigma(\tau)$ *is not a variable is said to be* of the first kind *if* $\Sigma(\tau)$ *contains at least one variable that is not an unknown, and* of the second kind *if* $\Sigma(\tau)$ *contains no variables other than unknowns.*

Note that any redex-I of the second kind may also be viewed as part of a redex-III. We now show that a single inequality can only give rise to finitely many reductions:

Lemma 4.3 *Every instance of USUP comprising a single inequality $\tau \leq \mu$, with unknowns $\vec{\alpha}$, and $(\mathrm{Vars}(\tau) \cap \mathrm{Vars}(\mu)) \setminus \vec{\alpha} = \emptyset$, is solvable by the USUP redex procedure (that is, the USUP redex procedure will terminate on such an input).*

Proof We bound the number of redex reductions that can be performed in $\tau \leq \mu$:

- *The number of redex-I reductions of the first kind in $\tau \leq \mu$ is bounded by the number of leaf nodes in τ (i.e., by the number of variable occurrences in τ).* Every redex-I reduction causes at least one variable α in τ to be matched against a variable in μ. No further reduction will ever again cause this occurrence of α to be part of a redex-I. Hence there can be no more redex-I's of the first kind than leaves in τ. (Note that, because $\mathrm{Vars}(\tau) \cap \mathrm{Vars}(\mu) = \emptyset$, redex reduction does not change τ.)

- *The number of redex-II reductions that can occur in $\tau \leq \mu$ before a redex-I reduction must occur is bounded by $|\mathrm{Vars}(\mu)|$.* This is because each redex-II reduction replaces at least one variable in μ; hence it decreases $|\mathrm{Vars}(\mu)|$ by at least 1.

- *The number of unknown reductions that can take place between successive variable reductions is finite.* This is simply Lemma 4.2.

- *The number of redex-I reductions of the second kind in $\tau \leq \mu$ that can occur before an redex-I reduction of the first kind must occur is bounded by $|\mathrm{Vars}(\mu) \setminus \vec{\alpha}|$.* Each redex-I reduction of the first kind replaces a true variable (i.e., not an unknown) in $\mathrm{Vars}(\mu)$ with an expression containing only unknowns; hence, $|\mathrm{Vars}(\mu) \setminus \vec{\alpha}|$ decreases by 1.

Since the number of unknown reductions that can occur between variable reductions is bounded, the number of redex-II reductions and redex-I reductions of the second kind that can occur between redex-I reductions of the first kind is bounded, and the total number of redex-I reductions is bounded, the USUP redex procedure must eventually terminate. □

In analogy with Lemma 3.2, we need a result that states that redexes can only be induced along edges in the USUP graph:

Lemma 4.4 *Let $\Gamma = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^{N}$ be an instance of USUP, and suppose a redex reduction σ in an inequality $v_i = (\tau_i \leq \mu_i)$ induces a redex in an inequality $v_j = (\tau_j \leq \mu_j)$. Then there is an edge from v_i to v_j in $G(\Gamma)$.*

Proof We proceed according to the kind of redex induced:

- redex-I. If a redex-I is created in v_j then for some path Σ, $\Sigma(\mu_j\sigma)$ is a variable, and $\Sigma(\tau_j\sigma)$ is not a variable. Since the redex-I did not exist previously, $\Sigma(\tau_j)$ must be a variable. Hence, σ contains the replacement $[\Sigma(\tau_j\sigma)/\Sigma(\tau_j)]$. Since redex reductions indicate replacements of variables on the right-hand sides of inequalities in which they originate, or of unknowns on either side, it follows that either $\Sigma(\tau_j) \in \text{Vars}(\mu_i)$ or $\Sigma(\tau_j)$ is unknown. Either way, there is an edge from v_i to v_j.

- redex-II. If a redex-II is created in v_j, then for some paths Σ_1, Σ_2, $\Sigma_1(\tau_j\sigma) = \Sigma_2(\tau_j\sigma) = \alpha$ for some variable α and $\Sigma_1(\mu_j\sigma) \neq \Sigma_2(\mu_j\sigma)$. Since substitutions cannot "un-unify" two expressions, it follows that $\Sigma_1(\mu_j) \neq \Sigma_2(\mu_j)$, or at least one of these does not exist. In the former case, since the redex did not exist previously, we must have $\Sigma_1(\tau_j) \neq \Sigma_2(\tau_j)$. Hence, at least one of these was replaced during the redex reduction, and therefore either occurs in μ_j or is unknown. Either way, there is an edge from v_i to v_j. In the latter case, the non-existence of either $\Sigma_1(\mu_j)$ or $\Sigma_2(\mu_j)$ indicates that there is already at least one redex-I at that site, and reduction of this redex-I within v_j itself would create the redex-II anyway. Hence this redex-II does not arise strictly as a result of a reduction in v_i. In those cases in which it does, however, we always have $v_i \rightarrow v_j$.

- redex-III. If a redex-III is created in v_j, then for some path Σ, $\Sigma(\tau_j\sigma)$ contains only unknowns at the leaves, $\Sigma(\mu_j\sigma)$ exists and is not equal to $\Sigma(\tau_j\sigma)$, and no prefix of Σ has this property. Since the redex did not exist previously, and reductions cannot "un-unify" expressions, we conclude that $\Sigma(\tau_j)$ must contain a variable α. Then α is replaced in v_i, which implies that $\alpha \in \text{Vars}(\mu_i)$. Hence $v_i \rightarrow v_j$.

- redex-IV. If a redex-IV is created in v_j, then for some path Σ, $\Sigma(\mu_j\sigma)$ is an unknown $\underline{\gamma}$, and $\Sigma(\tau_j\sigma)$ exists, is not a variable, and is not equal to $\Sigma(\mu_j\sigma)$. Since the redex did not exist previously, either $\Sigma(\mu_j)$ is a variable, or $\Sigma(\tau_j\sigma)$ is a variable α. In the latter case, α is replaced in v_i; hence $\alpha \in \text{Vars}(\mu_i)$, and therefore $v_i \rightarrow v_j$. In the former case, if the latter case does not also hold, then there is a redex-I between $\Sigma(\tau_j)$ and $\Sigma(\mu_j)$. In this case, the reduction in v_i only changes the form of the redex in v_j (i.e., from redex-I to redex-IV), rather than introduce a new one.

□

We are now ready to establish our main result:

Theorem 4.6 *Let Γ be an instance of USUP such that $G(\Gamma)$ is R-acyclic. Then the USUP redex procedure terminates on input Γ.*

Proof The proof is similar to that of Theorem 3.3. As before, the invariance of R-acyclicity establishes a partial order $\sqsubseteq$ on the vertices in $G(\Gamma)$ that will not be violated by any reduction. Let $\leq$ be any total order consistent with $\sqsubseteq$ and number the vertices in Γ according to this order. Then for any vertex v_i, the only vertices v_j in which v_i can induce redexes are those for which $j > i$ (this is Lemma 4.4). We then proceed by induction on $(n_1, \ldots, n_N)$, under lexicographic ordering, where N is the number of inequalities in Γ, and for each i, n_i represents the number of redexes present in the inequality v_i. The lexicographic ordering of $(n_1, \ldots, n_N)$ is a well-ordering of the N-tuple. If a reduction occurs in vertex v_i, then n_i decreases by one, and all n_k for $k < i$ are unchanged. Thus the ordinal approaches $(0, \ldots, 0)$ with each redex reduction. If the procedure does not abort early due to an error, the ordinal will inevitably reach $(0, \ldots, 0)$, whereupon the entire instance contains no redexes, and the procedure terminates. □

Theorem 4.6 establishes the decidability of the problem R-AUSUP, consisting of R-acyclic USUP instances. As a consequence, the problem AUSUP, consisting of column-acyclic USUP instances, is also decidable.

4.5 Syntax-Directed Translation to USUP

We have formally established a generalization of SUP, which we have named USUP, together with a reduction procedure that is both sound and complete, outputs most general semiunifiers, terminates on all solvable instances and on all R-acyclic instances. These developments were done for the purpose of formulating a syntax-directed translation from rank-2 typability to a set of SUP-style constraints. We establish such a translation in this section.

4.5.1 Translation Rules

Because our translation from rank 2-typability to USUP is syntax-directed, we can avoid the English descriptions of the translation that we used in Section 2.4.3. Instead, we can formulate the translation as a formal system. Figure 4.1 presents our translation, which derives statements of the form $E \Rightarrow \Gamma$, where E is a labelled λ-term, and Γ is a set of USUP inequalities. Note that, because of our introduction of unknowns, it is no longer formally necessary to treat monomorphic variables and polymorphic variables differently during translation. In particular, we no longer

$$\frac{}{x \Rightarrow \{\langle \beta_x \leq \delta_x, \{\}\rangle\}}$$

$$\frac{M \Rightarrow \Gamma_1 \quad N \Rightarrow \Gamma_2}{MN \Rightarrow \{\langle \delta_M = \delta_N \rightarrow \delta_{MN}, \{\}\rangle\} \cup \Gamma_1 \cup \Gamma_2}$$

$$\frac{E \Rightarrow \Gamma}{\lambda^p x.E \Rightarrow \{\langle \delta_{\lambda^p x.E} = \beta_x \rightarrow \delta_E, \{\}\rangle\} \cup \Gamma}$$

$$\frac{E \Rightarrow \Gamma}{\lambda^p x : \tau.E \Rightarrow \{\langle \delta_{\lambda^p x:\tau.E} = \beta_x \rightarrow \delta_E, \{\}\rangle, \langle \beta_x = \tau, \{\}\rangle\} \cup \Gamma}$$

$$\frac{E \Rightarrow \Gamma}{\lambda^m x.E \Rightarrow \{\langle \delta_{\lambda^m x.E} = \beta_x \rightarrow \delta_E, \{\}\rangle\} \cup (\Gamma + \underline{x})}$$

Figure 4.1: Syntax-directed translation from rank-2 typability to USUP.

have to be careful to translate polymorphic variable occurrences as inequalities and monomorphic variable occurrences as equations. Instead, because unknowns will propagate, via redex-III and redex-IV reductions, across inequalities anyway, we can simply translate all variable occurrences as inequalities. A side-effect of this simplification is that we now formally label the type variable associated with a given term variable as a subscripted β, irrespective of whether the variable is monomorphic or polymorphic. In the context of discussion, however, we may continue to assign monomorphic term variables type variables labelled as subscripted γ's, for the sake of clarity.

It is also no longer formally necessary to make a distinction between λ^1- and λ^2-abstractions. As these are both simply polymorphic abstractions, we can refer to them as λ^p-abstractions. Analogously, we now call the λ^3-abstractions λ^m-abstractions, since they are monomorphic abstractions.

One way in which λ^1- and λ^2-abstractions differ from each other in the original formulation is that λ^2-abstractions may carry optional type annotations, whereas there is no similar facility for λ^1-abstractions. One feature of our present translation, in which these two kinds of abstraction are unified, is that λ^1-abstractions may now also carry type annotations. Though practically speaking, this extra freedom will not cause any problems, for the purpose of formally verifying equivalence with the previous formulation, we may simply assume, if we like, that even though λ^1-abstractions are permitted to carry type annotations, they never actually do.

Our translation for monomorphic abstractions makes use of the notation "$\Gamma + \underline{x}$", which we use to denote the addition of x as an unknown to each inequality in Γ. This notation is defined formally as follows:

Definition 4.7 ($\Gamma + \underline{x}$-notation) *Given a USUP instance Γ over a term algebra whose set $\mathbb{V}$ of variables contains the identifier x, we use the notation $\Gamma + \underline{x}$ to denote the USUP instance Γ', which is identical to Γ, except that the identifier x is regarded as an unknown throughout Γ':*

$$\begin{aligned} \emptyset + \underline{x} &= \emptyset \\ (\{\langle \tau \leq \mu, \vec{\alpha} \rangle\} \cup \Gamma) + \underline{x} &= \{\langle \tau \leq \mu, \vec{\alpha} \cup \{x\} \rangle\} \cup (\Gamma + \underline{x}) \end{aligned}$$

With this definition in place, the rule for monomorphic abstractions now simply reads that we translate a monomorphic abstraction by translating its body, adding x as an unknown to the resulting inequalities, and then adding an additional inequality relating the parameter and body type of the abstraction to the type of the abstraction itself.

Finally, the syntax-directed translation does not directly address type annotations on free variables. Under the original, non-syntax-directed translation procedure, free variables obtain

their types through a programmer-supplied type environment, which we translate into ASUP constraints simply by creating an equality between the type variable assigned to the free term variable, and its assigned type. This step has been omitted from our presentation, simply because the type environment is external to the program, and therefore cannot be treated in a syntax-directed way. However, there are at least two easy ways to accommodate it. The first is simply to take the same approach as the original translation, and add equations for the free variables' types to the instance at the end (accompanied by an empty set of unknowns). The second is to take a closure of the term over its free variables, so that they become λ^2-variables (now λ^p-variables), whose binding abstractions can be annotated with the free variables' assigned types. In this way, the syntax-directed translation can be made to cover free variables as well. For our purposes in the remainder of this chapter, we shall assume that the latter approach has been taken—that all source programs have been closed over their free variables, so that no free variables remain.

For convenience in later sections, we introduce notation to capture the translation from typability to USUP:

Definition 4.8 *Let E be a labelled λ-term and Δ a type environment with $FTV(E) \subseteq \mathrm{dom}(\Delta)$. We define $USUP(E, \Delta)$ to be the USUP instance obtained by applying our translation procedure to the term E under the type environment Δ. When Δ is understood without ambiguity, or unimportant to the discussion, we simply write $USUP(E)$ to denote $USUP(E, \Delta)$.*

Comparing our new translation procedure with the original procedure, we see that our procedure is considerably more concise, easier to understand, and fully syntax-directed—there is a one-to-one correspondence between inequalities and the syntax elements they represent. Further, the size of the USUP instance is linear[3] in the size of the source program. In the remaining sections, we show that the USUP instance output by our translation procedure actually captures the intended semantics of rank 2-typability, and produces terminating instances.

4.5.2 Soundness and Completeness

In Section 4.4.1, we showed that the USUP redex procedure is sound and complete (when it terminates) with respect to the definition of a solution of a USUP instance. The other relevant

[3]Strictly speaking, the size of the USUP instance is not quite linear in the size of the term, because each variable that plays the role of unknown is listed with every inequality in which it is an unknown. Hence, the size of the representation is dependent on the number of unknowns times the size of their scope. However, this non-linearity can be removed using a tree-based representation of the problem instance, so that the repetition of unknowns is avoided.

notion of soundness and completeness for USUP concerns whether the USUP representation of a λ-term faithfully represents the rank 2-typability of the term. It is this question that we address in this section.

Before we address questions of soundness and completeness, however, we first present type rules for rank 2-typability in System F, so that we have a semantics with which to compare our translation. These rules are presented in Figure 4.2. We assume that a labelling step for λ-

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \\ \pi & ::= & \tau \mid \forall\alpha.\pi \\ \rho & ::= & \pi \mid \pi \to \rho \mid \forall\alpha.\rho \end{array}$$

$$\Delta ::= [\,] \mid \Delta,\ x : \pi$$

$$\frac{}{\Delta \vdash x\colon \Delta(x)}\ [\text{var}] \qquad \frac{\Delta \vdash M\colon \pi \to \rho \quad \Delta \vdash N\colon \pi}{\Delta \vdash MN\colon \rho}\ [\text{app}]$$

$$\frac{\Delta,\ x : \tau_1 \vdash E : \tau_2}{\Delta \vdash \lambda^m x.E : \tau_1 \to \tau_2}\ [\text{m-abs}] \qquad \frac{\Delta,\ x : \forall.\tau \vdash E : \rho}{\Delta \vdash \lambda^p x.E : (\forall.\tau) \to \rho}\ [\text{p-abs}]$$

$$\frac{\Delta,\ x : \forall.\tau\sigma \vdash E : \rho}{\Delta \vdash \lambda^p x : \forall.\tau.E : (\forall.\tau\sigma) \to \rho}\ [\text{p-abs-annot}]$$

$$\frac{\Delta \vdash E : \forall\alpha.\rho}{\Delta \vdash E : \rho[\tau/\alpha]}\ [\text{spec}] \qquad \frac{\Delta \vdash E : \rho}{\Delta \vdash E : \forall\alpha.\rho}\ [\text{gen}]\ (\alpha \text{ not free in } \Delta)$$

Figure 4.2: Type rules for the rank 2 fragment of System F.

abstractions has taken place before the type rules are applied. Hence the rank 2 restrictions about which abstractions are allowed to be polymorphic have already been applied, and the rules address how a term is typed with those restrictions in place. Note also that, as with our presentation of the translation to USUP, we use the notation λ^p to denote both λ^1- and λ^2-abstractions, and λ^m

to denote λ^3-abstractions. The metavariables τ, π, and ρ run, respectively, over rank 0, rank 1, and rank 2 types. Also note that we disregard the order in which quantified variables are listed on a quantified type, and will freely specialize them in any order we wish. This convenience can be justified by repeated application of rules [spec] and [gen]; however it is simpler to merely adopt the convention that the list of quantified variables in a type is unordered.

At first glance, the occurrence of an arbitrary substitution σ in the premise and conclusion of rule [p-abs-annot] may seem unusual. However, it is needed in order to capture the notion that not every solution of a USUP instance is a most general solution, and therefore, even though the parameter x in the rule is annotated with the type $\forall.\tau$, we may consider assigning x a less general type when typing the body E. The result is still a valid type of the abstraction under relaxed assumptions, though not the expected one. If σ is, in fact, a most general solution, then it is straightforward to show that, at least when the λ^p-abstraction is not accompanied by an argument (i.e., it is in fact a λ^2-abstraction, which is the only kind of abstraction for which the original KW permits annotations), we obtain $\tau\sigma = \tau$, and the expected type results.

Soundness of the Translation

To establish soundness, intuitively we must show that, for an expression E, if $USUP(E)$ is solved by a substitution σ, which, when applied to the type variable δ_E, representing the type of E, produces a type τ, then τ is derivable as a type for E from the type rules for rank 2 System F.

However, this formulation is not quite right, because for any σ solving $USUP(E)$, $\delta_E\sigma$ will not contain any quantifiers, and is therefore always a rank 0 type. Furthermore, we cannot reintroduce the quantifiers simply by prepending a universal quantifier for each type variable onto the expression—such an action can only create rank 1 types. Instead, quantifiers must be introduced in such a way that they can be nested to the left of at most one $\rightarrow$-functor. In order to guide the way that quantifiers are reintroduced into the open type $\delta_E\sigma$, we employ the following function, which compares a type with a term (for which it is presumably a type, once quantifiers are introduced) and uses the form of the term to guide the introduction of $\forall$-quantifiers into the type:

Definition 4.9 *For a given labelled λ-term E and type τ, define $Q(E,\tau)$ as follows:*

$$\begin{aligned} Q(x,\tau) &= \tau \\ Q(MN,\tau) &= \tau' \text{ if } Q(M,\alpha\to\tau)=\pi\to\tau', \text{ where } \alpha \text{ is any variable} \\ Q(\lambda^p x.E',\tau_1\to\tau_2) &= (\forall.\tau_1)\to Q(E',\tau_2) \\ Q(\lambda^m x.E',\tau_1\to\tau_2) &= \tau_1\to Q(E',\tau_2)\ . \end{aligned}$$

The type τ, used as input to Q, is assumed to be large enough that $Q(E,\tau)$ is fully defined by the above equations.

The function Q essentially pattern-matches a type against a term, quantifying any portion that corresponds to the parameter type of a polymorphic abstraction, and otherwise leaving the expression unchanged. Our desired soundness result, then, is that if σ solves $USUP(E)$, then $Q(E,\delta_E\sigma)$ is a derivable type for E.

Before we proceed to establish soundness for our translation procedure, we first present the following two technical results about the function Q:

Lemma 4.5 *If an expression N occurs as a subterm of a function argument in a larger rank 2 term E, and is not itself paired with an argument, then $Q(N,\tau)=\tau$.*

Proof If N is a variable, the result is immediate. If N is an abstraction, it cannot be a polymorphic abstraction, by the labelling rules for the rank 2 fragment of System F. Hence, N is a monomorphic abstraction $\lambda^m z.N'$, τ can be written as $\tau_1\to\tau_2$ for some τ_1, τ_2, and $Q(N,\tau)=\tau_1\to Q(N',\tau_2)$. Since N' also occurs as a subterm of a function argument, and is not itself paired with an argument, we have by induction that $Q(N',\tau_2)=\tau_2$. Hence $Q(N,\tau)=\tau_1\to\tau_2=\tau$. If N is an application PR, then we have $Q(P,\alpha\to\tau)=\pi\to\tau'$ for some π, τ', and $Q(PR,\tau)=\tau_2$. We argue that $\tau=\tau_2$. If $P=x$ for some variable x, then $Q(P,\alpha\to\tau)=\alpha\to\tau$, and therefore, $\tau=\tau_2$. If $P=\lambda x.E$ (either monomorphic or polymorphic), then $Q(P,\alpha\to\tau)=\pi\to Q(E,\tau)$ for some π. By induction, since $Q(E,\tau)$ is a subterm of a function argument, but has no argument itself, we have $Q(E,\tau)=\tau$, and therefore, $\tau=\tau_2$. If $P=E_1E_2$, then $Q(P,\alpha\to\tau)=\tau_3$ if $Q(E_1,\alpha\to\alpha\to\tau)=\pi'\to\tau_3$ for some π'. By induction, $\alpha\to\tau=\tau_3$, hence $Q(P,\alpha\to\tau)=\alpha\to\tau$, and therefore, $\tau=\tau_2$, from which the result follows. □

Note that the conditions outlined for N in the statement of Lemma 4.5 are precisely those which describe the labelling of an abstraction as a λ^3-abstraction (now a λ^m-abstraction).

Lemma 4.6 *Let $E = MN_1 \cdots N_k$ be a labelled λ-term. Then $Q(M, \tau_1 \to \cdots \to \tau_{k+1}) = (\forall ?\tau_1) \to \cdots \to (\forall ?\tau_k) \to Q(E, \tau_{k+1})$, where the notation $\forall ?\tau$ denotes either τ or $\forall.\tau$.*

Proof The proof is by induction on the size of M. There are four cases to consider:

- M is a variable x. Then $Q(M, \tau_1 \to \cdots \to \tau_{k+1}) = \tau_1 \to \cdots \to \tau_{k+1}$. Note also that by definition,

$$\begin{aligned} Q(M, \tau_1 \to \cdots \to \tau_{k+1}) &= \pi_1 \to Q(MN_1, \tau_2 \to \cdots \to \tau_{k+1}) \text{ for some } \pi_1 \\ &\cdots \\ &= \pi_1 \to \cdots \to \pi_k \to Q(MN_1 \cdots N_k, \tau_{k+1}) \text{ for some } \pi_1, \ldots, \pi_k \\ &= \pi_1 \to \cdots \to \pi_k \to Q(E, \tau_{k+1}) \end{aligned}$$

  Hence, $Q(E, \tau_{k+1}) = \tau_{k+1}$, and the result follows.

- M is a monomorphic abstraction, $\lambda^m x.M'$. Then $Q(M, \tau_1 \to \cdots \to \tau_{k+1}) = \tau_1 \to Q(M', \tau_2 \to \cdots \to \tau_{k+1})$. We also have

$$Q(M, \tau_1 \to \cdots \to \tau_{k+1}) = \pi_1 \to Q(MN_1, \tau_2 \to \cdots \to \tau_{k+1})$$

  for some π_1; hence, $\pi_1 = \tau_1$ and $Q(M', \tau_2 \to \cdots \to \tau_{k+1}) = Q(MN_1, \tau_2 \to \cdots \to \tau_{k+1})$. By induction, since M' is smaller than M, we have

$$Q(M', \tau_2 \to \cdots \to \tau_{k+1}) = (\forall ?\tau_2) \to \cdots \to (\forall ?\tau_k) \to Q(M'N_2 \cdots N_k, \tau_{k+1}) \ .$$

  Since $Q(M', \tau_2 \to \cdots \to \tau_{k+1}) = Q(MN_1, \tau_2 \to \cdots \to \tau_{k+1})$, then we also have

$$\begin{aligned} Q(M'N_2 \cdots N_k, \tau_{k+1}) &= Q(MN_1N_2 \cdots N_k, \tau_{k+1}) \\ &= Q(E, \tau_{k+1}) \ , \end{aligned}$$

  so that

$$Q(M', \tau_2 \to \cdots \to \tau_{k+1}) = (\forall ?\tau_2) \to \cdots \to (\forall ?\tau_k) \to Q(E, \tau_{k+1}) \ .$$

  Hence,

$$Q(M, \tau_1 \to \cdots \to \tau_{k+1}) = \tau_1 \to (\forall ?\tau_2) \to \cdots \to (\forall ?\tau_k) \to Q(E, \tau_{k+1}) \ ,$$

  as required.

- M is a polymorphic abstraction, $\lambda^p x.M'$. Then the reasoning is identical to the case that M is a monomorphic abstraction, except that we now have $Q(M, \tau_1 \to \cdots \to \tau_{k+1}) = (\forall.\tau_1) \to Q(M', \tau_2 \to \cdots \to \tau_{k+1})$. The argument then proceeds as above.

- M is an application $M'N'$. Then M' is smaller than M, and by induction,

$$\begin{aligned} Q(M', \alpha \to \tau_1 \to \cdots \to \tau_{k+1}) &= (\forall ?\alpha) \to (\forall ?\tau_1) \to \cdots \to (\forall ?\tau_k) \to Q(M'N'N_1 \cdots N_k, \tau_{k+1}) \\ &= (\forall ?\alpha) \to (\forall ?\tau_1) \to \cdots \to (\forall ?\tau_k) \to Q(E, \tau_{k+1}) \ . \end{aligned}$$

  Hence,

$$\begin{aligned} Q(M, \tau_1 \to \cdots \to \tau_{k+1}) &= Q(M'N', \tau_1 \to \cdots \to \tau_{k+1}) \\ &= (\forall ?\tau_1) \to \cdots \to (\forall ?\tau_k) \to Q(E, \tau_{k+1}) \ , \end{aligned}$$

  which is the expected result.

Thus, in all cases, we obtain $Q(M, \tau_1 \to \cdots \to \tau_{k+1}) = (\forall ?\tau_1) \to \cdots \to (\forall ?\tau_k) \to Q(E, \tau_{k+1})$, as required. □

Definition 4.10 *For a quantified type ρ, denote by $Q^{-1}\rho$ the rank 0 type obtained by stripping all quantifiers (including all nested quantifiers) from ρ.*

The soundness of our translation procedure is now a consequence of the following theorem:

Theorem 4.7 *Let E be a labelled λ-term in which each variable $x \in FV(E)$ is labelled as either monomorphic or polymorphic. Let Δ be a type environment such that $FV(E) \subseteq \mathrm{dom}(\Delta)$, and for each monomorphic (resp. polymorphic) $x \in FV(E)$, $\Delta(x) = \tau$ (resp. $\Delta(x) = \forall.\tau$) for some rank 0 type τ. Suppose that $USUP(E, \Delta)$ possesses a solution σ. Then E possesses a type derivation, such that for each subexpression E' of E, the statement $\Delta\sigma \vdash E' : Q(E', \delta_{E'}\sigma)$ is derivable from the rules in Figure 4.2, where for every $x \in FV(E)$, $(\Delta\sigma)(x) = \beta_x\sigma$ if x is monomorphic and $Q^{-1}((\Delta\sigma)(x))\sigma = \beta_x\sigma$ if x is polymorphic (throughout the above, the notation $\forall.\tau$ denotes quantification over all of $FTV(\tau) \setminus FTV(\Delta)$).*

Proof The proof is by structural induction, dispatching on the form of the expression E:

- E is a variable x. Since x is necessarily a free variable, then by hypothesis, x has a binding in Δ. Then, depending on whether x is monomorphic or polymorphic, we proceed as follows:

- x is polymorphic, i.e., β_x is a true variable, and not an unknown. Then $\Delta(x) = \forall.\tau$ for some rank 0 type τ. The instance $\Gamma := USUP(x, \Delta)$ contains the inequalities $\beta_x \leq \delta_E$ and $\beta_x = \tau$. Since $Q(x, \tau) = \tau$ for every τ, it suffices to show that $\Delta \vdash E : \delta_E\sigma$ is derivable for every σ that solves Γ. Now, if σ solves Γ, then there is a substitution σ_1 such that $\beta_x\sigma\sigma_1 = \delta_E\sigma$ and $\beta_x\sigma = \tau\sigma$. Then $\tau\sigma\sigma_1 = \delta_E\sigma$. By rule [var], $\Delta \vdash E : \forall\vec{\alpha}.\tau$ is derivable, where $\vec{\alpha} = FTV(\tau) \setminus FTV(\Delta)$. Then $\Delta\sigma \vdash E : \forall.\vec{\alpha'}\tau\sigma$ (i.e., $\Delta\sigma \vdash E : \forall.\vec{\alpha'}\beta_x\sigma$) is derivable, where $\vec{\alpha'} = FTV(\tau\sigma) \setminus FTV(\Delta\sigma)$. Since the action of σ_1 on $\beta_x\sigma$ is completely determined by its action on $\vec{\alpha'}$ (all other identifiers in Vars($\beta_x\sigma$) are unknowns), we can apply rule [spec] and specialize each $\alpha \in \vec{\alpha'}$ to $\alpha\sigma$, and arrive at $\Delta\sigma \vdash x : \beta_x\sigma\sigma_1$, and since $\beta_x\sigma\sigma_1 = \delta_E\sigma$, we obtain the desired result.
- x is monomorphic, i.e., β_x is an unknown. Then $\Delta(x) = \tau$ for some rank 0 type τ. The instance $\Gamma := USUP(x, \Delta)$ contains the inequalities $\underline{\beta_x} \leq \delta_E$ and $\underline{\beta_x} = \tau$. Since $Q(x, \tau) = \tau$ for every τ, it suffices to show that $\Delta \vdash E : \delta_E\sigma$ is derivable for every σ that solves Γ. Now, if σ solves Γ, then there is a substitution σ_1 such that $\underline{\beta_x}\sigma\sigma_1 = \delta_E\sigma$ and $\underline{\beta_x}\sigma = \tau\sigma$. Since dom($\sigma_1$) contains no unknowns, the first equation becomes $\underline{\beta_x}\sigma = \delta_E\sigma$. Then we obtain $\tau\sigma = \delta_E\sigma$. By rule [var], $\Delta \vdash E : \tau$ is derivable. Then $\Delta\sigma \vdash E : \tau\sigma$, i.e., $\Delta\sigma \vdash E : \delta_E\sigma$ is derivable.

- E is an application MN. Then $USUP(E, \Delta)$ contains the equality $\delta_M = \delta_N \to \delta_E$. Then for any solution σ of $USUP(E, \Delta)$, we will have $\delta_M\sigma = \delta_N\sigma \to \delta_E\sigma$. By induction, the statement $\Delta\sigma \vdash M : Q(M, \delta_M\sigma)$ is derivable, and for each $x \in FV(M)$, $(\Delta\sigma)(x) = \beta_x\sigma$ if x is monomorphic and $Q^{-1}((\Delta\sigma)(x))\sigma = \beta_x\sigma$ if x is polymorphic. Also by induction, the statement $\Delta\sigma \vdash N : Q(N, \delta_N\sigma)$ is derivable, and for each $x \in FV(N)$, $(\Delta\sigma)(x) = \beta_x\sigma$ if x is monomorphic and $Q^{-1}((\Delta\sigma)(x))\sigma = \beta_x\sigma$ if x is polymorphic. Since N is the argument of M (hence does not itself have an argument), we have, by Lemma 4.5, $Q(N, \delta_N\sigma) = \delta_N\sigma$. By Lemma 4.6, we have $Q(M, \delta_M\sigma) = Q(M, \delta_N\sigma \to \delta_E\sigma) = (\forall?\delta_N\sigma) \to Q(E, \delta_E\sigma)$, which gives rise to two possibilities:
  - $Q(M, \delta_N\sigma \to \delta_E\sigma) = \delta_N\sigma \to Q(E, \delta_E\sigma)$. Then $\Delta\sigma \vdash M : \delta_N\sigma \to Q(E, \delta_E\sigma)$ is derivable, and by rule [app], since $E = MN$, we obtain $\Delta\sigma \vdash E : Q(E, \delta_E\sigma)$, as needed.
  - $Q(M, \delta_N\sigma \to \delta_E\sigma) = (\forall.\delta_N\sigma) \to Q(E, \delta_E\sigma)$. Then $\Delta\sigma \vdash M : (\forall.\delta_N\sigma) \to Q(E, \delta_E\sigma)$ is derivable. By applying rule [gen] once for each $\alpha \in FTV(\delta_N\sigma) \setminus FTV(\Delta)$, we obtain $\Delta\sigma \vdash N : \forall.\delta_N\sigma$. Then by rule [app], we again obtain $\Delta\sigma \vdash E : Q(E, \delta_E\sigma)$.

- E is a monomorphic abstraction $\lambda^m x.M$. Then $USUP(E, \Delta)$ contains the equality $\delta_E = \beta_x \to \delta_M$, and β_x is unknown (i.e., monomorphic) throughout $USUP(M, \Delta_M)$, where $\Delta_M = (\Delta, x : \beta_x)$.. Thus, for any solution σ of $USUP(E, \Delta)$, we have $\delta_E\sigma = \beta_x\sigma \to \delta_M\sigma$. By induction, the statement $\Delta_M\sigma \vdash M : Q(M, \delta_M\sigma)$ is derivable. Then by rule [m-abs], we can derive $\Delta\sigma \vdash \lambda^m x.M : \beta_x\sigma \to Q(M, \delta_M\sigma)$. By definition, $Q(E, \delta_E\sigma) = Q(\lambda^m x.M, \beta_x\sigma \to \delta_M\sigma) = \beta_x\sigma \to Q(M, \delta_M\sigma)$, and we obtain $\Delta\sigma \vdash E : Q(E, \delta_E\sigma)$.

- E is a polymorphic abstraction $\lambda^p x.M$. Let σ be a substitution that solves $USUP(E, \Delta)$. The instance $USUP(E, \Delta)$ contains the equality $\delta_E = \beta_x \to \delta_M$, and β_x is variable (i.e., polymorphic) throughout $USUP(M, \Delta_M)$, where $\Delta_M = (\Delta, x : \forall.\beta_x\sigma)$. Thus, for any solution σ of $USUP(E, \Delta)$, we have $\delta_E\sigma = \beta_x\sigma \to \delta_M\sigma$. By induction, the statement $\Delta_M\sigma \vdash M : Q(M, \delta_M\sigma)$ is derivable, where for each $y \in FV(M)$, $(\Delta_M\sigma)(y) = \beta_y\sigma$ if y is monomorphic, and $Q^{-1}((\Delta_M\sigma)(y))\sigma = \beta_y\sigma$ if y is polymorphic. Then by rule [p-abs], we can derive $\Delta\sigma \vdash \lambda^p x.M : (\forall.\beta_x\sigma) \to Q(M, \delta_M\sigma)$. By definition, $Q(E, \delta_E\sigma) = Q(\lambda^p x.M, \beta_x\sigma \to \delta_M\sigma) = (\forall.\beta_x\sigma) \to Q(M, \delta_M\sigma)$, and we obtain $\Delta \vdash E : Q(E, \delta_E\sigma)$.

- E is an annotated polymorphic abstraction $\lambda^p x : (\forall.\tau).M$. Let σ be a substitution that solves $USUP(E, \Delta)$. The instance $USUP(E, \Delta)$ contains the equalities $\delta_E = \beta_x \to \delta_M$ and $\beta_x = \tau$, and β_x is variable (i.e., polymorphic) throughout $USUP(M, \Delta_M)$, where $\Delta_M = (\Delta, x : \forall.\tau\sigma)$. For any solution σ of $USUP(E, \Delta)$, we have $\delta_E\sigma = \beta_x\sigma \to \delta_M\sigma$ and $\beta_x\sigma = \tau\sigma$. Hence $\delta_E\sigma = \tau\sigma \to \delta_M\sigma$. By induction, the statement $\Delta_M \vdash M : Q(M, \delta_M\sigma)$ is derivable, where for each $y \in FV(M)$, $(\Delta_M\sigma)(y) = \beta_x\sigma$ if y is monomorphic, and $Q^{-1}((\Delta_M\sigma)(y))\sigma = \beta_y\sigma$ if y is polymorphic. Then by rule [p-abs-annot], we can derive $\Delta \vdash \lambda^p x : \forall.\tau.M : (\forall.\tau\sigma) \to Q(M, \delta_M\sigma)$. By definition, $Q(E, \delta_E\sigma) = Q(\lambda^p x : \forall.\tau.M, \tau\sigma \to \delta_M\sigma) = (\forall.\tau\sigma) \to Q(M, \delta_M\sigma)$. Hence, $\Delta \vdash E : Q(E, \delta_E\sigma)$, as required.

Having exhausted all cases, we conclude, by structural induction, that every solution σ of $USUP(E)$ yields a valid type derivation of E, in the manner laid out in the statement of the theorem. □

In particular, the theorem implies that whenever $USUP(E)$ is solvable, then E is rank 2-typable; moreover, if σ solves $USUP(E)$, then $Q(E, \delta_E\sigma)$ is a derivable type for E. This is the soundness result we sought.

Completeness of the Translation

Establishing completeness for our translation is a matter of showing that whenever a λ-term E is rank 2-typable with type ρ, then there is a substitution σ that solves $USUP(E)$, such that $\delta_E\sigma = \rho$. However, we immediately run into a difficulty analogous to our experience with soundness—the type ρ of E is a rank 2 type, and possibly contains quantifiers, which may be nested within $\rightarrow$-functors (though at most one level deep on the left). On the other hand, $\delta_E\sigma$ does *not* contain quantifiers. Instead, we wish to show that, given a term E with rank 2 type ρ, there is a substitution σ that solves $USUP(E)$, such that $\delta_E\sigma = \tau_\rho$, where τ_ρ denotes the rank 0 type obtained by stripping ρ of all quantifiers. Equivalently, we want $\rho = Q(E, \delta_E\sigma)$.

In particular, the completeness theorem is as follows:

Theorem 4.8 *Let E be a labelled λ-term, in which each free variable is labelled as either monomorphic or polymorphic. Suppose there exists a type environment Δ whose domain contains $FV(E)$, such that each monomorphic variable in $FV(E)$ has a rank 0 binding in Δ, and each polymorphic variable in $FV(E)$ has a rank 1 binding in Δ. Furthermore, suppose there is a rank 2 type ρ such that $\Delta \vdash E : \rho$ is derivable from the rules in Figure 4.2. Then there exists a substitution σ such that the following conditions hold:*

1. *σ is a solution of $USUP(E, \Delta)$;*
2. *σ is an identity on $FTV(\Delta)$;*
3. *$\delta_E\sigma = Q^{-1}(\rho)$ (up to variable renaming);*
4. *for each variable $\alpha \in \mathrm{Vars}(\delta_E\sigma)$, whose corresponding variable in $Q^{-1}(\rho)$ has rank at most 1: α does not occur on the left-hand side of any inequality in $USUP(E, \Delta)\sigma$ (i.e., the result of reducing $USUP(E, \Delta)$) in which α is not an unknown.*

Proof The proof is by induction on the type derivation for $\Delta \vdash E : \rho$. We dispatch based on the last rule applied in the derivation:

- Rule [var]. If rule [var] is at the root of the derivation, then the derivation must consist solely of the application of rule [var], and therefore, E is a variable x. Then by rule [var], $\Delta(x) = \rho$. $USUP(E, \Delta)$ contains the inequalities $\beta_x \leq \delta_E$ (since $E = x$, δ_E and δ_x are identical) and $\beta_x = Q^{-1}(\rho)$. Then there are two possibilities:

- β_x is a variable. Then take $\sigma = [Q^{-1}(\rho)'/\delta_E, Q^{-1}(\rho)/\beta_x]$ (recall that $Q^{-1}(\rho)'$ is the result of consistently replacing the variables in $Q^{-1}(\rho)$ with fresh ones), and let σ' be the substitution that maps $Q^{-1}(\rho)$ to $Q^{-1}(\rho)'$. Then $\beta_x\sigma\sigma' = \delta_E\sigma$ and $\beta_x\sigma = Q^{-1}(\rho)\sigma$; hence σ solves $USUP(E, \Delta)$, and since $\mathrm{dom}(\sigma) = \{\beta_x, \delta_E\}$, σ is an identity on $FTV(\Delta)$. Furthermore, $\delta_E\sigma = Q^{-1}(\rho)$, up to renaming, as required. Finally, since the variables in $Q^{1-}(\rho)'$ are fresh, they do not occur on any left-hand sides.
- β_x is an unknown. Then take $\sigma[Q^{-1}(\rho)/\delta_E, Q^{-1}(\rho)/\beta_x]$, and take σ' to be the identity substitution. Then $\beta_x\sigma\sigma' = \delta_E\sigma$, $\beta_x\sigma = Q^{-1}(\rho)\sigma$, and $\mathrm{dom}(\sigma')$ contains no unknowns. Hence, σ solves $USUP(E, \Delta)$, $\mathrm{dom}(\sigma) \cap FTV(\Delta) = \emptyset$, and $\delta_E\sigma = Q^{-1}(\rho)$, as required. Finally, since β_x is an unknown, every identifier in $Q^{-1}(\rho)$ is an unknown in $USUP(E, \Delta)\sigma$; hence, the final requirement of the theorem is satisfied vacuously.

- Rule [app]. If rule [app] is at the root of the derivation, then E is an application MN. Then there is a type π such that $\Delta \vdash M : \pi \to \rho$ and $\Delta \vdash N : \pi$. By induction, there are substitutions σ_M and σ_N that solve $USUP(M, \Delta)$ and $USUP(N, \Delta)$ respectively, are identity maps on $FTV(\Delta)$, and satisfy $\delta_M\sigma_M = Q^{-1}(\pi \to \rho) = Q^{-1}(\pi) \to Q^{-1}(\rho)$ and $\delta_N\sigma_N = Q^{-1}(\pi)$ (both up to (consistent) variable renaming). By restricting their domains, if necessary, we may assume that $\mathrm{dom}(\sigma_M) \subseteq \mathrm{Vars}(USUP(M, \Delta))$ and $\mathrm{dom}(\sigma_N) \subseteq \mathrm{Vars}(USUP(N, \Delta))$. Since bound variables in E are assumed to be distinctly named, the only variables that occur in both M and N are those that are free in both M and N. If a variable x lies in $FV(M) \cap FV(N)$, then x is either globally free, or it is bound in an abstraction whose body contains both M and N. Either way, $x \in \mathrm{dom}(\Delta)$, and therefore, $\beta_x\sigma_M = Q^{-1}(\Delta(x)) = \beta_x\sigma_N$, i.e., σ_M and σ_N agree where their domains intersect. Then the set σ_E defined by $\sigma_E = \sigma_M \cup \sigma_N$ is a substitution, with $\sigma_E|_{dom(\sigma_M)} = \sigma_M$ and $\sigma_E|_{dom(\sigma_N)} = \sigma_N$, so that σ_E solves both $USUP(M, \Delta)$ and $USUP(N, \Delta)$ and is an identity on $FTV(\Delta)$. The instance $USUP(E, \Delta)$ is made up of both $USUP(M, \Delta)$ and $USUP(N, \Delta)$, as well as the additional inequality $\delta_M = \delta_N \to \delta_E$. Since δ_E occurs in neither $\mathrm{Vars}(USUP(M, \Delta))$ nor $\mathrm{Vars}(USUP(N, \Delta))$, let $\sigma = [Q^{-1}(\rho)/\delta_E] \circ \sigma_E$. Then σ also solves both $USUP(M, \Delta)$ and $USUP(N, \Delta)$, and is also an identity on $FTV(\Delta)$. We then have $\delta_M\sigma = \delta_M\sigma_M = Q^{-1}(\pi) \to Q^{-1}(\rho)$, and $\delta_N\sigma = \delta_N\sigma_N = Q^{-1}(\pi)$. Also, by construction, $\delta_E\sigma = Q^{-1}(\rho)$ (all three of these taken modulo (consistent) variable renaming). Putting these together, we obtain $\delta_M\sigma = \delta_N\sigma \to \delta_E\sigma$; therefore σ solves $USUP(E, \Delta)$, with $\delta_E\sigma = Q^{-1}(\rho)$, as required.

It remains to show that the variables in $Q^{-1}(\rho)$ do not occur on any left-hand sides in $USUP(E, \Delta)\sigma$, except possibly where they are unknowns. By induction, the variables in $Q^{-1}(\pi) \to Q^{-1}(\rho)$ do not occur in any left-hand sides (except for unknown occurrences) in $USUP(M, \Delta)\sigma_M$, and similarly for $Q^{-1}(\pi)$ and $USUP(N, \Delta)\sigma_N$. The non-rank-2 variables in $Q^{-1}(\pi) \to Q^{-1}(\rho)$ form a superset of those in $Q^{-1}(\rho)$ (note that *all* variables in $Q^{-1}(\pi)$ are non-rank-2); hence, unknown occurrences aside, the non-rank-2 variables in $Q^{-1}(\rho)$ do not occur on left-hand sides in $USUP(M, \Delta)\sigma_M$, except for unknown occurrences. Let α be a non-rank-2 variable in $Q^{-1}(\rho)$. If α has a non-unknown occurrence on a left-hand side in $USUP(N, \Delta)\sigma_N$ (indeed, any occurrence at all in $USUP(N, \Delta)\sigma_N$), then since α would then occur in *both* $USUP(M, \Delta)\sigma_M$ and $USUP(N, \Delta)\sigma_N$, α must originate higher in the parse tree than the entire application. Suppose, then, that α is a subexpression of some $\beta_x\sigma_M$ $(= \beta_x\sigma_N)$, where x is a variable with a monomorphic binding occurrence. Then β_x is an unknown, and the result follows vacuously—α is then also an unknown throughout $USUP(M, \Delta)\sigma_M$ and $USUP(N, \Delta)\sigma_N$ and therefore has no occurrences of interest. If, instead, x has a polymorphic binding occurrence, then β_x only occurs on left-hand sides throughout $USUP(MN, \Delta)$, except for an equality $\beta_x = \tau$ that sets β_x by environment lookup. The variables in τ are distinct from all variables in $USUP(MN, \Delta)$. Let $\sigma_1 = [\tau/\beta_x]$. Then $USUP(MN, \Delta)$ is solvable with solution $\sigma_2 \circ \sigma_1$ for some σ_2 if and only if $USUP(MN, \Delta)\sigma_1$ is solvable with solution σ_2. By Theorem 3.6 (the Canonical Solutions Theorem), σ_2 can be assumed to be canonical without affecting its operation on any variable on a right-hand side. Hence, we can take $\sigma_N = \sigma_2 \circ \sigma_1$ with no effect on the value of $\delta_N\sigma_N$. Since the variables of τ are distinct from all those in the rest of $USUP(MN, \Delta)$, they occur only on left-hand sides in $USUP(MN, \Delta)\sigma_1$. Hence σ_N (and also σ_M, from previous reasoning) do not replace any variable in τ; hence apart from replacing β_x with τ, they do not replace β_x. In particular, then, $\delta_N\sigma$ does not contain β_x, or any other variable on a left-hand side throughout $USUP(MN, \Delta)$. Thus, variables in $Q^{-1}(\rho)$ do not occur on any left-hand sides in $USUP(E, \Delta)\sigma_E$. Since σ only adds to σ_E a replacement of δ_E (which has exactly one occurrence—on a right-hand side) with $Q^{-1}(\rho)$, the same holds for σ, as required.

- Rule [m-abs]. If rule [m-abs] is at the root of the derivation, then E is a monomorphic abstraction $\lambda^m x.M$ and x is monomorphic in M (i.e., β_x is unknown in $USUP(M, (\Delta, x : \tau))$ for any τ). Then there are rank 0 types τ_x and τ_M such that $\rho = \tau_x \to \tau_M$, and $\Delta, x : \tau_x \vdash M : \tau_M$ is derivable. Let Δ_M denote the type environment $(\Delta, x : \tau_x)$. By

induction, there is a substitution σ_M that solves $USUP(M, \Delta_M)$, such that $\delta_M\sigma_M = \tau_M$, up to renaming, and σ_M is an identity on $FTV(\Delta_M)$. The instance $USUP(E, \Delta)$ contains all of $USUP(M, \Delta_M)$, as well as the equality $\delta_E = \beta_x \to \delta_M$. If x has at least one occurrence in M, then $USUP(M, \Delta_M)$ contains the equality $\beta_x = \tau_x$. Then $\beta_x\sigma_M = \tau_x\sigma_M$. Otherwise, β_x has no occurrences in $USUP(M, \Delta_M)$ and we can let $\sigma_{M,x} = [\tau_x/\beta_x] \circ \sigma_M$, and continue the argument using $\sigma_{M,x}$. For simplicity, and without loss of generality, however, we will simply assume that $\beta_x \in \mathrm{Vars}(USUP(M, \Delta_M))$, and therefore, $\beta_x \in \mathrm{dom}(\sigma_M)$. Since δ_E has no occurrences in $USUP(M, \Delta_M)$, define the substitution σ as $[Q^{-1}(\rho)/\delta_E] \circ \sigma_M$. Then σ also solves $USUP(M, \Delta_M)$, and since $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma_M) \cup \{\delta_E\}$, σ is an identity on $FTV(\Delta_M)$, and therefore also on $FTV(\Delta)$. Since σ and σ_M are identities on $FTV(\Delta_M)$, we have $\tau_x\sigma = \tau_x\sigma_M = \tau_x$. Finally, we have

$$\begin{aligned}
\delta_E\sigma &= Q^{-1}(\rho) \\
&= \rho \\
&= \tau_x \to \tau_M \\
&= \tau_x\sigma_M \to \tau_M \\
&= \beta_x\sigma_M \to \delta_M\sigma_M \\
&= \beta_x\sigma \to \delta_M\sigma \ ,
\end{aligned}$$

as required.

We now show that the non-rank-2 variables in $Q^{-1}(\rho) = \tau_x \to \tau_M$ do not occur on any left-hand sides in $USUP(E, \Delta)$, unknown occurrences aside. By induction, the non-rank-2 variables in τ_M do not occur in any left-hand sides in $USUP(M, \Delta_M)\sigma_M$, unknown occurrences aside. Since σ only adds a replacement of δ_E (whose only occurrence is on a right-hand side) to σ_M, and since $USUP(E, \Delta)$ does not introduce onto any left-hand sides any variable whose only occurrences in $USUP(M, \Delta_M)$ (unknown occurrences aside) are on right-hand sides, the non-rank-2 variables in τ_M do not occur on any left-hand sides in $USUP(E, \Delta)\sigma_E$. As for β_x, since E is a monomorphic abstraction, any occurrences of β_x in $USUP(M, \Delta_M)$ are unknown occurrences, which we may disregard. Further, β_x's only other occurrence in $USUP(E, \Delta)$ is within the equality $\delta_E = \beta_x \to \delta_M$, which is a right-hand side occurrence. Thus, $Q^{-1}(\rho)$ contains no variables with left-hand side occurrences outside unknown occurrences, as required.

- Rule [p-abs]. If rule [p-abs] is at the root of the derivation, then E is a polymorphic

abstraction $\lambda^p x.M$ and x is polymorphic in M (i.e., β_x is variable in $USUP(M, (\Delta, x : \forall.\tau))$ for any τ). Then there is a rank 1 type $\forall.\tau_x$ and a rank 2 type ρ_M such that $\rho = (\forall.\tau_x) \to \rho_M$, and the statement $\Delta, x : (\forall.\tau_x) \vdash M : \rho_M$ is derivable. Let Δ_M denote the type environment $(\Delta, x : \forall.\tau_x)$. By induction, there is a substitution σ_M that solves $USUP(M, \Delta_M)$, such that $\delta_M\sigma_M = Q^{-1}(\rho_M)$ after renaming, and σ_M is an identity on $FTV(\Delta, x, \forall.\tau_x)$. The instance $USUP(E, \Delta)$ contains all of $USUP(M, \Delta_M)$, plus the equality $\delta_E = \beta_x \to \delta_M$. If x has at least one occurrence in M, then $USUP(M, \Delta_M))$ contains the equality $\beta_x = \tau_x$. Then $\beta_x\sigma_M = \tau_x\sigma_M$. Otherwise, β_x has no occurrences in $USUP(M, \Delta_M)$ and we can let $\sigma_{M,x} = [\tau_x/\beta_x] \circ \sigma_M$, and continue the argument using $\sigma_{M,x}$. For simplicity, and without loss of generality, however, we will simply assume that $\beta_x \in \mathrm{Vars}(USUP(M, \Delta_M))$, and therefore, $\beta_x \in \mathrm{dom}(\sigma_M)$. Since δ_E has no occurrences in $USUP(M, \Delta_M)$, define the substitution σ as $[Q^{-1}(\rho)/\delta_E]\circ\sigma_M$. Then σ also solves $USUP(M, \Delta_M)$, and since $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma_M)\cup\{\delta_E\}$, σ is an identity on $FTV(\Delta_M)$, and therefore also on $FTV(\Delta)$. Since σ and σ_M are identities on $FTV(\Delta_M)$, we have $\tau_x\sigma = \tau_x\sigma_M = \tau_x$. Finally, we have

$$\begin{aligned}
\delta_E\sigma &= Q^{-1}(\rho) \\
&= \tau_x \to Q^{-1}(\rho_M) \\
&= \tau_x\sigma_M \to Q^{-1}(\rho_M) \\
&= \beta_x\sigma_M \to \delta_M\sigma_M \\
&= \beta_x\sigma \to \delta_M\sigma \ ,
\end{aligned}$$

as required.

We now show that the non-rank-2 variables in $Q^{-1}(\rho) = \tau_x \to Q^{-1}(\rho_M)$ do not occur on any left-hand sides in $USUP(E, \Delta)$, unknown occurrences aside. By induction, the non-rank-2 variables in $Q^{-1}(\rho_M)$ do not occur in any left-hand sides in $USUP(M, \Delta_M)\sigma_M$, unknown occurrences aside. Since σ only adds a replacement of δ_E (whose only occurrence is on a right-hand side) to σ_M, and since $USUP(E, \Delta)$ does not introduce onto any left-hand sides any variable whose only occurrences in $USUP(M, \Delta_M)$ (unknown occurrences aside) are on right-hand sides, the non-rank-2 variables in τ_M do not occur on any left-hand sides in $USUP(E, \Delta)\sigma_E$. As for β_x, since E is a polymorphic abstraction, any occurrences of $\beta_x\sigma$ in ρ are rank 2 occurrences; hence τ_x is excluded from consideration. Thus, $Q^{-1}(\rho)$ contains no non-rank-2 variables with left-hand side occurrences outside unknown occurrences, as required.

- Rule [p-abs-annot]. If rule [p-abs-annot] is at the root of the derivation, then E is an annotated polymorphic abstraction $\lambda x : (\forall.\tau).M$, and x is polymorphic in M (i.e., β_x is variable in $USUP(M, (\Delta, x : \forall.\tau\sigma))$ for any σ). Then there is a rank 2 type ρ_M and a substitution σ_x such that $\rho = (\forall.\tau\sigma_x) \to \rho_M$ and the statement $\Delta, x : \forall.\tau\sigma_x \vdash M : \rho_M$ is derivable. Let Δ_M denote the type environment $(\Delta, x : \forall.\tau\sigma_x$. By induction, there is a substitution σ_M that solves $USUP(M, \Delta_M))$, such that $\delta_M\sigma_M = Q^{-1}(\rho_M)$, and σ_M is an identity on $FTV(\Delta_M)$. The instance $USUP(E)$ contains all of $USUP(M, \Delta_M)$, plus the inequalities $\beta_x = \tau\sigma_x$ and $\delta_E = \beta_x \to \delta_M$. If x has at least one occurrence in M, then $USUP(M, \Delta_M)$ contains the equality $\beta_x = \tau\sigma_x$. Then $\beta_x\sigma_M = \tau\sigma_x\sigma_M$. Otherwise, β_x has no occurrences in $USUP(M, \Delta_M)$ and we can let $\sigma_{M,x} = [\tau\sigma_x/\beta_x] \circ \sigma_M$, and continue the argument using $\sigma_{M,x}$. For simplicity, and without loss of generality, however, we will simply assume that $\beta_x \in \text{Vars}(USUP(M, \Delta_M))$, and therefore, $\beta_x \in \text{dom}(\sigma_M)$. Hence, the inequality $\beta_x = \tau\sigma_x$, introduced in $USUP(E, \Delta)$, contributes nothing new to the instance. We may therefore restrict our attention to the inequality $\delta_E = \beta_x \to \delta_M$. Since δ_E has no occurrences in $USUP(M, \Delta_M)$, define the substitution σ as $[Q^{-1}(\rho)/\delta_E] \circ \sigma_M$. Then σ also solves $USUP(M, \Delta_M)$, and since $\text{dom}(\sigma) = \text{dom}(\sigma_M) \cup \{\delta_E\}$, σ is an identity on $FTV(\Delta_M)$, and therefore also on $FTV(\Delta)$. Since σ and σ_M are identities on $FTV(\Delta_M)$, we have $\tau\sigma_x\sigma = \tau\sigma_x\sigma_M = \tau\sigma_x$. Finally, we have

$$\begin{aligned}
\delta_E\sigma &= Q^{-1}(\rho) \\
&= \tau\sigma_x \to Q^{-1}(\rho_M) \\
&= \tau\sigma_x\sigma_M \to Q^{-1}(\rho_M) \\
&= \beta_x\sigma_M \to \delta_M\sigma_M \\
&= \beta_x\sigma \to \delta_M\sigma \ ,
\end{aligned}$$

  as required.

  The argument that no non-rank-2 variables in $Q^{-1}(\rho)$ have any occurrences on a left-hand side in $USUP(E, \Delta)\sigma$, unknown occurrences aside, is identical to case [p-abs].

- Rule [spec]. If rule [spec] is at the root of the derivation, then there is a rank 2 type ρ_E, a rank 0 type τ, and a type variable α such that $\rho = \rho_E[\tau/\alpha]$ and $\Delta \vdash E : \forall\alpha.\rho_E$ is derivable. By induction, there is a substitution σ_E that solves $USUP(E, \Delta)$, such that $\delta_E\sigma = Q^{-1}(\rho_E)$, and σ_E is an identity on $FTV(\Delta)$. The type variable α is quantified at the outermost scope in $\forall\alpha.\rho_E$; hence α is not a rank-2 variable in $Q^{-1}(\rho)$. Hence σ_E can be

chosen such that α has no occurrences on any left-hand sides in $USUP(E,\Delta)\sigma_E$. We can write $USUP(E,\Delta) = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i}\rangle\}_{i=1}N$ for some N, $\tau_1,\ldots,\tau_N$, $\mu_1,\ldots,\mu_N$. Then there exist substitutions $\sigma_{E_1},\ldots,\sigma_{E_N}$ such that

$$\begin{aligned} \tau_1\sigma_E\sigma_{E_1} &= \mu_1\sigma_E \\ &\cdots \\ \tau_N\sigma_E\sigma_{E_N} &= \mu_N\sigma_E\ . \end{aligned}$$

Therefore, we have

$$\begin{aligned} \tau_1\sigma_E\sigma_{E_1}[\tau/\alpha] &= \mu_1\sigma_E[\tau/\alpha] \\ &\cdots \\ \tau_N\sigma_E\sigma_{E_N}[\tau/\alpha] &= \mu_N\sigma_E[\tau/\alpha]\ . \end{aligned}$$

Since α does not occur on any left-hand sides in $USUP(E,\Delta)\sigma_E$, we can introduce $[\tau/\alpha]$ on each left-hand side without affecting the equalities:

$$\begin{aligned} \tau_1\sigma_E[\tau/\alpha]\sigma_{E_1}[\tau/\alpha] &= \mu_1\sigma_E[\tau/\alpha] \\ &\cdots \\ \tau_N\sigma_E[\tau/\alpha]\sigma_{E_N}[\tau/\alpha] &= \mu_N\sigma_E[\tau/\alpha]\ . \end{aligned}$$

But this system now says that $[\tau/\alpha]\circ\sigma_E$ solves $USUP(E,\Delta)$. Furthermore,

$$\delta_E([\tau/\alpha]\circ\sigma_E) \;=\; \delta_E\sigma_E[\tau/\alpha] \;=\; \tau_\rho[\tau/\alpha]\ .$$

Since α is a quantified variable in $\forall\alpha.\rho_E$, and quantification can only be introduced by application of rule [gen], it follows that α is not free in Δ. By induction, σ_E is an identity on $FTV(\Delta)$. Since σ only adds to σ_E a replacement of α, which is not free in Δ, it follows that σ is also an identity on $FTV(\Delta)$. Also by induction, each non-rank-2 variable in $Q^{-1}(\rho)$ does not occur on any left-hand side in $USUP(E,\Delta)$, unknown occurrences aside. Any variables τ has in common with $Q^{-1}(\rho)$ obviously have this property as well. For any variables in τ but not in $Q^{-1}(\rho)$, either these variables, or the variables in $USUP(E,\Delta)$ can be renamed such that the two sets of variable names are disjoint. As a result, any non-rank-2 variables in $Q^{-1}(\rho)[\tau/\alpha]$ have no occurrences on any left-hand sides in $USUP(E,\Delta)\sigma_E[\tau/\alpha]$, unknown occurrences aside, which is the needed result.

- Rule [gen]. If rule [gen] is at the root of the derivation, then there is rank 2 type ρ such that $\Delta \vdash E : \rho$ is derivable. By induction, there is a substitution σ_E that solves $USUP(E, \Delta)$, such that $\delta_E \sigma = Q^{-1}(\rho)$, σ is an identity on $FTV(\Delta)$, and any non-rank-2 variables in $Q^{-1}(\rho)$ have no occurrences on left-hand sides in $USUP(E, \Delta)\sigma_E$, unknown occurrences aside. Then, since $Q^{-1}(\forall\alpha.\rho) = Q^{-1}(\rho)$, the same substitution σ_E can be used to satisfy the theorem for the statement $\Delta \vdash E : \forall\alpha.\rho$, and the result follows immediately.

Having exhausted the possible forms of a valid type derivation for an expression E, we conclude by induction that a solution σ exists for $USUP(E, \Delta)$, according to the conditions laid out in the statement of the theorem. □

The statement and proof of the completeness theorem are long and technical—much of the technical awkwardness arises from the fact that certain variables in the unquantified type $Q^{-1}(\rho)$ are meant to be rank 2 variables, while others are meant to be of lower rank; hence, they must be treated differently from one another. For example, in the context of the [spec] rule, only non-rank-2 variables are eligible for substitution—indeed, the rank 2 variables arising from polymorphic abstractions (i.e., rules [p-abs] and [p-abs-annot]) do not satisfy the condition of the theorem requiring no occurrences on a left-hand side. This condition, however, is essential for the application of the additional substitution in the [spec] rule to yield a valid solution for the USUP instance.

Despite its complexity, the completeness theorem implies a relatively straightforward result: any type ρ derivable for an expression E arises as a solution for $USUP(E)$, which is the result we sought.

4.5.3 Termination

Our syntax-directed translation from rank 2-typability to USUP is sound and complete. In order to establish USUP, together with the associated translation procedure, as a viable alternative to the original SUP translation, it remains to establish a termination result. In particular, it remains to show that for any labelled λ-term E, the USUP redex procedure terminates on input $USUP(E)$.

In Section 4.4.2, we showed that the USUP redex procedure terminates on all USUP instances that possess a solution; hence, whenever a labelled λ-term E is rank 2-typable, the USUP redex procedure will terminate on input E, and we will obtain a valid type for E. For the general case, however, we need a more general result. In Section 4.4.2, we also showed that the USUP

redex procedure terminates on all R-acyclic instances of USUP. Hence, establishing R-acyclicity for $G(USUP(E))$, for an arbitrary E, would give us the result we seek.

It turns out, however, that even for a simple, θ-normal, source term, this result (that is, R-acyclicity in the graph of the corresponding USUP instance) is false. Consider the following example:

$$E = (\lambda^p y.y)(\lambda^m z.z) \ .$$

Then $USUP(E)$ contains the following inequalities[4]:

$$\begin{aligned}
\alpha \to \alpha &\leq \delta_{\lambda^p y.y} \to (\delta_{\lambda^m z.z} \to \delta_E) \\
\alpha \to \alpha &\leq \delta_{\lambda^p y.y} \to (\beta_y \to \delta_y) \\
\beta_y &\leq \delta_y \\
\alpha \to \alpha &\leq \delta_{\lambda^m z.z} \to (\beta_z \to \delta_z) \\
\underline{\beta_z} &\leq \delta_z \ .
\end{aligned}$$

Consider now the graph $G = G(USUP(E))$. The relevant portions of G for our purposes are the subgraphs containing the second and third, and fourth and fifth inequalities. These subgraphs are presented in Figure 4.3. The upper edge in Figure 4.3 exists because the variable β_y occurs

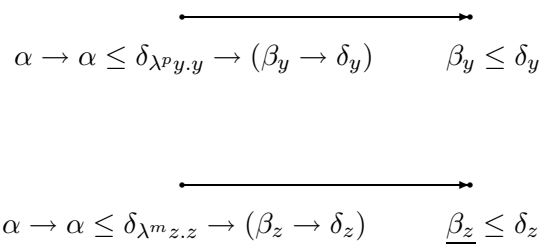

Figure 4.3: Portions of the USUP graph for the expression $E = (\lambda^p y.y)(\lambda^m z.z)$.

on the right-hand side of $\alpha \to \alpha \leq \delta_{\lambda^p y.y} \to (\beta_y \to \delta_y)$ and on the left-hand side of $\beta_y \leq \delta_y$. Hence every variable on the right-hand side of $\alpha \to \alpha \leq \delta_{\lambda^p y.y} \to (\beta_y \to \delta_y)$ is R' related to every variable on the right-hand side of $\beta_y \leq \delta_y$. In particular, $\delta_y R' \delta_y$. Since every variable is R-related to itself, via a path of length 0, we have $\delta_y R' \delta_y R \delta_y$, which is a violation of the condition for R-acyclicity. Similarly, in the lower edge, we have $\delta_z R' \delta_z R \delta_z$. In general, whenever an identifier

[4]For illustrative purposes, equalities are written explicitly here as inequalities.

occurs on the right-hand sides of two distinct inequalities that are joined by a directed path, there is an R-acyclicity violation.

In the case of the lower edge in Figure 4.3, the R-acyclicity violation is not a serious problem:

Proposition 4.1 *If $\underline{\gamma}$ is an unknown, then for every expression τ, the inequalities $\underline{\gamma} \leq \tau$ and $\underline{\gamma} = \tau$ have the same set of solutions.*

Proof Let σ be a substitution, and suppose σ solves $\underline{\gamma} \leq \tau$. Then there is a substitution σ' such that $\gamma\sigma\sigma' = \tau\sigma$, where $\mathrm{dom}(\sigma')$ contains no unknowns. In the context of this inequality, then, we have $\mathrm{dom}(\sigma') \cap \mathrm{Vars}(\gamma\sigma) = \emptyset$. Hence, $\gamma\sigma\sigma' = \gamma\sigma$, and therefore, $\gamma\sigma = \tau\sigma$. The second inequality is, in fact, $\alpha \to \alpha \leq \gamma \to \tau$, where α is a fresh inequality (hence $\alpha \notin \mathrm{dom}(\sigma)$). Applying σ gives $\alpha\sigma \to \alpha\sigma \leq \gamma\sigma \to \tau\sigma$, which simplifies to $\alpha \to \alpha \leq \gamma\sigma \to \tau\sigma$. Taking $\sigma' = [\gamma\sigma/\alpha]$, we have $\alpha\sigma' \to \alpha\sigma' = \gamma\sigma \to \tau\sigma$; hence σ solves $\underline{\gamma} = \tau$. Conversely, if σ solves $\underline{\gamma} = \tau$, then there is a substitution σ' such that $\alpha\sigma\sigma' \to \alpha\sigma\sigma' = \gamma\sigma \to \tau\sigma$. Then we have $\gamma\sigma = \alpha\sigma\sigma' = \tau\sigma$. Applying σ to $\underline{\gamma} \leq \tau$ gives $\underline{\gamma}\sigma = \tau\sigma$, and since $\underline{\gamma}\sigma = \tau\sigma$, we can take $\sigma' = []$, and we have $\underline{\gamma}\sigma\sigma' = \tau\sigma$. Since $\mathrm{dom}(\sigma') = \emptyset$, it contains no unknowns. Hence, σ solves $\underline{\gamma} \leq \tau$. □

Proposition 4.2 *If an unknown $\underline{\gamma}$ within an R-AUSUP instance Γ only occurs on right-hand sides within the inequalities in which it is an unknown, then $\underline{\gamma}$ may be simply treated as a variable, without affecting the principal solution (if one exists) of Γ; moreover $\underline{\gamma}$ is indistinguishable from an ordinary variable by the USUP redex procedure.*

Proof Let $\Gamma = \{\langle \tau_i \leq \mu_i, \vec{\alpha_i} \rangle\}_{i=1}^N$. Let Γ_S denote the R-ASUP instance derived from Γ by treating all unknowns as ordinary variables. Then a substitution σ solves Γ iff for all i, there is a substitution σ_i, such that $\tau_i\sigma\sigma_i = \mu_i\sigma$, and $\mathrm{dom}(\sigma_i) \cap \mathrm{Vars}(\vec{\alpha_i}\sigma) = \emptyset$, and a substitution σ_S solves Γ_S iff for all i, there is a substitution σ_{Si}, such that $\tau_i\sigma\sigma_{Si} = \mu_i\sigma$. We show that if σ_S is a principal solution for Γ_S, then σ_S solves Γ. Suppose, for some i, that $\underline{\gamma} \in \vec{\alpha_i}$, but $\underline{\gamma} \notin \mathrm{Vars}(\tau_i)$. If some variable $\alpha \in \mathrm{Vars}(\underline{\gamma}\sigma_S)$ has an occurrence in τ_i, then some variable $\beta \in \mathrm{Vars}(\tau_i)$ was at some point replaced, via redex reduction, by an expression containing α. Hence, $\beta R \alpha$ and $\alpha R \beta$. Also, we have $\underline{\gamma} R \alpha$ and $\alpha R \underline{\gamma}$. Now, if β has no occurrences on a right-hand side, then there is a solution σ'_S of Γ that does not replace β and is otherwise equivalent to σ in terms of its actions on right-hand sides. Assume, therefore, that β does have an occurrence on a right-hand side. Then $\beta R' \underline{\gamma}$, which gives $\beta R' \underline{\gamma} R \alpha R \beta$, which contradicts R-acyclicity. Hence, no variable in $\mathrm{Vars}(\underline{\gamma}\sigma_S)$ has an occurrence in τ_i. Therefore, σ_{Si} can be chosen such that its domain contains

no occurrences of any variable in Vars($\underline{\gamma}\sigma$), and we find that σ_S solves Γ. The reverse direction (i.e., that σ solves Γ_S) is trivial, as USUP is a superset of SUP.

For the second part of the claim, consider the actions of the four kinds of redex reduction on an unknown $\underline{\gamma}$ that only has occurrences on right-hand sides within inequalities in which it is an unknown. In the case of a redex-I, any occurrence $\underline{\gamma}$ on the right-hand side will be ignored, because it is not strictly a variable. Redex-II reductions do not distinguish between unknowns and variables; hence $\underline{\gamma}$ will be treated the same under redex-II reduction, whether or not it is labelled as an unknown. Redex-III reduction does not apply, as it only treats unknown occurrences on left-hand sides. Redex-IV reduction reduces an unknown occurrence on the right-hand side with a structural copy of the corresponding expression on the left-hand side—this is precisely the reduction performed under redex-I reduction for ordinary variables. Hence, redex-IV reduction makes up for the behaviour that is lacking in redex-I reduction, so that, indeed, the redex procedure treats γ in the same way as $\underline{\gamma}$. □

Because of Proposition 4.1, we can replace $\underline{\beta_z} \leq \delta_z$ in Figure 4.3 with $\underline{\beta_z} = \delta_z$, without changing the solution, or the solvability, of the USUP instance. In so doing, because, $\underline{\beta_z} = \delta_z$ is actually the inequality $\alpha \to \alpha \leq \underline{\beta_z} \to \delta_z$ (where α is fresh), the unknown $\underline{\beta_z}$ moves from the left-hand side of the inequality to the right-hand side. As a result, the edge between $\alpha \to \alpha \leq \delta_{\lambda^m z.z} \to (\beta_z \to \delta_z)$ and what was formerly $\underline{\beta_z} \leq \delta_z$ no longer exists. Hence, we no longer have $\delta_z R' \delta_z$, and the corresponding R-acyclicity violation is gone. Thus, for convenience, we can simply assume that the USUP translation outputs the equality instead of the inequality, so that the R-acyclicity violation does not occur.

Furthermore, because of Proposition 4.2, for subexpressions E_i of the source program E, such that $\mathit{USUP}(E_i)$ only contains unknown occurrences on right-hand sides (in particular, after applying the convenience transformation afforded by Proposition 4.1), we can regard the USUP instance $\mathit{USUP}(E_i)$ as equivalent to the underlying SUP instance, and dispense with discussion of unknowns in the context of E_i. We call the simplifying assumptions afforded by Propositions 4.1 and 4.2 the *convenience assumptions*.

In the case of the top edge in Figure 4.3, because β_y is *not* an unknown, the inequality $\beta_y \leq \delta_y$ is not equivalent to the equality $\beta_y = \delta_y$, and the R-acyclicity violation is not so easily eliminated. Note, however, that the R-acyclicity violation in the upper edge does not lead to non-termination. Instead it reduces as follows. Reduction of the redex-I in the last inequality (which then becomes

solved) yields

$$\begin{aligned}
\alpha \to \alpha &\leq \delta_{\lambda^p y.y} \to (\delta_{\lambda^m z.z} \to \delta_E) \\
\alpha \to \alpha &\leq \delta_{\lambda^p y.y} \to (\beta_y \to \delta_y) \\
\beta_y &\leq \delta_y \\
\alpha \to \alpha &\leq \delta_{\lambda^m z.z} \to (\beta_z \to \beta_z) \ .
\end{aligned}$$

Reduction of the redex-II in the fourth inequality (which then becomes solved) yields

$$\begin{aligned}
\alpha \to \alpha &\leq \delta_{\lambda^p y.y} \to ((\beta_z \to \beta_z) \to \delta_E) \\
\alpha \to \alpha &\leq \delta_{\lambda^p y.y} \to (\beta_y \to \delta_y) \\
\beta_y &\leq \delta_y \ .
\end{aligned}$$

Reduction of the redex-II in the second inequality (which then becomes solved) yields

$$\begin{aligned}
\alpha \to \alpha &\leq (\beta_y \to \delta_y) \to ((\beta_z \to \beta_z) \to \delta_E) \\
\beta_y &\leq \delta_y \ .
\end{aligned}$$

Reduction of the redex-II in the first inequality (which then becomes solved) yields

$$\begin{aligned}
\alpha \to \alpha &\leq ((\beta_z \to \beta_z) \to \delta_E) \to ((\beta_z \to \beta_z) \to \delta_E) \\
\beta_z \to \beta_z &\leq \delta_E \ .
\end{aligned}$$

Finally, a redex-I reduction yields

$$\begin{aligned}
\alpha \to \alpha &\leq ((\beta_z \to \beta_z) \to (\beta \to \beta)) \to ((\beta_z \to \beta_z) \to (\beta \to \beta)) \\
\beta_z \to \beta_z &\leq \beta \to \beta \ ,
\end{aligned}$$

at which point the entire instance is solved. The image of δ_E under the solution yields the final type, $\forall\beta.\beta \to \beta$, as expected. Thus, even though we do not have R-acyclicity for this problem instance, we do have termination.

In general, any time a polymorphic abstraction $E = \lambda^p x.M$ actually makes use of its argument x, this kind of R-acyclicity violation will occur. If E makes use of x, then there will be at least one inequality $\beta_x \leq \delta_x$ in $USUP(E)$. But then M's type will in general be dependent upon x's type, so that δ_x and δ_M are R-related. Further, because of the equality $\delta_E = \beta_x \to \delta_M$, δ_M and

δ_x will be R'-related, thus yielding an R-cycle comprising at least one non-trivial path. Our task, then, is to show that these violations can never lead to non-termination.

It is worth pointing out that the original KW translation does not suffer from this acyclicity violation, because it never actually represents the type of a polymorphic abstraction in the translation; for an expression

$$\lambda^2 x_1 \cdots x_m.(\lambda^1 y_1.(\cdots((\lambda^1 y_n.E_{n+1})E_n)\cdots)E_2)E_1 \ ,$$

the KW translation only translates each subexpression E_i into ASUP, and then adds additional equalities to match up the parameter and argument types for each λ^1-abstraction. The final result type is given as the image of E_{n+1} under the translation, and the final parameter types are added on at the end by prepending the bindings for each x_j in a user-supplied environment. By never explicitly including the types of the λ^1- and λ^2-abstractions in the translation, the KW translation avoids the kind of acyclicity violation we have encountered above; had they explicitly translated the polymorphic abstractions, the acyclicity violations (which violate not only R-acyclicity, but, as a consequence, ASUP-acyclicity as well) would indeed have surfaced.

To begin our analysis, we will assume that the expression E to be typed is θ-normal (i.e., in the form presented above, in which no expression E_i contains a redex), and then relax this assumption as our analysis progresses. Our first observation is the following:

Proposition 4.3 *For an expression*

$$E = \lambda^2 x_1 \cdots x_m.(\lambda^1 y_1.(\cdots((\lambda^1 y_n.E_{n+1})E_n)\cdots)E_2)E_1$$

in some environment, each USUP(E_i) is R-acyclic. Further, the concatenation of all USUP(E_i) is also R-acyclic.

Proof The proof follows immediately from the R-acyclicity of the modified SUP translation from Section 3.3.1—aside from the introduction of unknowns (which have no effect on an instance's graph structure) and some variable name changes, the two translation procedures produce identical (U)SUP instances for each E_i. □

Thus, termination is assured for the parts of $USUP(E)$ corresponding to each E_i. Now, for each redex $(\lambda^1 y_i.M)E_i$, in addition to the inequalities in $USUP(E_i)$ and $USUP(M)$, we also have the following two inequalities:

$$\begin{aligned} \delta_{\lambda^1 y_i.M} &= \delta_{E_i} \to \delta_{(\lambda^1 y_i.M)E_i} \\ \delta_{\lambda^1 y_i.M} &= \beta_{y_i} \to \delta_M \ . \end{aligned}$$

All other occurrences of β_{y_i} are in inequalities of the form $\beta_{y_i} \leq \delta_{y_i,k}$. In particular, all other occurrences of β_{y_i} are on left-hand sides, and none of them are unknown occurrences, as $\lambda^1 y_i.M$ is a polymorphic abstraction.

Now, we claim that, if this R-acyclicity violation leads to an infinite reduction sequence, then that reduction sequence must include substitutions that replace β_{y_i}. To see this, consider the replacement of β_{y_i} in $\delta_{\lambda^1 y_i.M} = \beta_{y_i} \to \delta_M$ (or indeed in each of the $\beta_{y_i} \leq \delta_{y_i,k}$) by some fresh variable β_0, so that the edge created by β_{y_i} is broken. Under the assumption that E_i and M are already R-acyclic, the graph as a whole then becomes R-acyclic, and termination is then assured. The only difference between this USUP instance and the original is the ability of a substitution to propagate across the edge between $\delta_{\lambda^1 y_i.M} = \beta_{y_i} \to \delta_M$ and $\beta_{y_i} \leq \delta_{y_i,k}$, which the former possess and the latter lacks. Hence an infinite reduction sequence must involve the propagation of a substitution along one such edge. In particular, an infinite reduction sequence must involve a replacement of some β_{y_i}. But the redex procedure only replaces variables that have an occurrence on a right-hand side. Hence, a replacement of β_y can only take place via the two inequalities presented above. The effect, however, of such reductions, is that (after solved vertices are removed) the R-acyclicity violation disappears, and along with it, the potential for non-termination.

We formalize the situation as follows:

Proposition 4.4 *Let*

$$E = \lambda^2 x_1 \cdots x_m.(\lambda^1 y_1.(\cdots((\lambda^1 y_n.E_{n+1})E_n)\cdots)E_2)E_1$$

be a θ-normal, labelled λ-term. Then the USUP redex procedure terminates on input $USUP(E)$.

Proof For each $i \in \{1, \ldots, n+1\}$, let $\Gamma_i = USUP(E_i)$, and let Γ_i' be the corresponding underlying SUP instance. By the convenience assumptions, we may regard each Γ_i as equivalent to the corresponding Γ_i'. Then each Γ_i is R-acyclic, and so is the concatenation $\Gamma_1 \cdots \Gamma_{n+1}$. Hence, termination is assured for the instance $\Gamma_1 \cdots \Gamma_{n+1}$. To this instance, for each i, the instance $USUP(E)$ adds the following inequalities:

$$\begin{aligned} \delta_{\lambda^1 y_i.M_i} &= \beta_{y_i} \to \delta_{M_i} \\ \delta_{\lambda^1 y_i.M_i} &= \delta_{E_{i+1}} \to \delta_{(\lambda^1 y_i.M_i)E_{i+1}} \ , \end{aligned}$$

for each λ^1-abstraction, and for $j \in \{1, \ldots, m\}$ (i.e., for each λ^2-abstraction) the inequalities

$$\delta_{\lambda^2 x_i.N_j} = \beta_{x_j} \to \delta_{N_j} \ ,$$

where $M_n = E_{n+1}$, $M_i = (\lambda^1 y_{i+1}.M_{i+1})E_{i+1}$, $N_m = M_0$, and $N_j = \lambda^2 x_{j+1}.N_{j+1}$. If the redex procedure never reduces these inequalities, then they are effectively not present, and the procedure terminates. Hence, non-termination implies that the procedure must reduce at least one of these inequalities. Whatever the choice of i, or j, however, there are only two possible replacements:

- for some value of i, the reduction replaces $\delta_{\lambda^1 y_i.M_i}$. This reduction can either come from the inequality $\delta_{\lambda^1 y_i.M_i} = \beta_{y_i} \to \delta_{M_i}$ or the inequality $\delta_{\lambda^1 y_i.M_i} = \delta_{E_{i+1}} \to \delta_{(\lambda^1 y_i.M_i)E_{i+1}}$. In the former case, we obtain the substitution $[\beta_{y_i} \to \delta_{M_i}/\delta_{\lambda^1 y_i.M_i}]$, and in the latter case, we obtain the substitution $[\delta_{E_{i+1}} \to \delta_{(\lambda^1 y_i.M_i)E_{i+1}}/\delta_{\lambda^1 y_i.M_i}]$. Since the only two occurrences of $\delta_{\lambda^1 y_i.M_i}$ are within these two inequalities, reduction either way renders the originating inequality solved, and the other inequality becomes $\beta_{y_i} \to \delta_{M_i} = \delta_{E_{i+1}} \to \delta_{(\lambda^1 y_i.M_i)E_{i+1}}$. All other inequalities in $USUP(E)$ remain unchanged. Thus, an infinite reduction in $USUP(E)$ is still impossible, unless further reduction of one of the added inequalities takes place. If, for some i', the additional reduction replaces $\delta_{\lambda^1 y_{i'}.M_{i'}}$, then the reduction proceeds as outlined here, and still the remainder of the instance is unchanged. If instead, some $\lambda^2 x_j.N_j$ is reduced, we argue as in the next case. Otherwise, the additional substitution reduces the newly-established inequality, $\beta_{y_i} \to \delta_{M_i} = \delta_{E_{i+1}} \to \delta_{(\lambda^1 y_i.M_i)E_{i+1}}$, thereby yielding $[\delta_{E_{i+1}}/\beta_{y_i}, \delta_{(\lambda^1 y_i.M_i)E_{i+1}}/\delta_{M_i}]$. By the end of this substitution, however, both of the inequalities $\delta_{\lambda^1 y_i.M_i} = \beta_{y_i} \to \delta_{M_i}$ and $\delta_{\lambda^1 y_i.M_i} = \delta_{E_{i+1}} \to \delta_{(\lambda^1 y_i.M_i)E_{i+1}}$ will now be solved, and thus of no further interest. Furthermore, in the remainder of the instance, we will have equated β_{y_i} with $\delta_{E_{i+1}}$ and $\delta_{(\lambda^1 y_i.M_i)E_{i+1}}$ with δ_{M_i}. This, however, is precisely the way in which the original R-ASUP translation, from Section 3.3.1, treats β-redexes. Hence by the termination property for that translation, this reduction cannot produce a non-terminating USUP instance.

- for some value of j, the reduction replaces $\delta_{\lambda^2 x_j.N_j}$ or β_{x_j}. This reduction can either come from the inequality $\delta_{\lambda^2 x_j.N_j} = \beta_{x_j} \to \delta_{N_j}$ or from the inequality $\beta_{x_j} = \tau_j$, where $\forall.\tau_j$ is the user- or environment-supplied type annotation for x_j (if none is supplied, we can assume for uniformity that τ_j is identically equal to β_j, so that, effectively, the type $\perp$ was supplied). If $\delta_{\lambda^2 x_j.N_J}$ was replaced, then the replacement could only come from the inequality $\delta_{\lambda^2 x_j.N_j} = \beta_{x_j} \to \delta_{N_j}$, which produces the substitution $[\beta_{x_j} \to \delta_{N_j}/\delta_{\lambda^2 x_j.N_j}]$ (and is then rendered solved—and therefore no longer of interest—upon reduction). There is at most one other occurrence of $\delta_{\lambda^2 x_j.N_j}$ in the instance, within the inequality $\delta_{\lambda^2 x_{j-1}.N_{j-1}} = \beta_{x_{j-1}} \to \delta_{N_{j-1}}$, since $N_{j-1} = \lambda^2 x_j.N_j$. This inequality would then become $\delta_{\lambda^2 x_{j-1}.N_{j-1}} = \beta_{x_{j-1}} \to (\beta_{x_j} \to$

δ_{N_j}). Since this inequality is not part of $\Gamma_1 \cdots \Gamma_{n+1}$, it does not induce in infinite reduction without further reduction of the additional inequalities present in $USUP(E)$. If, instead, β_{x_j} is replaced, then the reduction comes from the inequality $\beta_{x_j} = \tau_j$, which yields the substitution $[\tau_j/\beta_j]$. This substitution alone has no effect on the R-acyclicity of the instance $\Gamma_1 \cdots \Gamma_{n+1}$, because the inequality $\beta_{x_j} = \tau_j$ is present in the original translation. However, it changes the inequality $\delta_{\lambda^2 x_j.N_j} = \beta_{x_j} \to \delta_{N_j}$ to $\delta_{\lambda^2 x_j.N_j} = \tau_j \to \delta_{N_j}$. Nevertheless, the small change in this inequality is immaterial, as reducing it still only affects the additional inequalities in $USUP(E)$, so that further reduction would still be required to effect an infinite reduction sequence.

In summary, none of the additional inequalities from $USUP(E)$, over and above those in $\Gamma_1 \cdots \Gamma_{n+1}$ can bring about an infinite reduction, without requiring more of the additional inequalities to be reduced as well. Since they are only finite in number, we conclude that an infinite reduction in $USUP(E)$ is simply not possible. □

Proposition 4.4 establishes termination for the USUP redex procedure on input $USUP(E)$, for any θ-normal, labelled expression E. We now expand on this result to accommodate expressions that are *not* θ-normal.

Proposition 4.5 *Let E be a labelled λ-term such that the USUP redex procedure terminates on input $USUP(E)$. Suppose $E' \to_{\theta_4} E$. Then the USUP redex procedure terminates on input $USUP(E')$.*

Proof There exists a context C, with a single hole, and subexpressions N and P, such that $E = C[\lambda^2 x.(\lambda^1 y.N)P]$, and $E' = C[(\lambda^1 y.\lambda^2 x.N)P]$. The context being the same in both cases, it is sufficient to show termination for $(\lambda^1 y.\lambda^2 x.N)P$. Denote by $USUP_E(N)$ and $USUP_{E'}(N)$, respectively, the USUP translation of N in the context of expressions E and E', and similarly for P. Since N occurs in the same scope in both E and E' (i.e., in scope of both x and y), we have $USUP_E(N) = USUP_{E'}(N)$. As for P, although P lies within x's scope in E but not in E', since variables are assumed to be distinctly named, we can be assured that P has no free occurrences of x, that would be captured upon θ-reduction. Hence, $USUP_E(P) = USUP_{E'}(P)$; in particular, the former actually contains no occurrence of $\beta_x \leq \delta_{x_i}$ for any occurrence x_i of x, because there are no such occurrences. Thus, the inequalities in $USUP_E(P)$ and $USUP_E(N)$ are common to

both $USUP(E)$ and $USUP(E')$. The following inequalities are unique to $USUP(E)$:

$$\begin{aligned} \delta_{\lambda^2 x.(\lambda^1 y.N)P} &= \beta_x \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \delta_P \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \beta_y \to \delta_N \ , \end{aligned}$$

and the following inequalities are unique to $USUP(E')$:

$$\begin{aligned} \delta_{\lambda^1 y.\lambda^2 x.N} &= \delta_P \to \delta_{(\lambda^1 y.\lambda^2 x.N)P} \\ \delta_{\lambda^1 y.\lambda^2 x.N} &= \beta_y \to \delta_{\lambda^2 x.N} \\ \delta_{\lambda^2 x.N} &= \beta_x \to \delta_N \ . \end{aligned}$$

If $USUP(E')$ contains an infinite reduction, therefore, it must arise as a result of reducing at least one of these three inequalities. Reducing either the first or the second inequality only replaces the variable $\delta_{\lambda^1 y.\lambda^2 x.N}$, which has no occurrences $USUP_{E'}(N)\,USUP_{E'}(P)$. Hence, this reduction alone cannot cause an infinite reduction sequence, without further reduction of the above inequalities. However, it will not prevent an infinite reduction either, and so we can apply the reduction[5] and work with the simplified set of inequalities:

$$\begin{aligned} \delta_P \to \delta_{(\lambda^1 y.\lambda^2 x.N)P} &= \beta_y \to \delta_{\lambda^2 x.N} \\ \delta_{\lambda^2 x.N} &= \beta_x \to \delta_N \ . \end{aligned}$$

Now, if an infinite reduction sequence is to result, it must come from reducing one or both of these inequalities. By the same reasoning, since the variable $\delta_{\lambda^2 x.N}$ has no other occurrences besides the two above, replacing it via reduction of the second inequality will neither cause nor prevent an infinite reduction, and so we go ahead and perform the replacement:

$$\delta_P \to \delta_{(\lambda^1 y.\lambda^2 x.N)P} = \beta_y \to (\beta_x \to \delta_N) \ .$$

Now, either reducing this inequality causes non-termination, or nothing does. Reduction would yield the substitution $[\delta_P/\beta_y, \beta_x \to \delta_N/\delta_{(\lambda^1 y.\lambda^2 x.N)P}]$. The replacement $[\delta_P/\beta_y]$ occurs by reducing the last two inequalities among those unique to $USUP(E)$; hence by termination for $USUP(E)$, this replacement cannot cause an infinite reduction in $USUP(E')$. We are left, therefore, with

[5] Whether we reduce the first or the second inequality, once the solved inequality is removed, the result is the same.

the replacement $[\beta_x \rightarrow \delta_N/\delta_{(\lambda^1 y.\lambda^2 x.N)P}]$. But $(\lambda^1 y.\lambda^2 x.N)P$ is the top-level expression, and has no other occurrence in $USUP(E')$. Hence, the replacement cannot cause non-termination. Therefore, the USUP redex procedure must terminate on input $USUP(E')$, as claimed. □

Thus, an expression that is some number of θ_4-reductions away from θ-normal form still gives rise to a terminating USUP instance. We continue in a similar vein for the other forms of θ-reduction.

Proposition 4.6 *Let E be a labelled λ-term such that the USUP redex procedure terminates on input $USUP(E)$. Suppose $E' \rightarrow_{\theta_3} E$. Then the USUP redex procedure terminates on input $USUP(E')$.*

Proof As in Proposition 4.5, we can ignore the surrounding context, and simply assume that $E = (\lambda^1 y.NP)Q$, and $E' = N((\lambda^1 y.P)Q)$. Let $USUP_E(N)$ and $USUP_{E'}(N)$ denote, respectively, the USUP translation of N in the context of E and E', and similarly for P and Q. Since P and Q have the same scope in both E and E', we have $USUP_E(P) = USUP_{E'}(P)$ and $USUP_E(Q) = USUP_{E'}(Q)$. As for N, it lies within y's scope in E, but not in E'. Since E' is the source term, however, by our unique naming assumption, N can contain no occurrences of y; hence $USUP_E(N)$ contains no occurrences of an inequality of the form $\beta_y \leq \delta_{y_i}$ for some occurrence y_i of y. Thus, $USUP_{E'}(N) = USUP_E(N)$ as well. By termination for $USUP(E)$, the concatenated instance $USUP_E(N)\,USUP_E(P)\,USUP_E(Q)$, which is common to both $USUP(E)$ and $USUP(E')$, cannot give rise to an infinite reduction sequence. The following inequalities are unique to $USUP(E)$:

$$\begin{aligned} \delta_{\lambda^1 y.NP} &= \delta_Q \rightarrow \delta_{(\lambda^1 y.NP)Q} \\ \delta_{\lambda^1 y.NP} &= \beta_y \rightarrow \delta_{NP} \\ \delta_N &= \delta_P \rightarrow \delta_{NP} \ , \end{aligned}$$

and the following inequalities are unique to $USUP(E')$:

$$\begin{aligned} \delta_N &= \delta_{(\lambda^1 y.P)Q} \rightarrow \delta_{N((\lambda^1 y.P)Q)} \\ \delta_{\lambda^1 y.P} &= \delta_Q \rightarrow \delta_{(\lambda^1 y.P)Q} \\ \delta_{\lambda^1 y.P} &= \beta_y \rightarrow \delta_P \ . \end{aligned}$$

Any reduction in $USUP(E')$ that gives rise to an infinite reduction sequence must occur as a result of reducing at least one of these three inequalities. As we argued in the proof of Proposition 4.5,

because the variable $\delta_{\lambda^1 y.P}$ has no occurrences other than above, we can replace it and thereby simplify the remaining inequalities:

$$\begin{aligned} \delta_N &= \delta_{(\lambda^1 y.P)Q} \to \delta_{N((\lambda^1 y.P)Q)} \\ \beta_y \to \delta_P &= \delta_Q \to \delta_{(\lambda^1 y.P)Q} \ . \end{aligned}$$

If we reduce the second inequality, we obtain the replacement $[\delta_Q/\beta_y, \delta_{(\lambda^1 y.P)Q}/\delta_P]$. The replacement $[\delta_Q/\beta_y]$ is available in $USUP(E)$, by reducing the first two inequalities presented above; hence, this replacement cannot lead to non-termination. The variable $\delta_{(\lambda^1 y.P)Q}$ only has occurrences within the above inequalities; hence replacing it can neither single-handedly cause nor prevent non-termination in the expression as a whole. The replacement yields a single remaining unsolved inequality:

$$\delta_N = \delta_P \to \delta_{N((\lambda^1 y.P)Q)} \ .$$

If there is to be a non-terminating reduction sequence in $USUP(E')$, then it must be a reduction of this inequality—in particular, the replacement $[\delta_P \to \delta_{N((\lambda^1 y.P)Q)}/\delta_N]$—that creates it. Within $USUP(E)$, the inequality $\delta_N = \delta_P \to \delta_{NP}$ gives rise to the similar replacement, $[\delta_P \to \delta_{NP}/\delta_N]$. The difference between these is merely that the replacement for E' has an occurrence of $\delta_{N((\lambda^1 y.P)Q}$, where the replacement for E has an occurrence of δ_{NP}. This difference is not surprising, as E does not contain $N((\lambda^1 y.P)Q)$ as a subexpression, and E' does not contain NP as a subexpression. However, these two variables actually denote the same type. As we have seen via the replacement $[\delta_{(\lambda^1 y.P)Q}/\delta_P]$, the variables $\delta_{(\lambda^1 y.P)Q}$ and δ_P have the same value in $USUP(E')$. Further, every reduction we have performed in E' has also been available in E; thus, if σ represents the substitutions performed so far, then $\delta_P\sigma$ has the same value in both $USUP(E)$ and $USUP(E')$. Therefore, $\delta_P\sigma$ in $USUP(E)$ is equal to $\delta_{(\lambda^1 y.P)Q}\sigma$ in $USUP(E')$. Similarly, $\delta_N\sigma$ has the same value in both $USUP(E)$ and $USUP(E')$ (before reduction of this last inequality). Therefore, $N((\lambda^1 y.P)Q)$ has the same type in E' as NP has in E, and $\delta_{N((\lambda^1 y.P)Q)}$ in $USUP(E')$ is equal to δ_{NP} in $USUP(E)$. Hence, the reduction $[\delta_P \to \delta_{N((\lambda^1 y.P)Q}/\delta_N]$ is performed in E, though using different names, and therefore, by termination in $USUP(E)$, the instance $USUP(E')$ must terminate as well. □

Proposition 4.7 *Let E be a labelled λ-term such that the USUP redex procedure terminates on input $USUP(E)$. Suppose $E' \to_{\theta_1} E$. Then the USUP redex procedure terminates on input $USUP(E')$.*

Proof As in Proposition 4.5, we can ignore the surrounding context, and simply assume that $E = (\lambda^1 y.NQ)P$, and $E' = ((\lambda^1 y.N)P)Q$. Let $USUP_E(N)$ and $USUP_{E'}(N)$ denote, respectively, the USUP translation of N in the context of E and E', and similarly for P and Q. Since N and P have the same scope in both E and E', we have $USUP_E(N) = USUP_{E'}(N)$ and $USUP_E(P) = USUP_{E'}(P)$. As for Q, it lies within y's scope in E, but not in E'. Since E' is the source term, however, by our unique naming assumption, Q can contain no occurrences of y; hence $USUP_E(Q)$ contains no occurrences of an inequality of the form $\beta_y \leq \delta_{y_i}$ for some occurrence y_i of y. Thus, $USUP_{E'}(Q) = USUP_E(Q)$ as well. By termination for $USUP(E)$, the concatenated instance $USUP_E(N)\,USUP_E(P)\,USUP_E(Q)$, which is common to both $USUP(E)$ and $USUP(E')$, cannot give rise to an infinite reduction sequence. The following inequalities are unique to $USUP(E)$:

$$\begin{aligned} \delta_{\lambda^1 y.NQ} &= \delta_P \to \delta_{(\lambda^1 y.NQ)P} \\ \delta_{\lambda^1 y.NQ} &= \beta_y \to \delta_{NQ} \\ \delta_N &= \delta_Q \to \delta_{NQ} \ . \end{aligned}$$

and the following inequalities are unique to $USUP(E')$:

$$\begin{aligned} \delta_{(\lambda^1 y.N)P} &= \delta_Q \to \delta_{((\lambda^1 y.N)P)Q} \\ \delta_{\lambda^1 y.N} &= \delta_P \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \beta_y \to \delta_N \ . \end{aligned}$$

Any reduction in $USUP(E')$ that gives rise to an infinite reduction sequence must occur as a result of reducing at least one of these three inequalities. As we argued in the proof of Proposition 4.5, because the variables $\delta_{\lambda^1 y.N}$ and $\delta_{(\lambda^1 y.N)P}$ have no occurrences other than above, we can replace them and thereby simplify the remaining inequalities:

$$\beta_y \to \delta_N \quad = \quad \delta_P \to (\delta_Q \to \delta_{((\lambda^1 y.N)P)Q}) \ .$$

If there is to be a non-terminating reduction sequence in $USUP(E')$, therefore, it must be a reduction of this inequality—in particular, the replacement $[\delta_P/\beta_y, \delta_Q \to \delta_{((\lambda^1 y.N)P)Q}/\delta_N]$—that creates it. The replacement $[\delta_P/\beta_y]$ arises from reducing the first two inequalities particular to $USUP(E)$, presented above. Hence, by termination for $USUP(E)$, this replacement cannot, on its own, lead to non-termination. Thus, non-termination can only come from the replacement $[\delta_Q \to \delta_{((\lambda^1 y.N)P)Q}/\delta_N]$. Within $USUP(E)$, there is the similar replacement $[\delta_Q \to \delta_{NQ}/\delta_N]$,

arising from reduction of the third inequality. From the inequalities for $USUP(E')$, we can derive $\delta_N = \delta_{(\lambda^1 y.N)P}$. Then by the same reasoning as in the proof of Proposition 4.6, the reduced value of $\delta_{((\lambda^1 y.N)P)Q}$ in $USUP(E')$ is equal to that of δ_{NQ} in $USUP(E)$. Therefore, a replacement equivalent to $[\delta_Q \rightarrow \delta_{((\lambda^1 y.N)P)Q}/\delta_N]$ is performed in $USUP(E)$. Hence, by the assumed termination for $USUP(E)$, we conclude that the redex procedure must terminate on input $USUP(E')$. □

Proposition 4.8 *Let E be a labelled λ-term such that the USUP redex procedure terminates on input $USUP(E)$. Suppose $E' \rightarrow_{\theta_2} E$. Then the USUP redex procedure terminates on input $USUP(E')$.*

Proof As in Proposition 4.5, we can ignore the surrounding context, and simply assume that $E = (\lambda^1 v.\lambda^3 z.N')(\lambda^3 w.P')$, and $E' = \lambda^3 z.(\lambda^1 y.N)P$, where $N' = N[vz/y]$ and $P' = P[w/z]$. Let $USUP_E(N')$ and $USUP_{E'}(N)$ denote, respectively, the USUP translation of N' in the context of E and N in the context of E', and similarly for P and P'. In the context of E', P lies within the scope of the monomorphic variable z; hence the variable β_z is unknown throughout $USUP_{E'}(P)$. In the context of E, P lies within the scope of the monomorphic variable w, so that the variable β_w is unknown throughout $USUP_E(P)$. Thus, since $P' = P[w/z]$, we have that $USUP_E(P')$ and $USUP_{E'}(P)$ are the same, except that every occurrence of β_z in $USUP_{E'}(P)$ becomes β_w in $USUP_E(P')$, and similarly for each δ_{z_i} and δ_{w_i}, accounting for each respective occurrence z_i of z in E' and w_i of w in E. Thus, modulo these variable renamings, the result of running the USUP redex procedure on input $USUP_{E'}(P)$ is identical to that of running it on input $USUP_E(P')$. More importantly, termination for E implies that the USUP redex procedure will not fall into an infinite reduction on input $USUP_{E'}(P)$. Similarly, $USUP_{E'}(N)$ and $USUP_E(N')$ are the same, except that for each occurrence y_i of y in N, the corresponding inequality $\beta_y \leq \delta_{y_i}$ in $USUP_{E'}(N)$ becomes the trio of inequalities

$$\begin{aligned} \delta_{v_i} &= \delta_{z_i} \rightarrow \delta_{v_i z_i} \\ \beta_v &\leq \delta_{v_i} \\ \underline{\beta_z} &\leq \delta_{z_i} \end{aligned}$$

in $USUP_E(N')$. (Note that β_z is unknown throughout $USUP_{E'}(N)$.) Further, any remaining occurrences of δ_{y_i} in $USUP_{E'}(N)$ become $\delta_{v_i z_i}$ in $USUP_E(N)$. Thus, if any reduction in $USUP_{E'}(N)$ is to cause non-termination, it can only be the inequality $\beta_y \leq \delta_{y_i}$, as this is the only inequality particular to $USUP_{E'}(N)$. However, this inequality, as it currently exists, contains no redexes.

Further, as N lies entirely within y's scope, not reduction in $USUP_{E'}(N)$ will cause a replacement of β_y; hence a redex will not be induced in $\beta_y \leq \delta_{y_i}$ as a result of reduction in $USUP_{E'}(N)$. Hence, the USUP redex procedure terminates on $USUP_{E'}(N)$.

Now, in addition, the following inequalities are particular to $USUP(E)$:

$$\begin{aligned} \delta_{\lambda^1 v.\lambda^3 z.N'} &= \delta_{\lambda^3 w.P} \to \delta_E \\ \delta_{\lambda^1 v.\lambda^3 z.N'} &= \beta_v \to \delta_{\lambda^3 z.N'} \\ \delta_{\lambda^3 z.N'} &= \beta_z \to \delta_{N'} \\ \delta_{\lambda^3 w.P'} &= \beta_w \to \delta_{P'} \ , \end{aligned}$$

and the following inequalities are particular to $USUP(E')$:

$$\begin{aligned} \delta_{E'} &= \beta_z \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \delta_P \to \delta_{(\lambda^1 y.N)P} \\ \delta_{\lambda^1 y.N} &= \beta_y \to \delta_N \ . \end{aligned}$$

Thus, if non-termination is to arise, it must come as a result of reducing one or more of these three latter inequalities. The variable $\delta_{\lambda^1 y.N}$ has no occurrences outside these three inequalities; hence reducing it cannot cause non-termination without further reduction. We therefore replace it in the above inequalities to obtain the simpler system

$$\begin{aligned} \delta_{E'} &= \beta_z \to \delta_{(\lambda^1 y.N)P} \\ \beta_y \to \delta_N &= \delta_P \to \delta_{(\lambda^1 y.N)P} \ . \end{aligned}$$

The variable $\delta_{E'}$ has no other occurrences in $USUP(E')$; hence, replacing it via the reduction in the first inequality has no effect on the rest of the instance, and in particular, no effect on termination. Thus, if non-termination is to result, it must come as a result of the replacement that arises from reducing the inequality $\beta_y \to \delta_N = \delta_P \to \delta_{(\lambda^1 y.N)P}$—namely, $[\delta_P/\beta_y, \delta_{(\lambda^1 y.N)P}/\delta_N]$. The effect of the replacement $[\delta_P/\beta_y]$ is to replace every inequality $\beta_y \leq \delta_{y_i}$ in $USUP(E')$ with $\delta_P \leq \delta_{y_i}$; more precisely, if σ denotes the replacements performed so far, then the inequalities become $\delta_P\sigma \leq \delta_{y_i}\sigma$. The effect of this replacement is the possible creation of additional redex-I's in these inequalities. Within $USUP(E)$, reducing the first two inequalities presented above gives

$\beta_v = \delta_{\lambda^3 w.P}$. Performing this replacement has an effect on the inequalities

$$\begin{aligned} \delta_{v_i} &= \delta_{z_i} \to \delta_{v_i z_i} \\ \beta_v &\leq \delta_{v_i} \\ \underline{\beta_z} &\leq \delta_{z_i} \end{aligned}$$

for each occurrence v_i of v. Specifically, if σ denotes the replacements performed so far, then each inequality $\beta_v \leq \delta_{v_i}$ becomes $\delta_{\lambda^3 w.P'}\sigma \leq \delta_{v_i}\sigma$. Then, since $\delta_{\lambda^3 w.P'}$ reduces to $\beta_w \to \delta_{P'}$, we obtain $\beta_w\sigma \to \delta_{P'}\sigma \leq \delta_{v_i}\sigma$. Then by the first inequality ($\delta_{v_i} = \delta_{z_i} \to \delta_{v_i z_i}$), we get $\beta_w\sigma \to \delta_{P'}\sigma \leq \delta_{z_i}\sigma \to \delta_{v_i z_i}\sigma$. Since β_w replaces β_y in $USUP(E)$, and $\delta_{v_i z_i}$ replaces δ_{y_i} in $USUP(E)$, the replacement of what stands for β_y with what stands for δ_P (i.e., $\delta_{P'}$) is reflected in $USUP(E)$. Note, however, that $\underline{\beta}_z$ is an unknown in $USUP(E')$; hence, any occurrence of δ_{z_i} in $USUP(E')$, representing the corresponding occurrence z_i of z in E', is also unknown. Hence, after the replacement $[\delta_P/\beta_y]$, the subexpressions in each $\delta_{y_i}\sigma$ corresponding to unknowns in $\delta_P\sigma$ themselves become unknown. Within $USUP(E)$, the occurrences of δ_{z_i} in $USUP(E')$ correspond to occurrences of δ_{w_i}, which are also unknown. After the replacement $[\delta_{\lambda^3 w.P'}/\beta_v]$, we eventually obtain $\beta_w\sigma \to \delta_{P'}\sigma \leq \underline{\beta_z}\sigma \to \delta_{v_i z_i}\sigma$. However, the occurrences of $\delta_{z_i}\sigma$ in $\delta_{P'}\sigma$, which were unknown within $USUP_{E'}(P)$, are no longer unknown at this higher scope. On the other hand, each occurrence of $\delta_{v_i}\sigma$ is matched with $\delta_{P'}\sigma$ and then applied to the monomorphic variable z, whose occurrences δ_{z_i} in $USUP(E)$ are all unknown. In particular, the expression

$$(\lambda^1 v.\lambda^3 z.N[vz/y])(\lambda^3 w.P[w/z])$$

gives rise to the following inequalities involving the variable β_v:

$$\begin{aligned} \beta_v &= \beta_w \to \delta_{P[w/z]} \\ \beta_v &= \underline{\beta_z} \to \delta_{N[vz/y]} \ . \end{aligned}$$

Thus, we obtain $\beta_w = \underline{\beta_z}$, so that β_w becomes an unknown. Then for each occurrence w_i of w, we obtain the inequality

$$\underline{\beta_w} \leq \delta_{w_i} \ ,$$

from which each δ_{w_i} becomes an unknown, and in particular, each δ_{w_i} is equal to $\underline{\beta_w}$. Hence, the occurrences of $\delta_{w_i}\sigma$ get matched with unknowns and become unknowns once again. Then occurrences of $\delta_{v_i z_i}$ corresponding to these unknowns in $\delta_{P'}$ are also unknown. Thus, the behaviour of the reduction $[\delta_P/\beta_y]$ in $USUP(E')$ is exactly reflected in $USUP(E)$. Hence, by termination for

$USUP(E)$, this replacement cannot result in non-termination for $USUP(E')$. The remaining possibility is the replacement $[\delta_{(\lambda^1 y.N)P}/\delta_N]$. The variable $\delta_{(\lambda^1 y.N)P}$, however, only has occurrences in the three inequalities particular to $USUP(E')$, and the variable δ_N only has a single occurrence in $USUP_{E'}(N)$. Furthermore, this single occurrence is on a right-hand side. Therefore, it can only induce a redex if if occurs in a position corresponding to an unknown on the left-hand side. Since the single inequality in which δ_N occurs within $USUP_{E'}(N)$ is an *equality*, however, the identifiers on the left-hand side are actual variables, and therefore, no redex in $USUP_{E'}(N)$ is induced by the replacement of δ_N by $\delta_{(\lambda^1 y.N)P}$, or vice versa. Thus, this replacement cannot bring about non-termination either, and we conclude, from termination in $USUP(E)$, that termination in $USUP(E')$ is guaranteed as well. □

What we have shown, by Propositions 4.5, 4.6, 4.7, and 4.8, is that if we start with any expression E for which $USUP(E)$ is guaranteed to make the USUP redex procedure terminate, then any step of θ-expansion will yield an expression E', for which $USUP(E')$ also makes the USUP redex procedure terminates. By iterating these results over several steps of θ-expansion, we have the following result:

Theorem 4.9 (Termination) *For any labelled λ-term E, the USUP redex procedure will terminate on input $USUP(E)$.*

Proof Every labelled λ-term E is a finite number of θ-reductions from a θ-normal form E_θ. By Proposition 4.4, the USUP redex procedure terminates on input $USUP(E_\theta)$. Then by Propositions 4.5, 4.6, 4.7, and 4.8, each θ-expansion of E_θ yields a USUP instance for which the redex procedure terminates. Further θ-expansions then yield further terminating USUP instances. Eventually, θ-expansion produces the original term E, at which point we conclude that, indeed, the USUP redex procedure terminates on input $USUP(E)$. □

Theorem 4.9 is the main result we sought in this section—that, like the ASUP and R-ASUP translation procedures we discussed previously, the USUP translation procedure also produces problem instances (now USUP problem instances) for which the redex procedure (now the USUP redex procedure) is guaranteed to terminate.

4.6 Chapter Summary

Motivated by the desire to formulate a truly syntax-directed translation from rank 2-typability in System F to a SUP-like problem, we found in this chapter that SUP on its own is inadequate for expressing the difference in behaviour between variables with monomorphic binding occurrences and those with polymorphic binding occurrences. Rather, the original SUP-based translation relies heavily on an expression being translated to θ-normal form before the translation begins. Once a term is in this form, the differences in behaviour between these two classes of variables do not manifest themselves and the SUP translation suffices.

If we truly want a syntax-directed translation, however we need a SUP-like problem to act as the target of the translation, in which there is some accommodation for monomorphic type variables. In this chapter, we presented USUP, which is an extension of SUP to include a new class of type variable, which we called *unknown*, whose purpose is precisely to capture the behaviour of monomorphic variables. We presented a solution semi-procedure for USUP, analogous to the redex procedure for SUP, and showed it to be sound and complete (when it terminates) with respect to the definition of a USUP solution. We showed termination for all solvable instances, and for the USUP analogue of R-acyclicity, which we called R-AUSUP.

By making USUP the target of a new translation procedure from typability, we obtain a procedure that is truly syntax-directed, and affords us several other simplifications over the original procedure as well. The new translation procedure is sound and complete with respect to the type rules for the rank 2 fragment of System F, and always produces terminating USUP instances when applied to λ-terms that have been labelled according to the rank-2 labelling procedure.

Among the goals of returning useful types, facilitating meaningful error messages, supporting link-time specialization, and being easy to state and understand, which we set forth for this thesis in Chapter 1, our introduction of USUP in this chapter fulfills the second and fourth of these. By virtue of being syntax-directed, the USUP translation procedure provides simply one rule for every element of abstract syntax for labelled λ-terms. In fact, because we are now able to unify the treatment of λ^1- and λ^2-abstractions, we can further reduce the number of needed translation rules. The translation procedure now has an elegant, highly compact presentation, as presented in Figure 4.1. Compared to the original, the new translation is certainly easier to understand, and to state.

Furthermore, by virtue of being syntax-directed, there is a reverse mapping between individual

inequalities in a USUP instance $USUP(E)$ and the elements of syntax in E that generated them. Thus, if an error is found in some inequality (say, an occurs-check violation) that renders the instance unsolvable, then the error can be traced directly back to an element of syntax in the original expression and then reported to the user. A similar error-reporting mechanism for the original SUP translation, while not technically impossible, would require a significant amount of bookkeeping, and be far from straightforward.

The two goals that remain—more expressive types, and link-time specialization—will be the subject of the next chapter, when we will build on our USUP translation, to introduce a new class of identifier that generalizes over the possible types with which a polymorphic abstraction might be annotated. Put another way, we abstract user annotations away from the USUP instance itself and thereby create templates for USUP instances, parameterized by the type annotations supplied by the user for polymorphic abstractions.

Chapter 5

Third-Order Types

Chapters 3 and 4 address what might be called "internal" deficiencies of the KW inference algorithm. By moving from ASUP to R-ASUP, and eventually to R-AUSUP, we developed a new translation procedure from rank 2-typability to a SUP-like problem, whose advantages over the original translation procedure include brevity, economy of variables and inequalities, and syntax-directedness. Consequently, further development of the algorithm to provide error-reporting and piecewise static analysis become easier under the new translation.

In this chapter, we will concern ourselves with what we might call the "external" deficiencies of the KW algorithm, which we have outlined previously. In particular, our focus in this chapter is to address the algorithm's current tendency to output degenerate types in the absence of user assistance, by introducing a richer language in which to express rank 2 types. We then demonstrate how our system supports separate compilation and link-time specialization of the types we output to match link-time-supplied arguments.

5.1 Motivating Example

Consider the following Haskell program, which is presented in Peyton Jones and Shields [47]:

```
foo :: ([Bool], [Char])
foo = let
       f x = (x [True, False], x ['a', 'b'])
      in
       f reverse
```

The function `f` cannot be typed in Haskell, because Haskell's rank 1 type system forbids `f` from requiring its parameter, `x`, to be polymorphic—but if `x` is not required to be polymorphic, then it cannot possibly be applied to both a list of boolean values and a list of characters.

In this particular case, the code may be rearranged so that it becomes typable:

```
foo :: ([Bool], [Char])
foo = let
       reverse = foldl (\x -> \y -> y:x) []
      in
       (reverse [True, False], reverse ['a', 'b'])
```

This rearrangement takes advantage of the polymorphic nature of `reverse` in typing the body of what was previously called `f`. By performing this rearrangement, we have gained typability at a severe cost—the parameter `x` is for all time bound to `reverse`, and we have lost the ability to abstract the computation as a separate function `f`. Hence we have lost the opportunity to create libraries of independently type-checked abstractions.

The solution presented by Peyton Jones and Shields is to allow the programmer to annotate the function `f` with a type that indicates that the parameter `x` should be polymorphic:

```
foo :: ([Bool], [Char])
foo = let
       f :: (forall a. [a] -> [a]) -> ([Bool], [Char])
       f x = (x [True, False], x ['a', 'b'])
      in
       f reverse
```

The algorithms they describe, which are implemented as language extensions in the Glasgow Haskell Compiler [12], then allow the type checking to proceed; further, the function `f` can be abstracted as a separately-compilable function, and applied to the function `reverse` or any other function of type `[a] -> [a]`.

Expressed in a notation similar to that of the rank 2 fragment of System F, the program becomes

$$(\lambda^p x.(x[\mathit{True}, \mathit{False}], x[\text{'a'}, \text{'b'}]))\ \mathit{reverse}\ ,$$

where the function *reverse* has the rank 1 type $\forall\alpha.[\alpha] \to [\alpha]$, and either has some supplied implementation or is treated as a free variable. In this case, even without the supplied annotation,

the rank 2 inference algorithm would produce the correct type, namely $([Bool], [Char])$, for the program, because the (U)SUP solution procedure uses the type information for the function *reverse* to provide a type for the parameter x, and then proceeds to complete the type derivation.

On the other hand, if we remove the argument *reverse* from the program and try to type the remaining program (i.e., the part that corresponds to the original `f` in isolation),

$$\lambda^p x.(x[True, False], x[\text{'a'}, \text{'b'}]) \ ,$$

we obtain the type $\bot \to \bot \times \bot$[1], i.e., $\forall\beta.(\forall\alpha.\alpha) \to \beta \times \beta$, which is of no practical use. This behaviour arises because, in the absence of the argument *reverse*, there is no type information available for the parameter x, and therefore no inequality in the (U)SUP instance that assigns a value to the variable β_x. Hence it remains as β_x and ultimately forms the parameter type $\forall.\beta_x$, i.e., $\bot$. Thus, in the absence of an argument to supply type information for the parameter of the abstraction, we are unable, under the current system, to produce a type that conveys any meaningful information. Further, even if the argument *reverse* were supplied later, after the type $\bot \to \bot \times \bot$ has been computed, there would be no way, based solely on this type, to recover sufficient information to determine whether the function application is well-typed, and if so, what the result type might be. Thus, the current inference procedure does not support separate compilation (i.e., type checking) of a function from its argument, at least not without a user-supplied annotation.

5.1.1 On Programmer Annotations

Programmer annotations are of great benefit for the purpose of documenting source code. Indeed, even within our own work, certain kinds of annotations could help to guide the inference procedure to a more specialized solution set, if necessary. However, programmer annotations supplied in advance of the type inference procedure may mask hidden opportunities for a greater amount of polymorphism in the resulting type, and therefore increased opportunities for code reuse. Consider, for example, the following Haskell code fragment:

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

[1] We assume that the infix binary functor $\times$, denoting cartesian product, binds more tightly than $\to$.

Though it is common in Haskell for programmers to annotate their functions with types, in this case, the programmer mistakenly did not choose the most general type possible for the function `sumList`. Had the programmer left the program unannotated and queried the Haskell interpreter for the type of `sumList`, he would have obtained the type

```
sumList :: (Num a) => [a] -> a
```

which indicates that the function `sumList`, with a more general annotation, could have been applied to a list of *any* numeric type, rather than simply lists of integers (perhaps the programmer came from an ML background and assumed that, as with ML, the `+` operator by default only applies to the type `Int`). Thus, the programmer-supplied annotation limits the domain of applicability of the function `sumList`, thereby eliminating potential opportunities for code reuse.

A reasonable use of programmer annotations, then, might be to attempt to perform fully-automated type inference on a given source program fragment first, and then annotate the source program with the resulting type. In this way, the type with which the programmer annotates the program is as general as the type inference engine is able to compute, and at the same time, human readers benefit from the documentation value provided by the annotations. On the other hand, if, due to technical limitations, a fully automated type inference process is unable to produce a satisfactory or sufficiently precise type on its own, a programmer might incrementally add annotation to his source program until the inference engine outputs a suitable type. Tools that implement this kind of on-demand inference could certainly be integrated into existing development environments.

As we have seen, the programmer annotation for the argument `x` in our original Haskell example arose from the second need mentioned above, namely the need to help the system towards deriving a type that it would not otherwise have been able to derive. As a side effect, we also reap the benefits of increased documentation provided by the annotation. However, as we observed in our discussion of the function `sumList`, this annotation also limits potential opportunities for code reuse. Consider the following three functions, any one of which could be a reasonable instantiation for the parameter `x`:

- *reverse*, which returns the reversal a list, and has type $\forall\alpha.[\alpha] \rightarrow [\alpha]$;
- *head*, which returns the first element of a list, and has type $\forall\alpha.[\alpha] \rightarrow \alpha$;
- *length*, which returns the length of a list, and has type $\forall\alpha.[\alpha] \rightarrow \mathit{Int}$.

The resulting types of the function `f` are summarized below:

Argument	Type of `f`
reverse	$(\forall\alpha.[\alpha] \to [\alpha]) \to ([Bool], [Char])$
head	$(\forall\alpha.[\alpha] \to \alpha) \to (Bool, Char)$
length	$(\forall\alpha.[\alpha] \to Int) \to (Int, Int)$

None of these types is more general than any other; furthermore, there is no type, at rank 2 or any higher rank, that both captures all of these types and is a valid type for `f`. Thus, any annotation the programmer chooses in order to help the type inference process along will necessarily limit the domain of applicability of the function `f`.

What we observe in this example is simply an instance of a more general phenomenon, in which the types a term possesses appear, in a sense, to be "incompatible" with one another. Further, there is no single type among those derivable for a given term that functions as a "most general" type, from which all others may be derived (this is in contrast with languages like ML that do have this property). This phenomenon, namely the lack of principal types, is a property of System F in general, and also of the rank 2 fragment, as observed by Kfoury and Wells [27]. It is perhaps for this reason that, in the absence of annotations or information about the type of its arguments, a polymorphic abstraction is typed under the assumption that its parameters have type $\bot$. This type seems as reasonable a type as any to choose as a default, and typing with $\bot$ as parameter type has the property that if a term is typable at all, then it is typable under this assumption.

It is our goal in this chapter to develop a notation for types, and associated inference algorithm, that is able to capture many, if not all, of these seemingly incompatible types for a given term, so that a polymorphic abstraction may be typed in isolation from its argument, and still be given a meaningful type. We introduce our notation in the next section.

5.2 Third-Order Notation

Rather than continue our development using our motivating example, which makes use of primitives not present in the core System F, we will use as our running example the smallest possible strictly rank 2-typable term—the self-application combinator,

$$SA := \lambda x.xx \ .$$

The term SA is not typable unless the parameter x is polymorphic; for if x had the monomorphic type τ_1, then it would also have to have the type $\tau_1 \to \tau_2$ for some τ_2, in order for the application xx to be well-typed. But only one of these can be the type of x, and the two expressions cannot be equal, as the latter is textually strictly longer than the former. Thus x must be polymorphic.

The USUP instance for SA (which, after labelling, becomes $\lambda^p x.xx$) is as follows:

$$\begin{aligned} \delta_{SA} &= \beta_x \to \delta_{xx} \\ \delta_{x_1} &= \delta_{x_2} \to \delta_{xx} \\ \beta_x &\leq \delta_{x_1} \\ \beta_x &\leq \delta_{x_2} \ . \end{aligned}$$

Solving the inequalities yields the following reduced instance:

$$\begin{aligned} \delta_{SA} &= \beta_x \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \ . \end{aligned}$$

The last two inequalities are solved, and the final result is the image of δ_{SA}, i.e., $\beta_x \to \delta_{xx}$. As the β_x indicates the type of the parameter, which is polymorphic, we quantify β_x inside the $\to$-functor, and obtain the following type:

$$\forall \delta_{xx}.(\forall \beta_x.\beta_x) \to \delta_{xx} \ .$$

Using canonical variable naming, we have

$$\forall \beta.(\forall \alpha.\alpha) \to \beta \ ,$$

i.e., $\bot \to \bot$. We see, then, as we noted previously, that by default we get a type that, while valid, is of no practical use, since no argument has type $\bot$[2]. On the other hand, there are many polymorphic functions that could legitimately be passed as an argument to SA, if it were given a type that could accommodate them.

The simplest such function is the identity function, $\lambda x.x$, of type $\forall \alpha.\alpha \to \alpha$. Adding the equality $\beta_x = \alpha \to \alpha$ to the instance above yields

$$\delta_{SA} = \beta_x \to \delta_{xx}$$

[2]Non-terminating expressions could justifiably be given the type $\bot$, but aside from the fact that there are no non-terminating expressions in core System F, we view such cases as degenerate and uninteresting.

$$\begin{aligned} \beta_x &\leq \delta_{x_2} \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \\ \beta_x &= \alpha \to \alpha \ , \end{aligned}$$

which, after reduction, becomes

$$\begin{aligned} \delta_{SA} &= (\alpha \to \alpha) \to \delta_{xx} \\ \alpha \to \alpha &\leq \delta_{x_2} \to \delta_{xx} \\ \alpha \to \alpha &\leq \delta_{x_2} \ . \end{aligned}$$

Reducing the redex-II in the second inequality yields

$$\begin{aligned} \delta_{SA} &= (\alpha \to \alpha) \to \delta_{x_2} \\ \alpha \to \alpha &\leq \delta_{x_2} \to \delta_{x_2} \\ \alpha \to \alpha &\leq \delta_{x_2} \ . \end{aligned}$$

Finally, reducing the redex-I in the third inequality yields

$$\begin{aligned} \delta_{SA} &= (\alpha \to \alpha) \to (\gamma \to \gamma) \\ \alpha \to \alpha &\leq (\gamma \to \gamma) \to (\gamma \to \gamma) \\ \alpha \to \alpha &\leq \gamma \to \gamma \ , \end{aligned}$$

and the instance is solved. From the image of δ_{SA} under the solution of the instance, we obtain the type

$$\forall.(\forall.\alpha \to \alpha) \to (\gamma \to \gamma) \ ,$$

i.e.,

$$\forall\gamma.(\forall\alpha.\alpha \to \alpha) \to (\gamma \to \gamma) \ .$$

If, instead, we had decided to apply SA to a function of type, say, $\forall\alpha.\alpha \to (\alpha \to \alpha)$, we would have obtained the type

$$\forall\gamma.(\forall\alpha.\alpha \to (\alpha \to \alpha)) \to ((\gamma \to (\gamma \to \gamma)) \to (\gamma \to (\gamma \to \gamma))) \ .$$

Here, as with the Peyton Jones and Shields example from Haskell, neither of these types is more general than the other; moreover, no System F type is expressive enough to capture them both, and still itself be a valid type for SA.

Nevertheless, there is some regularity between the assumed type of the parameter and the result type of the function, which is at least somewhat apparent when we compare these two possible types of *SA*. In order to capture this regularity in a single notation, our approach is to borrow some notation from System F_3 [13] (the so-called "third-order λ-calculus"). Under the simply-typed λ-calculus, variables are placeholders for terms. Under System F (also known as System F_2, or the second-order λ-calculus), variables may also be placeholders for types. Under System F_3, variables may now also stand for type *constructors*. A type constructor is any function that maps types to types. The familiar arrow-functor, $\rightarrow$, used to indicate function types, is a (binary) type constructor. Other functors, like the list and product functors, are also type constructors. Constructors, however, need not be simple functors. For example, the function T, defined on types as

$$T(\tau) \ := \ \tau \rightarrow (\tau \times \tau)$$

is also a constructor.

Suppose now that, instead of adopting the type assumption $\forall\alpha.\alpha \rightarrow \alpha$ or $\forall.\alpha.\alpha \rightarrow (\alpha \rightarrow \alpha)$ for x, we abstract over the type constructor for the return type and instead use the assumption $\forall\alpha.\alpha \rightarrow T(\alpha)$, where the variable T denotes a type constructor. The (U)SUP inequality associated with this assumption is $\beta_x = \alpha \rightarrow T(\alpha)$; hence the (U)SUP instance for *SA* with this assumption becomes

$$\begin{aligned} \delta_{SA} &= \beta_x \rightarrow \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \rightarrow \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \\ \beta_x &= \alpha \rightarrow T(\alpha) \ . \end{aligned}$$

Although we have not formalized what it means to solve a (U)SUP instance that contains this new class of identifier (i.e., constructor variables), we can, as a first step, substitute for β_x, as suggested by the final inequality, obtaining

$$\begin{aligned} \delta_{SA} &= (\alpha \rightarrow T(\alpha)) \rightarrow \delta_{xx} \\ \alpha \rightarrow T(\alpha) &\leq \delta_{x_2} \rightarrow \delta_{xx} \\ \alpha \rightarrow T(\alpha) &\leq \delta_{x_2} \ . \end{aligned}$$

If we now view the occurrences of $T(\alpha)$ and δ_{xx} in corresponding locations in the second inequality

as indicating a redex-I, reduction would then produce

$$\begin{aligned} \delta_{SA} &= (\alpha \to T(\alpha)) \to T(\beta) \\ \alpha \to T(\alpha) &\leq \delta_{x_2} \to T(\beta) \\ \alpha \to T(\alpha) &\leq \delta_{x_2} \ . \end{aligned}$$

Similarly, if we view the two occurrences of α on the left-hand side of the second inequality as indicating a redex-II, we obtain the instance

$$\begin{aligned} \delta_{SA} &= (\alpha \to T(\alpha)) \to T(\beta) \\ \alpha \to T(\alpha) &\leq \beta \to T(\beta) \\ \alpha \to T(\alpha) &\leq \beta \ , \end{aligned}$$

at which point the second inequality is solved, and need not be retained. However, the final inequality, by the same reasoning, now contains a redex-I, whose reduction yields the reduced instance

$$\begin{aligned} \delta_{SA} &= (\alpha \to T(\alpha)) \to T(\gamma \to T(\gamma)) \\ \alpha \to T(\alpha) &\leq \gamma \to T(\gamma) \ , \end{aligned}$$

at which point the entire instance is solved. From the reduced instance we recover the type

$$\forall\gamma.(\forall\alpha.\alpha \to T(\alpha)) \to T(\gamma \to T(\gamma)) \ .$$

Finally, quantification over the free constructor variable T yields the final type,

$$\forall T.\forall\gamma.(\forall\alpha.\alpha \to T(\alpha)) \to T(\gamma \to T(\gamma)) \ .$$

By specializing T, we can obtain the previous type derivations for SA, corresponding to our assumptions for the parameter x. In the first case, where x is assumed to have type $\forall\alpha.\alpha \to \alpha$, we have $T(\tau) = \tau$ for all τ, i.e., T is the identity constructor. By substituting this value for T, we obtain the type

$$\forall\gamma.(\forall\alpha.\alpha \to \alpha) \to (\gamma \to \gamma)$$

for SA, which is the same type we originally derived. In the second case, where x is assumed to have type $\forall\alpha.\alpha \to (\alpha \to \alpha)$, we have $T(\tau) = \tau \to \tau$ for all τ. By substituting this value for T, we obtain the type

$$\forall\gamma.(\forall\alpha.\alpha \to (\alpha \to \alpha)) \to ((\gamma \to (\gamma \to \gamma)) \to (\gamma \to (\gamma \to \gamma)))$$

for SA, which, again, is the type we originally presented. In this way, our notation permits us to generalize over two types for which there was no satisfactory generalization within the type system of System F itself. Furthermore, once the type of the argument is known, we simply determine the appropriate value of T to match the argument, and apply it throughout the generalized type to produce the correct specialization. In this way, we can see how the types produced by this system lend themselves to separate compilation and link-time type matching, more readily than those of the original KW.

Even this system, however, is not a complete solution to the type inference problem. There are a number of sources of incompleteness inherent in our choice of notation. Although the type assumption $\forall\alpha.\alpha \to T(\alpha)$ generalizes over the two types previously under consideration, it is not difficult to imagine rank 1 types that are *not* instances of this type assumption. For example, any type for which the parameter is not simply a type variable α, or any type generalized over two or more type variables, would not be covered by this assumption. Even our assumption that x is a function at all could be incorrect (though in the case of SA, the only non-function choice for the type of x that would result in a valid typing is $\perp$, which we view as safe to reject). All of these are sources of incompleteness for our system, which we discuss in greater detail in later sections. We will illustrate ways of partially overcoming these sources of incompleteness, resulting in an algorithm that, although still not fully complete with respect to all of a term's derivable types, will at least make a "best guess" of a generalized type for a given term. If the type it produces is unsatisfactory, the programmer may override it with an annotation.

5.3 Non-Simple Parameter Types and Symbolic Semiunification

We consider in this section the introduction of type assumptions for which the parameter type is not simply a type variable, as in $\forall\alpha.\alpha \to T(\alpha)$—as we observed in the last section, the fact that the parameter component of this type is a simple variable is one of several sources of incompleteness in our system, as we have presented it so far. Perhaps the simplest way to address this particular incompleteness is just to introduce a type constructor application in the parameter type as well, and proceed to solve the semiunification instance as before. Suppose, then, that we assign a type of $\forall\alpha.T_1(\alpha) \to T_2(\alpha)$ for the parameter x in SA. The USUP instance that arises from adding this type assumption for x is as follows:

$$\begin{aligned} \delta_{SA} &= \beta_x \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \to \delta_{xx} \end{aligned}$$

$$\begin{aligned} \beta_x &\leq \delta_{x_2} \\ \beta_x &= T_1(\alpha) \rightarrow T_2(\alpha) \ . \end{aligned}$$

Solving the last inequality yields the instance

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha) \rightarrow T_2(\alpha)) \rightarrow \delta_{xx} \\ T_1(\alpha) \rightarrow T_2(\alpha) &\leq \delta_{x_2} \rightarrow \delta_{xx} \\ T_1(\alpha) \rightarrow T_2(\alpha) &\leq \delta_{x_2} \ . \end{aligned}$$

There are two redex-I's in the second inequality, which, after reduction, leave a redex-II in the second inequality. Upon reducing all three of these, we obtain the reduced instance

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha) \rightarrow T_2(\alpha)) \rightarrow T_2(\beta) \\ T_1(\alpha) \rightarrow T_2(\alpha) &\leq T_1(\beta) \rightarrow T_2(\beta) \\ T_1(\alpha) \rightarrow T_2(\alpha) &\leq T_1(\beta) \ . \end{aligned}$$

At this point, the second inequality is now solved, and can therefore be removed from consideration:

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha) \rightarrow T_2(\alpha)) \rightarrow T_2(\beta) \\ T_1(\alpha) \rightarrow T_2(\alpha) &\leq T_1(\beta) \ . \end{aligned}$$

We are left, however, with the question of what to do with the final inequality, $T_1(\alpha) \rightarrow T_2(\alpha) \leq T_1(\beta)$. This inequality is clearly not solved, as the left-hand side is larger than the right-hand side, no matter what T_1 and T_2 happen to be. Thus, if we ignore the reductions demanded by this inequality, our inference procedure will be unsound. On the other hand, without knowing the identity of at least T_1, we do not know what reduction or reductions are necessary in order to solve this inequality.

To express the reductions encoded in this inequality, we introduce what we call *symbolic semiunifiers*:

Definition 5.1 (Symbolic Semiunifier) *Given type expressions τ and μ, possibly incorporating type constructor variables, the* symbolic semiunifier *(SSU) of τ and μ, written $SU(\tau, \mu)$ is defined as a most general substitution σ that solves the inequality $\tau \leq \mu$.*

Hence, for any τ and μ, $SU(\tau, \mu)$ has the property that there exists a substitution $\sigma_{(\tau,\mu)}$ such that $\tau SU(\tau, \mu)\sigma_{(\tau,\mu)} = \mu SU(\tau, \mu)$. Further, for any other substitution σ' that solves the inequality $\tau \leq \mu$, there is a substitution σ'' such that $\sigma'|_{\mathrm{Vars}(\tau \leq \mu)} = (\sigma'' \circ SU(\tau, \mu))|_{\mathrm{Vars}(\tau \leq \mu)}$.

Having introduced symbolic semiunification, we can now complete the type inference process for SA by applying an SSU to the USUP instance, thereby producing the reduced instance

$$\begin{aligned} \delta_{SA} S &= ((T_1(\alpha) \to T_2(\alpha)) \to T_2(\beta))S \\ (T_1(\alpha) \to T_2(\alpha))S &\leq T_1(\beta)S \ , \end{aligned}$$

where $S = SU(T_1(\alpha) \to T_2(\alpha), T_1(\beta))$. The application of the SSU solves the second inequality, leaving us with simply

$$\delta_{SA} S = ((T_1(\alpha) \to T_2(\alpha)) \to T_2(\beta))S \ .$$

Since the SSU S is defined only in the context of the inequality $T_1(\alpha) \to T_2(\alpha) \leq T_1(\beta)$, its domain is a subset of the variables in this inequality. More precisely, its domain is a subset of the variables on the right-hand side of the inequality, and the unknowns on either side of the inequality. In this case, the only such variable is β, and since we know that S cannot be simply an identity substitution, it follows that $\mathrm{dom}(S) = \{\beta\}$. In particular, $\mathrm{dom}(S)$ does not contain α or δ_{SA}.

Also note that since substitutions behave homomorphically across functor applications, we can push the SSU S inside the $\to$-applications on the right-hand side and obtain

$$\delta_{SA} S = (T_1(\alpha)S \to T_2(\alpha)S) \to T_2(\beta)S \ .$$

In fact, we can push the SSU S inside the constructor variable applications as well, and obtain

$$\delta_{SA} S = (T_1(\alpha S) \to T_2(\alpha S)) \to T_2(\beta S) \ .$$

Now, using the fact that $\mathrm{dom}(S)$ does not contain the variables α and δ_{SA}, we can simplify this inequality to

$$\delta_{SA} = (T_1(\alpha) \to T_2(\alpha)) \to T_2(\beta S) \ .$$

Now that the inequality is fully simplified, we can recover the type of SA:

$$\forall T_1.\forall T_2.\forall.(\forall \alpha.T_1(\alpha) \to T_2(\alpha)) \to T_2(\beta S) \ ,$$

where $S = SU(T_1(\alpha) \to T_2(\alpha), T_1(\beta))$. Notice that the quantification "$\forall\gamma$." that we previously employed in the type of SA has been replaced with simply a "$\forall$." quantification. The reason for the change is that the variables that are subject to the quantification are those in $\mathrm{Vars}(\beta S)$, and at the moment we do not know their identities.

Recalling one of our previous examples of arguments to SA, if we take $T_1(\tau) = \tau$ and $T_2(\tau) = \tau \to \tau$, for all τ, the type we have obtained for SA becomes

$$\forall.(\forall\alpha.\alpha \to (\alpha \to \alpha)) \to \beta S \to \beta S \ ,$$

where $S = SU(\alpha \to (\alpha \to \alpha), \beta)$. We are now able to solve the SSU and obtain $S = [\gamma \to (\gamma \to \gamma)/\beta]$. Substitution into the type above yields

$$\forall\gamma.(\forall\alpha.\alpha \to (\alpha \to \alpha)) \to (\gamma \to (\gamma \to \gamma)) \to (\gamma \to (\gamma \to \gamma)),$$

as before.

5.4 The Choice of Arity

By choosing $\forall\alpha.T_1(\alpha) \to T_2(\alpha)$ as the type for the parameter x in SA, we have implicitly made the assumption that x is a *unary* function, i.e., that x takes a single argument. One might wonder what might be gained by choosing a higher-arity—or, indeed, even a lower-arity—type assumption for the parameter x.

Suppose, therefore, that we choose a binary function type for x in SA:

$$\forall\alpha.T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha)) \ .$$

Then the (U)SUP instance for SA becomes

$$\begin{aligned} \delta_{SA} &= \beta_x \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \\ \beta_x &= T_1(\alpha) \to (T_2(\alpha) \to T_{22}(\alpha)) \ . \end{aligned}$$

Replacing β_x results in the instance

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha))) \to \delta_{xx} \\ T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha)) &\leq \delta_{x_2} \to \delta_{xx} \\ T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha)) &\leq \delta_{x_2} \ . \end{aligned}$$

Upon completely solving the second inequality, we obtain

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha))) \to (T_{21}(\beta) \to T_{22}(\beta)) \\ T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha)) &\leq T_1(\beta) \to (T_{21}(\beta) \to T_{22}(\beta)) \\ T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha)) &\leq T_1(\beta) \ . \end{aligned}$$

We then introduce an SSU to solve the final inequality, and we obtain the type

$$\forall T_1.\forall T_{21}.\forall T_{22}.\forall.(\forall\alpha.T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha))) \to (T_{21}(\beta S) \to T_{22}(\beta S)) \ ,$$

where $S = SU(T_1(\alpha) \to (T_{21}(\alpha) \to T_{22}(\alpha)), T_1(\beta))$. The same type can, however be obtained from the original by specializing T_2 to $T_{21} \to T_{22}$ (i.e., assigning, for each τ, $T_2(\tau) = T_{21}(\tau) \to T_{22}(\tau)$). Thus, we gain nothing by assuming a higher-arity type for x in this case. On the other hand, if we choose a zero-arity type for x, say $\forall\alpha.T_0(\alpha)$, the (U)SUP instance becomes

$$\begin{aligned} \delta_{SA} &= \beta_x \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \to \delta_{xx} \\ \beta_x &\leq \delta_{x_2} \\ \beta_x &= T_0(\alpha) \ . \end{aligned}$$

Replacing β_x produces the instance

$$\begin{aligned} \delta_{SA} &= T_0(\alpha) \to \delta_{xx} \\ T_0(\alpha) &\leq \delta_{x_2} \to \delta_{xx} \\ T_0(\alpha) &\leq \delta_{x_2} \ . \end{aligned}$$

Redex-I reduction in the third inequality yields

$$\begin{aligned} \delta_{SA} &= T_0(\alpha) \to \delta_{xx} \\ T_0(\alpha) &\leq T_0(\beta) \to \delta_{xx} \\ T_0(\alpha) &\leq T_0(\beta) \ . \end{aligned}$$

Introducing an SSU to solve the second inequality produces the final type

$$\forall T_0.\forall.(\forall\alpha.T_0(\alpha)) \to \delta_{xx}S \ ,$$

where $S = SU(T_0(\alpha), T_0(\alpha) \to \delta_{xx})$. By specializing T_0 we can obtain all of the types we can obtain by specializing T_1 and T_2 in our original, unary type assumption. On the other hand, the SSU in this type encodes more redex reductions than the SSU in the type based on the unary assumption. Thus, more of the semiunification is deferred until link-time. In general, we prefer to do as much semiunification as possible at compile-time, as it allows the type returned by the algorithm to be more descriptive in terms of the structural relationship between the parameter and result types of the function in question. Further, in the case of nullary type assumptions, the only type captured by these that is not captured by a higher-arity assumption is $\perp$, which regard as degenerate. Thus, we can safely prefer the unary assumption over the nullary assumption.

On the other hand, consider a function like $\lambda x.xxx$, in which the first occurrence of x plays the role of a *binary* function. In this case, a unary type assumption for x misses redex reductions that would be available under a binary type assumption. Without any type assumptions, the reduced (U)SUP instance for $\lambda x.xxx$ is as follows:

$$\begin{aligned} \delta_{\lambda x.xxx} &= \beta_x \to \delta_{xxx} \\ \beta_x &\leq \delta_{x_2} \to \delta_{x_3} \to \delta_{xxx} \\ \beta_x &\leq \delta_{x_2} \\ \beta_x &\leq \delta_{x_3} \ . \end{aligned}$$

We can see now that a binary type assumption would match the right-hand side of the second inequality exactly, giving rise to three redex-I's, followed by redex-II's. Any lower arity assumption would result in a less precise match.

The form of the above (U)SUP instance suggests a way to decide on an appropriate arity for our type assumption for the parameter of a polymorphic abstraction. For any parameter x, we determine the arity of each right-hand side in the (U)SUP instance whose left-hand side is β_x:

$$\begin{aligned} \text{arity}(\alpha) &= 0 \text{ (where } \alpha \text{ is a variable)} \\ \text{arity}(\tau_1 \to \tau_2) &= 1 + \text{arity}(\tau_2) \ . \end{aligned}$$

We then assign a type assumption for x with arity equal to the maximum arity among right-hand sides whose left-hand side is x, as calculated above.

In summary, a lower arity assumption captures all of the types that any higher-arity assumption can express, at the expense of doing less computation at compile-time, thereby outputting less precise answers. On the other hand, there is an "upper bound" arity beyond which no further redex reductions appear. Therefore, we can achieve the descriptiveness of the higher arity

assumptions along with the the completeness of the lower arity assumptions (though, as we have suggested, there is no need to consider arities lower than 1) by returning the list of types associated with each arity assumption, so long as the potentially exponential size of the resulting output is not a problem. On the other hand, the default choice of arity for each function parameter, is itself a parameter by which the algorithm itself may be tuned, or may be overridden on a parameter-by-parameter basis by user annotations.

5.5 Solving SSU's: Single-Variable Case

For a given SSU, without knowing the identities of the type constructors involved, we cannot know precisely what redexes will need to be reduced, and therefore what the solution is. On the other hand, we may still be able to determine what the result of *reducing* by the SSU may be. Given type constructor variables T_1 and T_2, and type variables α and β, the (U)SUP inequality $T_1(\alpha) \leq T_2(\beta)$ is, in general, only solvable for certain values of T_1 and T_2. In general, the conditions involved are complex, but the single-variable case is easier to analyze. To formalize the necessary circumstances for the single-variable case, we introduce the following definition:

Definition 5.2 *Given unary type constructor variables T_1 and T_2, we write $T_1 \leq T_2$ if for all α, β, $T_2(\beta)$ is a substitution instance of $T_1(\alpha)$.*

Intuitively, T_2 is "bigger" than T_1 if the tree obtained by applying T_2 to some variable β is larger than the tree obtained by applying T_1 to some variable α, such that each occurrence of α in $T_1(\alpha)$ matches the same expression involving β in $T_2(\beta)$. The theorem establishing the conditions for the solvability of $T_1(\alpha) \leq T_2(\beta)$ now follows:

Theorem 5.1 *Let T_1 and T_2 be unary type constructors, and α and β be type variables. Then the (U)SUP inequality $T_1(\alpha) \leq T_2(\beta)$ has a solution if and only if either $T_1 \leq T_2$ or $T_2 \leq T_1$.*

Proof The "if" direction is obvious. Suppose, therefore, that the inequality $T_1(\alpha) \leq T_2(\beta)$ is solvable. Then there exist substitutions σ and σ_1 such that $T_1(\alpha)\sigma\sigma_1 = T_2(\beta)\sigma$, i.e., $T_1(\alpha\sigma\sigma_1) = T_2(\beta\sigma)$. We may assume that σ acts only on variables on the right-hand side, so that the equation reduces to $T_1(\alpha\sigma_1) = T_2(\beta\sigma)$. Now consider an occurrence of α in $T_1(\alpha)$, and the corresponding subexpression of $T_2(\beta)$, if it exists. Since σ_1 maps all occurrences of α to the same expression, it follows that every subexpression of $T_2(\beta)$ corresponding to an occurrence of α in $T_1(\alpha)$ is the same. Similarly, every subexpression of $T_1(\alpha)$ corresponding to an occurrence of β in $T_2(\beta)$ is

the same. Suppose now that there are paths Σ_1 and Σ_2 such that $\Sigma_1(T_1(\alpha)) = \Sigma_2(T_1(\alpha)) = \alpha$, $\Sigma_1(T_2(\beta))$ is an expression in β, and $\Sigma_2(T_2(\beta))$ does not exist. Since $\Sigma_2(T_2(\beta\sigma)) = \Sigma_2(T_1(\alpha\sigma_1))$, it follows that $|\beta\sigma| < |\alpha\sigma_1|$. On the other hand, $|\Sigma_1(T_1(\alpha))| = 1$ and $|\Sigma_1(T_2(\beta))| \geq 1$. Since $\Sigma_1(T_1(\alpha\sigma_1)) = \Sigma_1(T_2(\beta\sigma))$, we have $|\alpha\sigma_1| \geq |\beta\sigma|$, which is a contradiction. Therefore, all occurrences of α in $T_1(\alpha)$ correspond to expressions in $T_2(\beta)$, or none of them do. Similarly, all occurrences of β in $T_2(\beta)$ correspond to expressions in $T_1(\alpha)$, or none of them do. If all occurrences of α in $T_1(\alpha)$ correspond to expressions in $T_2(\beta)$, then since, as we observed, all such expressions are the same, there is a substitution σ_1 such that $T_1(\alpha\sigma_1) = T_2(\beta)$, i.e., $T_1 \leq T_2$. Similarly, if all occurrences of β in $T_2(\beta)$ correspond to expressions in $T_1(\alpha)$, then there is a substitution σ such that $T_1(\alpha) = T_2(\beta\sigma)$, i.e., $T_2 \leq T_1$. Since at least one of these conditions must hold, we have either $T_1 \leq T_2$ or $T_2 \leq T_1$. □

We therefore have a reasonably succinct characterization of the solvable single-variable SSU's. With this characterization in place, it is relatively straightforward to compute the value denoted by the SSU:

Theorem 5.2 *Let T_1 and T_2 be type constructors, and α and β be variables, such that the (U)SUP inequality $T_1(\alpha) \leq T_2(\beta)$ is solvable. Then the substitution that solves the inequality maps $T_2(\beta)$ to $T_3(\gamma)$, where γ is a fresh variable (or an alias for β itself, if no reduction is required), and T_3 is the larger of T_1 and T_2.*

Proof Since the inequality $T_1(\alpha) \leq T_2(\beta)$ is solvable, then by Theorem 5.1, either $T_1 \leq T_2$ or $T_2 \leq T_1$. In the former case, there is a substitution σ such that $T_1(\alpha)\sigma = T_2(\beta)$. Hence, the inequality is already solved, which implies that the SSU is simply the identity substitution. Hence, $T_2(\beta)$ is mapped to $T_2(\beta)$, which is equal to $T_3(\beta)$, since T_2 is the larger of T_1 and T_2. In the latter case, there is a substitution σ such that $T_1(\alpha) = T_2(\beta)\sigma$. Hence, the substitution $[\gamma/\alpha] \circ \sigma$ solves the inequality and maps $T_2(\beta)$ to $T_1(\gamma)$, which is equal to $T_3(\gamma)$, since T_1 is the larger of T_2 and T_3. In both cases, therefore, the solution of the SSU maps $T_2(\beta)$ as stated in the theorem. □

It is convenient to use the notation $\max(T_1, T_2)$ to denote the larger of T_1 and T_2. Then the application of $SU(T_1(\alpha), T_2(\beta))$ to the inequality $T_1(\alpha) \leq T_2(\beta)$ produces the solved inequality $T_1(\alpha) \leq max(T_1, T_2)(\gamma)$ (the renaming of β to γ in the case that the SSU is an identity is harmless).

Returning to our running example SA, the type we derived for it was

$$\forall T_1.\forall T_2.\forall.(\forall\alpha.T_1(\alpha) \to T_2(\alpha)) \to T_2(\beta S) \ ,$$

where $S = SU(T_1(\alpha) \to T_2(\alpha), T_1(\beta))$. We now know that S maps $T_1(\beta)$ to $\max(T_1 \to T_2, T_1)(\gamma)$. Assuming the SSU is solvable, we can only have $T_1 \leq T_1 \to T_2$; hence, S is the substitution that maps $T_1(\beta)$ to $T_1(\gamma) \to T_2(\gamma)$. To express the effect of S on β itself, we can employ the following notation:

Definition 5.3 (Inverse of a Type Constructor) *Let T be a type constructor. We denote by T^{-1} the function on types defined such that, for all τ, $T^{-1}(T(\tau)) = \tau$ (T^{-1} is undefined on types outside the image of T). We call T^{-1} the* inverse *of T.*

Then we can write $S = [T_1^{-1}(T_1(\gamma) \to T_2(\gamma))/\beta]$. Our type for SA then becomes

$$\forall T_1.\forall T_2.\forall\gamma.(\forall\alpha.T_1(\alpha) \to T_2(\alpha)) \to T_2(T_1^{-1}(T_1(\gamma) \to T_2(\gamma))) \ .$$

Even though the results and notation we have presented in this section have allowed us, at least in the case of SA, to compute a type that does not include SSU's, it is important to remain mindful of their influence on the final solution. In particular, even if we have been able to find a solution of an SSU $SU(T_1(\alpha), T_2(\beta))$, thereby eliminating the SSU from the type itself, we must remember that the SSU also encodes a constraint on the identities of T_1 and T_2 themselves—not every T_1 and T_2 lend themselves to an SSU that solves them. Thus, even though the SSU itself is not present in the final solution, the constraint that the SSU must exist remains as a condition on the existence of a solution.

Having made this point, however, we now point out that if we ever insert an SSU solution of the form $\max(T_1, T_2)(\gamma)$ into a type, then we have made an implicit assertion about T_1 and T_2. In particular, $\max(T_1, T_2)$ cannot exist unless either $T_1 \leq T_2$ or $T_2 \leq T_1$, and this is precisely the condition characterizing the solvability of $T_1(\alpha) \leq T_2(\beta)$, and therefore also the existence of the SSU. Hence, in this case, an explicit side condition is not needed, for if T_1 and T_2 violate the side condition, then $\max(T_1, T_2)$, and therefore the type as a whole, is undefined.

In the case of the type we presented above for SA, we have included the subexpression $T_1^{-1}(T_1(\gamma) \to T_2(\gamma))$. Such an expression is only defined if $T_1(\gamma) \to T_2(\gamma)$ is equal to $T_1(\tau)$ for some τ. Since $T_1(\tau)$ can be written as $T_1(\alpha)[\tau/\alpha]$, we have $T_1(\gamma) \to T_2(\gamma) = T_1(\alpha)[\tau/\alpha]$. Hence, the existence of the subexpression $T_1^{-1}(T_1(\gamma) \to T_2(\gamma))$ implies that $T_1 \leq T_1 \to T_2$, which implies that the SSU must exist. Again, therefore, an explicit side condition is not needed.

As we have seen in this section, SSU's involving unary type constructors lend themselves particularly well to analysis, to the point that we can often eliminate them entirely from the types we return. The corresponding analysis for higher-arity type constructors is considerably more complicated and unfortunately does not produce succinct solutions. We discuss issues surrounding higher-arity type constructors in a later section.

5.6 The Core Algorithm

In this section, we formally present our third-order inference algorithm, which we have been presenting gradually throughout the previous sections, before considering enhancements to the core algorithm in later sections.

To begin with, we must clarify what constitutes a redex, now that type constructor variables have been introduced into (U)SUP. Now, as before, the definition of a redex is expressed in terms of paths; hence our first step is to update our definition of a path:

Definition 5.4 (Path) *For a term algebra comprising a set $\mathbb{F}$ of functors, and a set $\mathbb{T}$ of functor variables, also with associated arities, a path (denoted by Σ) is a string over the set*

$$\{f_i | f \in \mathbb{F}, 1 \leq i \leq \text{arity}(f)\} \cup \{T_i | T \in \mathbb{T}, 1 \leq i \leq \text{arity}(T)\}$$

that acts as a partial function on terms as follows:

$$\begin{aligned} \epsilon(\tau) &= \tau \text{ for all } \tau \\ (\Sigma f_i)(f(\tau_1, \ldots, \tau_{\text{arity}(f)})) &= \Sigma(\tau_i)\ (1 \leq i \leq \text{arity}(f)) \\ (\Sigma T_i)(T(\tau_1, \ldots, \tau_{\text{arity}(T)})) &= \Sigma(\tau_i)\ (1 \leq i \leq \text{arity}(T)), \end{aligned}$$

where τ ranges over terms and ϵ is the empty path.

Thus, paths now include constructor variables as well. This updated definition of a path allows us to capture any subexpression of a given (unquantified) type expression, either inside or outside of a constructor variable application. With this definition in place, we now obtain definitions of the four classes of USUP redex for free.

By allowing redex reductions to examine the arguments of a type constructor variable, we are making the implicit assumption that the constructor variable actually makes use of the arguments, i.e., that the arguments actually appear somewhere in the type. If this is not the case, then

$$
\begin{array}{rcl}
\langle \mathit{Type} \rangle & ::= & \forall T.\langle \mathit{Type} \rangle \mid \forall.\rho \\
\rho & ::= & \pi \rightarrow \rho \mid \pi \\
\pi & ::= & \forall \alpha.\pi \mid \tau \\
\tau & ::= & \tau \rightarrow \tau \mid T(\tau, \ldots, \tau) \mid \tau\ S \mid \alpha \\
S & ::= & SU(\tau, \tau)
\end{array}
$$

Figure 5.1: Grammar for types produced by the third-order inference algorithm.

the algorithm may potentially make unnecessary reductions. For the algorithm as it currently exists, however, (i.e., without nullary constructors like *Int* and *Bool*, and without multi-argument constructor variables), this assumption is valid. We will reexamine this issue later, when we consider enhancements to the core system. It many cases, as we shall see, the additional reductions do not manifest themselves in the final type.

Figure 5.1 presents a grammar for the language of types our system produces. As in Section 4.5.2, we use the metavariables τ, π, and ρ, to refer, respectively, to rank 0, rank 1, and rank 2 types. The metavariables α, T, and S range, respectively, over type variables, type constructor variables, and SSU's. Note that, in order to keep the grammar simple, we have specified that SSU's be written inline. In actual discussion, and in any practical system, however, we would use placeholders for SSU's in the actual type, and then provide a lookup table for their actual values. The use of placeholders for SSU's, instead of inlining them, leads to concision in the returned type, making it easier to both read and write.

Though the grammar we present assumes only the existence of the concrete constructor "$\rightarrow$" for functions, it is straightforward to include other constructors, such as those for tuples and lists, into the language of types as well. These, however, are not essential to the core presentation and are omitted for the sake of brevity. We explicitly discuss nullary constructors, like *Int* and *Bool*, in a later section.

Figure 5.2 presents pseudocode for the third-order inference algorithm. As we have outlined in the previous sections, the algorithm consists of translating the given term E into a USUP instance Γ, and then solving Γ via the USUP redex procedure. For any polymorphically-bound variables for which type information has not been provided, either in the form of user-supplied annotations, or implicitly because the argument is present, we create a type assumption, based on

ALGORITHM TypeOf
INPUT: λ-term E, type environment Δ, $FV(E) \subseteq \mathrm{dom}(\Delta)$
OUTPUT: 3rd-order rank 2 type for E

$E_l \leftarrow E$, with all abstractions labelled m or p
$\Gamma \leftarrow USUP(E_l, \Delta)$
if Γ is not solvable then
    abort(error)
end if
$\sigma \leftarrow$ USUPRedexProcedure(Γ)
$\Gamma' \leftarrow \Gamma\sigma$
for each β_x occurring only on left-hand sides in Γ (other than in $\delta_{\lambda^p x.E} = \beta_x \rightarrow \delta_E$)
    $arity \leftarrow 0$
    for each inequality $\beta_x \leq \mu_i$ in Γ'
        $arity \leftarrow \max(arity, \mathrm{arity}(\mu_i))$
    end for
    let α_x be a fresh type variable, $T_{x,0}, \ldots, T_{x,arity}$ be fresh constructor variables
    $\Gamma' \leftarrow \Gamma' \cup \langle \beta_x = T_{x,0}(\alpha) \rightarrow \cdots \rightarrow T_{x,arity}(\alpha), \vec{\alpha} \rangle$,
        where $\vec{\alpha}$ is the set of unknowns associated with the inequality $\delta_{\lambda^p x.E} = \beta_x \rightarrow \delta_E$ in Γ
end for
$\sigma' \leftarrow$ USUPRedexProcedure(Γ')
$\Gamma'' \leftarrow \Gamma'\sigma'$
for each unsolved inequality $\tau \leq \mu$ in Γ''
    $\sigma' \leftarrow SU(\tau, \mu) \circ \sigma'$
    $\Gamma'' \leftarrow \Gamma'' SU(\tau, \mu)$ (simplify expressions if possible)
end for
final substitution is $\sigma' \circ \sigma$
return $\forall.Q(E_l, \delta_{E_l}\sigma\sigma')$, quantified over all type constructors introduced in the procedure

Figure 5.2: Third-order inference procedure.

the maximum arity exhibited for that variable throughout the instance, and add the appropriate type assumptions for these variables to the reduced Γ. We then continue reduction as long as we are able to find redexes, according to our extended notion of what constitutes a path. For each inequality that is not solved when no redexes remain, we create an SSU that expresses its solution, which we then apply throughout the instance. Once we have finished, we apply the accumulated substitution to the variable δ_E, producing an expression τ. We then add quantifiers to τ by evaluating $Q(E, \tau)$, as expressed in Section 4.5.2, and finally quantify over all free type variables and constructor variables. Any available algebraic simplifications of the type may be performed along the way (in particular, pushing SSU's inside functor applications), but are not essential to the operation of the procedure.

5.6.1 Soundness

We show in this section that the inference procedure presented in Figure 5.2 is sound, in that it only produces types for a term that are indeed derivable for that term from the type rules of the rank 2 fragment of System F, as presented in Figure 4.2. In fact, this result, as we have just stated it, is clearly false, as the rank 2 type rules for System F possess no facility for outputting types containing constructor variable applications and SSU's. Rather, what we must show is that, given a type output by our algorithm for a term E, every instantiation of the type constructor variables in our type either produces, after SSU's are evaluated, a type that is derivable for E from the type rules in Figure 4.2, or is undefined because the instantiated SSU's have no solution. The soundness theorem, then, is as follows:

Theorem 5.3 (Soundness) *Let E be a λ-term and Δ a type environment such that $FV(E) \subseteq \mathrm{dom}(\Delta)$, and let $\forall T_1 \cdots T_n.\rho$ be the type returned by the third-order inference algorithm on input E, Δ. Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be type constructors. Then the statement $\Delta \vdash E : \rho'$ is derivable from the type rules in Figure 4.2, where the type ρ' is defined by $\rho[\mathcal{T}_1/T_1, \ldots, \mathcal{T}_n/T_n]$ (with all SSU's evaluated and substituted).*

Proof Suppose that, instead of expressions equating each β_x with an expression involving constructor variables, we add expressions containing the actual constructors themselves. In other words, suppose we instantiate the constructor variables to actual constructors right away, and then do the USUP reduction. Then, by Theorem 4.7, the resulting type, if one exists, is derivable for the expression E. The key, then, is to show that if we use constructor variables instead of actual constructors, and then instantiate the constructor variables in the same way at the end, we

obtain the same type, or at least a compatible type. We begin by showing that redex reduction commutes with constructor instantiation. In particular, let Γ be a USUP instance that contains constructor variables, and let F be some function on USUP instances (with a given set of constructor variables) that instantiates the constructor variables in the given instance, and evaluates the SSU's, to yield a real System F type. Then we wish to show that for any redex reduction σ of Γ, we have $F(\Gamma\sigma) = F(\Gamma)\sigma$.

We first note that if the redex in question does not involve variables found inside constructor variable applications, then the result is trivial, as the constructor applications may then be thought of as "black boxes" whose contents are irrelevant. We may therefore assume that any redex reduction is one that requires looking inside constructor variable applications. Let $[\tau/\alpha]$, where α is either a true variable or an unknown, be a replacement performed during redex reduction. Suppose the constructor variable T is replaced by a constructor $\mathcal{T} = \lambda\tau_1, \ldots, \tau_k.(\cdots)$. Let $\Sigma_{\mathcal{T}_j}$ denote a path such that, for all $\beta_1, \ldots, \beta_k$, $\Sigma_{\mathcal{T}_j}(\mathcal{T}(\beta_1, \ldots, \beta_k)) = \beta_j$ (in other words, let $\Sigma_{\mathcal{T}_j}$ denote a path to an occurrence of the j-th argument in $\mathcal{T}$). Let $\Gamma' = \{\langle\tau'_i \leq \mu'_i, \vec{\alpha_i}\rangle\}_{i=1}^{N}$ be the result of instantiating T to $\mathcal{T}$ in Γ. Then for all $\Sigma_{\mathcal{T}_j}$ defined as above, and paths Σ and Σ', we have $(\Sigma T_j \Sigma')(\tau_i) = (\Sigma\Sigma_{\mathcal{T}_j}\Sigma')(\tau'_i)$, and similarly for μ_i and μ'_i, when these expressions exist. Let $(\Sigma T_j \Sigma')(\tau_h)$ be an occurrence of α. Then after redex reduction, $(\Sigma T_j \Sigma')(\tau_h[\tau/\alpha])$ will be a an occurrence of τ. Then, after constructor instantiation, each $(\Sigma\Sigma_{\mathcal{T}_j}\Sigma')(\tau'_h[\tau/\alpha])$ will be an occurrence of τ. On the other hand, if the instantiation happens first, then for each $\Sigma_{\mathcal{T}_j}$ (if any), $(\Sigma\Sigma_{\mathcal{T}_j}\Sigma')(\tau'_h)$ will be an occurrence of α. After the replacement $[\tau/\alpha]$, each $(\Sigma\Sigma_{\mathcal{T}_j}\Sigma')(\tau'_h)[\tau/\alpha]$ will be an occurrence of τ. Hence the replacement commutes with the constructor instantiation. Iteration over several replacements (as redex-II and redex-III reductions typically involve compound substitutions) and several instantiations gives us our desired result.

On the other hand, there are some redexes that are only present when the constructor variables are instantiated. These are the reductions that are encoded as SSU's when the constructor variables are not instantiated. An SSU, however, by definition, represents the substitution that solves a given inequality, once the constructor variables are instantiated. Even if the constructor variables had been instantiated from the beginning, we could have chosen to reserve reduction of those redexes that would have been part of the SSU's until the end, since the USUP redex procedure does not prescribe a reduction strategy. Hence, the types we obtain by doing instantiation at the end (i.e., by instantiating constructor variables in the types returned by the algorithm) are equivalent to those we would have obtained by using concrete constructors from the beginning. Therefore, each instantiation of a type output by the third-order inference algorithm, if the

instantiation is defined, produces a type derivable for the expression E. □

As a consequence of Theorem 5.3, we can be confident that the types output by the third-order inference procedure are valid types for the source term under consideration.

5.6.2 Completeness

Completeness of the third order inference procedure would imply that every type that can be derived for a term E from the rank 2 type rules for System F is captured by the output from the procedure. As we have already noted, this statement is false, as there are several sources of incompleteness within the procedure. On the other hand, the original KW inference procedure is incomplete as well—in the absence of programmer annotations, it can only output function types parameterized by $\perp$, and the set of programmer annotations available to it (i.e., rank 1 System F types) cannot capture all of the types that third-order types can. We have explored some of the sources of incompleteness in our system in the previous sections, and continue to explore sources of incompleteness, and certain ways to address them, in sections to come.

5.6.3 Termination

Termination for the third-order inference procedure is trivial, as it follows directly from our termination result for the USUP redex procedure, when applied to the USUP translation of a λ-term E, as outlined in Section 4.5.3.

5.7 Constants

To be applicable to real-world programming tasks, any type inference procedure must accommodate constant types, like *Int* and *Bool*, and our system is no different. For the purposes of USUP translation and reduction, constant types may be regarded as nullary functors. Hence, they are neither variables nor unknowns, and are therefore unrestricted in where they may occur. The theory of USUP and USUP reduction has included nullary functors from the beginning; hence no theoretical development is necessary before they may be included in the present system. At the same time as we consider the introduction of constant types in this section, we will also begin to include other type constructors, like tuple and list constructors, in our discussion as well.

One (perhaps unexpected) consequence of introducing constant types into the language, is that where they occur, they may be able to provide us with more information about the set of

type constructors to which a constructor variable may be instantiated, and perhaps even deduce their values, thereby potentially removing some SSU's from computed types, or at least reducing their complexity. Consider, for example, the following SSU:

$$SU(T_1(\alpha) \to T_2(\alpha), [Bool] \to T_2(\beta)) \ .$$

Because $T_1(\alpha)$ is matched against $[Bool]$, there are only three values of T_1 for which this SSU has a solution:

- $T_1 = \lambda\tau.\tau$;
- $T_1 = \lambda\tau.[\tau]$;
- $T_1 = \lambda\tau.[Bool]$.

In particular, $T_1(\alpha)$ must be an anti-instance of $[Bool]$. Suppose, in addition, that in the same (U)SUP instance, we have the following SSU:

$$SU(T_1(\alpha) \to T_2(\alpha), [Int] \to T_2(\beta)) \ .$$

Now, in addition to being an anti-instance of $[Bool]$, $T_1(\alpha)$ must also be an anti-instance of $[Int]$. Hence, $T_1(\alpha)$ must be a *common anti-instance* of $[Bool]$ and $[Int]$, of which there are two:

- $T_1 = \lambda\tau.\tau$;
- $T_1 = \lambda\tau.[\tau]$.

We are left with the question of which of these two potential instantiations of T_1 is the most appropriate. We favour the latter, in which T_1 is identical to the list constructor, as it most closely matches the actual uses of the parameter x modelled by these SSU's (presumably, these particular SSU's arose because the parameter x was applied exclusively to lists in the source program). Further, the identity constructor is an anti-instance of all constructors; hence choosing this constructor would imply that we should choose it in all cases, the effect of which would be to simply erase the constructor variables on parameters. Thus, we choose the *most specific common anti-instance* of the two expressions $[Bool]$ and $[Int]$, which is the list constructor. This constructor represents a "greatest lower bound" of these two expression under a subsumption preorder. We will see how the behaviour associated with the more general assumption $T_1(\tau) = \tau$ can be recaptured when we discuss our linking procedure, in a later section.

ALGORITHM mscai
INPUT: type expressions τ_1, τ_2, variable α
OUTPUT: most specific common anti-instance of τ_1 and τ_2, parameterized by α

$\text{mscai}(\tau, \tau, \alpha) \;=\; \tau$
$\text{mscai}(f_1(\tau_{11}, \ldots \tau_{1m}), f_2(\tau_{21}, \ldots, \tau_{2n}), \alpha) \;=$
  if $f_1 = f_2$ and $m = n$ then
    let
      $\tau_{31} \;=\; \text{mscai}(\tau_{11}, \tau_{21}, \alpha)$
      $\ldots$
      $\tau_{3m} \;=\; \text{mscai}(\tau_{1m}, \tau_{2m}, \alpha)$
      τ'_{1i}, τ'_{2i} $(1 \le i \le m)$ be such that $\tau_{1i} = \tau_{3i}[\tau'_{1i}/\alpha]$, $\tau_{2i} = \tau_{3i}[\tau'_{2i}/\alpha]$
    in
      if $|\{\tau'_{1i} \mid 1 \le i \le m, \alpha \in FTV(\tau_{3i})\}| \le 1$ and $|\{\tau'_{2i} \mid 1 \le i \le m, \alpha \in FTV(\tau_{3i})\}| \le 1$ then
        $f_1(\tau_{31}, \ldots, \tau_{3m})$
      else α
  else α
$\text{mscai}(\tau, \alpha, \alpha) \;=\; \text{mscai}(\tau, \alpha, \alpha) \;=\; \alpha$

Figure 5.3: Algorithm for computing the most specific univariate common anti-instance of two expressions.

"Anti-unification" algorithms for computing most specific common anti-instances first appeared in Reynolds [53] and Plotkin [49]; more recent work was done by Østvold [44]. However, these algorithms are not quite suited to our current purposes. Consider, for example, the following type expressions:

$$(Int, Bool) \qquad (Bool, Int)\ .$$

Their most specific common anti-instance is (α, β). However, this expression contains two variables, and therefore specializing a type constructor variable to one that outputs an expression like (α, β) does not fit our current restriction that type constructors be unary.

We present an algorithm for computing the most specific *univariate* common anti-instance of two expressions in Figure 5.3. The algorithm, much like Østvold's, works by finding, given two

expressions, the points in their parse trees where they diverge from one another, and replacing the trees rooted at those points with a type variable. However, if, at any point, more than a single type variable is required to produce an anti-instance, the algorithm moves up a level in the parse tree and replaces everything beneath it with a variable, including the two subtrees that would have required different type variables in order to anti-unify the expressions.

What we have been proposing, in this discussion, is to simplify (and perhaps eliminate) SSU's $SU(\tau, \mu)$ by matching constructor variables in τ against corresponding subexpressions in μ, or vice versa. Matching between terms with variables in the functor position is essentially second-order unification, which was shown by Goldfarb [14] to be undecidable. Thus, we must be conservative in what we choose to attempt to match, in order to be assured of termination. We therefore only attempt to instantiate a constructor variable T if it occurs in a corresponding subexpression to an expression containing no constructor or type variables.

Although restricting our attention, when instantiating a constructor variable T, to constant expressions assures us of termination, we cannot ignore other contexts in which T might occur, or we might give up completeness in our procedure unnecessarily. Consider the following three SSU's:

$$SU(T_1(\alpha) \to T_2(\alpha), [Bool] \to T_2(\beta))$$
$$SU(T_1(\alpha) \to T_2(\alpha), [Char] \to T_2(\gamma))$$
$$SU(T_1(\alpha) \to T_2(\alpha), (T_3(\delta) \times T_4(\delta)) \to T_2(\epsilon)) \ .$$

If, in attempting to instantiate T_1, we consider only the first two SSU's (as T_1 only occurs in corresponding positions with constant expressions in these two), then we instantiate T_1 to the most specific univariate common anti-instance of $[Bool]$ and $[Char]$, which is $\lambda\tau.[\tau]$. Upon substituting this value for T_1, the first two SSU's become solvable, and the third becomes

$$SU([\alpha] \to T_2(\alpha), (T_3(\delta) \times T_4(\delta)) \to T_2(\epsilon)) \ ,$$

which is not solvable because of a functor mismatch between $[-]$ and $\times$. However, this SSU is solvable for an *anti-instance* of our choice for T_1, namely $\lambda\tau.\tau$, the identity constructor. Then the third SSU becomes

$$SU(\alpha \to T_2(\alpha), (T_3(\delta) \times T_4(\delta)) \to T_2(\epsilon)) \ ,$$

which is now solvable. The other two SSU's, of course, remain solvable as well.

Therefore, after anti-unifying the constant expressions occurring in corresponding positions to a particular constructor variable T, we must consider the *non-constant* expressions with which

T occurs in corresponding positions, and anti-instantiate our current estimate for T until all of the functor mismatches disappear (which is certain to happen when T is instantiated to the identity constructor, and may happen sooner). The procedure is summarized in Figure 5.4. The procedure getSubst searches through the given set of SSU's for a type constructor that occurs in corresponding positions with one or more constant expressions, and returns a specialization of T to their most specific univariate common anti-instance. Note that we have employed a slight abuse of notation, by applying our previously defined function mscai to a set of expressions, rather than just a pair of expressions. The intended semantics are that if the set has cardinality one, the lone element is returned. If the set has cardinality two, then the behaviour is the normal operation of mscai. For larger cardinalities, mscai anti-unifies the expressions in pairs until a single anti-instance is reached.

After finding the most specific univariate common anti-instance of the constant expressions, the procedure then matches the result, through the function refine, against the non-constant expressions, taking anti-instances as necessary. Once this process is complete, the result is the necessary specialization for the constructor variable T. The procedure then continues, processing other constructor variables, until no further non-trivial replacements are found.

We return now to our motivating example from Section 5.1, rephrased as a λ-term, and without the supplied `reverse` argument:

$$f = \lambda^p x.(x[\mathit{True}, \mathit{False}], x[\text{`a'}, \text{`b'}]) \ .$$

The associated USUP instance (assuming, for the sake of brevity, that the subexpressions $[\mathit{True}, \mathit{False}]$ and [`a', `b'] are immediately recognized as being of type $[\mathit{Bool}]$ and $[\mathit{Char}]$, respectively) is as follows:

$$\begin{aligned}
\delta_f &= \beta_x \to \delta_{(x[\mathit{True},\mathit{False}],x[\text{`a'},\text{`b'}])} \\
\delta_{(x[\mathit{True},\mathit{False}],x[\text{`a'},\text{`b'}])} &= \delta_{x[\mathit{True},\mathit{False}]} \times \delta_{x[\text{`a'},\text{`b'}]} \\
\delta_{x_1} &= [\mathit{Bool}] \to \delta_{x[\mathit{True},\mathit{False}]} \\
\delta_{x_2} &= [\mathit{Char}] \to \delta_{x[\text{`a'},\text{`b'}]} \\
\beta_x &\leq \delta_{x_1} \\
\beta_x &\leq \delta_{x_2} \ .
\end{aligned}$$

ALGORITHM inst
INPUT: Set $\mathcal{S}$ of SSU's
OUTPUT: List of most specific instantiations on type constructors that avoid functor mismatches

inst($\mathcal{S}$) =
  let σ = getSubst($\mathcal{S}$)
  in if σ is trivial (or $\mathcal{S} = \mathcal{S}\sigma$) then nil else σ :: inst($\mathcal{S}\sigma$)

getSubst($\mathcal{S}$) =
  let
    $P = \{\langle \Pi\tau, \Pi\mu\rangle \mid (SU(\tau,\mu) \in \mathcal{S}$ or $SU(\mu,\tau) \in \mathcal{S}),\ \Pi$ is a path, $\Pi\tau$ and $\Pi\mu$ exist$\}$
    $T(\alpha) \in \{T(\alpha) \mid \exists\tau.\langle T(\alpha), \tau\rangle \in P,\ \tau$ is constant$\}$
    $\vec{\tau} = \{$constant $\tau \mid \langle T(\alpha), \tau\rangle \in P\}$
  in
    $[\lambda\beta.\text{refine}(T, \text{mscai}(\vec{\tau}, \beta), \mathcal{S}, \alpha)/T]$, where β is a fresh variable
  except when no such T, τ_1, τ_2 exist $\Rightarrow$ []

refine(T, τ, $\mathcal{S}$, α) =
  refine-rec($\tau, \{\Pi\mu \mid \exists\tau.SU(\tau,\mu) \in \mathcal{S}, \Pi\tau = T\} \cup \{\Pi\tau \mid \exists\mu.SU(\tau,\mu) \in \mathcal{S}, \Pi\mu = T\}, \alpha$)

refine-rec(α, $\mathcal{S}$, α) = α
refine-rec($f(\tau_1, \ldots, \tau_n)$, $\mathcal{S}$, α) =
  if $\exists f'(\tau'_1, \ldots, \tau'_m) \in \mathcal{S}$ such that $f' \neq f$ or $m \neq n$ then α
  else
    let
      τ'_i = refine-rec(τ_i, $\{f_i\mu \mid \mu \in \mathcal{S}\}$, α) for each $1 \leq i \leq n$
      τ''_i $(1 \leq i \leq n)$ be such that $\tau'_i = \tau_i[\tau''_i/\alpha]$
    in
      if $|\{\tau''_i \mid 1 \leq i \leq m, \alpha \in \text{FTV}(\tau'_i)\}| \leq 1$ then $f(\tau'_1, \ldots, \tau'_n)$ else α
    where $f_1, \ldots, f_n$ are the path selection operators for the functor f

Figure 5.4: Constructor variable instantiation

Simplification yields

$$\begin{aligned}\delta_f &= \beta_x \to (\delta_{x[True,False]} \times \delta_{x[\text{`a'},\text{`b'}]}) \\ \beta_x &\leq [Bool] \to \delta_{x[True,False]} \\ \beta_x &\leq [Char] \to \delta_{x[\text{`a'},\text{`b'}]} \ .\end{aligned}$$

Introducing a unary assumption for β_x yields, after replacement,

$$\begin{aligned}\delta_f &= (T_1(\alpha) \to T_2(\alpha)) \to (\delta_{x[True,False]} \times \delta_{x[\text{`a'},\text{`b'}]}) \\ T_1(\alpha) \to T_2(\alpha) &\leq [Bool] \to \delta_{x[True,False]} \\ T_1(\alpha) \to T_2(\alpha) &\leq [Char] \to \delta_{x[\text{`a'},\text{`b'}]} \ .\end{aligned}$$

Redex-I reduction yields

$$\begin{aligned}\delta_f &= (T_1(\alpha) \to T_2(\alpha)) \to (T_2(\beta) \times T_2(\gamma)) \\ T_1(\alpha) \to T_2(\alpha) &\leq [Bool] \to T_2(\beta) \\ T_1(\alpha) \to T_2(\alpha) &\leq [Char] \to T_2(\gamma) \ .\end{aligned}$$

Introducing SSU's leaves us with the type

$$\forall T_1.\forall T_2.\forall.(\forall\alpha.T_1(\alpha) \to T_2(\alpha)) \to (T_2(\beta S_1) \times T_2(\gamma S_2)) \ ,$$

where $S_1 = SU(T_1(\alpha) \to T_2(\alpha), [Bool] \to T_2(\beta))$ and $S_2 = SU(T_1(\alpha) \to T_2(\alpha), [Char] \to T_2(\gamma))$. Note that we have used our knowledge of the domains of these SSU's to avoid superfluous application of them in the final type. In particular, we only applied S_1 to β, and we only applied S_2 to γ.

Upon examining the two SSU's S_1 and S_2, we see that the constructor T_1 is matched in S_1 against the constant expression $[Bool]$, and in S_2 against the constant expression $[Char]$. Their most specific common anti-instance (which happens to be univariate) is $[\epsilon]$, for some variable ϵ. We therefore assign $T_1 = \lambda\epsilon.[\epsilon]$ (since there are no other SSU's, there is no need to refine this estimate) and substitute this instantiation into the two SSU's:

$$\begin{aligned}S_1 &= SU([\alpha] \to T_2(\alpha), [Bool] \to T_2(\beta)) \\ S_2 &= SU([\alpha] \to T_2(\alpha), [Char] \to T_2(\gamma)) \ .\end{aligned}$$

In each SSU, there is now a redex-II, reducing β to $Bool$ in S_1 and γ to $Char$ in S_2, whereupon both SSU's become solved. These reductions then lead to a final type of

$$\forall T.(\forall\alpha.[\alpha] \to T(\alpha)) \to (T(Bool) \times T(Char)) \ ,$$

where T_2 has been renamed to T for convenience.

In Section 5.1, we mentioned three possible candidate arguments for the function f: *reverse*, *head*, and *length*, with respective types $\forall\alpha.[\alpha] \to [\alpha]$, $\forall\alpha.[\alpha] \to \alpha$, and $\forall\alpha.[\alpha] \to Int$. For these arguments, then, T is, respectively, $\lambda\tau.[\tau]$, $\lambda\tau.\tau$, and $\lambda\tau.Int$. Substituting these for T in the type we derived for f yields the following specialized types, respectively:

$$(\forall\alpha.[\alpha] \to [\alpha]) \to ([Bool] \times [Char])$$
$$(\forall\alpha.[\alpha] \to \alpha) \to (Bool \times Char)$$
$$(\forall\alpha.[\alpha] \to Int) \to (Int \times Int)\ .$$

These are precisely the types that would have been obtained for f, had the arguments above been supplied from the beginning, or at least their types supplied via annotation. Note again, however, that any annotation for one of the above arguments would not have been able to capture the types of the other two arguments, whereas the third-order type captures all of them. Also recall that the ordinary KW inference procedure would have simply output the uninteresting type $\bot \to \bot \times \bot$ for f.

Note that the soundness of the specialization procedure follows directly from the soundness of the type inference procedure itself. Since every specialization of a type containing constructor variables is guaranteed to be either derivable or undefined, and we are simply performing specializations, we can be sure that the types we produce are valid for the given source term.

5.8 Link-Time Specialization

We turn now to the question of checking the type of a separately-compiled rank 2 function against that of a supplied rank 1 argument for compatibility. This kind of checking, which conceptually takes place at link-time (though it may also occur at compile-time) is necessary in order to support separate compilation. We have already seen evidence in previous sections that link-time type checking for separately-compile function and argument is possible; we formalize the procedure in this section.

In Section 5.7, when we were considering which, among a range of possible specializations for a constructor variable T, to choose, we chose the more specific $\lambda\tau.[\tau]$ over the less specific $\lambda\tau.\tau$, with the expectation that we would be able to recover any expressive loss at link-time. We explore the consequences of choosing one specialization or another here. For concreteness, we use the two candidate constructors $\lambda\tau.[\tau]$ and $\lambda\tau.\tau$ that we had discussed previously, but the same argument holds for any two constructors, in which one is a specialization of the other.

A specialization of $T_1 = \lambda\tau.[\tau]$ leads to a parameter assumption of $\forall\alpha.[\alpha] \to T_2(\alpha)$, whereas a specialization of $T_1 = \lambda\tau.\tau$ leads to a parameter assumption of $\forall\alpha.\alpha \to T_2(\alpha)$. Suppose the intended function argument has type $\forall\alpha.[\alpha] \to \tau_1$, for some τ_1. Then this type is compatible with the first parameter assumption, but not the second—a parameter assumption of $\forall\alpha.\alpha \to T_2(\alpha)$ demands that the argument be a function applicable to *all* types, rather than just list types. On the other hand, suppose the intended function argument has type $\forall\alpha.\alpha \to \tau_2$ for some τ_2. Then this type is compatible with the second parameter assumption, and if we specialize α to $[\beta]$, thereby obtaining the type $\forall\beta.[\beta] \to \tau_2[[\beta]/\alpha]$, we find that the type is also compatible with the first parameter assumption. Thus, the set of argument types admitted by the more specific assumption strictly contains those admitted by the more general assumption.

For a concrete setting in which to explore these candidate specializations, we use two inequalities from the Haskell example:

$$\begin{aligned} T_1(\alpha) \to T_2(\alpha) &\leq [Bool] \to T_2(\beta) \\ T_1(\alpha) \to T_2(\alpha) &\leq [Char] \to T_2(\gamma)\ . \end{aligned}$$

As we observed in Section 5.7, the only two values of T_1 for which these inequalities can possess solutions are $\lambda\tau.\tau$ and $\lambda\tau.[\tau]$.

As we observed before, if we specialize T_1 to $\lambda\tau.[\tau]$, then the inequalities reduce to

$$\begin{aligned} [\alpha] \to T_2(\alpha) &\leq [Bool] \to T_2(Bool) \\ [\alpha] \to T_2(\alpha) &\leq [Char] \to T_2(Char)\ . \end{aligned}$$

If we specialize T_1 to $\lambda\tau.\tau$, then the inequalities reduce to

$$\begin{aligned} \alpha \to T_2(\alpha) &\leq [Bool] \to T_2([Bool]) \\ \alpha \to T_2(\alpha) &\leq [Char] \to T_2([Char])\ . \end{aligned}$$

In essence, the removal of the list constructor on the left is balanced by the additional list constructor on the right, in each inequality.

As we have already observed, an argument type of $\forall\alpha.[\alpha] \to \tau_1$ would not be compatible with the second set of inequalities. On the other hand, an argument type of $\forall\alpha.\alpha \to \tau_2$ is compatible with both, provided that, in the first case, α is specialized to some $[\beta]$. In that case, every occurrence of α in τ_2 becomes an occurrence of $[\beta]$. This may be modelled in the first set of

inequalities by specializing T_2 to $\lambda\tau.T_3([\tau])$ for some new constructor variable T_3:

$$\begin{aligned} [\alpha] \to T_3([\alpha]) &\leq [Bool] \to T_3([Bool]) \\ [\alpha] \to T_3([\alpha]) &\leq [Char] \to T_3([Char]) \ . \end{aligned}$$

The specialization is permitted because the constructor variables will eventually be universally quantified, and a specialization of this time must ultimately take place (if only implicitly) for an argument of type $\forall\alpha.\alpha \to \tau_2$. Upon performing this specialization, we now see that the result is identical to that of the second set of inequalities; hence the result is equivalent, and therefore there is, in fact, no advantage to choosing the more general specialization for the constructor variable T_1.

In addition to justifying our choice of constructor specialization, the above discussion provides an overview of the processes that must take place during link-time argument checking. In particular, we see that, not only the constructor variables, but also some type variables must be specialized so that the parameter type of a rank 2 function matches the rank 1 type of its argument. As we have already noted, matching between terms that contain variables in the functor position is an instance of second-order unification, which is undecidable in general. Recently, however, Stirling [57, 56] confirmed a long-standing conjecture of Huet that the higher-order *matching* problem, in which all (free) variables occur on one side of an equality, is decidable. The matching problem we have before us, though not quite a subproblem of the Huet/Stirling matching problem, is similarly specialized, in the following ways:

- only the rank 2 function's type will contain constructor variables;
- we may only specialize type variables that occur as part of the rank 1 argument's type.

Here, although both the function's parameter type and the argument's type may contain variables, the higher-order variables (i.e., the constructor variables, which are second-order) only occur on one side of the equality between these two expressions. Furthermore, there is no nesting of constructor variable applications within the assumed parameter type.

The procedure for instantiating constructor variables in the parameter specification and type variables in the argument type, so that the two expressions match, is outlined in Figure 5.5. The procedure is parameterized by a working set of parameter-argument pairs to be satisfied. On the initial call, we supply a singleton set consisting of the parameter/argument pair we wish to match. The algorithm then takes apart the expressions by stripping off functors and replacing the entire

ALGORITHM match
INPUT: set $\mathcal{S}$ of pairs $\langle \tau_P, \tau_A \rangle$ of parameter and argument type expressions
OUTPUT: substitution on constructors in each τ_P and variables in each τ_A that unifies $\mathcal{S}$.

match =
  if $\mathcal{S}$ is empty then []
  if each $\langle \tau_P, \tau_A \rangle$ in $\mathcal{S}$ is of the form $\langle T(\alpha), \beta \rangle$ or $\langle \alpha, \beta \rangle$ then
    if $\exists$ a pair $p = \langle T(\alpha), \beta \rangle \in \mathcal{S}$
      match$((\mathcal{S} \setminus \{p\})[\lambda\tau.\tau/T]) \circ [\lambda\tau.\tau/T]$
    else
      let $\langle \alpha, \beta \rangle \in \mathcal{S}$ in match$((\mathcal{S} \setminus \{\langle \alpha, \beta \rangle\})[\alpha/\beta]) \circ [\alpha/\beta]$
  else if $\exists$ a pair $p = \langle f(\tau_{P_1}, \ldots, \tau_{Pn}), g(\tau_{A_1}, \ldots, \tau_{Am}) \rangle \in \mathcal{S}$ then
      if $f \neq g$ or $m \neq n$ then
        error
      else
        match$((\mathcal{S} \setminus \{p\}) \cup \{\langle \tau_{P1}, \tau_{A1} \rangle, \ldots, \langle \tau_{Pn}, \tau_{An} \rangle\})$
  else if $\exists$ a pair $p = \langle T(\alpha), f(\tau_{A1}, \ldots, \tau_{An}) \rangle \in \mathcal{S}$ then
    let
      $T_1, \ldots, T_n$ be fresh constructor variables
      $\sigma \;=\; [\lambda\tau.f(T_1(\tau), \ldots T_n(\tau))/T]$
    in
      match$((\mathcal{S} \setminus \{p\})\sigma \cup \{\langle T_1(\alpha), \tau_{A1} \rangle, \ldots, \langle T_n(\alpha), \tau_{An} \rangle\}) \circ \sigma$
  else if $\exists$ a pair $p = \langle f(\tau_{P1}, \ldots, \tau_{Pn}), \beta \rangle \in \mathcal{S}$ then
    let
      $\beta_1, \ldots, \beta_n$ be fresh variables
      $\sigma \;=\; [f(\beta_1, \ldots, \beta_n))/\beta]$
    in
      match$((\mathcal{S} \setminus \{p\})\sigma \cup \{\langle \tau_{P1}, \beta_1 \rangle, \ldots, \langle \tau_{Pn}, \beta_n \rangle\}) \circ \sigma$
  else if $\exists$ a pair $\langle \alpha, f(\tau_1, \ldots \tau_n) \rangle \in \mathcal{S}$ then
    error

Figure 5.5: Link-time matching algorithm.

functor application pair in the set by pairs matching off corresponding arguments. In so doing, the algorithm decreases the total size of elements in the working set. If the algorithm detects a functor mismatch, it signals an error. Otherwise, after matching all corresponding functions with each other and decomposing, if functor applications remain in the working set, then there are three possibilities:

- a type constructor application $T(\alpha)$ in the parameter specification matches a functor application $f(\tau_1, \ldots, \tau_n)$ in the argument type. In this case, we specialize $T(\alpha)$ to the functor application $f(T_1(\alpha), \ldots, T_n(\alpha))$, where $T_1, \ldots, T_n$ are fresh constructor variables, decompose the now-matching functor applications, and continue. In this way, the total number of functor occurrences in the working set decreases;

- a functor application $f(\tau_{P1}, \ldots, \tau_{Pn})$ in the parameter specification matches a variable β in the argument specification. In this case, we specialize β to $f(\beta_1, \ldots, \beta_n)$, where $\beta_1, \ldots, \beta_n$ are fresh type variables, decompose the now-matching functor applications, and continue. Again, the total number of functor occurrences in the working set decreases;

- a variable in the parameter specification matches a functor application in the argument type. In this case, matching the expressions would require a type variable replacement in the parameter specification, which we do not permit; hence, we signal an error.

Once all functor applications have been removed from the working set, all that remain are pairs of the form $\langle T(\alpha), \beta \rangle$ or $\langle \alpha, \beta \rangle$. At this point, we specialize all constructors T to identity constructors, and then for each $\langle \alpha, \beta \rangle$, set β to α.

As the steps outlined above always decrease the number of functor occurrences until none remain, then decrease the number of constructor variable occurrences until none remain, and finally decrease the number of pairs in the instance (because solving the pair $\langle \alpha, \beta \rangle$ unifies α and β, rendering the pair no longer useful, and the algorithm then removes it), the procedure must terminate.

5.9 When Constructors may Ignore their Arguments

The introduction of constant types in Section 5.7 not only provides essential practical expressiveness, but also allows us to solve for some of our type constructor variables, thereby simplifying and/or eliminating SSU's. On the other hand, the presence of constant types like *Int* and *Bool*

give rise to the possibility that constructor variables may themselves be specialized to constant functors, and thereby ignore their arguments—if a constructor T may be specialized to, say, *Int*, then we can no longer be certain that a constructor application $T(\alpha)$ actually contains an occurrence of α.

Since we can no longer be sure that the variable α is guaranteed to occur in the expression $T(\alpha)$ once T is specialized, any redexes that arise based on the occurrence of α in $T(\alpha)$ may not actually be present under the nullary specialization of T; hence, in reducing the redex, we may actually perform an unnecessary substitution, and thereby introduce a source of incompleteness into the system.

For example, if a USUP instance contains the inequality[3]

$$T(\alpha \to \alpha) \ \leq \ T(\beta) \ ,$$

then there is a redex-I replacing β with $\gamma \to \gamma$, where γ is a fresh variable. However, if T is specialized to a constant functor, then the redex-I does not exist, and occurrences of β within the USUP instance will have been replaced unnecessarily.

On the other hand, if *all* occurrences of β are as arguments of the constructor variable T, then just as the occurrence of $\alpha \to \alpha$ that occasioned the redex-I disappears, so too do all of the identifiers that would be affected by the redex-I. Hence, there is no loss of completeness in reducing the redex, as the effect of its reduction would be completely erased by specializing T to a constant expression. The same result holds true for redex-III and redex-IV reductions as well.

For redex-II reductions, there are potentially two type constructors involved, one or both of which might be instantiated to a constant. Consider, for example, the following inequality:

$$T_1(\alpha) \to T_2(\alpha) \ \leq \ T_1(\beta) \to T_2(\gamma) \ .$$

Here, the two occurrences of α on the left create a redex-II, calling for the unification of β and γ. If both T_1 and T_2 are specialized so that they use their arguments, then there is, of course, no problem; the redex will still exist. If both T_1 and T_2 are specialized to constants, then the redex disappears, but so long as all occurrences of β and γ are as part of arguments to T_1 and T_2, the effect of reducing the redex-II disappears as well.

[3]As we shall see, such an inequality cannot actually arise during the execution of the third-order inference procedure; the intent of this example is merely to illustrate the effect of nullary instantiation on arbitrary redex reduction.

The situation becomes more interesting when one, but not both, of the constructor variables is specialized to a constant. In such cases, the redex-II disappears, but its effect remains visible, through the unification of β and γ. For the sake of concreteness, let us assume that T_1 is specialized to a constant, and T_2 is not. To satisfy the redex-II, we have two choices: we can replace β with γ, or we can replace γ with β. If we replace β with γ, then the effect is erased within the subexpression $T_1(\beta)$ on the right-hand side, because T_1 is a constant (again, the effect would potentially be visible in any occurrences of β not inside of T_1). On the other hand, if we replace γ with β, then the effect is visible in T_2, whose argument is now β, rather than γ. If, as we have been supposing, however, all occurrences of β are as part of arguments to T_1, then the observable effect is simply a renaming of γ to β throughout the instance, which is harmless.

The effect of the redex-II reduction is less harmless, however, if the replacement is not simply a unification of two variables. Consider, for example, the following inequality:

$$T_1(\alpha) \to T_2(\alpha) \;\leq\; T_1(\beta \to \beta) \to T_2(\gamma) \;.$$

In this case, the redex-II calls for γ to be replaced by $\beta \to \beta$. There is no choice of direction in which to perform the replacement, and the result of the replacement is not a simple renaming. In such cases, the effect of the redex reduction is indeed visible, even though the redex itself never materializes.

In general, therefore, redex-II reduction can introduce substitutions that might not have been necessary if certain constructors had been instantiated to constants. In the case of the third-order inference procedure itself, however, we make the following observations:

Theorem 5.4 *Suppose that, throughout the execution of the third-order inference procedure, no variable occurs both as part of an argument of a constructor variable, and outside all constructor variable applications. Then during the execution of the third-order inference procedure, before SSU application, there will never be an occurrence of a subterm of the form $T(\tau)$ where τ is a functor application.*

Proof Suppose such a subterm occurs. It does not occur as a result of introducing third-order type assumptions for parameters, by design. Therefore, an occurrence of $T(\tau)$ can only arise as the result of a redex reduction, of which there are four possibilities:

- redex-I reduction. Then $T(\tau)$ occurs as a result of a replacement $[\tau/\alpha]$ for some α (it cannot occur as a result of a replacement $[T(\tau)/\alpha]$, as that would imply that $T(\tau)$ already

had an occurrence). The existence of the redex implies that there must be an inequality $\tau_i \leq \mu_i$ and path Σ such that $\Sigma(\tau_i)$ is a renaming τ_0 of τ and $\Sigma(\mu_i) = \alpha$. The path Σ cannot contain the constructor T. Hence, α occurs outside of T. On the other hand, for the subexpression $T(\tau)$ to be created, there must be an occurrence somewhere in the instance of the subexpression $T(\alpha)$. But then α occurs both inside and outside of a constructor variable, which contradicts our hypothesis.

- redex-IV reduction. The reasoning is the same as for redex-I reduction.
- redex-III reduction. The substitution solving a redex-III may be broken into a sequence of replacements. For each such replacement, the argument is the same as for redex-I and redex-IV reduction.
- redex-II reduction. Again, $T(\tau)$ must arise as a result of some replacement $[\tau/\alpha]$. There is an inequality $\tau_i \leq \mu_i$, and paths $\Sigma_1 \neq \Sigma_2$ such that $\Sigma_1(\tau_i) = \Sigma_2(\tau_i) = \beta$, and unifying $\Sigma_1(\mu_i)$ and $\Sigma_2(\mu_i)$ brings about the replacement $[\tau/\alpha]$. Thus, there is a path Π such that (without loss of generality) $\Pi\Sigma_1(\mu_i) = \alpha$ and $\Pi\Sigma_2(\mu_i) = \tau$. The path $\Pi\Sigma_2$ may not contain a constructor variable. However, in order for the subexpression $T(\tau)$ to arise, there must be an occurrence of the subexpression $T(\alpha)$ somewhere; hence the path $\Pi\Sigma_1$ must contain T. Since $\Pi\Sigma_2$ does not contain T, it follows that Σ_1 contains T, but Σ_2 does not. But then the two occurrences $\Sigma_1(\tau_i)$ and $\Sigma_2(\tau_i)$ of β differ in that the former occurs within a constructor variable application, and the latter does not, which contradicts our hypothesis. Hence, the expression $T(\tau)$ cannot arise.

In all cases, then, so long as no variable occurs both inside and outside constructor applications, there can be no occurrences of a compound expression as the argument of a constructor application. □

Note that, at the time that third-order type assumptions are added to the USUP instance during the third-order inference procedure, the variables do indeed have the property that those that occur as constructor variable arguments are disjoint from those that do not.

Theorem 5.5 *During the third-order inference procedure, if constructor variables are not specialized, then no variable that occurs as part of a constructor variable application will ever occur outside a constructor variable application, and vice versa.*

Proof Let α occur as part of a constructor variable application. Suppose a reduction $[\tau/\beta]$, where α is a subexpression of τ, causes α to escape the constructor variable application, such that

previously no variable occurred both inside and outside a constructor variable application. Then before reduction, there existed an occurrence of β outside of a constructor variable application. The argument proceeds according to the type of redex whose reduction includes the substitution $[\tau/\beta]$:

- redex-I reduction. Then there was an inequality $\tau_i \leq \mu_i$ and a path Σ such that $\Sigma(\tau_i)$ is some renaming τ_0 of τ, and $\Sigma(\mu_i) = \beta$. Since τ_0 contains α (actually only possible if α is an unknown, since redex-I reduction creates fresh variables), and there was previously no occurrence of a variable both inside and outside a constructor variable application, the path Σ must include a constructor. But then β occurred inside a constructor variable application. As we have already observed, β also had an occurrence outside of any constructor variable application, which contradicts our hypothesis. Hence, α cannot escape via redex-I reduction.

- redex-IV reduction. The argument is the same as for redex-I reduction.

- redex-III reduction. The argument is the same as for redex-I and redex-IV reduction.

- redex-II reduction. Then there was an inequality $\tau_i \leq \mu_i$ and paths $\Sigma_1 \neq \Sigma_2$ such that $\Sigma_1(\tau_i) = \Sigma_2(\tau_i) = \gamma$ for some variable γ, and there is a path Π such that (without loss of generality) $\Pi\Sigma_1(\mu_i) = \beta$ and $\Pi\Sigma_2(\mu_i) = \tau$. The path $\Pi\Sigma_2$ must contain a constructor variable; otherwise α would already have an occurrence outside a constructor variable application. On the other hand, $\Pi\Sigma_1$ cannot contain a constructor variable; otherwise, β would have occurrences both inside and outside of a constructor variable application. Hence, Σ_1 contains no constructor variables and Σ_2 contains at least one constructor variable. Thus, the two occurrences $\Sigma_1(\tau_i)$ and $\Sigma_2(\tau_i)$ of γ differ, in that one is inside a constructor variable application, and the other is not, which is a contradiction. Hence, α cannot escape redex-II reduction.

Suppose now that α occurs outside a constructor variable application, and the reduction $[\tau/\beta]$, where α is a subexpression of τ, cause α to appear inside some constructor variable application, such that previously no variable occurred both inside and outside a constructor variable application. Then before reduction, there existed an occurrence of β inside of a constructor variable application. The argument is almost identical to the previous case, with "outside" and "inside" interchanged. The details are presented below. As before, we proceed according to the type of redex whose reduction includes the substitution $[\tau/\beta]$:

- redex-I reduction. Then there was an inequality $\tau_i \leq \mu_i$ and a path Σ such that $\Sigma(\tau_i)$ is some renaming τ_0 of τ, and $\Sigma(\mu_i) = \beta$. Since τ_0 contains α (actually only possible if α is an unknown, since redex-I reduction creates fresh variables), and there was previously no occurrence of a variable both inside and outside a constructor variable application, the path Σ cannot include a constructor. But then β has an occurrence outside of any constructor variable application. As we have already observed, β also had an occurrence inside of a constructor variable application, which contradicts our hypothesis. Hence, α cannot escape via redex-I reduction.

- redex-IV reduction. The argument is the same as for redex-I reduction.

- redex-III reduction. The argument is the same as for redex-I and redex-IV reduction.

- redex-II reduction. Then there was an inequality $\tau_i \leq \mu_i$ and paths $\Sigma_1 \neq \Sigma_2$ such that $\Sigma_1(\tau_i) = \Sigma_2(\tau_i) = \gamma$ for some variable γ, and there is a path Π such that (without loss of generality) $\Pi\Sigma_1(\mu_i) = \beta$ and $\Pi\Sigma_2(\mu_i) = \tau$. The path $\Pi\Sigma_2$ cannot contain a constructor variable; otherwise α would already have an occurrence inside a constructor variable application. On the other hand, $\Pi\Sigma_1$ must contain a constructor variable; otherwise, β would have occurrences both inside and outside of a constructor variable application. Hence, Σ_1 contains at least one constructor variable and Σ_2 contains no constructor variables. Thus, the two occurrences $\Sigma_1(\tau_i)$ and $\Sigma_2(\tau_i)$ of γ differ, in that one is inside a constructor variable application, and the other is not, which is a contradiction. Hence, α cannot escape redex-II reduction.

In conclusion, the set of variables occurring inside some constructor variable application, and the set of variables occurring outside of all constructor variable applications remain disjoint throughout the running of the third-order inference procedure. □

Corollary 7 *During the running of the third-order inference procedure, the argument of a constructor variable application will never be a functor application.*

Proof The conclusion of the corollary holds when the inference procedure inserts the third-order type assumptions. By Theorem 5.4, so long as no variable occurs both inside of a type constructor application and outside of all type constructor applications, a functor application will not arise as the argument of a constructor variable application. By Theorem 5.5, no variable will ever occur

both inside a constructor variable application and outside of all constructor variable applications, from which the result follows. □

As a consequence of this last result, we can be assured that inequalities of the form

$$T_1(\alpha) \to T_2(\alpha) \ \leq \ T_1(\beta \to \beta) \to T_2(\gamma)$$

or

$$T_1(\alpha \to \alpha) \ \leq \ T_2(\beta)$$

(or similar inequalities with more complex constructor variable arguments than $\alpha \to \alpha$ or $\beta \to \beta$) cannot occur. Thus, redex reductions in which the expressions in the replacement lie within constructor variable applications can only involve variable-for-variable substitutions, and in particular cannot be redex-I reductions.

Having established that a variable cannot lie both inside a constructor variable application and outside all constructor variable applications (and cannot move between these two possibilities), we turn now to the question of when a variable may move from one constructor variable application to another. As redex-I, redex-III, and redex-IV all require that the same path be applied to both sides of an inequality in order to find a redex, any variables that propagate across an inequality due to these kinds of reductions will find themselves within the same constructor variable application on the other side of the inequality as they lay within before the reduction. Hence, it is only the redex-II reduction that can cause an identifier to propagate to other constructor variable applications. Our example

$$T_1(\alpha) \to T_2(\alpha) \ \leq \ T_1(\beta) \to T_2(\gamma) \ ,$$

in fact, illustrates this phenomenon. Here, depending on the direction of the reduction, either β propagates to T_2 or γ propagates to T_1. Suppose T_1 will be specialized to a constant, and T_2 will not. Then the substitution (say we perform $[\beta/\gamma]$) will have no more effect than a simple renaming, provided that all occurrences of β are currently within an application of T_1. If this is not true, then there is some occurrence $T_3(\beta)$ somewhere in the instance, and a reduction either propagated β from T_3 to T_1, or from T_1 to T_3. We show below, however, that such a propagation is not possible.

Before we proceed with the result, we make the following observation: given type constructor variables T_1 and T_2 and type variables α and β, if there is a redex calling for the unification of $T_1(\alpha)$ and $T_2(\beta)$, then it is *not* a valid conclusion that $T_1 = T_2$ and $\alpha = \beta$. Suppose, for example,

that $T_1 = \lambda\tau.Int$, $T_2 = \lambda\tau.\tau$, $\beta = Int$, and $\alpha \neq Int$. Then $T_1(\alpha) = Int = T_2(\beta)$, even though $T_1 \neq T_2$ and $\alpha \neq \beta$. Thus it is *not* a valid reduction step to unify constructor variables[4].

We now proceed with our result:

Theorem 5.6 *Let T_1 and T_2 be type constructors, and suppose that they were introduced by the third-order inference procedure as part of* different *parameter type assumptions. Then at no point will T_1 and T_2 occur together on a left-hand side with the same argument.*

Proof Since T_1 and T_2 arise from different parameter annotations (say of x and y), the procedure introduces two distinct inequalities[5]:

$$\begin{aligned} \beta_x &= \cdots\ T_1(\alpha)\ \cdots \\ \beta_y &= \cdots\ T_2(\alpha)\ \cdots \end{aligned}$$

After β_x and β_y are replaced, the instance contains some number of inequalities whose left-hand sides are $(\cdots\ T_1(\alpha)\ \cdots)$, and some number of inequalities whose left-hand sides are $(\cdots\ T_2(\alpha)\ \cdots)$, but none whose left-hand sides contain both $T_1(\alpha)$ and $T_2(\alpha)$. Since these subexpressions now reside only on left-hand sides (assuming the equalities on β_x and β_y, now solved, have been removed), no reduction will replace them. Therefore, if T_1 and T_2 are to appear together on some left-hand side with the same argument, they must at least propagate across the inequalities on which they reside, which can only happen via redex-I or redex-IV reduction. Either way $T_1(\alpha)$ becomes $T_1(\beta)$ and $T_2(\alpha)$ becomes $T_2(\gamma)$, where β and γ are fresh variables (or unknowns), and are, in particular, distinct. Eventually, if T_1 and T_2 are to occur together on a left-hand side with the same argument, however, their arguments will have to be unified via some redex reduction. Suppose, without loss of generality, that the redex reduction in question unifies β and γ. By Theorem 5.5, β and γ cannot occur outside a constructor variable application. Further, β (and similarly γ) cannot appear as the argument of a constructor T_3 not originally associated with β_x (respectively β_y for γ)—otherwise, there would have to be an inequality such that either

- $T_1(\delta)$ and $T_3(\delta)$ occur together on a left-hand side, for some δ (in order to create the needed redex-II); or

[4]Note, however, that for single-variable constructor variables that are not permitted to ignore their arguments, the conclusion is valid.

[5]We could have chosen to use a different argument variable in each inequality, rather than α both times, but it is not strictly necessary, as after reduction (and removal of the solved inequalities), α will reside only on left-hand sides.

- a variable δ has two occurrences on a left-hand sides in positions that create a redex-II calling for the unification of $T_1(\epsilon)$ and $T_3(\zeta)$ for some ϵ, ζ; or
- T_1 and T_3 occur in corresponding positions on opposite sides of the same inequality, and the argument of the constructor on the left-hand side is an unknown (in order to create a redex-I that would copy the unknown across the inequality, thereby unifying the two arguments).

In the first case, there must have already existed constructor variables arising from different annotations, but residing together on A left-hand side with the same argument—but this cannot happen if the unification of β and γ creates the first such situation. In the second and third cases, the unification cannot proceed without also unifying T_1 and T_3 themselves—which is not permitted.

Therefore, β can only occur as an argument of a type constructor associated with the parameter x and similarly for γ and the parameter y. Without loss of generality, we can take the constructors of interest to be T_1 and T_2, respectively. Thus, the unification of β and γ can occur in exactly the three ways outlined above, and as before, each either implies a pre-existing violation of the conclusion of the theorem, or that T_1 and T_2 would be unified. Thus, β and γ cannot be unified, and the result follows. □

Corollary 8 *A variable α that occurs as an argument of a constructor T_1 associated with the annotation for a parameter x cannot propagate through redex reduction such that it becomes the argument of a constructor T_2 associated with a parameter $y \neq x$, unless redex reduction unifies T_1 and T_2.*

Proof Suppose the result were false. Then redex reduction propagates α to T_2, either directly or in stages. At some point, the redex reduction must propagate α to some T_3, not associated with x, by unifying α with some variable β occurring as the argument of T_3. As in the proof of Theorem 5.6, the only way such a redex can occur (without requiring that constructor variables be unified) is if, for some variable δ, $T_1(\delta)$ and $T_3(\delta)$ occur together on a left-hand side, in order to create the needed redex-II. This condition, however, violates Theorem 5.6. Thus α cannot propagate to a constructor associated with a different parameter. □

Returning now to our discussion of the effect of potentially constant constructor variables on redex-II reduction, consider again our example:

$$T_1(\alpha) \to T_2(\alpha) \ \leq \ T_1(\beta) \to T_2(\gamma) \ ,$$

where T_1 is specialized to a constant and T_2 is not. We now know that the parameter β will not occur as an argument of any constructor T_3 associated with a different annotation from that with which T_1 is associated. Thus, if any occurrence of β is to persist when T_1 is a constant, it must be because the constructor T_3 that contains it is, in fact, associated with the same parameter as T_1 and T_2, so that the inequality on which the annotation appears on the left looks, in fact, more like the following:

$$T_1(\alpha) \to T_2(\alpha) \to T_3(\alpha) \leq T_1(\beta) \to T_2(\gamma) \to T_3(\beta) \ .$$

In this case, however, even if T_1 is constant, so long as T_3 is not constant, then the redex-II can be said to occur between T_3 and T_2, rather than T_1 and T_2. Thus, the effect of the redex reduction does not disappear, but, in fact, neither does the redex.

In conclusion, as we have shown, no loss of completeness is introduced by the addition of constant types to the inference procedure, as the erasure of a redex by a constant constructor instantiation is also guaranteed to erase the *effect* of reducing the redex. Thus, we can employ, without reservation, the version of the algorithm that includes constant constructors.

5.10 Multi-Argument Constructor Variables

The third-order notation that we have been developing throughout this chapter has, as we have seen, the ability to abstract over types in ways not possible within ordinary System F types. The expressive power of these types facilitates code reuse and separate compilation to a greater degree than is possible in systems based solely on System F types, by expanding the range of arguments to which a function f may be applied, without having to derive a new type for f.

On the other hand, the restriction that all type constructor variables introduced by the third order inference procedure be *unary* imposes a limit on its expressive power that is not experienced by systems that use ordinary System F types. The types of common library functions, like *map* and *apply*, cannot be expressed in full generality using only a single type variable. Rather, their principal (rank 1) types are $\forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$ and $\forall\alpha.\forall\beta.(\alpha \to \beta) \to (\alpha \to \beta)$, respectively. However, without at least bivariate type constructors, we cannot support such functions (in full generality) as arguments of a rank 2 function. Other well-known functions require even more type variables in order to express their principal types. To accommodate such functions requires support within the third-order inference procedure for multi-argument type constructors.

From the start, there are two major difficulties in establishing support for multi-argument type constructors:

- we do not know from the outset just how many arguments a given type constructor variable should have;
- a constructor variable taking multiple parameters may turn out to be simply a projection onto one of them, thereby ignoring some of its arguments. As a result, we may reduce redexes that ultimately never existed, and thereby give up a measure of completeness.

As an example of the second difficulty, consider the constructor application $T(\alpha, \beta)$. Two valid instantiations of T include the projections $T(\alpha, \beta) = \alpha$ and $T(\alpha, \beta) = \beta$. In each case, one of the arguments of T (respectively, β and α) disappears. If the presence of the argument that disappears had created a redex somewhere, then potentially an unnecessary substitution will have been performed.

In light of what we observed in Section 5.9, we now know that if a redex is reduced because a constructor argument disappears, then so too do the effects of reducing the redex—the theorems and proofs in Section 5.9 transfer to the multi-argument case without any difficulty at all. Thus, as with the possibility of constant constructors, we do not need to be afraid to reduce any redex; completeness will not suffer. Indeed the multi-argument case may be viewed as a generalization of the constant constructor case. In the former, a projection reduces the effective arity of the constructor variable by at least one; in the latter, instantiation of a constructor to a constant reduces its effective arity from one to zero.

We are left, then, with the question of how many arguments to give any particular constructor variable. Because we know that projections do not pose any threat to completeness, we can now say that we need not worry about having too many arguments; those we do not use will simply disappear on instantiation. On the other hand, if the number of arguments is too small, then there will be valid arguments to the given function that would be rejected because their types contain too many type variables. Indeed, for any chosen number of constructor arguments, such an outcome is possible.

The obvious way to avoid such potential incompleteness, then, is to give every constructor variable *infinitely many* arguments. Consider our USUP instance for SA, with infinitary con-

structor variables:

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots)) \to \delta_{xx} \\ T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots) &\leq \delta_{x_2} \to \delta_{xx} \\ T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots) &\leq \delta_{x_2} \ . \end{aligned}$$

After two redex-I reductions in the second inequality, we obtain

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots)) \to T_2(\gamma_1, \gamma_2, \ldots) \\ T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots) &\leq T_1(\beta_1, \beta_2, \ldots) \to T_2(\gamma_1, \gamma_2, \ldots) \\ T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots) &\leq T_1(\beta_1, \beta_2, \ldots) \ . \end{aligned}$$

Now, an infinite sequence of redex-II reductions unifies each β_i with each γ_i, and the instance becomes

$$\begin{aligned} \delta_{SA} &= (T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots)) \to T_2(\beta_1, \beta_2, \ldots) \\ T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots) &\leq T_1(\beta_1, \beta_2, \ldots) \to T_2(\beta_1, \beta_2, \ldots) \\ T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots) &\leq T_1(\beta_1, \beta_2, \ldots) \ . \end{aligned}$$

The second inequality is now solved. Introducing an SSU to complete the solution of the last inequality leaves only the first inequality,

$$\delta_{SA} = (T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots)) \to T_2(\beta_1 S, \beta_2 S, \ldots) \ ,$$

where $S = SU(T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots), T_1(\beta_1, \beta_2, \ldots))$. Note that the domain of S includes only the variables β_i; hence, in the first inequality, we need only apply it to the variables β_i. From this inequality, we obtain the final type

$$\forall T_1.\forall T_2.\forall.(\forall(\alpha_1, \alpha_2, \ldots).T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots)) \to T_2(\beta_1 S, \beta_2 S, \ldots) \ ,$$

where $S = SU(T_1(\alpha_1, \alpha_2, \ldots) \to T_2(\alpha_1, \alpha_2, \ldots), T_1(\beta_1, \beta_2, \ldots))$.

The sequence of reductions we outlined above cannot, as given, be codified directly into an algorithm, because it is infinite. A direct implementation, assuming it had a representation for infinitary constructor variables, would get caught in the infinite sequence of redex-II reductions and not proceed further. However, the following observation makes the reduction sequence above feasible: each β_i undergoes the same replacements as all other β_j, and similarly for each γ_i. Thus,

even though there are infinitely many replacements, *they are all the same*. Thus we can encode the argument sequences, and perform the reductions, *lazily*, by only performing the reductions on one constructor argument, and duplicating it as needed to match the eventual function argument at link-time. In this way, we can restructure the type inference for SA as follows:

$$\begin{aligned} \delta_{SA} &= (T_1(\vec{\alpha}) \to T_2(\vec{\alpha})) \to \delta_{xx} \\ T_1(\vec{\alpha}) \to T_2(\vec{\alpha}) &\leq \delta_{x_2} \to \delta_{xx} \\ T_1(\vec{\alpha}) \to T_2(\vec{\alpha}) &\leq \delta_{x_2} \ . \end{aligned}$$

Since all of the arguments of T_1 will be treated the same, we use a vector to represent the entire sequence of arguments. The reduction then proceeds as before, and we arrive at the type

$$\forall T_1.\forall T_2.\forall.(\forall\vec{\alpha}.T_1(\vec{\alpha}) \to T_2(\vec{\alpha})) \to T_2(\vec{\beta}S) \ ,$$

where $S = SU(T_1(\vec{\alpha}) \to T_2(\vec{\alpha}), T_1(\vec{\beta}))$.

For a concrete example of the use of this type we have derived for SA consider the well-known function *map*, of type $\forall\alpha.\forall\beta.(\alpha \to \beta) \to ([\alpha] \to [\beta])$. If we apply SA to *map*, we specialize T_1 to $\lambda(\tau_1, \tau_2, \ldots).\tau_1 \to \tau_2$ and T_2 to $\lambda(\tau_1, \tau_2, \ldots).[\tau_1] \to [\tau_2]$. Notice how the parameter lists of these constructors are essentially *lazy* lists, from the front of which we take only those parameters that we actually need. Then solving the SSU amounts to solving the inequality

$$(\alpha \to \beta) \to ([\alpha] \to [\beta]) \ \leq \ \gamma \to \delta \ .$$

After all redexes are reduced, the inequality becomes

$$(\alpha \to \beta) \to ([\alpha] \to [\beta]) \ \leq \ (\epsilon \to \zeta) \to ([\epsilon] \to [\zeta]) \ .$$

Now, substituting for T_1, T_2, and S in the type of SA, we obtain the type

$$\forall\epsilon.\forall\zeta.(\forall\alpha.\forall\beta.(\alpha \to \beta) \to ([\alpha] \to [\beta])) \to [\epsilon \to \zeta] \to [[\epsilon] \to [\zeta]] \ .$$

As can be verified with any ML[6] or Haskell implementation, the type of `map map` is indeed $\forall\epsilon.\forall\zeta.[\epsilon \to \zeta] \to [[\epsilon] \to [\zeta]]$. Thus the inference algorithm yields for us precisely the type we would expect.

[6]So long as the value restriction is ignored.

5.10.1 Multi-Argument Constructor Variables and Constant Types

We have so far considered the interaction of multi-argument constructor variables with the core inference algorithm, and the results have been favourable. We consider now how constant types, whose effect on the core algorithm was considerable, interact with the multi-argument version of the inference procedure.

As we have already observed, constructor variable instantiations that project away some or all of their arguments do not pose a problem for either nullary or multi-argument type constructors; similarly, no such problem arises from using them in combination.

The interesting effect of combining higher-arity and nullary constructor variables is on our ability to specialize constructors when they are matched against constant expressions. Recall that if, for example we are presented with the SSU's

$$SU(T_1(\alpha) \to T_2(\alpha), (Int \times Bool) \to T_2(\beta))$$
$$SU(T_1(\alpha) \to T_2(\alpha), (Bool \times Int) \to T_2(\gamma)) \ ,$$

then $T_1(\alpha)$ specializes to the most specific univariate common anti-instance of $Int \times Bool$ and $Bool \times Int$, which is α itself.

However, now that our type constructors are not required to be unary, we no longer need to insist on univariate common anti-instances of constant expressions; rather, we can simply take the most specific common anti-instance, and employ as many variables as necessary.

Consider, therefore, the following modification of our original Haskell example:

```
foo :: ((Char, Bool), (Bool, Char))
foo = let
       f x = (x (True, 'a'), x ('b', False))
      in
       f (\ (x,y) -> (y,x))
```

Now, rather than apply the polymorphic argument `x` of `f` to two lists of different types, we apply it to two *tuples* of different types. One possible such function is supplied—it simply swaps the elements of a tuple.

Separating the function from the supplied argument, and translating into a System F setting, yields the function

$$f = \lambda x.(x(True, \text{`a'}), x(\text{`b'}, False)) \ .$$

The reduced USUP instance for f is then

$$\begin{aligned} \delta_f &= \beta_x \to (\delta_{x(\mathit{True},\text{`a'})} \times \delta_{x(\text{`b'},\mathit{False})}) \\ \beta_x &\leq (\mathit{Bool} \times \mathit{Char}) \to \delta_{x(\mathit{True},\text{`a'})} \\ \beta_x &\leq (\mathit{Char} \times \mathit{Bool}) \to \delta_{x(\text{`b'},\mathit{False})} \ . \end{aligned}$$

We make a unary assumption for x, and obtain

$$\begin{aligned} \delta_f &= (T_1(\vec{\alpha}) \to T_2(\vec{\alpha})) \to (\delta_{x(\mathit{True},\text{`a'})} \times \delta_{x(\text{`b'},\mathit{False})}) \\ T_1(\vec{\alpha}) \to T_2(\vec{\alpha}) &\leq (\mathit{Bool} \times \mathit{Char}) \to \delta_{x(\mathit{True},\text{`a'})} \\ T_1(\vec{\alpha}) \to T_2(\vec{\alpha}) &\leq (\mathit{Char} \times \mathit{Bool}) \to \delta_{x(\text{`b'},\mathit{False})} \ . \end{aligned}$$

Reduction then gives

$$\begin{aligned} \delta_f &= (T_1(\vec{\alpha}) \to T_2(\vec{\alpha})) \to (T_2(\vec{\beta}) \times T_2(\vec{\gamma})) \\ T_1(\vec{\alpha}) \to T_2(\vec{\alpha}) &\leq (\mathit{Bool} \times \mathit{Char}) \to T_2(\vec{\beta}) \\ T_1(\vec{\alpha}) \to T_2(\vec{\alpha}) &\leq (\mathit{Char} \times \mathit{Bool}) \to T_2(\vec{\gamma}) \ . \end{aligned}$$

Associated with this reduced instance are the two SSU's

$$SU(T_1(\vec{\alpha}) \to T_2(\vec{\alpha}), (\mathit{Bool} \times \mathit{Char}) \to T_2(\vec{\beta}))$$

and

$$SU(T_1(\vec{\alpha}) \to T_2(\vec{\alpha}), (\mathit{Char} \times \mathit{Bool}) \to T_2(\vec{\gamma})) \ ,$$

which are (except for the vector notation) the SSU's we presented earlier in this section. Our constructor specialization procedure, upon consideration of these SSU's, specializes $T_1(\vec{\alpha})$ to the most specific common anti-instance of $\mathit{Bool} \times \mathit{Char}$ and $\mathit{Char} \times \mathit{Bool}$, which is $\alpha_1 \times \alpha_2$ for some variables α_1, α_2. Since T_1 is parameterized by the vector $\vec{\alpha}$, it follows that the variables α_1 and α_2 belong to $\vec{\alpha}$. Hence we specialize the vector $\vec{\alpha}$, throughout the USUP instance, to $\alpha_1, \alpha_2, \vec{\alpha}$, and similarly for $\vec{\beta}$ and $\vec{\gamma}$. Doing this specialization, while specializing $T_1(\alpha_1, \alpha_2, \vec{\alpha})$ to $\alpha_1 \times \alpha_2$, gives

$$\begin{aligned} \delta_f &= ((\alpha_1 \times \alpha_2) \to T_2(\alpha_1, \alpha_2, \vec{\alpha})) \to (T_2(\beta_1, \beta_2, \vec{\beta}) \times T_2(\gamma_1, \gamma_2, \vec{\gamma})) \\ (\alpha_1 \times \alpha_2) \to T_2(\alpha_1, \alpha_2, \vec{\alpha}) &\leq (\mathit{Bool} \times \mathit{Char}) \to T_2(\beta_1, \beta_2, \vec{\beta}) \\ (\alpha_1 \times \alpha_2) \to T_2(\alpha_1, \alpha_2, \vec{\alpha}) &\leq (\mathit{Char} \times \mathit{Bool}) \to T_2(\gamma_1, \gamma_2, \vec{\gamma}) \ . \end{aligned}$$

Four redex-II reductions (two in each of the second and third inequalities) then yield

$$\begin{aligned}
\delta_f &= ((\alpha_1 \times \alpha_2) \to T_2(\alpha_1, \alpha_2, \vec{\alpha})) \\
&\qquad \to (T_2(Bool, Char, \vec{\beta}) \times T_2(Char, Bool, \vec{\gamma})) \\
(\alpha_1 \times \alpha_2) \to T_2(\alpha_1, \alpha_2, \vec{\alpha}) &\leq (Bool \times Char) \to T_2(Bool, Char, \vec{\beta}) \\
(\alpha_1 \times \alpha_2) \to T_2(\alpha_1, \alpha_2, \vec{\alpha}) &\leq (Char \times Bool) \to T_2(Char, Bool, \vec{\gamma}) \ .
\end{aligned}$$

The last two inequalities are now solved, and we now recover the type[7]

$$\forall T.\forall \vec{\beta}.\forall \vec{\gamma}.(\forall(\alpha_1, \alpha_2, \vec{\alpha}).\alpha_1 \times \alpha_2 \to T(\alpha_1, \alpha_2, \vec{\alpha})) \to (T(Bool, Char, \vec{\beta}) \times T(Char, Bool, \vec{\gamma}))$$

for the function f. If we now specialize T to $\lambda(\tau_1, \tau_2, \vec{\tau}).\tau_2 \times \tau_1$, then we obtain the specialized type

$$(\forall(\alpha_1, \alpha_2).\alpha_1 \times \alpha_2 \to \alpha_2 \times \alpha_1) \to ((Char \times Bool) \times (Bool \times Char))$$

for the function f. This type is precisely the type that our Haskell function `f` requires, in order to accept the function `\ (x, y) -> (y, x)` as its argument `x`. On the other hand, if the argument is `\ (x, y) z -> (y, x, z)` (of type `(a, b) -> c -> (b, a, c)`), then by specializing T to $\lambda(\tau_1, \tau_2, \tau_3, \vec{\tau}).\tau_3 \to \tau_2 \times \tau_1 \times \tau_3$, then we obtain the type

$$\begin{aligned}
&\forall(\beta, \gamma).(\forall(\alpha_1, \alpha_2, \alpha_3).\alpha_1 \times \alpha_2 \to (\alpha_3 \to \alpha_2 \times \alpha_1 \times \alpha_3)) \\
&\qquad \to ((\beta \to Char \times Bool \times \beta) \times (\gamma \to Bool \times Char \times \gamma))
\end{aligned}$$

for f, precisely as expected.

5.10.2 Link-Time Specialization

To complete the description of our extension of the core algorithm to accommodate multi-argument constructor variables, it remains to codify the function-argument matching we performed by hand in the last two examples of the previous section, as the algorithm in Figure 5.5 only addresses the single-argument case. For the most part, the algorithm adapts to the multi-argument case by simply replacing each $T(\alpha)$ with $T(\vec{\alpha})$. The difficulty, however, lies with the pairs $\langle T(\alpha), \beta \rangle$, as, though is clear that T must be specialized to a projection, it is not immediately clear onto which of its arguments T should project.

Of course, in our two aforementioned examples, the question of how to project a constructor variable did not arise—having been specialized by matching against constant expressions, the

[7] Recall that we give the cartesian product operator $\times$ a higher precedence than the function constructor $\to$.

constructor variable was already gone. In general, however, the link-time matching procedure will eventually reach a point where all of the pairs in the working set $\mathcal{S}$ have the form $\langle T(\vec{\alpha}), \beta\rangle$, and we must address the question of how to project T.

Consider, for instance, our earlier example of applying *SA* to the function *map*. The type we derived for *SA* was

$$\forall(T_1, T_2).\forall.(\forall\vec{\alpha}.T_1(\vec{\alpha}) \to T_2(\vec{\alpha})) \to T_2(\vec{\beta}S) \ ,$$

where $S = SU(T_1(\vec{\alpha}) \to T_2(\vec{\alpha}), T_2(\vec{\beta}))$. If we wish to apply *SA* to the function *map*, whose type is $\forall(\alpha, \beta).(\alpha \to \beta) \to ([\alpha] \to [\beta])$, then we call our link-time algorithm with the initial working set

$$\mathcal{S} = \{\langle T_1(\vec{\alpha}) \to T_2(\vec{\alpha}), (\alpha \to \beta) \to ([\alpha] \to [\beta])\rangle\} \ .$$

Eventually, the algorithm arrives at a working set

$$\mathcal{S}' = \{\langle T_{11}(\vec{\alpha}), \alpha\rangle, \langle T_{12}(\vec{\alpha}), \beta\rangle, \langle T_{210}(\vec{\alpha}), \alpha\rangle, \langle T_{220}(\vec{\alpha}), \beta\rangle\} \ ,$$

where $T_1 = T_{11} \to T_{12}$ and $T_2 = [T_{210}] \to [T_{220}]$. At this point it is clear that, for example, T_1 must be specialized to a projection, and the argument onto which T_1 projects must then be equal to α. Put another way, α and β must both belong to the sequence $\vec{\alpha}$. Assume, therefore, that $\vec{\alpha} = \alpha, \beta, \vec{\alpha}'$. Then the working set becomes

$$\mathcal{S}' = \{\langle T_{11}(\alpha, \beta, \vec{\alpha}'), \alpha\rangle, \langle T_{12}(\alpha, \beta, \vec{\alpha}'), \beta\rangle, \langle T_{210}(\alpha, \beta, \vec{\alpha}'), \alpha\rangle, \langle T_{220}(\alpha, \beta, \vec{\alpha}'), \beta\rangle\} \ .$$

Then it becomes clear that T_{11} and T_{210} project onto their first argument, and T_{12} and T_{220} project onto their second argument. The type we obtain for *SA* is then

$$\forall.(\forall(\alpha, \beta).(\alpha \to \beta) \to ([\alpha] \to [\beta])) \to ([\beta_1 S] \to [\beta_2 S]) \ ,$$

where $\vec{\beta} = \beta_1, \beta_2, \vec{\beta}'$ and $S = SU((\alpha \to \beta) \to ([\alpha] \to [\beta]), [\beta_1] \to [\beta_2])$. Note that if, on the other hand, we had decided that $\vec{\alpha} = \beta, \alpha, \vec{\alpha}'$, as could happen, depending on the order in which we examine the members of the working set $\mathcal{S}'$, then we would have found that T_{11} and T_{210} project onto their second argument, and T_{12} and T_{220} project onto their first argument. In the end, the type we would have obtained is

$$\forall.(\forall(\alpha, \beta).(\alpha \to \beta) \to ([\alpha] \to [\beta])) \to ([\beta_1 S] \to [\beta_2 S]) \ ,$$

where $\vec{\beta} = \beta_1, \beta_2, \vec{\beta}'$ and $S = SU((\alpha \to \beta) \to ([\alpha] \to [\beta]), [\beta_1] \to [\beta_2])$. This type is, of course, identical to the previous type for *SA*.

Upon reflection, it is clear why this result must be true in general. In choosing a specialization of a sequence $\vec{\alpha}$ and a particular projection to which to specialize a constructor variable T, what matters is not the particular instantiations of $\vec{\alpha}$ and T, but the resulting value of $T_(\vec{\alpha})$. Instantiations are, in fact, chosen so that, if $\mathcal{S}$ contains the pair $\langle T(\vec{\alpha}), \beta \rangle$, then upon instantiation, $T(\vec{\alpha})$ is equal to β. Thus, the resulting type cannot help but be the same, irrespective of the particular instantiations chosen for $\vec{\alpha}$ and T.

Figure 5.6 presents the multi-argument version of our link-time specialization algorithm. As we outlined in our example, the algorithm is identical to the previous formulation until the working set $\mathcal{S}$ contains only pairs of the form $\langle T(\vec{\alpha}), \beta \rangle$ or $\langle \alpha, \beta \rangle$. At this point, in the former case, if β has been explicitly named as part of $\vec{\alpha}$, then the projection to which to map T is clear; otherwise, we update $\vec{\alpha}$ to explicitly include β. The projection to which to map T will then be apparent upon the next iteration. Pairs of the form $\langle \alpha, \beta \rangle$ are not handled any differently.

In summary, then, we see that the core third-order inference algorithm scales up to accommodate multi-argument type constructors without any difficulty at all.

5.11 Solving SSU's: Multi-Variable Case

We consider in this section an analysis, similar to the one in Section 5.5, in which we attempt to determine characteristics of the solution of an SSU, this time in the multi-variable case. The analysis is more complex in the multi-variable case, as, unlike in the single-variable case, the solvability of an SSU $SU(T_1(\vec{\alpha}), T_2(\vec{\beta}))$ is no longer simply a matter of one constructor being larger than the other, in the sense of the ordering $\leq$ on constructors, established in Section 5.5. Suppose, for example, that we specialize T_1 and T_2 as follows:

$$\begin{aligned} T_1 &= \lambda(\tau_1, \tau_2, \tau_3, \vec{\tau}).\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3) \\ T_2 &= \lambda(\tau_1, \tau_2, \tau_3, \vec{\tau}).(\tau_1 \rightarrow \tau_2) \rightarrow \tau_3 \ . \end{aligned}$$

Then neither $T_1 \leq T_2$ nor $T_2 \leq T_1$ holds according to the definition given in Section 5.5—for any substitution σ, neither $T_1(\vec{\alpha}\sigma) = T_2(\vec{\beta})$ nor $T_1(\vec{\alpha}) = T_2(\vec{\beta}\sigma)$ can be true. Rather, to obtain equality, some substitution must be performed within each constructor application. In this particular case, instantiating T_1 and T_2 as outlined, and attempting to equate the instantiations, gives rise to the following equality:

$$\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_3) \ = \ (\beta_1 \rightarrow \beta_2) \rightarrow \beta_3 \ .$$

ALGORITHM multi-match
INPUT: set $\mathcal{S}$ of pairs $\langle \tau_P, \tau_A \rangle$ of parameter and argument type expressions
OUTPUT: substitution on constructors in each τ_P and variables in each τ_A that unifies $\mathcal{S}$.

multi-match =
  if $\mathcal{S}$ is empty then []
  if each $\langle \tau_P, \tau_A \rangle$ in $\mathcal{S}$ is of the form $\langle T(\vec{\alpha}), \beta \rangle$ or $\langle \alpha, \beta \rangle$ then
    if $\exists$ a pair $p = \langle T(\alpha_1, \ldots, \alpha_n, \vec{\alpha}), \beta \rangle \in \mathcal{S}$
      if $\beta = \alpha_i$ for some i then
        multi-match$((\mathcal{S} \setminus \{p\})[\lambda(\tau_1, \ldots, \tau_n, \vec{\tau}).\tau_i/T]) \circ [\lambda(\tau_1, \ldots, \tau_n, \vec{\tau}).\tau_i/T]$
      else
        multi-match$(\mathcal{S}[(\beta, \vec{\alpha}')/\vec{\alpha}]) \circ [(\beta, \vec{\alpha}')/\vec{\alpha}]$
    else
      let $\langle \alpha, \beta \rangle \in \mathcal{S}$ in multi-match$((\mathcal{S} \setminus \{\langle \alpha, \beta \rangle\})[\alpha/\beta]) \circ [\alpha/\beta]$
  else if $\exists$ a pair $p = \langle f(\tau_{P_1}, \ldots, \tau_{Pn}), g(\tau_{A_1}, \ldots, \tau_{Am}) \rangle \in \mathcal{S}$ then
      if $f \neq g$ or $m \neq n$ then error
      else
        multi-match$((\mathcal{S} \setminus \{p\}) \cup \{\langle \tau_{P1}, \tau_{A1} \rangle, \ldots, \langle \tau_{Pn}, \tau_{An} \rangle\})$
  else if $\exists$ a pair $p = \langle T(\vec{\alpha}), f(\tau_{A1}, \ldots, \tau_{An}) \rangle \in \mathcal{S}$ then
    let
      $T_1, \ldots, T_n$ be fresh constructor variables
      $\sigma \;=\; [\lambda\vec{\tau}.f(T_1(\vec{\tau}), \ldots T_n(\vec{\tau}))/T]$
    in
      multi-match$((\mathcal{S} \setminus \{p\})\sigma \cup \{\langle T_1(\vec{\alpha}), \tau_{A1} \rangle, \ldots, \langle T_n(\vec{\alpha}), \tau_{An} \rangle\}) \circ \sigma$
  else if $\exists$ a pair $p = \langle f(\tau_{P1}, \ldots, \tau_{Pn}), \beta \rangle \in \mathcal{S}$ then
    let
      $\beta_1, \ldots, \beta_n$ be fresh variables
      $\sigma \;=\; [f(\beta_1, \ldots, \beta_n))/\beta]$
    in
      multi-match$((\mathcal{S} \setminus \{p\})\sigma \cup \{\langle \tau_{P1}, \beta_1 \rangle, \ldots, \langle \tau_{Pn}, \beta_n \rangle\}) \circ \sigma$
  else if $\exists$ a pair $\langle \alpha, f(\tau_1, \ldots \tau_n) \rangle \in \mathcal{S}$ then error

Figure 5.6: Link-time matching algorithm, multi-argument constructor variable version.

The most general solution of this equation is

$$\sigma = [\beta_1 \to \beta_2/\alpha_1, \alpha_2 \to \alpha_3/\beta_3] \ ,$$

which, as we can see, includes replacements on *both* sides of the equality. Thus, neither $T_1 \leq T_2$ nor $T_2 \leq T_1$ is true. On the other hand, the semiunification instance $T_1(\vec{\alpha}) \leq T_2(\vec{\beta})$, with this instantiation of T_1 and T_2, is solvable. The particular inequality we obtain is

$$\alpha_1 \to (\alpha_2 \to \alpha_3) \ \leq \ (\beta_1 \to \beta_2) \to \beta_3 \ .$$

In this inequality, there is a redex-I between $\alpha_2 \to \beta_a$ and β_3. Reducing it yields the inequality

$$\alpha_1 \to (\alpha_2 \to \alpha_3) \ \leq \ (\beta_1 \to \beta_2) \to (\gamma_1 \to \gamma_2) \ ,$$

where γ_1 and γ_2 are fresh variables, and the inequality is now solved.

Unfortunately, there appears to be no simple characterization of the set of specializations of the constructor variables T_1 and T_2 for which the SSU $SU(T_1(\vec{\alpha}), T_2(\vec{\beta}))$ has a solution, other than the obvious criterion that calls for simple unifiability between $T_1(\vec{\alpha})$ and $T_2(\vec{\beta})$[8] [9].

On the other hand, our characterization of the result of applying an SSU in the single-variable case has an analogue in the multivariable case:

Proposition 5.1 *Let T_1 and T_2 be type constructor variables. Suppose that, for specific constructors $\mathcal{T}_1$ and $\mathcal{T}_2$, replacing T_1 and T_2 respectively, the SSU $SU(T_1(\vec{\alpha}), T_2(\vec{\beta}))$ has a solution σ. Then $\mathcal{T}_2(\vec{\beta})\sigma = T_3(\vec{\gamma})$, where $\mathcal{T}_3 = \mathcal{T}_1 \bigvee \mathcal{T}_2$ according to the ordering $\leq$ established for constructors in Section 5.5, and the renaming of $\vec{\beta}$ to $\vec{\gamma}$ may be omitted if $\mathcal{T}_2 \geq \mathcal{T}_1$.*

Proof Suppose σ solves $\mathcal{T}_1(\vec{\alpha}) \leq \mathcal{T}_2(\vec{\beta})$. Then there is a substitution σ_1 such that $\mathcal{T}_1(\vec{\alpha})\sigma\sigma_1 \leq \mathcal{T}_2(\vec{\beta})\sigma$, i.e., $\mathcal{T}_1(\vec{\alpha}\sigma\sigma_1) \leq \mathcal{T}_2(\vec{\beta}\sigma)$. Let $\mathcal{T}_3$ be the constructor such that $\mathcal{T}_3(\vec{\gamma}) = \mathcal{T}_2(\vec{\beta}\sigma)$. Then by definition, $\mathcal{T}_3 \geq \mathcal{T}_2$ and $\mathcal{T}_3 \geq \mathcal{T}_1$. Suppose now that, for some constructor $\mathcal{T}_4$, we have $\mathcal{T}_4 \geq \mathcal{T}_2$ and $\mathcal{T}_4 \geq \mathcal{T}_1$. Then there are substitutions σ' and σ'_1 such that $\mathcal{T}_1(\vec{\alpha}\sigma'_1) = T_4(\vec{\delta})$ and $\mathcal{T}_2(\vec{\beta}\sigma') = T_4(\vec{\delta})$. Since σ' and σ'_1 have disjoint domains, we may rewrite the first of these equations

[8] The left-hand side and right-hand side of a single inequality, if they have no variables in common, are semiunifiable if and only if they are unifiable.

[9] The single-argument criterion that either $T_1 \leq T_2$ or $T_2 \leq T_1$ is also nothing more than a statement of unifiability between $T_1(\alpha)$ and $T_2(\beta)$. However, the criterion that either $T_1 \leq T_2$ or $T_2 \leq T_1$ provides an elegant characterization of just what it takes for $T_1(\alpha)$ and $T_2(\beta)$ to be unifiable; such a characterization, unfortunately, does not seem to exist in the multi-argument case.

as $\mathcal{T}_1(\vec{\alpha}\sigma\sigma'_1) = \mathcal{T}_4(\vec{\delta})$. Hence σ semiunifies $\mathcal{T}_1(\vec{\alpha})$ and $\mathcal{T}_2(\vec{\beta})$. Since the output of an SSU is a most general semiunifier, there is a substitution σ'' such that for every variable ϵ in $\vec{\alpha}$ or $\vec{\beta}$, $\epsilon\sigma' = \epsilon\sigma\sigma''$. Therefore, $\mathcal{T}_4(\vec{\delta}) = \mathcal{T}_3(\vec{\gamma}\sigma'')$, from which we obtain $\mathcal{T}_3 \leq \mathcal{T}_4$. □

If we prefer, therefore, we may use least upper bound notation to denote SSU's in the final type we return. Practically speaking, however, we expect that in the majority of cases, constructor specialization by matching against constant type expressions will allow most SSU's to be solved at compile-time.

5.12 The Full Algorithm

Pseudocode for the full third-order inference algorithm, including support for multi-argument type constructors and constructor variable specialization against constant expressions, appears in Figures 5.7 and 5.8.

The mainline inference algorithm in Figure 5.7 differs from its predecessor primarily in the call to the function inst (Figure 5.8), and then in applying the resulting substitutions to simplify the returned type. The constructor variable instantiation algorithm in Figure 5.8 is a multivariate version of our previous constructor variable instantiation procedure; the refinement part of the algorithm is now a specialized version of Østvold's anti-unification algorithm, modified to accommodate constructor variables appropriately.

5.13 Completeness

Earlier in this chapter, we noted several sources of incompleteness in the third-order inference procedure. Since then, we have addressed these sources of incompleteness, with varying degrees of success. In this section, we examine in detail the notion of completeness, as it applies to our algorithm.

To put the issue of completeness in perspective, we must recall the extent to which the original algorithm KW was shown to be complete. Of course, given the need for annotations (which, if employed, could then only give a parameter one of several possible types), KW has no hope of a full completeness property. However, Kfoury and Wells [27] show that KW has a property they call *weak principality*—that, *given the programmer-supplied annotations on parameters*, the result type they compute for a source term is most general—all other result types are merely substitution instances of theirs (again, given the annotations).

ALGORITHM FullTypeOf
INPUT: λ-term E, type environment Δ, $FV(E) \subseteq \mathrm{dom}(\Delta)$
OUTPUT: 3rd-order rank 2 type for E

$E_l \leftarrow E$, with all abstractions labelled m or p
$\Gamma \leftarrow USUP(E_l, \Delta)$
if Γ is not solvable then
  abort(error)
end if
$\sigma \leftarrow$ USUPRedexProcedure(Γ)
$\Gamma' \leftarrow \Gamma\sigma$
for each β_x occurring only on left-hand sides in Γ (other than in $\delta_{\lambda^p x.E} = \beta_x \to \delta_E$)
  $arity \leftarrow \max_{(\beta_x \le \mu_i) \in \Gamma'}(\mathrm{arity}(\mu_i))$
  let $\vec{\alpha_x}$ be a sequence of fresh type variables, $T_{x,0}, \ldots, T_{x,arity}$ be fresh constructor variables
  $\Gamma' \leftarrow \Gamma' \cup \langle \beta_x = T_{x,0}(\vec{\alpha_x}) \to \cdots \to T_{x,arity}(\vec{\alpha_x}), \vec{\alpha} \rangle$,
    where $\vec{\alpha}$ is the set of unknowns associated with the inequality $\delta_{\lambda^p x.E} = \beta_x \to \delta_E$ in Γ
end for
$\sigma' \leftarrow$ USUPRedexProcedure(Γ')
$\Gamma'' \leftarrow \Gamma'\sigma'$
$\mathcal{S} \leftarrow \{SU(\tau, \mu) \mid \tau \le \mu \text{ is unsolved in } \Gamma''\}$
$\sigma_{SU} \leftarrow \mathrm{inst}(\mathcal{S})$ (see Figure 5.8)
while $\exists$ unsolved $SU(\tau, \mu) \in \mathcal{S}\sigma_{SU}$ that is now completely solvable with solution $\sigma_{\tau \le \mu}$
  $\sigma_{SU} \leftarrow \sigma_{\tau \le \mu} \circ \sigma_{SU}$
end while
while $\exists$ unsolved $SU(\tau, \mu) \in \mathcal{S}\sigma_{SU}$ that is not completely solvable
  $\sigma_{SU} \leftarrow SU(\tau, \mu) \circ \sigma_{SU}$
end while
final substitution is $\sigma_{SU} \circ \sigma' \circ \sigma$
return $\forall.Q(E_l, \delta_{E_l}\sigma\sigma'\sigma_{SU})$, quantified over all remaining type constructors in the expression

Figure 5.7: The full third-order inference procedure.

ALGORITHM inst
INPUT: Set $\mathcal{S}$ of SSU's
OUTPUT: List of most specific constructor variable instantiations that avoid functor mismatches

inst($\mathcal{S}$) = let σ = getSubst($\mathcal{S}$) in (if $\mathcal{S} = \mathcal{S}\sigma$ then nil else σ :: inst($\mathcal{S}\sigma$))

getSubst($\mathcal{S}$) =
  let
    $P = \{\langle \Pi\tau, \Pi\mu\rangle \mid (SU(\tau,\mu) \in \mathcal{S}$ or $SU(\mu,\tau) \in \mathcal{S}),\ \Pi$ is a path, $\Pi\tau$ and $\Pi\mu$ exist$\}$
    $T(\vec{\alpha}) \in \{T(\vec{\alpha}) \mid \exists\tau.\langle T(\vec{\alpha}), \tau\rangle \in P,\ \tau$ is constant$\}$
    $\vec{\tau} = \{$constant $\tau \mid \langle T(\vec{\alpha}), \tau\rangle \in P\}$
    $\tau_0 =$ mscai($\vec{\tau}$) (Note: standard mscai)
  in $[\lambda(FTV(\tau_0))$.refine$(T, \tau_0, \mathcal{S})/T]$
  except when no such T, $\vec{\tau}$ exist $\Rightarrow$ []

refine(T, τ, $\mathcal{S}$) =
  fst(refine-rec($\tau, \{\Pi\mu \mid \exists\tau.SU(\tau,\mu) \in \mathcal{S}, \Pi\tau = T\} \cup \{\Pi\tau \mid \exists\mu.SU(\tau,\mu) \in \mathcal{S}, \Pi\mu = T\}$, []))

refine-rec($f(\tau_1, \ldots, \tau_n)$, $\mathcal{S}$, σ) =
  if $\exists f'(\tau'_1, \ldots, \tau'_m) \in \mathcal{S}$ such that $f' \neq f$ or $m \neq n$ then
    if $\exists\beta.\beta\sigma = f(\tau_1, \ldots, \tau_n)$ then $\langle\beta, \sigma\rangle$
    else let β be a fresh variable in $\langle\beta, [f(\tau_1, \ldots \tau_n)/\beta] \circ \sigma\rangle$
  else let
      $\langle\tau'_1, \sigma_1\rangle$ = refine-rec(τ_1, $\{f_1\mu \mid \mu \in \mathcal{S}\}$, σ)
      $\langle\tau'_2, \sigma_2\rangle$ = refine-rec(τ_2, $\{f_2\mu \mid \mu \in \mathcal{S}\}$, σ_1)
      $\ldots$
      $\langle\tau'_n, \sigma_n\rangle$ = refine-rec(τ_n, $\{f_n\mu \mid \mu \in \mathcal{S}\}$, σ_{n-1})
    in $\langle f(\tau'_1, \ldots, \tau'_n), \sigma_n\rangle$
    where $f_1, \ldots, f_n$ are the path selection operators for the functor f
refine-rec(x, _, σ) = $\langle x, \sigma\rangle$

Figure 5.8: Constructor variable instantiation—multivariate version.

We can make the same claim about the third-order procedure—that given any annotations that either the procedure itself or the programmer supplies, the computed result type will be at least as general as any other. Without third-order annotations, the result follows from the completeness of the USUP translation, and the principality of the output from the USUP redex procedure, both of which were established in Chapter 4. In the presence of third-order annotations, because we showed in Section 5.6.1 that redex reduction commutes with constructor variable instantiation, establishing a completeness result amounts to showing that no more redexes are reduced by keeping the constructor variables uninstantiated than by instantiating the constructor variables immediately. Indeed, any redex found when constructor variables are uninstantiated will arise after instantiation, unless constructor variables can be instantiated to constructors that ignore their arguments. In such cases, however, we showed in Section 5.9 that the effect of reducing a redex is erased along with the redex itself; hence reducing a redex that may be erased is not a threat to completeness. We therefore have the following result:

Theorem 5.7 (Weak principality) *Given a set of third-order type annotations for polymorphic variables in a source term E, let τ represent the final result type computed by the third-order inference procedure for E. Let σ_T represent a substitution that instantiates constructor variables to known constructors. Let τ' be a derivable final result type for $E\sigma_T$. Then there is a substitution σ' such that $\tau' = \tau\sigma_T\sigma'$.*

Thus, modulo our chosen annotations, we compute most general result types. However, in contrast with algorithm KW, the annotations we furnish express an entire range of types in a single notation. In particular, an annotation such as $\forall\vec{\alpha}.T_1(\vec{\alpha}) \to T_2(\vec{\alpha}) \to T_3(\vec{\alpha})$ covers all closed binary function types. There are, therefore, only two sources of incompleteness in our procedure:

- types with lower arities than our chosen annotations;
- types containing free type variables that are quantified globally.

We discussed the first source of incompleteness in Section 5.4; the second we leave to future investigation, but seems to be a straightforward application of unknowns. Modulo these two limitations, however, the full third-order inference procedure produces every System F type that can be derived for a given term.

5.14 Chapter Summary

Our goal in this chapter was to address what is perhaps the most fundamental limitation of Algorithm KW—its inability, in the absence of assistance from the programmer, to produce types of practical value for a given term. Because the exact type of a given function f's argument must be known and supplied as a parameter annotation before f can be typed, the potential for both code reuse and separate compilation is limited.

By observing certain regularities between the programmer-supplied annotations for a parameter x of f and the ensuing result types of f, we were led to abstract the action of supplying annotations by using placeholders to stand for type constructors, in a manner reminiscent of the third-order λ-calculus. Abstracting specific annotations into constructor variable applications made explicit the relationship between a particular annotation and its associated result type. In this way, a single third-order annotation encompasses infinitely many ordinary types, and facilitates code reuse in ways not possible with simple System F annotations.

The core algorithm was shown to be sound, but admits several sources of incompleteness. However, by examining the potential incompletenesses of the algorithm in detail, we discovered that there are really only two: the possibility of lower-arity arguments than our chosen annotations allow, and the possibility of free type variables within annotations, that are quantified globally. As we have noted, the former can be eased by programmer intervention, telling the type inference engine that a lower-arity assumption is needed. Alternatively, since there are only finitely many lower-arity assumptions, a type inference system could, in principle, return all types associated with lower-arity assumptions, along with the default.

The latter restriction, though we have not investigated it in detail here, appears to be addressable via the use of unknowns.

In summary, the inference algorithm we presented in this chapter computes large sets of practically-useful types for a given term, and supports specialization of separately-computed function and argument types for "link-time" matching. We have thus met our first and third goals, as set out in Chapter 1. As the second and fourth goals were addressed in Chapters 3 and 4, we have therefore produced a type inference procedure that meets all of our objectives.

Chapter 6

Conclusions and Future Work

This thesis has explored the problem of unannotated type inference for the rank 2 fragment of System F. The existing rank 2 type inference algorithm KW, due to Kfoury and Wells, relies on user annotations in order to output types that match real-world uses of the source program. In the absence of any help from the programmer, Algorithm KW is essentially a decision procedure, answering the equivalent of "yes" or "no" to the question of typability for a given source term.

Even in the presence of user annotations, however, even though the output from KW is of practical use, it excludes many other types that could have been given to the source term. Moreover, the algorithm itself, which is a reduction from typability to the acyclic semiunification problem, suffers from technical limitations that make error-reporting and piecewise analysis difficult, and make the reduction itself difficult to understand.

It has been the goal of this thesis to address these limitations of Algorithm KW to create a rank 2 inference algorithm for System F that is ready to be adopted into existing functional language implementations.

6.1 Contributions

We outline in this section the specific contributions we have made in this thesis. In Chapter 1, we outlined the following four goals, whose fulfillment would constitute a successful formulation of a rank 2 type inference algorithm for System F:

- returning a large (i.e., infinite) set of practically useful types for a given expression, rather than a single useless type;

- facilitating the production of clear and descriptive error messages whenever the algorithm fails, by providing a clear correspondence between subexpressions of the input program and components of the semiunification instance to which it maps;
- supporting link-time specializations of the types it produces, to facilitate well-typed separate compilation and linking; and
- being easy to state and understand.

In Chapter 3, we devised a graph-theoretic framework in which to reason about instances of SUP. After formulating a graph-theoretic equivalent of the ASUP acyclicity condition, we showed how the condition could be generalized, resulting in R-acyclicity, a condition that is of a more clearly cyclic flavour than the ASUP criteria. The corresponding SUP subproblem, which we call R-ASUP, is decidable and strictly contains ASUP. Moreover, by using R-ASUP rather than ASUP as the target of the translation, we were able to translate the rank 2 typability problem to SUP instance that are quadratically smaller than the previous ASUP translations. Moreover, the translation is more intuitive, as it omits variables and inequalities that were present for no other reason than to satisfy the ASUP acyclicity constraint. In this way, we made progress towards our fourth objective, and made further progress possible in Chapter 4.

Our goal in Chapter 4 was to build on our results from Chapter 3 and construct a syntax-directed translation from typability to a set of SUP-like constraints. It soon became apparent that even SUP in its unrestricted form is not well equipped to function as the target of such a translation. In particular, no element of SUP (either variable or functor) can satisfactorily model the behaviour of monomorphic variables that outscope polymorphic variables—for these, we devised a new class of identifier that was sometimes variable and sometimes constant. Yet, even when this new kind of identifier functioned as a constant, it could still be the target of substitutions, in a particular, well-defined way. We called such identifiers unknowns, and the related extension to SUP we called USUP.

In the remainder of Chapter 4, we gave an extension of the SUP redex procedure for use with USUP and showed it sound and complete (when it terminates) with respect to the definition of a solution of a USUP instance. We showed that the USUP redex procedure terminates on the USUP analogue of R-acyclic instances. Finally, we gave a translation from labelled rank 2 System F terms to USUP, and showed the translation to be sound and complete. Finally, we showed that the USUP redex procedure terminates on all targets of the USUP translation, thereby establishing an effective, syntax-directed type inference procedure for the rank 2 fragment of System F. The

procedure is not only concise, but also compact to state, and its syntax-directed nature provides a one-to-one map between source fragments and subsets of the associated USUP instance, thereby allowing characteristics of the USUP instance to be translated back into statements about the source program (in particular, error messages). Being syntax directed, the translation also seems to be quite easy to follow. In this way, the introduction of USUP fulfills our second and fourth objectives.

In Chapter 5 we introduced a third-order notation for types, in which there is a class of variables that can stand for type *constructors*, as opposed to ordinary types. This notation could then be used to create type expressions that generalize over a host of System F types that would not otherwise have been compatible with each other. We demonstrated that by the use of constructor variables, we obtained not only templates for entire sets of System F types, but also for entire sets of USUP instances in which to perform redex reductions. In a sense, by introducing constructor variables into USUP instances, we were performing infinitely many type derivations at once, using the constructor variables to abstract away the differences among them. As a result, at the end of the redex reduction process, we produce infinitely many types for a given expression. Those aspects of the redex reduction that rely on the particulars of the constructors being employed (and therefore cannot be carried out before constructor variable instantiation), we package as symbolic semiunifiers (SSU's), representing a promise to perform the remaining redex reductions when it becomes possible.

Once the core algorithm was established and proved sound, we introduced extensions to it. We first extended the algorithm to handle constant types like *Int* and *Bool*. As a side effect of this introduction, we found that it became possible to specialize certain constructors (as any constructor that is to be semiunified against a constant expression can only have a small range of values) and thereby simplify, or even eliminate, many SSU's.

The introduction of constant types, however, gave rise to the possibility that a constructor variable could be specialized to a constructor that ignores its argument; then any redex that depends on the presence of that argument would not actually exist upon constructor variable instantiation, and an unneeded redex reduction would have taken place. As a result, redex reduction under potentially constant constructor variables poses a potential threat to completeness. As we discovered, however, if an instantiation of a constructor variable to a constant erases a redex, it also erases the effect of reducing the redex. Hence, the potential threat to completeness is, in fact, a red herring.

The possibility that a constructor could ignore its argument was a potential completeness threat, not only for nullary constructors, but also for multi-argument constructors. Once this threat was eliminated, the way became clear for the adoption of multi-argument constructor variables into the inference procedure. The major question surrounding such an adoption was how many arguments to give a particular constructor variables? Perhaps surprisingly, the answer turned out to be infinity—because of the regularity with which a constructor's arguments are reduced, the parameter lists may simply be viewed as lazy sequences, from which we take as many elements as we need.

We thus obtained an inference procedure capable of handling constructor variables of arbitrary arity, thereby covering almost the entire set of types a given source term could have. In fact, as we observed in Section 5.13, the only remaining sources of incompleteness in our algorithm are as follows:

- types with lower arities than our chosen annotations;
- types containing free type variables that are quantified globally.

The first source of incompleteness can be addressed via user annotations or by returning a set of third-order types. The second is left to future work, but can most likely be addressed via the use of unknowns.

In summary, the major contributions of this thesis are as follows:

- the introduction of a graph-theoretic framework in which to study SUP instances;
- the notion of R-acyclicity as a termination criterion;
- the introduction of the class of identifiers we call "unknowns" for modelling "semi-constant" behaviour;
- a sound and complete syntax-directed translation from rank 2 typability instances to a set of SUP-like constraints;
- a notation for types in the rank 2 fragment of System F that abstracts over type constructors and thereby facilitates unassisted type inference, and provides previously-impossible abstractions over sets of seemingly unrelated types;
- a third-order inference algorithm for the rank 2 fragment of System F, which, while not incompatible with user annotations, is able, unassisted to compute sets of types for a given

term that, modulo the two restrictions outlined above, encompass the entire set of derivable types for the term;

- a parameter-argument matching procedure that facilitates separate compilation and type inference for a function from its eventual argument.

Opportunities for further development of this work appear in the next section.

6.2 Future Work

There are several ways in which the work outlined in this thesis could be extended. We outline a selection of these here.

6.2.1 The Remaining Incompleteness

As noted above, our system does not allow for polymorphic type annotations that contain free type variables that are quantified globally. Global quantification is an indication of rank 1 polymorphism; hence it seems most appropriate that such type variables be treated as monomorphic. It would perhaps be most convenient, then, to model such variables as unknowns.

Under such a model, however, the same set of unknowns would be part of several parameter type assumptions, in violation of some of the preconditions of results established in Section 5.9. It may be possible, therefore, that effects of redex reductions on these common unknowns could linger, even if the redexes themselves were erased by particular constructor variable instantiations. Thus, even though introducing globally quantified type assumption variables into the model addresses a current source of incompleteness, our treatment of such variables via unknowns may still leave some lingering incompleteness. Whether the results in Section 5.9 could be extended in some way to handle such unknown occurrences, so that we can eliminate this potential lingering incompleteness, requires further study.

6.2.2 Extending R-Acyclicity

As R-ASUP is a strict superset of ASUP (and similarly for R-AUSUP and AUSUP), it is natural to ask whether a larger class of problems in the application domain (in our case type inference for System F) become solvable, by virtue of having images under the SUP translation that lie within R-ASUP, but not ASUP. For example, there might be strictly rank 3-typable terms whose typability question can be phrased as an R-ASUP instance, but not as an ASUP instance.

The answer, as far as we can tell, however, is negative for R-ASUP. On the other hand, since the full SUP is undecidable, there exist infinitely many decidable subsets of SUP between R-ASUP and the full SUP. Similarly, even though rank 3 type inference for System F is undecidable, there are surely sets of terms not typable at ranks below 3, for which we can formulate typing procedures. Whether a correspondence can be found between such hypothetical subsets of SUP and sublanguages of System F, is a question for further research.

6.2.3 Mixed-Rank Type Inference

In reality, not all source terms are best served by a rank 2 type. Consider, for example, the identity function, $\lambda x.x$. The third-order, rank 2 type our procedure would produce for this term is

$$\forall T.\forall\vec{\beta}.(\forall\vec{\alpha}.T(\vec{\alpha})) \to T(\vec{\beta}) \ .$$

However, the identity function also possesses the rank 1 type

$$\forall\alpha.\alpha \to \alpha \ ,$$

which is the type that users of ML or Haskell would expect for such a function. Moreover, if we allow impredicative instantiation on this type, then we can specialize the variable α to the rank 1 type $\forall\vec{\alpha}.T(\vec{\alpha})$, and obtain the type

$$(\forall\vec{\alpha}.T(\vec{\alpha})) \to (\forall\vec{\alpha}.T(\vec{\alpha})) \ ,$$

which is equivalent to the rank 2 type that the third-order procedure produces for the function. It appears, then, that even in a rank 2 setting, the most appropriate type for the identity function is, in fact, the rank 1 type.

More generally, this example suggests that whenever a term possesses both a rank 1 type and a rank 2 type, we should prefer the rank 1 type. An important further research effort, then, would be to adapt the third-order inference algorithm to output rank 1 types when possible, and rank 2 types only when necessary. One simple adaptation of the algorithm is as follows:

for a source term E:
  attempt to find a rank 1 type for E
  if success then
    return rank 1 type for E
  else
    return rank 2 type for E

However aside from the inelegance of simply trying increasingly large ranks until the term becomes typable, this naive approach is not sufficiently general. Consider, for example, the following function:

$$E \;=\; \lambda x.\lambda y.(xx, y3)\ .$$

The function E is rank 2-typable, but due to the occurrence of the subterm xx, whose typability requires that the parameter x be polymorphic, E cannot be rank 1-typable. On the other hand, there is no reason why the parameter y, whose only occurrence is as a function applied to an integer, must be polymorphic. What we would like, therefore, is a type for the function E, in which the parameter x is polymorphic and the parameter y is monomorphic. Producing such a type is not simply a matter of iteratively increasing the rank of the inference procedure until the term becomes typable. Rather, we need a means of determining, as part of the inference process, which parameters should be polymorphic, and which should be monomorphic.

In particular, we would like criteria for determining, within a USUP translation of a source term, whether a particular variable occurs in contexts requiring polymorphism, or whether monomorphism will suffice. It seems most appropriate to carry out such an analysis on the USUP instance, rather than on the source term, as the USUP instance makes the relationships among the types in the source term explicit.

For a given parameter x in a term E, the USUP inequalities associated with the occurrences of x in E are of the form

$$\begin{array}{rcl} \beta_x & \leq & \delta_{x_1} \\ & \cdots & \\ \beta_x & \leq & \delta_{x_k}\ . \end{array}$$

If x is to be treated as monomorphic, then these become

$$\begin{array}{rcl} \beta_x & = & \delta_{x_1} \\ & \cdots & \\ \beta_x & = & \delta_{x_k}\ . \end{array}$$

This system is solvable if and only if $\delta_{x_1}, \ldots, \delta_{x_k}$ are unifiable. A criterion for deciding whether a term may be treated as of rank 1, then, may be to reduce the USUP instance as far as possible, and then to determine whether the images of $\delta_{x_1}, \ldots, \delta_{x_k}$ after reduction are collectively unifiable.

The question of which parameters must be polymorphic and which can be monomorphic may not always be easy to answer. Consider, for example, the following term:

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

In this term, the parameter x and y cannot both be monomorphic—the occurrence of the subterm xy demands that x and y have, respectively, the types $\tau_1 \to \tau_2$ and τ_1 for some τ_1, τ_2, and the occurrence of the subterm yx demands the reverse. On the other hand, it is not necessary for both x and y to be polymorphic; polymorphism is only really needed for one of x and y—though by symmetry, which of these we choose to make polymorphic does not matter.

Note that changing the inequalities $\beta_x \leq \delta_{x_i}$ into equalities $\beta_x = \delta_{x_i}$ is equivalent to simply treating β_x throughout the USUP instance as an unknown.

6.2.4 Incorporation into a Real Programming Language

As we have shown in this thesis, we have a type inference procedure for the rank 2 fragment of System F that is capable of performing fully automated inference on programs which, in existing systems, require programmer assistance; moreover, the types that our procedure produces generalize over a wider range of potential argument types than in existing systems.

Before this algorithm can be adopted into an existing mainstream programming language like ML or Haskell, several practical issues must be resolved, related to compatibility with existing features of these languages.

One such example is recursive function definitions. We have not considered in this work the possibility of performing type inference on a recursively-defined function. While we do not anticipate any real difficulty here—the solution may simply be a matter of adopting a fixed-point operator of type $\forall\alpha.(\alpha \to \alpha) \to \alpha$—investigation is needed to determine whether any troublesome issues arise.

We might ask whether, in addition to functions, data constructors might also be permitted to be of rank 2. Consider the following example ML `datatype` declaration:

```
datatype 'a t = T of 'a
```

In this simple example, the data constructor `T` has type `'a -> 'a t`, i.e., $\forall\alpha.\alpha \to t(\alpha)$. In general, data constructors have rank 1 polymorphic types. However, if we permit impredicative instantiation of their type variables, then we can obtain higher rank 2 types. For example, if, in

the type of `T`, we instantiate α to $\forall\beta.\beta \to \beta$, then `T` obtains the rank 2 type

$$(\forall\beta.\beta \to \beta) \to t(\forall\beta.\beta \to \beta) \ .$$

The functor t now has a polymorphic argument. Thus, we now have the ability to create, for example, container types for polymorphic data, just as we have seen that we could embed polymorphic types inside the product and list constructors.

On the other hand, if we have a `datatype` declaration like

```
datatype 'a t = T of ('a -> 'a) -> 'a
```

then impredicative instantiation once again gives the functor t a polymorphic argument; this time, however, the type is shorthand for a rank 3 type, as can be seen from the type of the data constructor `T` (`(('a -> 'a) -> 'a) -> 'a t`), which would then be of rank 4.

We see, then, that impredicative instantiation can only be permitted on those type constructors for which such instantiation would not result in higher-rank types. Fortunately, this condition should be verifiable by inspecting the form of the `datatype` declaration.

The type system of the Haskell programming language is considerably more complex than the mostly straightforward Hindley-Milner type system of ML. Specific innovations whose interactions with third-order inference must be studied include

- higher-kinded constructors;
- type classes;
- programmer-assisted higher-rank type inference.

The third innovation mentioned above is actually a language extension that is available in the Glasgow Haskell Compiler, and is not part of standard Haskell.

The fact that Haskell already recognizes higher-kinded variables (the classes `Monad` and `Functor`, for example, operate over type *constructors*, rather than ordinary types) suggests that much of the necessary groundwork for adopting third-order inference into an implementation may already have been laid. Note, however, that Haskell does not do any higher-kinded type inference. Being extensions of System F, such problems (i.e., unassisted type inference at higher orders) are undecidable in their full generality.

Type classes provide a uniform interface to the typically ad hoc practice of overloading, in which a given (function) name may have multiple types, with a separate implementation for

each. Type classes might be viewed as providing a constrained form of quantification for type variables, in which instances may only be chosen from types that are known to belong to a given class. As such, intuition suggests that, at least on a superficial level, third-order rank 2 types and type classes are able to coexist. We could conceive, however, of more complex interactions between type classes and third-order, rank 2 types. For example, since they indicate a limited form of quantification, perhaps class annotations might also be permitted to be of rank 2, as in the following type, for example:

```
Num b => (Num a => a -> T(a)) -> T(b)
```

The degree to which inference for types of this form is possible needs further study.

Finally, the Glasgow Haskell Compiler already supports type inference at arbitrary ranks (as described in Peyton Jones and Shields [47]), as long as the needed type annotations are provided by the user. Their inference procedure, of course, does not include constructor variables. We might wonder, therefore, whether we might be able to integrate our work with theirs. By adding constructor variables to their arbitrary rank type annotations, perhaps we might be able to achieve useful abstractions over types, with associated code reuse opportunities, at higher ranks as well as at rank 2.

6.2.5 Fourth-Order Types?

The success of third-order types at not only performing unassisted type inference on rank 2-typable terms, but also providing abstraction opportunities not present within the type system of core System F, raises the question of whether similar benefit might be obtained by considering an even higher order notation for types.

Under the so-called fourth-order λ-calculus (or System F_4 [13]), variables may stand, not only for types and type constructors, but also for functions that operate on type constructors. Thus, fourth-order types offer abstraction over "constructor constructors". It is not clear what, if any benefit might be obtained from such constructions. On the one hand, the expressive power of such a facility is surely immense; on the other hand, as we add orders of complexity to the system, we begin to reach a point where programmers cease to be able to contemplate just what such types could mean, and how to generate them.

Hofstadter [20, pp. 687–688] describes a modified game of chess in which, for his turn, a player may make a standard move, or change the rules of the game. The ways in which the rules may be changed are governed by meta-rules. A third option available to the player would

then be to change the meta-rules, the permissible changes being governed by meta-meta-rules. Though we may consider adding as many levels of complexity to the game as we like, Hofstadter suggests that even at the level of meta-meta-rules (which govern how we may modify meta-rules), the possibilities become difficult to imagine. In a similar way, the adoption of constructor constructors into a type inference procedure (i.e., a fourth-order system) might give rise to a system of comparable complexity to a chess game with meta-meta-rules, and may therefore lie beyond the limits within which most humans would be willing (or perhaps even able) to operate.

It is not entirely clear whether fourth-order types are the point where such a limit occurs; however, intuition suggests that such types may indeed lie beyond casual comprehension, and therefore would not likely be readily adopted by a programming community at large.

Bibliography

[1] Matthias Baaz. Note on the existence of most general unifiers. *Arithmetic, Proof Theory, and Logical Complexity*, pages 20–29, 1993.

[2] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392, 2001.

[3] Hans-J. Boehm. Partial polymorphic type inference is undecidable. In *Proceedings of the 26th Symposium on Foundations of Computer Science*, pages 339–345. IEEE, October 1985.

[4] Pascal Brisset. Avoiding dynamic type checking in a polymorphic logic programming language. In *Symposium on Logic Programming*, page 674, 1994.

[5] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv fur Mathematische Logik und Grundlagen-Forschung*, 19:139–156, 1978.

[6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[7] Jochen Dörre and William C. Rounds. On subsumption and semiunification in feature algebras. *Journal of Symbolic Computation*, 13:441–461, 1992.

[8] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 281–292. Association for Computing Machinery, ACM Press, 2004.

[9] Martin Emms and Hans Leiß. Extending the type checker of SML by polymorphic recursion: A correctness proof. Technical Report CIS-Bericht-96-101, Universität München Centrum für Informations- und Sprachverarbeitung, 1997.

[10] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensidive flow analysis using instantiation constraints. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 253–263. ACM Press, 2000.

[11] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *ESOP '88: 2nd European Symposium on Programming*, number 300 in Lecture Notes in Computer Science, pages 94–114. Springer-Verlag, 1988.

[12] The GHC Team. The glorious Glasgow Haskell Compilation System user's guide, version 6.6.1 [Online]. Available at <http://www.haskell.org/ghc/docs/latest/users_guide.pdf> [Accessed June 30, 2007], 2007.

[13] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[14] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.

[15] Friedrich Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers, New Brunswick, New Jersey, May 1989.

[16] Fritz Henglein. Semi-unification. Technical Report (SETL Newsletter) 223, New York University, April 1988.

[17] Fritz Henglein. Type inference and semi-unification. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 184–197. Association for Computing Machinery, ACM Press, 1988.

[18] Fritz Henglein. Fast left-linear semi-unification. In *ICCI'90: Proceedings of the international conference on Advances in computing and information*, pages 82–91, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[19] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[20] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, New York, 20th-anniversary edition, 1999.

[21] Trevor Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, MIT, August 1995.

[22] Mark P. Jones. From Hindley-Milner types to first-class structures. In *Proceedings of the Haskell Workshop*, La Jolla, California, June 1995. Yale University Research Report YALEU/DCS/RR-1075.

[23] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *Lecture Notes in Computer Science*, pages 435–454, Berlin/New York, 1988. Springer-Verlag.

[24] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science (LICS)*, jun 1989.

[25] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102:83–101, 1993.

[26] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the Association for Computing Machinery*, 41(2):368–398, March 1994.

[27] 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 *1994 ACM Conference on LISP and Functional Programming*, pages 196–207. ACM Press, 1994.

[28] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *ACM Symposium on Principles of Programming Languages*, pages 161–174, 1999.

[29] Didier Le Botlan and Didier Rémy. ML^{F}: Raising ML to the power of System F. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional Programming*, pages 27–38. ACM Press, August 2003.

[30] Hans Leiß. Decidability of semi-unification in two variables. Technical Report INF-2-ASE-9-89, Siemens, Munich, 1989.

[31] Daniel Leivant. Polymorphic type inference. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98, New York, NY, USA, 1983. ACM Press.

[32] Brad Lushman and Gordon V. Cormack. A *more* direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. Technical Report CS-2006-08, University of Waterloo, 2006.

[33] Brad Lushman and Gordon V. Cormack. A larger decidable semiunification problem. In *Proceedings of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '07)*, pages 143–152. ACM Press, 2007.

[34] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[35] Bernard Meltzer and Donald Michie, editors. *Machine Intelligence 5*. Edinburgh University Press, 1970.

[36] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge Massachusetts, 1997.

[37] John C. Mitchell. Coercion and type inference. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185. ACM Press, 1984.

[38] J. H. Morris. *Lambda-calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[39] A. Mycroft. Polymorphic type schemes and recursive definitions. In Paul and Robinet, editors, *International Symposium on Programming*, pages 217–228, 1984.

[40] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67. ACM Press, 1996.

[41] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, January/March 1999.

[42] Martin Odersky, Chritoph Zenger, and Matthias Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages*, pages 41–53, 2001.

[43] Alberto Oliart and Wayne Snyder. A fast algorithm for uniform semi-unification. In *15th Conference on Automated Deduction (CADE-15)*, number 1421 in LNAI. Springer-Verlag, 1998.

[44] Bjarte M. Østvold. A functional reconstruction of anti-unification. Technical Report DART/04/04, Norwegian Computing Center, April 2004.

[45] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[46] Simon Peyton Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), January 2003.

[47] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. Submitted to *Journal of Functional Programming*, April 2004.

[48] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 252–265. Association for Computing Machinery, ACM Press, January 1998.

[49] Gordon D. Plotkin. A note on inductive generalization. In Meltzer and Michie [35], pages 153–163.

[50] François Pottier. A semi-syntactic soundness proof for HM(X). Technical Report 4150, INRIA, March 2001.

[51] François Pottier and Didier Rémy. The essence of ML type inference (Chapter 10). In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 389–489. The MIT Press, 2005.

[52] Pavel Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.

[53] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In Meltzer and Michie [35], pages 135–151.

[54] John C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 1974.

[55] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[56] Colin Stirling. Decidability of higher-order matching [Online]. Available at <http://homepages.inf.ed.ac.uk/cps/lmcs.ps> [Accessed July 9, 2007], 2006.

[57] Colin Stirling. A game-theoretic approach to deciding higher-order matching. In *33rd International Colloquium on Automata, Languages, and Programming, Part II*, number 4052 in Lecture Notes in Computer Science, pages 348–359. Springer, 2006.

[58] Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, New Haven, Connecticut, 2000.

[59] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.