Introduction to Subtyping

Jakob Rehof

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark
Electronic mail: rehof@diku.dk

September 6, 1996

Abstract

An introduction to the theory of subtyping based on the simply typed lambda calculus. Material covered is roughly the theory as developed in [24], [26], [21], [18], [10], [11], [12], [38], [29], [32], [33].

Contents

Chapter 1

Introduction

This report is an introduction to the theory of subtyping based on the simply typed lambda calculus, as developed by Mitchell, Fuh and Mishra. The report contains both published material and new material. The known material covered is roughly the theory as developed in [24], [26], [21], [18], [10], [11], [12], [41], [38], [33]. Most of the new material is in Chapter 4 which is based on [32] and [33]. Moreover, Chapter 3 contains (especially in Section 3.1) an approach to the material covered there, which is probably not found in the literature.

We assume that the reader has some familiarity with lambda calculus, type systems and type inference problems for the simply typed lambda calculus. For background reading we recommend [3], [17] for the pure lambda calculus; for background material on typed lambda calculi we recommend [25], [4]. We also use elementary theory of partial orders, and here [7] is a sufficient source.

The material presented here focuses on *syntactic and structural aspects* of subtyping systems. This means, for instance, that we say very little about the model theory for subtyping systems, and while we do touch on the complexity theoretic analysis of subtype inference, this is done in a rather summarizing manner. In contrast, the issues we devote most of the space to, could be summarized under the headings:

1. *Basic properties of the subtyping logic*
2. *Basic syntactic properties of the type system*
3. *The structure of subtyping derivations*
4. *The problem of presenting and simplifying subtypings*

The item mentioned last in the list above appears to be of considerable pragmatic interest and enjoys much attention in current research in subtyping. This issue is treated both from a logical and an algorithmic perspective.

The literature on subtyping is becoming vast, and much relevant work could not be touched upon directly in this report. Such work includes [6], [19], [37], [36], [9], [28], [2]; the reader is referred to these works for further reference.

Technical preliminaries

We mention a few pervasive notational conventions from the outset. We assume a denumerable set $\mathcal{V}$ of type variables, ranged over by $\alpha, \beta, \gamma, \ldots$ If P is a set of *base types* (disjoint from $\mathcal{V}$) we let b range over P, and the set of types $T_P(\mathcal{V})$ ranged over by τ, σ is defined by

$$\tau ::= \alpha \mid b \mid \tau \rightarrow \tau'$$

Typical base types are `int`, the type of integers, `real`, the type of reals and `bool`, the type of the booleans. The set of *ground types* T_P is obtained from $T_P(\mathcal{V})$ by removing all types containing type variables. Type variables and base types together are referred to as *atomic types* (or just *atoms*, for short); we let A range over atomic types.

If $f : A \rightarrow B$ is a partial function from A to B then $Dm(f)$ denotes the domain of f. A *type susbtitution* S is a function mapping type variables to types, and a substitution is lifted homomorphically to types. The *support* of S, written $\mathrm{Supp}(S)$, is $\{\alpha \in \mathcal{V} \mid S(\alpha) \neq \alpha\}$, *i.e.*, the set of variables not mapped to themselves by S. If the support of S is finite with $\mathrm{Supp}(S) = \{\alpha_1, \ldots, \alpha_n\}$, then we sometimes write just $S = \{\alpha_1 \mapsto S(\alpha_1), \ldots, \alpha_n \mapsto S(\alpha_n)\}$. If $V \subseteq \mathcal{V}$ then the *restriction of S to V*, denoted $S\mid_V$, is the substitution S' such that $S'(\alpha) = S(\alpha)$ for $\alpha \in V$ and $S'(\alpha) = \alpha$ for $\alpha \notin V$. If $S(\alpha)$ is an *atom* (i.e., a constant in P or a variable) for all α then S is called an *atomic substitution.* If $S(\alpha) \neq S(\beta)$ for all $\alpha, \beta \in V$ with $\alpha \neq \beta$, then S is said to be *injective on V*. If S maps all variables in V to variables and S is injective on V, then S is called a *renaming on V*.

Chapter 2

Simple subtyping

2.1 Subtype logic

In subtyping we think of the the base types in P as being partially ordered. A *partially ordered set*, or *poset* for short, is a pair $(P, \leq_P)$ consisting of a set P and a binary relation $\leq_P$ on P satisfying the following axioms:

1. *Reflexivity :* $\forall x \in P.\ x \leq_P x$
2. *Transitivity :* $\forall x, y, z \in P.\ x \leq_P y, y \leq_P z \Rightarrow x \leq_P z$
3. *Anti-symmetry :* $\forall x, y \in P.\ x \leq_P y, y \leq_P x \Rightarrow x = y$

Let $(P, \leq_P)$ be a finite poset. We often write just P for the poset and $\leq$ for the ordering when the meaning is clear from context. P is intended to define a basic subtype ordering on type constants from which a partial order can be induced, in a systematic manner parametric in P, on the set $T_P(\mathcal{V})$ of types. Intuitively, $\tau \leq \tau'$ shall mean that τ is a *subtype* of τ', indicating that any object of type τ can also be regarded as an object of type τ'. Given P we induce a proof system for judgements of the form

$$C \vdash_P \tau \leq \tau'$$

meaning that, under the subtyping hypotheses in C, we can derive the subtyping relation $\tau \leq \tau'$. The set of hypotheses C is a finite set of formal subtype inequalities of the form $\tau \leq \tau'$; such a set is often called a *coercion set* or a *constraint set*. The proof system for the subtype logic is given in Figure 2.1.

[CONST] $\quad C \vdash_P b \leq b' \quad$ provided $b \leq_P b'$

[REF] $\quad C \vdash_P \tau \leq \tau$

[HYP] $\quad C \cup \{\tau \leq \tau'\} \vdash_P \tau \leq \tau'$

[TRANS] $$\frac{C \vdash_P \tau \leq \tau' \quad C \vdash_P \tau' \leq \tau''}{C \vdash_P \tau \leq \tau''}$$

[ARROW] $$\frac{C \vdash_P \tau_1' \leq \tau_1 \quad C \vdash_P \tau_2 \leq \tau_2'}{C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Figure 2.1: Subtype Logic

It is sometimes convenient to think of P as a coercion set or, equivalently, as a directed graph (a *digraph* for short. A finite digraph $G = (V, E)$ can be regarded as the coercion set $\{x \leq y \mid (x, y) \in E\}$ and, conversely, a coercion set C can be regarded as the digraph $\{(x, y) \mid x \leq y \in C\}$. For a partial order $(P, \leq_P)$ we can think of P in terms of its *covering graph* $Cov(P) = (P, E_P)$ where E_P is the *covering relation* on P wrt. $\leq_P$, i.e., $(x, y) \in E_P$ iff $x <_P y$ and $x \leq_P z <_P y \Rightarrow z = x$; the covering relation specifies the *Hasse diagram* of P, see [7] for more information on the covering relation. An alternative way of thinking of P as a coercion set is as the transitive closure of the graph of the covering relation; perhaps this is most to the point, since $(P, \leq_P)$ is certainly transitively closed. Observe that we have

$$C \cup P \vdash_{\emptyset} \tau \leq \tau' \text{ if and only if } C \vdash_P \tau \leq \tau'$$

where P appears as a coercion set to the left.

Note that rule [ARROW] is *contra-variant* in the domain type and *co-variant* in the range type.

Example 2.1.1 Let $P = \{\texttt{int}, \texttt{real}\}$ with the ordering $\texttt{int} \leq_P \texttt{real}$. Then we have $\vdash_P \texttt{real} \rightarrow \texttt{int} \leq \texttt{int} \rightarrow \texttt{real}$, whereas $\texttt{int} \rightarrow \texttt{int}$ and $\texttt{real} \rightarrow \texttt{real}$ are incomparable on the empty set of hypotheses. □

Note also that the subtype logic has no anti-symmetry axiom. This may appear strange in as much as we are reasoning about partial orders. There are probably at least the following two reasons why an anti-symmetry axiom is usually not part of the subtype logic:

1. It is possible to imagine subtype systems in which the underlying set P is only a pre-order. In fact, the theory developed in [26] does not assume a poset of ground types. A concrete example is *dynamic typing* as developed by Henglein (see [15] with more references) which can be regarded as a subtype system based on a pre-order of types, and, in a sense, the whole point of a dynamic type system is just this, that one does *not* have a poset of types.

2. Including an anti-symmetry axiom would be impractical, since the whole point of including it would be to exploit properties of the equality relation, such as substitutivity of equals for equals. This would mean that one would have to further axiomatize the properties of equality, and the logic would become more complicated. Rather than axiomatizing an equational theory in the logic, it is often sufficient and more convenient to reason about equality at the meta-level. An example of what this might mean occurs in Lemma 2.1.8 below (with Remark 2.1.9.)

3. As is indicated by Exercise 2, adding an anti-symmetry axiom would not give any new *in*equalities.

The arrow type constructor captures most of the interesting properties wrt. subtyping of other usual type constructors, such as pairs, lists etc. (which are co-variant), and in addition to this the contra-variant behaviour of the arrow raises special problems; therefore, we have only included the arrow among the type constructors here. The contra-variant behaviour of the arrow can be motivated by model-theoretic considerations; in particular, a simple set-theoretic view of types with $\leq$ as set-theoretic inclusion and arrow-types as function spaces will verify the subtype logic. We now give a short outline of this view.

Recall (from, e.g., [3] or [22], or see summary in [26]) that a *lambda model* $\mathcal{M} = (D, \cdot, \varepsilon)$ consists of a set, D, with a binary operation $\cdot$ on D, together with a distinguished "choice element" ε and elements $K, S \in D$, satisfying a number of algebraic conditions. Among the laws holding in any lambda model are: $(\varepsilon \cdot d) \cdot e = d \cdot e$ for all $d, e \in D$, and if $d \cdot e = d' \cdot e$ for all $e \in D$, then $\varepsilon \cdot d = \varepsilon \cdot d'$. The operation $\cdot$ can be regarded as an abstract "application" and ε an abstract "apply-function" (in the term model of the lambda calculus the element ε is just the combinator $\lambda x.\lambda y.xy$.) A *type environment* η for a model $\mathcal{M}$ (an $\mathcal{M}$-environment, for short) is a mapping from type variables and type constants to subsets of D, and the meaning of a type expression in a type environment is given by

$$\begin{array}{lcl} [\![b]\!]\eta & = & \eta(b) \\ [\![\alpha]\!]\eta & = & \eta(\alpha) \\ [\![\tau \to \tau']\!]\eta & = & \{d \in D \mid \forall e \in [\![\tau]\!]\eta.\, d \cdot e \in [\![\tau']\!]\eta\} \end{array}$$

For a coercion set C, model $\mathcal{M}$ and $\mathcal{M}$-environment η we write $\eta \models_{\mathcal{M}} C$ iff $[\![\tau]\!]\eta \subseteq [\![\tau']\!]\eta$ for all $\tau \leq \tau' \in C$. We write $\eta \models_{\mathcal{M}} \tau \leq \tau'$ as shorthand for $\eta \models_{\mathcal{M}} \{\tau \leq \tau'\}$. Further, we write $C \models \tau \leq \tau'$ iff it holds for every model $\mathcal{M}$ and every $\mathcal{M}$-environment η that $\eta \models_{\mathcal{M}} C$ implies $\eta \models_{\mathcal{M}} \tau \leq \tau'$. Then it is easy to see that the subtype logic is sound:

Theorem 2.1.2 *(Soundness)*

$$C \vdash_{\emptyset} \tau \leq \tau' \Rightarrow C \models \tau \leq \tau'$$

PROOF The proof is by induction on the derivation of $C \vdash_{\emptyset} \tau \leq \tau'$, and only the case [ARROW] is interesting. So suppose we have $C \vdash_{\emptyset} \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$, because $C \vdash_{\emptyset} \tau_1' \leq \tau_1$ and $C \vdash_{\emptyset} \tau_2 \leq \tau_2'$. By induction, we get $C \models \tau_1' \leq \tau_1$ and $C \models \tau_2 \leq \tau_2'$. Now assume $d \in [\![\tau_1 \to \tau_2]\!]\eta$, for arbitrary $\mathcal{M}$-environment η with $\eta \models_{\mathcal{M}} C$. So we have

$$\forall e \in [\![\tau_1]\!]\eta.\, d \cdot e \in [\![\tau_2]\!]\eta \tag{2.1}$$

Assume $e' \in [\![\tau_1']\!]\eta$, e' arbitrary in D. Then, by $C \models \tau_1' \leq \tau_1$, we have $e' \in [\![\tau_1]\!]\eta$. Then, by (2.1), we have $d \cdot e' \in [\![\tau_2]\!]\eta$, and hence, by $C \models \tau_2 \leq \tau_2'$, we get $d \cdot e \in [\![\tau_2']\!]\eta$. We have shown

$$\forall e' \in [\![\tau_1']\!]\eta.\, d \cdot e' \in [\![\tau_2']\!]\eta$$

hence $d \in [\![\tau_1' \to \tau_2']\!]\eta$, demonstrating the desired inclusion $[\![\tau_1 \to \tau_2]\!]\eta \subseteq [\![\tau_1' \to \tau_2']\!]\eta$. □

It is also possible to prove that the subtype logic is complete. However, this requires the construction of a typed term model for the lambda calculus; completeness of the subtype logic will then follow from completeness of the typed lambda calculus with subtyping and an equational rule. The interested reader is referred to [26] (in particular, Theorem 8 with Corollary 9 and 10) for the details, and we confine ourselves here to recording

Theorem 2.1.3 *(Completeness)*

$$C \models \tau \leq \tau' \Rightarrow C \vdash_{\emptyset} \tau \leq \tau'$$

PROOF See Corollary 10 in [26]. □

We note that it is certainly possible to take a more abstract view of subtyping, where the relation $\tau \leq \tau'$ is interpreted as just asserting that every τ-typed object "can be regarded", *in some sense*, as a τ'-typed object; this might just mean, say, that there exists a reasonable way of *converting* any τ-object into a τ'-object. For instance, asserting $\texttt{int} \leq \texttt{real}$ we might mean, not that the set of integers is a subset of the set of reals, but rather that any integer can be converted into a real by, say, a change of representation. As is suggested by this example, the particular view we take may have some very real operational consequences; this issue is often not addressed in the theory of subtyping. Conversion functions are often referred to as *coercion functions*, and correspondingly, the act of subtyping τ to a supertype τ' is often referred to as *coercing* τ to τ'. What is a "reasonable" way of converting things? Abstract conditions have been given by Reynolds in a categorical setting, see [34].

We now turn to elementary syntactic properties of the subtype logic. We write $\vdash_P \tau \leq \tau'$ for $\emptyset \vdash_P \tau \leq \tau'$, and we write $C \vdash_P C'$ to mean that, for every $\tau \leq \tau' \in C'$ we have $C \vdash_P \tau \leq \tau'$. We say that two coercion sets C and C' are *provably equivalent* (wrt. P), written $C \sim_P C'$, iff $C \vdash_P C'$ and $C' \vdash_P C$.

Derivability in the subtype logic is closed under substitution:

Lemma 2.1.4 *(Substitution) Let $S : \mathcal{V} \to T_P(\mathcal{V})$ be a substitution of types for type variables. Then $C \vdash_P \tau \leq \tau'$ implies $S(C) \vdash_P S(\tau) \leq S(\tau')$.*

PROOF By induction on the derivation of $C \vdash_P \tau \leq \tau'$ (details left for the reader.) □

The subtype logic gives rise to *structural subtyping*, which is so called because only structurally similar types can be compared, if no non-structural hypotheses (of the sort, e.g., $\alpha \leq \alpha \to \alpha$) are used. To make this precise, define the *matching* relation as follows: any two atoms A, A' match; two arrow types $\tau_1 \to \tau_2, \tau_1' \to \tau_2'$ match iff τ_1, τ_1' match and τ_2, τ_2' match. A coercion set C is called *atomic* iff all types in C are atoms. Then we have the following very important property:

Lemma 2.1.5 *(Match Lemma) Let C be atomic. Then $C \vdash_P \tau \leq \tau'$, implies that τ, τ' match.*

PROOF Easy induction on the derivation of $C \vdash_P \tau \leq \tau'$ (details left for the reader.) $\Box$

Lemma 2.1.6 *(Decomposition Lemma) Let C be atomic. Then $C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ if and only if $C \vdash_P \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'\}$.*

PROOF The right-to-left implication is obvious; the left-to-right implication is proven by induction on the length of the derivation of $C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$; we leave the details for the reader, only we do note that in the case where rule [TRANS] is used, one uses the Match lemma, as follows. If $C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ because

$$C \vdash_P \tau_1 \to \tau_2 \leq \tau'' \text{ and } C \vdash_P \tau'' \leq \tau_1' \to \tau_2'$$

then, by the Match Lemma, it must be the case that $\tau'' = \tau_1'' \to \tau_2''$. The result now follows by induction hypothesis. $\Box$

The Match Lemma is extremely useful and will be appealed to numerous times in what is to come. The lemma is often helpful in proofs by structural induction. An example of this occurs in the proof of Lemma 2.1.8 below.

Let C^* denote the reflexive, transitive closure of C. Then we have

Lemma 2.1.7 *If C is atomic and $A \neq A'$, then $C \vdash_P A \leq A'$ if and only if $A \leq A' \in (C \cup P)^*$.*

PROOF If $C \vdash_P A \leq A'$, then $A \leq A'$ can be deduced using only [CONST], [HYP], [REF] and [TRANS]. Hence, $A \leq A' \in (C \cup P)^*$. On the other hand, $C' = \{A \leq A' \mid C \vdash_P A \leq A'\}$ is evidently reflexive-transitive closed, hence $(C \cup P)^* \subseteq C'$. $\Box$

We say that C contains a *proper cycle* iff there are $A_1, \ldots, A_n$, $n \geq 2$ with $A_1 = A_n$ and $A_1 \leq A_2 \in C, \ldots, A_{n-1} \leq A_n \in C$ where we have $A_i \neq A_j$ for some i, j.

Lemma 2.1.8 *Let C be atomic. If $C \cup P$ contains no proper cycle, then*

$$C \vdash_P \tau \leq \tau', C \vdash_P \tau' \leq \tau \Rightarrow \tau = \tau'$$

PROOF We first prove that,

(∗) if $C \vdash_P \tau \leq \tau'$ and $C \vdash_P \tau' \leq \tau$ with $\tau \neq \tau'$, then there are atoms A, A' with $A \neq A'$ such that $C \vdash_P A \leq A'$ and $C \vdash_P A' \leq A$

The claim $(*)$ is shown by induction on τ, as follows. Assume $C \vdash_P \tau \leq \tau'$ and $C \vdash_P \tau' \leq \tau$ with $\tau \neq \tau'$. If $\tau = A$, an atom, then the Match Lemma (first part) shows that $\tau' = A'$, also an atom, and $(*)$ follows in this case. For the inductive step, if $\tau = \tau_1 \to \tau_2$, then the Match Lemma (first part) entails that $\tau' = \tau_1' \to \tau_2'$; the assumptions together with Match Lemma (second part) then imply that $C \vdash_P \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'\}$ and $C \vdash_P \{\tau_1 \leq \tau_1', \tau_2' \leq \tau_2\}$. Since $\tau \neq \tau'$, it must be the case that either $\tau_1 \neq \tau_1'$ or $\tau_2 \neq \tau_2'$. In either case, induction hypothesis yields the desired result. This completes the proof of $(*)$.

We now use $(*)$ to prove the lemma. By $(*)$ it is sufficient to prove that, whenever $C \vdash_P A \leq A'$, $C \vdash_P A' \leq A$ with $A \neq A'$, then $C \cup P$ must contain a proper cycle. But, under the assumption, we have (Lemma 2.1.7) $A \leq A', A' \leq A \in (C \cup P)^*$, so that $(C \cup P)^*$ contains a proper cycle; then $C \cup P$ must also contain a proper cycle. □

Remark 2.1.9 One thing which Lemma 2.1.8 tells us is that we can "implement" anti-symmetry in an atomic coercion set by eliminating cyclic constraints. A concrete example will be given in Section 3.3.1. □

We can now show

Lemma 2.1.10 *For every poset P, the set of ground types T_P as well as the set of types $T_P(\mathcal{V})$ are organized as posets by the definition*

$$\tau \leq \tau' \textit{ iff } \vdash_P \tau \leq \tau'$$

PROOF It is obvious that $(T_P, \leq)$ as well as $(T_P(\mathcal{V}), \leq)$ are both reflexive and transitive. Moreover, anti-symmetry for T_P will follow if we can show that $T_P(\mathcal{V})$ is anti-symmetric, because the ordering on T_P is just the ordering inherited from $T_P(\mathcal{V})$. But, since the empty set of subtype assumptions is certainly both atomic and contains no proper cycle, anti-symmetry of $(T_P(\mathcal{V}), \leq)$ follows directly from Lemma 2.1.8. □

Observe that we have, as a consequence of the Match Lemma that $\tau, \tau' \in T_P$ with $\vdash_P \tau \leq \tau'$ implies τ, τ' match. Hence, in the poset of ground types $(T_P, \leq)$, only matching types are comparable. Note that this relies on the fact that P contains only atomic inequalities.

Remark 2.1.11 Note that, even though we have $\vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \Rightarrow \vdash_P \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'\}$, we do *not* have $\{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'\} \vdash_P \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'\}$. □

We measure the size of types, inequalities and coercion sets by the measure $|\bullet|$, as follows:

$$\begin{array}{lcl} |A| & = & 1,\ A \text{ an atom} \\ |\tau \to \tau'| & = & 1 + |\tau| + |\tau'| \\ |\tau \leq \tau'| & = & |\tau| + |\tau'| \\ |C| & = & \sum_{\tau \leq \tau' \in C} |\tau \leq \tau'| \end{array}$$

Lemma 2.1.5 shows that, from atomic assumptions, only matching inequalities can be deduced. Conversely, as is shown in [26] (Lemma 18), any matching inequality can be deduced from atomic assumptions:

Lemma 2.1.12 *For any two matching type expressions τ, τ' there exists an atomic coercion set $C = \text{ATOMIC}(\tau \leq \tau')$ such that $C \vdash_P \tau \leq \tau'$; moreover, C is minimal in the sense that, for any other atomic set C' with $C' \vdash_P \tau \leq \tau'$, we have $C' \vdash_P C$. The set C can be computed in linear time.*

PROOF Define

$$\begin{array}{lcl} \text{ATOMIC}(A \leq A') & = & \{A \leq A'\},\ \text{where } A, A' \text{ are atoms} \\ \text{ATOMIC}(\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2') & = & \text{ATOMIC}(\tau_1' \leq \tau_1) \cup \text{ATOMIC}(\tau_2 \leq \tau_2') \end{array}$$

Since each recursive step of the algorithm above eliminates two arrows, it is easy to see that $\text{ATOMIC}(\tau \leq \tau')$ can be computed in time linearly bounded by the size of $\tau \leq \tau'$ (in fact, there can be at most $\lfloor \|C\|/2 \rfloor$ decomposition-steps starting from C, when $\|C\|$ measures the number of occurrences of the arrow in C.) Minimality follows easily from Lemma 2.1.5. □

Lemma 2.1.13 *The predicate $C \vdash_P \tau \leq \tau'$ with C atomic is decidable in cubic time.*

PROOF (Compare [26], Lemma 19.)
First let $C' = \text{ATOMIC}(\tau \leq \tau')$. Observe that this is well defined, unless the predicate fails, by Lemma 2.1.5, and (by the same lemma) $C \vdash_P \tau \leq \tau'$ iff $C \vdash_P C'$. Now use a cubic time procedure (see, e.g., [5] pp.558 ff., or [35] pp.475 f.) to build $(C \cup P)^*$ and test $C' \subseteq (C \cup P)^*$, using Lemma 2.1.7. □

The relation $\vdash_P$ is transitive:

Lemma 2.1.14 *If $C \vdash_P C'$ and $C' \vdash_P C''$, then $C \vdash_P C''$.*

PROOF $C \vdash_P C'$ shows that, every time rule [HYP] is used on a hypothesis $\tau_1' \leq \tau_2' \in C'$ in a proof of $C' \vdash_P \tau_1'' \leq \tau_2''$, we can replace that step by a proof of $C \vdash_P \tau_1' \leq \tau_2'$. □

Exercise 1 Prove that $[\![\tau\{\alpha \mapsto \tau'\}]\!]\eta = [\![\tau]\!]\eta\{\alpha \mapsto [\![\tau']\!]\eta\}$ □

Exercise 2 1. Would the definition

$$\tau \leq \tau' \text{ iff } C \vdash_P \tau \leq \tau'$$

organize T_P or $T_P(\mathcal{V})$ as posets, for arbitrary fixed atomic coercion set C?

2. Let Q be a *pre-ordered set* (*i.e.*, $\leq_Q$ is reflexive and transitive.) Let $\sim_Q$ be the equivalence relation on Q given by: $x \sim_Q y$ iff $x \leq_Q y \leq_Q x$. Show that $Q/\sim_Q$ is a poset with the ordering $[x] \leq [y]$ iff $[x] = [y]$ or $x \leq_Q y$ (here $[x]$ denotes the $\sim_Q$-equivalence class represented by x.)

3. Show that $[x] \leq [y]$ if and only if $x \leq_Q y$.

$\Box$

Exercise 3 Prove that, if $\vdash_P A \leq A'$ is provable, then there exists a proof of $\vdash_P A \leq A'$ which does not use rule [TRANS]. $\Box$

Exercise 4 Assume $C \vdash_P \alpha \leq \tau$ and $C \vdash_P \tau \leq \alpha$. Show that, under this assumption, one has

1. $C \vdash_P \tau' \leq \tau'\{\alpha \mapsto \tau\}$ for all τ'
2. $C \vdash_P \tau'\{\alpha \mapsto \tau\} \leq \tau'$ for all τ'
3. $C \vdash_P C\{\alpha \mapsto \tau\}$

$\Box$

2.2 Interpretations

A mapping $\rho : \mathcal{V} \rightarrow T_P$ with P a finite poset, is called a *valuation* in T_P. A valuation is tacitly lifted to an *interpretation* $[\![\bullet]\!]\rho : T_P(\mathcal{V}) \rightarrow T_P$ of $T_P(\mathcal{V})$, as follows:

$$\begin{array}{lcl} [\![b]\!]\rho & = & b \\ [\![\alpha]\!]\rho & = & \rho(\alpha) \\ [\![\tau \rightarrow \tau']\!]\rho & = & [\![\tau]\!]\rho \rightarrow [\![\tau']\!]\rho \end{array}$$

In other words, a valuation is an atomic ground substitution, and accordingly we occasionally write $\rho(\tau)$ for $[\![\tau]\!]\rho$. For ρ a valuation, $\tau, \tau' \in T_P(\mathcal{V})$, we write

$$\rho \models_P \tau \leq \tau'$$

iff $\vdash_P [\![\tau]\!]\rho \leq [\![\tau']\!]\rho$ holds. If C is a coercion set we write

$$\rho \models_P C$$

iff $\rho \models_P \tau \leq \tau'$ for all $\tau \leq \tau'$ in C. In that case we say that ρ *satisfies* C. We say that C is *satisfiable* (over P) iff there exists a valuation satisfying C, i.e., iff

$$\exists \rho : \mathcal{V} \to T_P.\ \rho \models_P C$$

The notion of satisfiability gives rise to interesting computational problems. If P is a poset and C is a constraint set over P, we call the ordered pair (P, C) a *satisfiability problem*; the computational problem is that of deciding, for given (P, C), whether C is satisfiable over P. In more detail, we can distinguish the following two variations:

- (SSI) *Given a finite poset P and a constraint set C over P, is C satisfiable over P?*
- (P-SSI) *Given a constraint set C over a fixed poset P, is C satisfiable over P?*

The name SSI is an abbreviation of "satisfiability of a system of inequalities". The problem SSI is sometimes called the *uniform* satisfiability problem of inequalities; if inequalities are restricted to be atomic, the satisfiablility problem is sometimes called POSAT or *flat* SSI. Satisfiability problems for partial orders are important in the complexity theoretic analysis of subtype inference problems, and they will be discussed again in Section 3.4.

We write

$$C \models_P C'$$

(*C entails C'*) iff every valuation, which satisfies C, also satisfies C', i.e., iff

$$\forall \rho : \mathcal{V} \to T_P.\ \rho \models_P C \Rightarrow \rho \models_P C'$$

We write $C \models_P \tau \leq \tau'$ as shorthand for $C \models_P \{\tau \leq \tau'\}$. Observe that $\emptyset \models_P \tau \leq \tau'$ (shorthand notation: $\models_P \tau \leq \tau'$) is equivalent to the condition that $\rho \models_P \tau \leq \tau'$ for *every* valuation ρ. In this case we say that $\tau \leq \tau'$ is *valid.* A coercion set C over P is called *consistent* with P iff it is conservative over P, i.e., iff

$$\forall b, b' \in P.\ C \vdash_P b \leq b' \Rightarrow \vdash_P b \leq b'$$

i.e., iff every inequality among constants in P derivable from C already holds in P.

Lemma 2.2.1 *(Soundness) If $C \vdash_P \tau \leq \tau'$ then $C \models_P \tau \leq \tau'$.*

PROOF By the substitution property and transitivity for $\vdash_P$ we get that $C \vdash_P \tau \leq \tau'$ and $\rho \models_P C$ together imply $\rho \models_P \tau \leq \tau'$, for every valuation ρ. Now assuming $C \vdash_P \tau \leq \tau'$, the implication

$$\rho \models_P C \Rightarrow \rho \models_P \tau \leq \tau'$$

is immediate, showing $C \models_P \tau \leq \tau'$, as desired. □

Remark 2.2.2 It is easy to see that the "completeness" property $C \models_P \tau \leq \tau' \Rightarrow C \vdash_P \tau \leq \tau'$ does *not* hold in general, even though C is satisfiable (if C is not satisfiable, $C \models_P \tau \leq \tau'$ holds for arbitrary $\tau \leq \tau'$, and completeness clearly fails here.) Take $P = \{b_1, b_2, \perp\}$ ordered by $\perp < b_1$, $\perp < b_2$; then clearly $\{\alpha \leq b_1, \alpha \leq b_2\} \models_P \alpha \leq \perp$, but not $\{\alpha \leq b_1, \alpha \leq b_2\} \vdash_P \alpha \leq \perp$. □

Proposition 2.2.3 *If $P \neq \emptyset$, then $\models_P \tau \leq \tau'$ implies $\vdash_P \tau \leq \tau'$. If C is satisfiable, $\tau, \tau' \in T_P$ and $C \models_P \tau \leq \tau'$, then $\vdash_P \tau \leq \tau'$.*

PROOF The first claim is proven by structural induction on τ. The assumption $\models_P \tau \leq \tau'$ means that we have $\vdash_P [\![\tau]\!]\rho \leq [\![\tau']\!]\rho$ for every valuation ρ. We proceed by cases according to the form of τ:

Case $\tau = \alpha$. We claim that $\tau' = \alpha$, by the assumptions; $\vdash_P \tau \leq \tau'$ obviously follows if this is true. To see that it is true, suppose first that $\tau' = \beta$ with $\beta \neq \alpha$; let $b \in P$ (by $P \neq \emptyset$) and let $\rho(\alpha) = b$, $\rho(\beta) = b \rightarrow b$, then we have $\rho \not\models \alpha \leq \beta$, contradicting the assumption that $\models_P \tau \leq \tau'$, so this case is impossible; suppose next that $\tau' = b'$, a constant; then we find again a valuation ρ not satisfying the inequality by taking $\rho(\alpha) = b \rightarrow b$; finally suppose that $\tau' = \tau_1' \rightarrow \tau_2'$. Then we can choose any constant $b \in P$ for α which will invalidate the assumption that $\models_P \tau \leq \tau'$ as before. No other cases are possible.

Case $\tau = b$, a constant. We argue, in analogy with the previous case (details left for reader), that the assumption $\models_P \tau \leq \tau'$ entails $\tau = b'$ with $b \leq_P b'$, and the claim is true.

Case $\tau = \tau_1 \rightarrow \tau_2$. Here $\models_P \tau \leq \tau'$ implies that $\tau' = \tau_1' \rightarrow \tau_2'$ with $\models_P \tau_1' \leq \tau_1$ and $\models_P \tau_2 \leq \tau_2'$. The claim now follows by induction.

For the second claim, $C \models_P \tau \leq \tau'$ implies $\vdash_P [\![\tau]\!]\rho \leq [\![\tau']\!]\rho$ for some ρ, hence $\vdash_P \tau \leq \tau'$. □

Remark 2.2.4 Observe that we have

$$\{\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'\} \models_P \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'\}$$

since we have the implication (Lemma 2.1.5)

$$\vdash_P \tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2' \Rightarrow \vdash_P \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'\}$$

Compare with Remark 2.1.11. □

Lemma 2.2.5 *If C is satisfiable, then C is consistent.*

PROOF Assume C satisfiable, and suppose $C \vdash_P b \leq b'$. Then (by soundness) we have $C \models_P b \leq b'$. Since C is satisfiable, this entails that for some ρ we have $\rho \models_P b \leq b'$, i.e., $\vdash_P b \leq b'$. □

Remark 2.2.6 Observe that consistency does not, in general, imply satisfiability. As a counter-example, take $P = \{b_1, b_2\}$ discretely ordered. Then the set $C = \{\alpha \leq b_1, \alpha \leq b_2\}$ is certainly consistent; however, C is not satisfiable, because α must be mapped to a lower bound on b_1 and b_2, and no such element exists in P. □

The following important property was shown in both [21] and [38].

Lemma 2.2.7 *(Lincoln-Mitchell, Tiuryn) Let P be a lattice and let C be an atomic set over P. Then C consistent implies C satisfiable.*

PROOF Assume C consistent; we construct a satisfying valuation ρ. For every variable α in C we define the set $\downarrow_C(\alpha)$ by

$$\downarrow_C(\alpha) = \{b \in P \mid C \vdash_P b \leq \alpha\}$$

Then define $\rho(\alpha) = \bigsqcup \downarrow_C(\alpha)$ for each α in C. We claim that ρ satisfies C. To see this, we consider the following cases

1. For $b \leq b' \in C$, we have $C \vdash_P b \leq b'$, hence $\vdash_P b \leq b'$ by consistency, hence any valuation satisfies $b \leq b'$.
2. For $b \leq \alpha \in C$, we have $b \in \downarrow_C(\alpha)$, and so $b \leq \rho(\alpha)$.
3. For $\alpha \leq \beta \in C$, we have $\downarrow_C(\alpha) \subseteq \downarrow_C(\beta)$ and hence $\rho(\alpha) \leq \rho(\beta)$.
4. For $\alpha \leq b \in C$, we have

$$\forall b' \in \downarrow_C(\alpha).\ C \vdash_P b' \leq b$$

by transitivity, and therefore, by consistency, we have $b' \leq b$ for every $b' \in \downarrow_C(\alpha)$, hence b is an upper bound on $\downarrow_C(\alpha)$ and $\rho(\alpha) \leq b$ follows.

□

Note that Lemma 2.1.13, Lemma 2.2.5 and Lemma 2.2.7 together imply that the problem POSAT is polynomial time decidable when P is a lattice. This argument is used in [21] and [38].

Remark 2.2.8 The argument given above for polynomial time decidability of POSAT over a lattice is not optimal; the reduction to consistency checking leads to a decision procedure which relies on transitive closure, which means (at the current state of the art) that the procedure will be at least cubic in the size of the coercion set. However, it turns out that one can attack the satisfiability problem directly and solve the problem in *linear* time, when P is a lattice. See [31] for more details. □

Exercise 5 A valuation ρ in T_P is called an *atomic valuation* iff ρ maps every variable to an element of P.

Prove that an atomic constraint set C is satisfiable if and only if C is satisfied by some atomic valuation. $\square$

Exercise 6 (Pratt-Tiuryn)
Let Q be a poset. A function $f : Q \to Q$ is called a *retraction* of Q onto $f(Q)$ iff f is idempotent ($f \circ f = f$) and monotone. A poset $(Q, \leq_Q)$ is said to *extend* a poset $(P, \leq_P)$ iff $P \subseteq Q$ and $p \leq_P p'$ implies $p \leq_Q p'$ for all $p, p' \in P$. If we are given finite posets P, Q with Q an extension of P, we may ask whether there exists a retraction f from Q onto P; if so, we say that P is a *retract* of Q. We call the ordered pair (P, Q) a *retractability problem.*

Let (P, Q) be a given retractability problem. Then we can define a constraint set $C_{P,Q}$ as follows: to each element $q \in Q \setminus P$ we assign a fresh variable α_q. We define a function $\phi : Q \to P \cup \mathcal{V}$ by $\phi(q) = \alpha_q$, if $q \in Q \setminus P$, and $\phi(q) = q$, if $q \in P$. Then we define $C_{P,Q} = \{\phi(q) \leq \phi(q') \mid q \leq_Q q'\}$.

1. Show that $f : X \to X$ is idempotent if and only if $f(X) = fix(f)$, the set of fixed points of f.

2. Show that, for every retractability problem (P, Q) it is the case that P is a retract of Q if and only if $C_{P,Q}$ is satisfiable over P.

$\square$

Exercise 7 (Pratt-Tiuryn)
Given finite poset P and constraint set C over P we construct pre-ordered sets Q^1_C, Q^2_C and Q_C as follows. The set of elements of Q^1_C is $P \cup \mathrm{Var}(C)$. The ordering of Q^1_C is defined by taking $A \leq A'$ in Q^1_C iff $C \vdash_P A \leq A'$. Now define the equivalence relation $\sim$ on Q^1_C by: $A \sim A'$ iff $A \leq A' \leq A$ holds in Q^1_C. Write $[A]$ for the equivalence class wrt. $\sim$ with representative A. Let $Q^2_C = Q^1_C / \sim$ with the ordering $[A] \leq [A']$ iff $A \leq A'$ in Q^1_C or $[A] = [A']$. If it holds for all $q' \in Q^2_C$ that q' contains at most one element of P, then we call Q^2_C a *pre-extension* of P. If Q^2_C is a pre-extension of P, then we define Q_C as follows. First define the function $\varepsilon : Q^2_C \to P \cup \mathrm{Var}(C)$ by: $\varepsilon(q') = p$, if $p \in P$ and $p \in q'$; $\varepsilon(q') = \alpha$, if no $p \in P$ is in q', and where α is a fixed, chosen variable in q'. So, ε chooses a canonical representative. Note that ε is well defined since Q^2_C is a pre-extension. Now let $Q_C = \varepsilon(Q^2_C)$, under the ordering $\varepsilon(q') \leq \varepsilon(q'')$ iff $q' \leq q''$ in Q^2_C.

1. Show that Q^1_C is a pre-order, Q^2_C is a poset and Q_C is a poset (assuming Q^2_C to be a pre-extension.)

2. Show that ε is a bijection of Q^2_C onto Q_C, and that Q_C extends P.

3. Show that, for every satisfiability problem (P, C) it is the case that C is satisfiable over P if and only if Q_C^2 is a pre-extension of P and P is a retract of Q_C.

$\square$

2.3 Simply typed λ-calculus with subtyping

Given a set of base types P, we assume a set B_P of *typed base constants*, such as numbers, arithmetic functions etc. Elements in B_P have the form $c : \tau$ with $\tau \in T_P$, meaning that τ is the type of the constant c. Note that only ground types in T_P are allowed for constants. The set Λ of λ-terms, ranged over by M, N is defined by

$$M ::= x \mid c \mid \lambda x.M \mid M\ M'$$

where x ranges over a denumerable set of term variables.

Given a finite poset P and a set B_P of base constants, we can define a simply typed λ-calculus with subtyping generated from the *signature* $\Sigma_P = (P, B_P)$; the calculus generated from Σ_P will be referred to as $\lambda_{\leq}(\Sigma_P)$, and it is defined by the typing rules in Figure 2.2. The rules define derivable judgements of the form

$$C, \Gamma \vdash M : \tau$$

where C is a set of subtyping assumptions (a coercion set) and Γ is a set of type assignments $x : \tau$, assigning types to term variables; we assume always that no term variable x occurs more than once in Γ, so Γ can be treated as a finite map from term variables to type expressions. The intended meaning of a judgement is that, under the subtyping assumptions C and the type assumptions Γ, the type τ can be assigned to the term M.

$$[\textsc{var}] \qquad C, \Gamma \cup \{x : \tau\} \vdash x : \tau$$

$$[\textsc{base}] \qquad C, \Gamma \vdash c : \tau \qquad \text{where } c : \tau \in B_P$$

$$[\textsc{abs}] \qquad \frac{C, \Gamma \cup \{x : \tau\} \vdash M : \tau'}{C, \Gamma \vdash \lambda x.M : \tau \to \tau'}$$

$$[\textsc{app}] \qquad \frac{C, \Gamma \vdash M : \tau \to \tau' \quad C, \Gamma \vdash N : \tau}{C, \Gamma \vdash M\ N : \tau'}$$

$$[\textsc{sub}] \qquad \frac{C, \Gamma \vdash M : \tau \quad C \vdash_P \tau \leq \tau'}{C, \Gamma \vdash M : \tau'}$$

Figure 2.2: Simply typed λ-calculus $\lambda_{\leq}(\Sigma_P)$

The *simply typed λ-calculus* (without constants) is obtained from $\lambda_{\leq}(\Sigma_P)$ by removing the rules [SUB] (and [BASE]) and removing all coercion sets from the rules.

If $S : \mathcal{V} \to T_P(\mathcal{V})$ is a type substitution and Γ a set of type assumptions, we let $S(\Gamma) = \{x : S(\tau) \mid x : \tau \in \Gamma\}$. Assumption sets are considered functions from term variables to types, with $\Gamma(x) = \tau$ iff $x : \tau \in \Gamma$ (and it is assumed that every term variable has at most one occurrence in any assumption set.) We write $dom(\Gamma) = \{x \mid x : \tau \in \Gamma \text{ for some } \tau\}$. If X is a set of term variables, $X \subseteq dom(\Gamma)$, then the restriction of Γ to X is $\Gamma\mid_X = \{x : \tau \in \Gamma \mid x \in X\}$.

Typings are closed under type substitutions:

Lemma 2.3.1 *(Type substitution) Let $S : \mathcal{V} \to T_P(\mathcal{V})$ be a substitution. Then $C, \Gamma \vdash M : \tau$ implies $S(C), S(\Gamma) \vdash M : S(\tau)$.*

PROOF By induction on the derivation of $C, \Gamma \vdash M : \tau$. In the case [SUB] we use Lemma 2.1.4. □

Typings depend on type assumptions for all and only those variables that occur free in the term. Let $FV(M)$ denote the set of free variables in M. Then we have

Lemma 2.3.2 *If $C, \Gamma \vdash M : \tau$, then $FV(M) \subseteq dom(\Gamma)$, and $C, \Gamma' \vdash M : \tau$ with $\Gamma' = \Gamma\mid_{FV(M)}$.*

PROOF Easy induction on the derivation of $C, \Gamma \vdash M : \tau$. □

Typings are closed under substitution of terms respecting type assumptions.

Lemma 2.3.3 *(Term substitution) If $C, \Gamma \cup \{x : \tau\} \vdash M : \tau'$ and $C, \Gamma \vdash N : \tau$, then $C, \Gamma \vdash M\{x \mapsto N\} : \tau$*

PROOF By structural induction on M. □

Typings are closed under $\beta\eta$-reduction

Lemma 2.3.4 *(Subject reduction) If $C, \Gamma \vdash M : \tau$ and $M \to^*_{\beta\eta} M'$, then $C, \Gamma \vdash M' : \tau$.*

PROOF (Sketch, a more detailed proof can be found in [26].) One shows that the lemma holds when the reduction consists of a single step of either β or η. One then gets the result by induction in the length of the $\beta\eta$-reduction sequence. To see that the lemma holds for a single reduction, we proceed as follows. First show that the lemma holds when M is a redex, and M' is the result of contracting that redex (use Lemma 2.3.3 in the case of β-reduction; use Lemma 2.3.2 in the case of η-reduction.) Then show that the lemma holds for all terms of the form $C[R]$, where R is the contracted redex and

C is an arbitrary term context. This is easily done by induction on the context, using the first part. □

Note that Lemma 2.3.4 implies that the rule

$$[\eta] \quad \frac{C, \Gamma \vdash \lambda x.M\,x : \tau}{C, \Gamma \vdash M : \tau} \quad x \text{ not free in } M$$

is *admissible* [1]

It is possible to infer typings in the type system with subtyping of Figure 2.2. In figure 2.3 we give a type inference algorithm, which we call $\mathcal{U}$ [2] and which is due to Mitchell [26]. The algorithm uses a standard unification procedure UNIFY, which finds a most general unifier for a set of pairs of types. In case no unifier exists, UNIFY returns FAIL, and in case this happens the algorithm is tacitly assumed to abort with output FAIL. Thus, the algorithm either produces a judgement $C, \Gamma \vdash M : \tau$, or it outputs FAIL; the latter can only happen in case the unification-step fails.

[1] In the sense of [17], where an inference rule is called admissible if, whenever the premises of the rule are derivable, then so is the conclusion.

[2] The name is meant to be suggestive of the name *unrestricted subtyping*, which will be explained later.

```
$\mathcal{U}[\![x]\!]$       $=$  $\{\alpha \leq \beta\}, \{x : \alpha\} \vdash x : \beta$
                where $\alpha, \beta$ are fresh type variables.

$\mathcal{U}[\![c]\!]$       $=$  $\{\tau \leq \alpha\}, \emptyset \vdash c : \alpha$
                where $c : \tau \in B_P$ and $\alpha$ is a fresh type variable

$\mathcal{U}[\![\lambda x.M]\!]$  $=$  **let** $C, \Gamma \vdash M : \tau' = \mathcal{U}[\![M]\!]$
                **in**
                  **if** $x : \tau \in \Gamma$ for some $\tau$
                  **then** $C \cup \{\tau \to \tau' \leq \alpha\}, \Gamma \setminus \{x : \tau\} \vdash \lambda x.M : \alpha$
                  **else** $C \cup \{\beta \to \tau' \leq \alpha\}, \Gamma \vdash \lambda x.M : \alpha$
                  where $\alpha, \beta$ are fresh type variables
                **end**

$\mathcal{U}[\![M\ N]\!]$   $=$  **let** $C_1, \Gamma_1 \vdash M : \sigma = \mathcal{U}[\![M]\!]$
                **and** $C_2, \Gamma_2 \vdash N : \tau = \mathcal{U}[\![N]\!]$
                     with type variables disjoint from $\mathcal{U}[\![M]\!]$

                **and** $S =$ UNIFY $(\{\tau' = \tau'' \mid x : \tau' \in \Gamma_1, x : \tau'' \in \Gamma_2\} \cup \{\sigma = \tau \to \alpha\})$
                     where $\alpha$ is a fresh type variable
                **in**
                  $S(C_1) \cup S(C_2) \cup \{S(\alpha) \leq \beta\}, S(\Gamma_1) \cup S(\Gamma_2) \vdash M\ N : \beta$
                  where $\beta$ is a fresh type variable
                **end**
```

Figure 2.3: Type Inference Algorithm (Mitchell)

Instance relations

A typing judgement $C', \Gamma' \vdash M : \tau'$ is said to be an *instance* of $C, \Gamma \vdash M : \tau$ iff there exists a type substitution S such that the following conditions are satisfied:

1. $C' \vdash_P S(C)$
2. $C' \vdash_P S(\tau) \leq \tau'$
3. $\forall x \in dom(\Gamma).\ C' \vdash_P \Gamma'(x) \leq S(\Gamma(x))$

If j_1 and j_2 are judgements we write $j_1 \prec j_2$ iff j_2 is an instance of j_1; we write $j_1 \approx j_2$ iff $j_1 \prec j_2$ and $j_2 \prec j_1$. We say that t_2 is an *instance of* t_1 *under substitution* S, written $t_1 \prec_S t_2$, if $t_1 \prec t_2$ with the three conditions above holding with substitution S.

In some of the litterature on subtyping a weaker notion of instance is used. We shall call this relation the *weak instance* relation, and it is defined in the following way: a typing judgement $C', \Gamma' \vdash M : \tau'$ is said to be a *weak instance* of $C, \Gamma \vdash M : \tau$ iff there exists a type substitution S such that the following conditions are satisfied:

1. $C' \vdash_P S(C)$
2. $S(\tau) = \tau'$
3. $S(\Gamma) \subseteq \Gamma'$

Clearly, the weak instance relation is contained in the instance relation; so "weakness" here means that the weaker relation relates *fewer* judgements. We shall discuss the relation between the two instance relations more carefully later in this section. The weak instance relation was used by Fuh and Mishra in [12] and by Mitchell in [26]. The instance relation was introduced by Fuh and Mishra in [11] and was called *lazy instance* there; this relation is also discussed by Hoang and Mitchell in [18].

We have the following strengthening of Lemma 2.3.1:

Lemma 2.3.5 *Typings are closed under the instance relation, i.e., if* $C, \Gamma \vdash M : \tau$ *and* $C, \Gamma \vdash M : \tau \prec C', \Gamma' \vdash M : \tau'$*, then* $C', \Gamma' \vdash M : \tau'$

PROOF The proof is by induction on the derivation of $C, \Gamma \vdash M : \tau$, using Lemma 2.1.4. More details can be found in [12]. $\square$

It follows that typings are also closed under weak instance, since the weak instance relation is contained in the instance relation.

The following two theorems establish soundness and completeness of algorithm $\mathcal{U}$. Together, the two theorems establish correctness of the algorithm. We begin with soundness, which is easy. Soundness means that whenever the algorithm outputs a typing judgement, then that judgement is derivable in the proof system of Figure 2.2.

Theorem 2.3.6 *(Soundness of $\mathcal{U}$) If $C, \Gamma \vdash M : \tau = \mathcal{U}[\![M]\!]$, then $C, \Gamma \vdash M : \tau$ is derivable.*

PROOF Easy structural induction on M. □

Completeness of an algorithm generally means that the algorithm does not miss any solutions to a problem. In our case we can prove the quite strong completeness property that, for every typable term M, every derivable typing for M is a weak instance of $\mathcal{U}[\![M]\!]$. Mitchell [26] gives a nice proof of this, and we include a proof here, since it is fundamental and technically slightly involved.

Theorem 2.3.7 *(Completeness of $\mathcal{U}$) If $C, \Gamma \vdash M : \tau$, then $\mathcal{U}[\![M]\!] \neq$ FAIL and $C, \Gamma \vdash M : \tau$ is a weak instance of $\mathcal{U}[\![M]\!]$.*

PROOF The proof is by structural induction on M.

Case $\boxed{M = x}$ Here $C, \Gamma \vdash x : \tau$ must hold because $x : \tau' \in \Gamma$, for some τ', with $C \vdash_P \tau' \leq \tau$, by [VAR] and [SUB]. Let $\{\alpha \leq \beta\}, \{x : \alpha\} \vdash x : \beta = \mathcal{U}[\![x]\!]$. Define the substitution T by $T = \{\alpha \mapsto \tau', \beta \mapsto \tau\}$. Then

1. $C \vdash_P T(\alpha \leq \beta)$
2. $T(\{x : \alpha\}) \subseteq \Gamma$
3. $T(\beta) = \tau$

which shows that $C, \Gamma \vdash x : \tau$ is a weak instance of $\mathcal{U}[\![x]\!]$.

Case $\boxed{M = c}$ Here $C, \Gamma \vdash c : \tau$ must hold because $(c : \tau') \in B_P$ and $C \vdash_P \tau' \leq \tau$, by [BASE] and [SUB]. Let $\{\tau' \leq \alpha\}, \emptyset \vdash c : \alpha = \mathcal{U}[\![c]\!]$, and define substitution T by $T(\alpha) = \tau$. Then it is easy to see that $C, \Gamma \vdash c : \tau$ is a weak instance of $\mathcal{U}[\![c]\!]$ under substitution T.

Case $\boxed{M = \lambda x.M'}$ Here $C, \Gamma \vdash \lambda x.M' : \tau$, which must hold because

$$C, \Gamma \cup \{x : \tau'\} \vdash M' : \tau'' \text{ with } C \vdash_P \tau' \to \tau'' \leq \tau$$

by [ABS] and [SUB]. Let

$$C_1, \Gamma_1 \vdash M' : \sigma = \mathcal{U}[\![M']\!]$$

Then, by induction, there exists a substitution S such that

$$C \vdash_P S(C_1), S(\Gamma_1) \subseteq \Gamma \cup \{x : \tau'\}, S(\sigma) = \tau''$$

Let α, β be the fresh variables introduced in $\mathcal{U}[\![\lambda x.M']\!]$. Since α occurs in neither of C_1, Γ_1, σ, we can assume w.l.o.g. that $S(\alpha) = \tau$.

We now consider the two possible subcases, $x : \sigma' \in \Gamma_1$ (for some σ') and $x \notin dom(\Gamma_1)$, corresponding to each of the two branches of the conditional in $\mathcal{U}[\![\lambda x.M']\!]$.

- *Subcase* $x : \sigma' \in \Gamma_1$. Let

  $$C_1 \cup \{\sigma' \to \sigma \leq \alpha\}, \Gamma_1 \setminus \{x : \sigma'\} \vdash \lambda x.M' : \alpha$$

  be the output of $\mathcal{U}[\![\lambda x.M']\!]$ in this case (first branch of conditional.) We wish to show that this judgement has the judgement $C, \Gamma \vdash \lambda x.M' : \tau$ as a weak instance under substitution S, *i.e.*, we must show

  1. $C \vdash_P S(C_1 \cup \{\sigma' \to \sigma \leq \alpha\})$
  2. $S(\Gamma_1 \setminus \{x : \sigma'\}) \subseteq \Gamma$
  3. $S(\alpha) = \tau$

  We already have (3). To see that we have (1), observe that it must be the case that $S(\sigma') = \tau'$, because $S(\Gamma_1) \subseteq \Gamma \cup \{x : \tau'\}$; furthermore, we already have $C \vdash_P S(C_1)$, and moreover

  $$S(\sigma' \to \sigma \leq \alpha) \Leftrightarrow S(\sigma') \to S(\sigma) \leq S(\alpha) \Leftrightarrow \tau' \to \tau'' \leq \tau$$

  which we already know to hold. We have shown (1). Finally, to see that (2) holds, we already have $S(\Gamma_1) \subseteq \Gamma \cup \{x : \tau'\}$, from which (2) follows directly.

- *Subcase* $x \notin dom(\Gamma_1)$. Since β is fresh, we can assume w.l.o.g. that $S(\beta) = \tau'$. Let

  $$C_1 \cup \{\beta \to \sigma \leq \alpha\}, \Gamma_1 \vdash \lambda x.M' : \alpha$$

  be the output of $\mathcal{U}[\![\lambda x.M']\!]$ in this case (second branch of conditional.) To see that $C, \Gamma \vdash \lambda x.M' : \tau$ is a weak instance of this judgement, first observe that we already have $S(\Gamma_1) \subseteq \Gamma \cup \{x : \tau'\}$, from which (by $x \notin dom(\Gamma_1)$) we immediately get $S(\Gamma_1) \subseteq \Gamma$. Also, we have $S(\alpha) = \tau$, and $C \vdash_P S(C_1)$, so we only need to check that $C \vdash_P S(\beta \to \sigma \leq \alpha)$. But

  $$S(\beta \to \sigma \leq \alpha) \Leftrightarrow S(\beta) \to S(\sigma) \leq S(\alpha) \Leftrightarrow \tau' \to \tau'' \leq \tau$$

  which we already know to hold, and we are finished.

Case $\boxed{M = M'N'}$ Here $C, \Gamma \vdash M'N' : \tau$ must follow from typings

$$C, \Gamma \vdash M' : \tau_1 \to \tau_2 \text{ and}$$
$$C, \Gamma \vdash N' : \tau_1 \text{ with}$$
$$C \vdash_P \tau_2 \leq \tau$$

by [APP] and [SUB]. Let

$$C_1, \Gamma_1 \vdash M' : \sigma = \mathcal{U}[\![M']\!]$$

$$C_2, \Gamma_2 \vdash N' : \tau' = \mathcal{U}[\![N']\!]$$

Then, by induction, there exist substitutions S_1, S_2 such that

$$C \vdash_P S_1(C_1), S_1(\Gamma_1) \subseteq \Gamma, S_1(\sigma) = \tau_1 \to \tau_2$$

$$C \vdash_P S_2(C_2), S_2(\Gamma_2) \subseteq \Gamma, S_2(\tau') = \tau_1$$

Since $\mathcal{U}$ chooses typings with disjoint variables for M' and N', we can compose S_1 and S_2. Define the substitution T by

$$T(\gamma) = \begin{cases} S_1(\gamma) & \text{if } \gamma \text{ occurs in } \mathcal{U}[\![M']\!] \\ S_2(\gamma) & \text{if } \gamma \text{ occurs in } \mathcal{U}[\![N']\!] \\ \tau_2 & \text{if } \gamma = \alpha \\ \tau & \text{if } \gamma = \beta \end{cases}$$

where α, β are the fresh type variables introduced in $\mathcal{U}[\![M'N']\!]$. Since T agrees with S_1 on variables in $\mathcal{U}[\![M']\!]$ and with S_2 on variables in $\mathcal{U}[\![N']\!]$, we have

$$C \vdash_P T(C_1), T(\Gamma_1) \subseteq \Gamma, T(\sigma) = \tau_1 \to \tau_2$$

$$C \vdash_P T(C_2), T(\Gamma_2) \subseteq \Gamma, T(\tau') = \tau_1$$

so that both instance-related typings for M' and N' are so related by the single substitution T.

Now, it is easy to see that, whenever $\mathcal{U}[\![M'']\!] = C'', \Gamma'' \vdash M'' : \tau''$, then $dom(\Gamma'') = FV(M'')$; hence, by Lemma 2.3.2 together with $T(\Gamma_1), T(\Gamma_2) \subseteq \Gamma$, T must be a unifier for the set

$$\{\tau' = \tau'' \mid x : \tau' \in \Gamma_1, x : \tau'' \in \Gamma_2\}$$

Moreover, since we have

$$T(\sigma) = \tau_1 \to \tau_2 = T(\tau') \to T(\alpha)$$

T also unifies $\{\sigma = \tau' \to \alpha\}$. In total, we have shown that T is a unifier for exactly that set for which the substitution S in $\mathcal{U}[\![M'N']\!]$ is a most general unifier, and therefore there exists a substitution V such that

$$T = V \circ S$$

It follows that we have

$$C \vdash_P V \circ S(C_1 \cup C_2), V \circ S(\Gamma_1 \cup \Gamma_2) \subseteq \Gamma, V \circ S(\alpha) = \tau_2$$

This shows that the typing $C, \Gamma \vdash M'N' : \tau_2$ is a weak instance of the typing

$$S(C_1) \cup S(C_2), S(\Gamma_1) \cup S(\Gamma_2) \vdash M'N' : S(\alpha)$$

under substitution V. This last mentioned typing judgement is very close to $\mathcal{U}[\![M'N']\!]$, we only need to take care of the fresh variable β and the subtyping $C \vdash_P \tau_2 \leq \tau$.

Since S is a most general unifier for a set in which β does not occur, we must have $S(\beta) = \beta$, and since $T(\beta) = \tau$, we must have $V(\beta) = \tau$, by $T = V \circ S$. Finally, we have $V(S(\alpha)) = T(\alpha) = \tau_2$, and so we get

$$C \vdash_P \tau_2 \leq \tau \Leftrightarrow C \vdash_P V(S(\alpha)) \leq V(\beta)$$

and hence we conclude that $C \vdash_P V(S(\alpha) \leq \beta)$. It follows then, by previous results, that

$$C \vdash_P V(S(C_1) \cup S(C_2) \cup \{S(\alpha) \leq \beta\})$$

In summa, we have established that

1. $C \vdash_P V(S(C_1) \cup S(C_2) \cup \{S(\alpha) \leq \beta\})$
2. $V(S(\Gamma_1) \cup S(\Gamma_2)) \subseteq \Gamma$
3. $V(\beta) = \tau$

which means that $C, \Gamma \vdash M'N' : \tau$ is a weak instance of $\mathcal{U}[\![M'N']\!]$. □

The completeness property proven for $\mathcal{U}$ in Theorem 2.3.7 above is quite strong, since $\mathcal{U}$ will not only find some typing for a term, whenever there is one, but it will invariably find a *most general typing*, from which *every* possible typing of the given term can be systematically generated by instantiation (under the weak instance relation.)

Principal typings

The concept of a most general typing, or, as it is also called, a *principal typing*, is so important that we shall give it a definition. The notion of principality can be defined whenever we have a suitable *instance relation* $\prec_{inst}$ on typing judgements (we have already seen that several instance relations may be possible.) In general, we require instance relations to be (*i*) independent of the terms appearing in judgements and that (*ii*) derivability is closed under the instance relation. Hence we say that a binary relation $\prec_{inst}$ on typing judgements is a *sound instance relation* iff the following hold:

1. $C, \Gamma \vdash M : \tau \prec_{inst} C', \Gamma' \vdash M : \tau'$ if and only if $C, \Gamma \vdash N : \tau \prec_{inst} C', \Gamma' \vdash N : \tau'$ for any term N.
2. If $C, \Gamma \vdash M : \tau \prec_{inst} C', \Gamma' \vdash M : \tau'$ and $C, \Gamma \vdash M : \tau$ is derivable, then $C', \Gamma' \vdash M : \tau'$ is derivable.

This generic definition of sound instance relations was introduced by Hoang and Mitchell in [18]. The intuitive idea is that a judgement is more general than any of its instances. Clearly, the particular instance relations introduced so far (weak instance and the relation $\prec$) are both sound.

Whenever we have an instance relation $\prec_{inst}$ between typing judgements, we say that a judgement $C, \Gamma \vdash M : \tau$ is a *principal typing* for M wrt. $\prec_{inst}$ iff the following holds

1. $C, \Gamma \vdash M : \tau$ is a provable judgement
2. for every other provable judgement $C', \Gamma' \vdash M : \tau'$ we have $C, \Gamma \vdash M : \tau \prec_{inst} C', \Gamma' \vdash M : \tau'$

We say that a type system has the *principality property* iff every typable term has a principal typing. The completeness theorem is a constructive proof that our subtype system has the principality property, both under weak instance and under $\prec$:

Corollary 2.3.8 *(Principality) Any term M which has a typing has a principal typing under both weak instance and $\prec$.*

PROOF Suppose M has a typing. Then $\mathcal{U}$ constructs a principal typing under weak instance for M, by Theorem 2.3.7. Since the weak instance relation is contained in $\prec$, it follows that the typing constructed by $\mathcal{U}$ is also principal under $\prec$. □

We shall discuss further the notion of principality in Section 2.5 below.

2.4 Variations of subtyping

It could be said that, in itself, our subtype system of Figure 2.2 does not give a very interesting classification of terms. If we consider any term M well-typed for which there exists a coercion set C, a type environment Γ and a type τ such that $C, \Gamma \vdash M : \tau$, we find that *every* pure λ-term is well-typed. To see this, let the type assignment $\Gamma(M)$ for a term M be given by $\Gamma(M) = \{x : \alpha \mid x \in FV(M)\}$, and prove by induction on pure term M that $\{\alpha \leq \alpha \to \alpha, \alpha \to \alpha \leq \alpha\}, \Gamma(M) \vdash M : \alpha$ holds for every M. Moreover, suppose we have the base types `int` and `real` in P, ordered by `int` $\leq$ `real`, together with integer and real constants and the successor function `succ` : `int` $\to$ `int`. Then we can type an expression such as `succ 1.5`, since we have $\{$`real` $\leq$ `int`$\}, \emptyset \vdash$ `succ 1.5` : `int`. What this says is the perfectly reasonable *hypothetical* statement that, *if* `real` $\leq$ `int`, *then* `succ 1.5` has type `int`. However, in many cases we are not satisfied with such conditioned assertions, and instead we should want something like `succ 1.5` to fail with a static type error, because we do not allow `real` to be a subtype of `int`. The way to achieve this would be to impose some restrictions on C in derivable statements $C, \Gamma \vdash M : \tau$ for M to be regarded as well-typed by such statements. The restrictions would naturally require C to respect the ordering on base types in P in some way. Two main proposals have been put forward for restricting coercion sets in well typings, and they lead to two forms of subtyping, which we shall call *weak subtyping* and *strong subtyping*, respectively. The system given in Figure 2.2 by itself imposes no restrictions on coercion sets at all, and following Mitchell [26] we call this form of subtyping *unrestricted subtyping.* We then have the following variations of subtyping:

- *Unrestricted subtyping:* no restrictions on C
- *Weak subtyping:* require C to be consistent with P
- *Strong subtyping:* require C to be satisfiable over P

In weak subtyping we should reject `succ1.5` as ill typed, because any statement $C, \emptyset \vdash \texttt{succ1.5} : \tau$ would have to include $\texttt{real} \leq \texttt{int}$ as a consequence of C contradicting (non-conservatively extending) the ordering of P. On the other hand, suppose P is the set

$$P = \{\texttt{real}, \texttt{int}, \texttt{bool}\}$$

ordered by $\texttt{int} \leq \mathit{ptxtreal}$, and suppose M is the term

$$M = \lambda x.\mathit{if}\ x\ \mathit{then}\ \texttt{sqrt}\ x\ \mathit{else}\ \texttt{0}$$

Then the judgement

$$\{\alpha \leq \texttt{bool}, \alpha \leq \texttt{real}\}, \emptyset \vdash_P M : \alpha \rightarrow \texttt{real}$$

is a valid weak typing for M, since the coercion set shown is consistent with P.

In contrast, in strong subtyping, both of the programs, `succ 1.5`, and M mentioned above would be rejected as ill- typed, since both terms can only be typed using coercion sets which are not satisfiable. In fact, it is easy to see that strong subtyping amounts to requiring a term to be typable under *no* subtype hypotheses at all, *i.e.*, we must have $\emptyset, \Gamma \vdash M : \tau$, for some Γ, τ, for M to be well-typed.

Lemma 2.4.1 *If $C, \Gamma \vdash M : \tau$ with C satisfiable, then there exists a substitution $\rho : \mathcal{V} \rightarrow T_P$ such that $\emptyset, \rho(\Gamma) \vdash M : \rho(\tau)$.*

PROOF Since C is satisfiable there exists a valuation ρ such that $\rho \models_P C$, *i.e.*, $\vdash_P \rho(C)$ (regarding ρ as a substitution.) By the assumption that $C, \Gamma \vdash M : \tau$, we get from Lemma 2.3.1 that $\rho(C), \rho(\Gamma) \vdash M : \rho(\tau)$. Since $\vdash_P \rho(C)$, it easily follows that $\emptyset, \rho(\Gamma) \vdash M : \rho(\tau)$. □

In the sequel, we shall occasionally distinguish weak and strong subtyping explicitly, by writing sequents in weak subtyping with $\vdash_W$ and sequents of strong subtyping with $\vdash_S$; a sequent $C, \Gamma \vdash_W M : \tau$, then, means just that $C, \Gamma \vdash M : \tau$ and C is consistent; a sequent $C, \Gamma \vdash_S M : \tau$ means just that $C, \Gamma \vdash M : \tau$ and C is satisfiable. Unrestricted subtyping is discussed extensively by Mitchell in [26]; Fuh and Mishra assume strong subtyping in [12], but weak subtyping in [11]; Lincoln and Mitchell discuss both variations in [21], but Hoang and Mitchell assume strong subtyping in [18]; Tiuryn [38] is relevant for strong subtyping. The terms weak and strong subtyping are our terms, so they are not found in the litterature. In [21] strong subtyping is called *subtyping with fixed subtype ordering*, and weak subtyping is called *subtyping with varying subtype ordering*. The reader be warned that not all papers make the distinction between weak and strong subtyping explicitly.

There is another kind of restriction on constraint sets which is very important and which has been discussed extensively in the litterature. This restriction leads to systems of *atomic subtyping*, and it is defined by the requirement

- *Atomic subtyping:* require C to be atomic

In principle, this restriction can be combined with any of the above mentioned variants, to give atomic, (and otherwise) unrestricted subtyping; atomic, weak subtyping; atomic, strong subtyping. Note, though, that every strongly typable term is automatically typable under atomic assumptions (use Lemma 2.4.1 together with the fact that $\emptyset$ is certainly atomic) and hence, the additional requirement of atomicity does not change the set of typable terms in strong subtyping. In contrast, the requirement of atomicity in combination with weak subtyping would rule out a program such as (`1 2`) as ill typed; it would still allow the program M_0 to type. Even though the requirement of atomicity does not change the set of typable terms in strong subtyping, the requirement is not redundant, because it may well change the typings a term may have, and this, as we shall see in more detail later, can be significant.

Atomic subtyping is very important, for at least the following reasons (several of them given by Mitchell in [26])

1. A large number of practically interesting subtype systems are atomic.

2. As will be seen later, atomic subtyping is a conservative extension of the simply typed calculus; this suggests that extending existing languages, such as *ML*-like languages, with atomic subtyping would allow the extended language to be understood in relatively much the same way as the original language could be understood.

3. Since typings in atomic subtyping have more properties, a number of technical problems become more tractable with atomic subtyping. To mention three important examples (which will be discussed extensively later) we have, first, the existence of normalization theorems for

typings in atomic subtyping; second, the possibility of minimizing the representation of coercion sets; third, known polynomial complexity of various decision problems and inference problems for certain (but not all) interesting atomic subtyping systems.

In our view, the point mentioned last is probably the most important one. The remainder of the present introduction will emphasize atomic subtyping heavily. Observe that the notions of instance and principality still make sense, in the obvious way, when all typings are restricted to having only atomic coercion sets.

Finally, we should mention important restrictions along yet a third dimension. This concerns restricting the poset, P, so as to get more algebraic properties. This usually means requiring P to be (something like) a lattice:

- *Lattice subtyping:* require P to be a lattice.

We have already seen one indication of how the lattice property can be exploited. This occurred in Lemma 2.2.7, which, together with Lemma 2.2.5, implies that weak and strong subtyping are identical for atomic subtyping over a lattice. We shall return to such restrictions again later.

2.5 Presenting typings

This section discusses various issues having to do with the notion of principality for subtyping systems. As we shall see, the addition of subtyping to the simply typed lambda calculus leads to a break-down of a number of pleasing properties of principal typings enjoyed by the simply typed calculus. This underlies some practically significant problems which seem to be inherent for subtyping systems.

In the case of strong subtyping we have seen (Lemma 2.4.1) that any well-typed term M has a typing under the empty set of subtype hypotheses. So, we may ask, why not drop the subtype hypotheses altogether from typing judgements? This raises the more general question

- Why are coercion sets necessary in subtype systems?

First note the obvious fact that $\emptyset, \Gamma \vdash M : \tau$ does not imply that M types without subtyping (*i.e.*, uses of rule [SUB]) at all, because the ordering of the poset T_P may be exploited in an essential way in the typing, when constants are present in M. In fact, it seems that we can do quite a lot of subtyping without any subtype hypotheses. The answer to the question rather lies in the fact that *principality fails* if we drop coercion sets from typing judgements. To see this, consider the following example

Example 2.5.1 (Fuh and Mishra) Suppose we have the subtype ordering P given as `int` $\leq$ `real`, and let *twice* $= \lambda f.\lambda x.f(f\ x)$. With no subtype hypotheses allowed, the best option for a principal type for *twice* would be the type $\tau = (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. However, *twice* also has the type $(\texttt{real} \rightarrow \texttt{int}) \rightarrow (\texttt{real} \rightarrow \texttt{int})$, and there is no substitution S such that $\vdash_P S(\tau) \leq (\texttt{real} \rightarrow \texttt{int}) \rightarrow (\texttt{real} \rightarrow \texttt{int})$. Hence *twice* has no principal typing under $\prec$ (and, hence, none under weak instance) when subtype hypotheses are not allowed. $\square$

Exercise 8 Let P and τ be as in Example 2.5.1. Prove the claim made there that there exists no substitution S such that $\vdash_P S(\tau) \leq (\texttt{real} \rightarrow \texttt{int}) \rightarrow (\texttt{real} \rightarrow \texttt{int})$. $\square$

On the other hand, once we allow coercion sets in typings, we shall be forced to include non-empty coercion sets in principal typings, as the following exercise shows:

Exercise 9 (Hoang and Mitchell) Let $M = f(f\ x)$, let $\prec_{inst}$ be any sound instance relation, and let the set P of base types be empty. Prove that M has no principal typing (wrt. $\prec_{inst}$) in which the coercion set is empty by constructing a typing $C', \Gamma' \vdash M : \tau'$ such that no typing of the form $\emptyset, \Gamma \vdash M : \tau$ has that typing as an instance under $\prec_{inst}$.

(*Hint:* try with the judgement $\{\alpha \leq \beta, \gamma \leq \beta, \gamma \leq \delta\}, \{x : \alpha, f : \beta \rightarrow \gamma\} \vdash f(f\ x) : \delta$ as $C', \Gamma' \vdash M : \tau'$.) $\square$

In the light of Example 2.5.1 and Exercise 9, one could say that it is still open whether there exists a sound notion of instance such that principal typings exist, when coercion sets are removed from the type system altogether (leaving only the base ordering T_P.) It must be said, though, that the relation $\prec$ appears to be quite strong, and one may doubt whether any natural notion of instance exists, which would meet this requirement.

Once we introduce coercion sets into typings in subtyping, a very important issue arises, namely the problem of *the size of principal typings*, and in particular, the size of coercion sets. As an example, note that algorithm $\mathcal{U}$ produces coercion sets of size proportional to *program size*, for *every program* ($\mathcal{U}$ generates new constraints at every syntactic construction.) This means that a typing generated by $\mathcal{U}$ become totally unreadable even for moderately sized program fragments. As we shall see later, the problem of blow-up of coercion sets is even more present in atomic subtyping systems, where standard algorithms may extract coercion sets of size exponential in the size of the term.

It is instructive to approach these problems by contrast with the simply typed lambda calculus. Adding subtyping with coercion sets to the simple typed lambda calculus turns out to brake one of the very pleasing properties of principal types in that calculus. Recall that the simple typed calculus has the principality property under the following notion of instance, $\prec_{simple}$: $\Gamma \vdash M : \tau \prec_{simple} \Gamma' \vdash M : \tau'$ iff there exists a substitution S such that

1. $S(\Gamma) \subseteq \Gamma'$
2. $S(\tau) = \tau'$

As is well known (see, *e.g.*, [4], [16]) there is a sound and complete type inference algorithm for the simply typed lambda calculus, and every typable term has a principal type. It follows from the definition of principality that one has:

Exercise 10 Show that, in the simply typed lambda calculus, one has:

1. Principal typings are unique up to renaming of type variables.
2. For any typable term M, the principal type of M is the syntactically shortest of all the types of M (*i.e.*, no shorter type exists for M.)

$\Box$

The strong uniqueness property of principal typings is preserved in ML (see [23]) which adds `let`-polymorphism to the simply typed lambda calculus, since principal ML-types are unque up to renaming, reordering of quantifiers and elimination of dummy quantifiers. However, when subtyping is added to the simply typed lambda calculus, the matter becomes radically different. Both of the nice properties mentioned in Exercise 10 brake down for the simply typed calculus with subtyping; this is shown in the following exercise.

Exercise 11 (Fuh and Mishra)
Assume atomic subtyping. Show that both of the following typings of the identity function are principal under weak instance:

$$\{\alpha \leq \gamma, \gamma \leq \beta\}, \emptyset \vdash \lambda x.x : \alpha \rightarrow \beta$$

$$\{\alpha \leq \beta\}, \emptyset \vdash \lambda x.x : \alpha \rightarrow \beta$$

Conclude that both are also principal under $\prec$. $\Box$

This shows that principal typings are unique only up to equivalence under mutual instantiation, and, in general, there will be infinitely many representations of the principal typing. Moreover, as will become clear in the course of this report, different presentations of the principal typing of the same term may well differ in very non-trivial ways.

This raises the important problem of finding *optimized representations* of such typings. Ideally, we should like to find some suitable notion of optimality, and we would like to have techniques for choosing an optimal representative in the equivalence class of all principal typings. In particular, we wish for very practical reasons to choose a representation which is as compact as possible. This leads to the study of techniques for *simplification of typings.* There are at least the following three reasons why type simplification in subtyping with coercion sets is important:

1. *Readability.* Large coercion sets make the type information unredable to humans.

2. *Efficiency of subtype inference.* In many applications, the efficiency of subtype inference algorithms will be sensitive to the size of coercion sets. For instance, when subtyping is added to *polymorphic* type systems, inference algorithms will typically copy the constraint set associated with the typing of a polymorphic function, producing one polymorphically instantiated copy for each use of the function.

3. *Explicitness.* Subtype simplification typically makes the information content of a typing more explicit by eliminating redundant subtyping constraints generated by inference algorithms.

The issue raised here is a major reason why one may want to find stronger notions of instance, as in passing from the weak instance relation to the relation $\prec$. The following exercises illustrate this:

Exercise 12 (Fuh and Mishra)
Consider the typing judgements $t_1 = \{\alpha \leq \beta, \beta \leq \alpha\}, \Gamma \vdash M : \alpha \rightarrow \beta$ and $t_2 = \emptyset, \Gamma\{\alpha \mapsto \beta\} \vdash M : \beta \rightarrow \beta$. Show that

1. $t_1 \approx t_2$

2. t_2 is a weak instance of t_1

3. t_1 is not a weak instance of t_2,

□

Exercise 13 (Hoang and Mitchell)
Assume atomic subtyping. Let $M = \lambda f.\lambda x.\lambda y.K(fx)(fy)$ with $K = \lambda x.\lambda y.x$ of type $\alpha \rightarrow \beta \rightarrow \alpha$. Consider the typing judgement t_1 for M:

$$\{\alpha \leq \beta, \gamma \leq \beta, \delta \leq \eta\}, \emptyset \vdash M : (\beta \rightarrow \delta) \rightarrow \alpha \rightarrow \gamma \rightarrow \eta$$

Show that there is a typing t_2 for M with empty coercion set, such that

1. t_1 is not a weak instance of t_2. Can any principal typing, under weak instance, for M have empty coercion set?

2. t_1 is an instance under $\prec$ of t_2. Is t_2 principal under the $\prec$-relation?

□

Exercise 14 (Fuh and Mishra)
Assume atomic subtyping. Let $comp = \lambda f.\lambda g.\lambda x.f(g\,x)$. Show that

1. The typing t_1:

   $$\{\delta \leq \alpha, \eta \leq \gamma, \beta \leq \upsilon\}, \emptyset \vdash \ \mathit{comp} : (\alpha \to \beta) \to (\gamma \to \delta) \to (\eta \to \upsilon)$$

   is principal for *comp* under weak instance

2. The typing t_2:

   $$\emptyset, \emptyset \vdash \ \mathit{comp} : (\alpha \to \beta) \to (\gamma \to \alpha) \to (\gamma \to \beta)$$

   is principal under $\prec$

3. The typing t_2 is not principal under weak instance

□

As illustrated by these excercises, there are quite simple and natural transformations (such as cycle elimination, in Excercise 12) on typings which cannot be performed on a typing *without falling out of the equivalence class* of the typing under weak instance; in contrast, quite non-trivial transformations (as suggested by Excercise 13 and Excercise 14) can be performed under the instance relation $\prec$, while remaining within the equivalence class of the transformed typing. The latter claim will be substantiated many times in the sequel. The demand that a typing transformation should transform a typing t into a typing t' which is still in the equivalence class of t appears to be fundamental; we do not wish to loose generality of a typing in choosing a better representative. Since $\prec$ is a larger relation than weak instance, more typings become instance related (equivalence classes become larger) under $\prec$, and therefore we may sometimes choose a better (more succinct) typing, which is equivalent to a given typing, under $\prec$. This whole issue will be taken up in a more systematic manner in the remainder of this report.

Chapter 3

Atomic subtyping

This chapter is devoted to the study of atomic subtyping systems. A judgement $C, \Gamma \vdash M : \tau$ with C atomic will be called an *atomic subtyping judgement.*

We have already observed, in Section 2.4, that every pure lambda-term has a typing under unrestricted subtyping. However, as mentioned earlier, atomic subtyping is a conservative extension of the simply typed lambda calculus:

Lemma 3.0.2 *(Conservative extension) If $C, \Gamma \vdash M : \tau$ with C atomic and M a pure lambda term (with no constants), then there exists a type substitution S such that $S(\Gamma) \vdash M : S(\tau)$ holds in the simply typed lambda calculus.*

PROOF We can assume that no type constants occur in the typing $C, \Gamma \vdash M : \tau$. As noted in [26], we can then take the substitution S which maps every variable in C to a single (arbitrary) variable. Clearly, we then have $\vdash_{\emptyset} S(C)$, and since Lemma 2.3.1 implies $S(C), S(\Gamma) \vdash M : S(\tau)$, we get $\emptyset, S(\Gamma) \vdash M : S(\tau)$, from which the claim follows. $\Box$

Observe that, if we allow typed constants in terms, then conservativity obviously does not hold. For example, the expression `sqrt2`, with `sqrt` : `real` $\rightarrow$ `real`, would typically not type without subtyping. Lemma 3.0.2 shows that we have the strong normalization property for atomic subtyping:

Lemma 3.0.3 *(Strong normalization) Every term which has a typing in atomic subtyping is β-strongly normalizing.*

PROOF By the strong normalization property for the simply typed lambda calculus (see [4]) together with Lemma 3.0.2, every pure lambda term with an atomic subtyping must be strongly normalizing. Now consider any term M with arbitrary typed constants, $c_1, \ldots, c_n$, and suppose $C, \Gamma \vdash M : \tau$ with C atomic. Then, choosing n fresh term variables $y_1, \ldots, y_n$, we obviously have $C, \Gamma \cup \{y_1 : \tau_{c_1}, \ldots, y_n : \tau_{c_n}\} \vdash M\{c_1, \ldots, c_n \mapsto y_1, \ldots, y_n\} : \tau$, where

$(c_i : \tau_{c_i}) \in TC_P$. Hence, $M\{c_1, \ldots, c_n \mapsto y_1, \ldots, y_n\}$ is typable and it is therefore strongly normalizing. It is easy to see that M is β-strongly normalizing (no δ-rules!) if and only if $M\{c_1, \ldots, c_n \mapsto y_1, \ldots, y_n\}$ is, and the result follows. □

3.1 Proof normalization

In this section we shall introduce *explicit subtyping coercions* and use them to build *subtyping proof terms*. Subtyping coercions are a special kind of constants, which we shall add to our language of terms, and they will be use to explicitly encode subtyping proof steps. The encoding allows us to study the structure of subtyping derivations (proofs). In the present section we use these techniques to prove that, in atomic subtyping, every subtyping proof can be transformed into particularly simple normal forms. The existence of normal form proofs has both practical and theoretical applications. Proof normalization is the process of transforming a typing derivation into normal form; we note from the outset that proof normalization as considered here does not involve β-reduction, but rather it relates to the structure of coercion in a derivation. This is distinct from the notion of proof normalization in typed lambda calculus which is studied under the heading of "Curry-Howard Isomorphism" (see [14] and [30]) where β-reduction is a central notion. A final word on notation: In this section we shall introduce an equational theory on terms with subtyping coercions, and for this equsality we use the symbol "=". In order to avoid confusion with syntactic equality of terms (hitherto denoted =), we shall use the symbol $\equiv$ for syntactic equality.

We now describe a language of *subtyping coercions*. We assume a set of *primitive coercions*, $\uparrow_A^{A'}$, one for each pair (A, A') of atomic types. If $A \leq_P A'$, then $\uparrow_A^{A'}$ is called a *base coercion*. The intention is that $\uparrow_A^{A'}$ represents a coercion function from A to A'. At every type τ there will be a special coercion id_τ, called the identity coercion, which coerces τ to itself. New coercions can be formed by composition ($\circ$) and by an arrow-constructor ($\rightarrow$.) Let κ range over subtyping coercion terms, then the set of terms is generated as follows

$$\kappa ::= \uparrow_A^{A'} \mid id \mid \kappa \circ \kappa' \mid \kappa \rightarrow \kappa'$$

Only a subset of the terms will be well-formed. A well formed coercion is assigned a *coercion signature* of the form $\tau \rightsquigarrow \tau'$, meaning that the term represents a coercion from τ to τ'. The proof system of Figure 3.1 defines provable judgements of the form

$$K \vdash_P \kappa : \tau \rightsquigarrow \tau'$$

Here K is a finite set of primitive coercions, and the intuitive meaning of such a judgement is that, assuming a poset P of base types, κ is a well formed coercion with signature $\tau \leadsto \tau'$ and κ is built from primitive coercions in K.

[CONST] $K \vdash_P {\uparrow}_b^{b'} : b \rightsquigarrow b'$ provided $b \leq_P b'$

[REF] $K \vdash_P id_\tau : \tau \rightsquigarrow \tau$

[HYP] $K \cup \{{\uparrow}_A^{A'}\} \vdash_P {\uparrow}_A^{A'} : A \rightsquigarrow A'$

[TRANS] $$\frac{K \vdash_P \kappa_1 : \tau \rightsquigarrow \tau' \quad K \vdash_P \kappa_2 : \tau' \rightsquigarrow \tau''}{K \vdash_P \kappa_2 \circ \kappa_1 : \tau \rightsquigarrow \tau''}$$

[ARROW] $$\frac{K \vdash_P \kappa_1 : \tau_1' \rightsquigarrow \tau_1 \quad K \vdash_P \kappa_2 : \tau_2 \rightsquigarrow \tau_2'}{K \vdash_P \kappa_1 \to \kappa_2 : (\tau_1 \to \tau_2) \rightsquigarrow (\tau_1' \to \tau_2')}$$

Figure 3.1: Coercion Formation

It is obvious that the system of Figure 3.1 is a mere repetition of the subtype logic in Figure 2.1 with the only difference that in Figure 3.1 we have introduced coercion terms as explicit "witnesses" of the proof steps in the subtype logic. More precisely, if C is an atomic coercion set, we write $C^\dagger = \{\uparrow_A^{A'} \mid A \leq A' \in C\}$. Then

Lemma 3.1.1 *Let C be an atomic coercion set. Then $C \vdash_P \tau \leq \tau'$ if and only if there exists a coercion κ such that $C^\dagger \vdash_P \kappa : \tau \leadsto \tau'$.*

PROOF By induction on the derivations of the judgements. □

Define a proof system for judgements of the form

$$K, \Gamma \vdash_{CO} M : \tau$$

by exchanging the rule [SUB] of Figure 2.2 with the rule [COERCE]:

$$[\text{COERCE}] \quad \frac{K, \Gamma \vdash_{CO} M : \tau \qquad K \vdash_P \kappa : \tau \leadsto \tau'}{K, \Gamma \vdash_{CO} [\kappa]M : \tau'}$$

and letting C range over finite sets of primitive coercions in the remainder of the rules of Figure 2.2. We write applied occurrences of coercions in special brackets, as $[\kappa]M$, to single out the coercions as special terms.

If M is a term possibly with applied coercions, we write $M^\diamond$ to denote the term which arises from M by erasing all coercions in M. If M, M' are two terms we write $M \cong M'$ iff $M^\diamond \equiv (M')^\diamond$. The system $\vdash_{CO}$ is a repetition of the subtype system in which the rule [COERCE] mimicks the rule [SUB]. In fact, it is easy to see that, for every subtyping proof tree Π with conclusion $C, \Gamma \vdash M : \tau$, there exists a unique completion $M^\dagger$ of M which is obtained from Π by a translation in which every single step of subproofs of the form $C \vdash_P \tau \leq \tau'$ with a subsequent step of [SUB] is translated into a coercion κ, where κ gets applied, in $M^\dagger$, to the term corresponding to the subproof to which the rule [SUB] was applied in Π; the coercion κ is likewise generated by mimicking each rule applied in the proof of $C \vdash_P \tau \leq \tau'$ using the corresponding rule for coercion formation, so that $C^\dagger \vdash \kappa : \tau \leadsto \tau'$. Obviously, $M^\dagger \cong M$. Defining $K^\diamond = \{A \leq A' \mid \uparrow_A^{A'} \in K\}$, we have the following correspondence between subtyping and typing with coercion terms, via the correspondence of proofs mentioned above:

Lemma 3.1.2 *Let C be atomic. Then*

1. $C, \Gamma \vdash M : \tau$ *if and only if* $C^\dagger, \Gamma \vdash_{CO} M^\dagger : \tau$.
2. $K^\diamond, \Gamma \vdash M^\diamond : \tau$ *if and only if* $K, \Gamma \vdash_{CO} M : \tau$.

PROOF Both claims are by easy induction on the derivations of the judgements. □

A term M possibly with coercions is called a *completion* (of $M^\diamond$.) Figure 3.2 shows an equational theory on coercions and completions. We write $E \vdash M = M'$ iff M is provably equal to M' in the equational theory of Figure 3.2. Then

Lemma 3.1.3 *If* $E \vdash M = M'$ *, then* $K, \Gamma \vdash_{CO} M : \tau$ *if and only if* $K, \Gamma \vdash_{CO} M' : \tau$ *.*

PROOF Every equation in Figure 3.2 is evidently type-preserving. □

$$
\begin{array}{lllll}
[\mathrm{C1}] & \kappa \circ (\kappa' \circ \kappa'') & = & (\kappa \circ \kappa') \circ \kappa'' \\
[\mathrm{C2}] & id \circ \kappa & = & \kappa \\
[\mathrm{C3}] & \kappa \circ id & = & \kappa \\
[\mathrm{C4}] & id_{\tau \to \tau'} & = & id_{\tau} \to id_{\tau'} \\
\\
[\mathrm{E1}] & [id]M & = & M \\
[\mathrm{E2}] & [\kappa'][\kappa]M & = & [\kappa' \circ \kappa]M \\
[\mathrm{E3}] & [\kappa_1 \to \kappa_2](\lambda x.M) & = & \lambda x.[\kappa_2]M\{x \mapsto [\kappa_1]x\} \\
[\mathrm{E4}] & ([\kappa_1 \to \kappa_2]M)\; N & = & [\kappa_2](M\; ([\kappa_1]N))
\end{array}
$$

Figure 3.2: Equational Theory

We shall now define proof transformations on subtyping proofs by prescribing rewrite relations on completions. The rewrite relations will be contained in the equational theory of Figure 3.2, and they will therefore (Lemma 3.1.3) transform derivable judgements into derivable judgements. Before we go on to define the rewite relations, we observe that

Lemma 3.1.4 *Assume* $K \vdash \kappa : \tau \leadsto \tau'$ *with a structured type in either domain or range. Then* $K \vdash \kappa : (\tau_1 \to \tau_2) \leadsto (\tau_1' \to \tau_2')$ *and, equivalently,*

1. $E \vdash \kappa = \kappa_1 \to \kappa_2$ *, where*
2. $K \vdash \kappa_1 : \tau_1' \leadsto \tau_1$ *and* $K \vdash \kappa_2 : \tau_2 \leadsto \tau_2'$

PROOF Follows from the Matching Lemma (Lemma 2.1.5) via the 1-1 correspondence between judgements $C \vdash_P \tau \leq \tau'$ and judgements $C^\dagger \vdash_P \kappa : \tau \leadsto \tau'$. □

An important message of the previous lemma is that any coercion with a structured type in the domain or range of its signature must itself be correspondingly structured (by removing identity coercions.) This property is what makes *atomic* subtyping special (and pleasingly so) from the proof-theoretic viewpoint. As we shall see soon, it has practically relevant consequences.

The first proof normalization we consider is extremely simple. It arises from normalizing completions under the left-to-right orientation of rule [E2] in Figure 3.2. We call it $\to_c$ (for "coercion compression"):

$$[\kappa][\kappa']M \to_c [\kappa' \circ \kappa]M$$

The rewrite relation induced by this rule is evidently terminating. It shows that every subtyping proof can be transformed into a proof in which rule [SUB] is never applied immediately after another application of [SUB]. In terms of completions M, it means that, whenever $\to_c$ is present, one can always assume without loss of generality that every subterm of $M^\diamond$ has at most one coercion applied to it in M; if the rules for the identity coercion are also present, one can assume that every subterm has exactly one coercion applied to it.

Our second rewrite system is more interesting; it performs what we shall call *leaf normalization*, and the rewrite relation is therefore called $\to_L$. It is defined in Figure 3.3. We emphasize that the relation $\to_L$ should be understood as *equivalence class rewriting*, in the sense of [8], modulo the equational theory on completions induced by the rule [LE1] of Figure 3.3.

$$
\begin{array}{llcl}
\text{[LE1]} & [\kappa'][\kappa]M & = & [\kappa' \circ \kappa]M \\
\text{[LR1]} & [id_\tau]M & \rightarrow & M \\
\text{[LR2]} & [\kappa](M\ N) & \rightarrow & ([id \rightarrow \kappa]M)N \\
\text{[LR3]} & [\kappa_1 \rightarrow \kappa_2](\lambda x.M) & \rightarrow & \lambda x.[\kappa_2]M\{x \mapsto [\kappa_1]x\}
\end{array}
$$

Let $\rightarrow_L$ be the rewite relation on completions induced by taking the rules [LR1], [LR2], [LR3] *modulo* the rule [LE1].

Figure 3.3: Leaf Normalization Rewriting System

Theorem 3.1.5 *(Leaf-normalization) The rewrite relation $\rightarrow_L$ is strongly normalizing.*

PROOF Rules [LR2] and [LR3] can only be applied a finite number of times, because they move coercions downwards in the syntax tree of the coercion erasure of the completion, and none of the other rules move them upwards. Hence, after a finite number of steps, only rule [LR1] can be applied, and since reduction under this rule (*modulo* the equation) is obviously terminating, there can be no infinite reduction sequence in $\rightarrow_L$. □

Exercise 15 The proof of the previous theorem is really by a lexicographic measure argument. Make that argument explicit. □

We can now derive Mitchell's normal form property for atomic subtyping (see [26] which proves the result without using explicit coercions.)

Corollary 3.1.6 *(Form of leaf-normal form proofs) For every provable atomic judgement $C, \Gamma \vdash M : \tau$ there exists a proof in which the rule [SUB] is only used immediately after [VAR] or [BASE].*

PROOF By Lemma 3.1.2 every proof can be represented as a proof on a completion $M^\dagger$; by Theorem 3.1.5 that completion has a normal form under $\rightarrow_L$, and by Lemma 3.1.3 and Lemma 3.1.2 the normal form completion still represents an equivalent typing under the given coercion assumptions and type assumptions. Using Lemma 3.1.4, inspection of normal form completions shows that

- In the typing proof of a normal form completion, no application of the rule [COERCE] can occur immediately after an application of [APP] or [ABS].

Using coercion compression, we can assume that the rule [COERCE] is never applied immediately after another application of rule [COERCE]. This proves the claim. □

Hence the name *leaf* normalization; a proof in leaf-normal form uses the rule [SUB] only at the leaves of the syntax tree.

We now introduce a third rewrite relation, $\rightarrow_A$, for normalizing subtype proofs. It is called *applicative normalization* and it is defined in Figure 3.4. Again, the rewrite relation should be understood as equivalence class rewriting.

$$
\begin{array}{lll}
[\mathrm{AE1}] & [\kappa'][\kappa]M & = [\kappa' \circ \kappa]M \\
[\mathrm{AR1}] & [id_\tau]M & \to M \\
[\mathrm{AR2}] & ([\kappa_1 \to \kappa_2]M)N & \to [\kappa_2](M\,([\kappa_1]N)) \\
[\mathrm{AR3}] & \lambda x.[\kappa]M & \to [id \to \kappa](\lambda x.M)
\end{array}
$$

Let $\to_A$ be the rewite relation on completions induced by taking the rules [AR1], [AR2], [AR3] *modulo* the rule [AE1].

Figure 3.4: Applicative Normalization Rewriting System

Theorem 3.1.7 *(Applicative normalization) The rewrite relation $\rightarrow_A$ is strongly normalizing.*

PROOF Rule [AR3] can only be applied a finite number of times, since it moves a coercion upwards, outside a λ-abstraction, and no rule moves a coercion inside a λ-abstraction. Hence, an infinite reduction sequence implies that there exists an infinite reduction sequence which does not use rule [AR3]. But this is easily seen to be impossible, since rule [AR2] is terminating, because it eliminates an arrow-coercion. $\square$

Fuh and Mishra [12] define the following notion of *normal typing judgements* in atomic subtyping. According to this definition, a judgement $C, \Gamma \vdash M : \tau$ is called normal iff

1. M is a constant c, and $c : \tau \in P$, or
2. M is a variable x, and $x : \tau \in \Gamma$, or
3. M is a λ-abstraction $\lambda x.N$, and $C, \Gamma \cup \{x : \tau_1\} \vdash N : \tau_2$ is a normal judgement and $\tau \equiv \tau_1 \rightarrow \tau_2$, or
4. M is an application $M_1 M_2$, and $C, \Gamma \vdash M_1 : \tau_1 \rightarrow \tau$ and $C, \Gamma \vdash M_2 : \tau_1'$ are normal judgements, and $C \vdash_P \tau_1' \leq \tau_1$.

In other words, in a judgement in applicative normal form, subtyping hypotheses are only used to coerce arguments passed to functions. Theorem 3.1.7 shows that every judgement can be transformed into a judgement in normal form in the sense of Fuh and Mishra. To see this, define for K a set of coercion terms and M a completion the *restriction of K wrt. M*, $K\mid_M$ to be the set of those primitive coercions which occur in both M and K and which are not base coercions. Then we have

Lemma 3.1.8 *If $K, \Gamma \vdash_{CO} [\kappa]M : \tau$ is (derivable) in applicative normal form with $K \vdash \kappa : \tau' \rightsquigarrow \tau$, then $(K\mid_M)^\diamond, \Gamma \vdash M^\diamond : \tau'$ is (derivable) in Fuh-Mishra normal form.*

PROOF It is obvious that, if $K, \Gamma \vdash_{CO} [\kappa]M : \tau$ is derivable, then so is $(K\mid_M)^\diamond, \Gamma \vdash M^\diamond : \tau'$.

Inspection of the reduction rules for $\rightarrow_A$ (together with Lemma 3.1.4 !) shows that, if M is in applicative normal form, then every coercion nested inside M must occur in front of a subterm occurrence in argument position. $\square$

As is stated in [12], we can always find a Fuh-Mishra normal typing, which is principal under $\prec$. In fact,

Lemma 3.1.9 *One has:*

1. *If* $K^\diamond, \Gamma \vdash ([\kappa]M)^\diamond : \tau$ *is principal under* $\prec$ *with* $K \vdash \kappa : \tau' \leadsto \tau$ *, then it is an instance of* $(K\mid_M)^\diamond, \Gamma \vdash M^\diamond : \tau'$ *which is also principal.*

2. *For every principal typing* t_1 *under* $\prec$ *there exists a Fuh-Mishra normal form typing* t_2 *which has* t_1 *as an instance (and hence* t_2 *is also principal under* $\prec$*.)*

PROOF Since we obviously have $K^\diamond \vdash \tau' \leq \tau$ and $K^\diamond \vdash (K\mid_M)^\diamond$, the first mentioned judgement is an instance, under $\prec$, of the second judgement, and the first claim follows.

Now suppose an arbitrary principal judgement is given; it can be written as $K^\diamond, \Gamma \vdash M^\diamond : \tau$ for some K, M. Then transform the completion judgement $K, \Gamma \vdash_{CO} M : \tau$ into an applicative normal form judgement $K, \Gamma \vdash_{CO} [\kappa]M' : \tau$ by Theorem 3.1.7, with $[\kappa]M'$ an applicative normal form of M and $K \vdash \kappa : \tau' \leadsto \tau$, and use Lemma 3.1.8 to show that the judgement $(K\mid_M)^\diamond, \Gamma \vdash (M')^\diamond : \tau'$ is a derivable subtyping judgement in Fuh-Mishra normal form; clearly, $K^\diamond, \Gamma \vdash ([\kappa]M')^\diamond : \tau$ is a principal typing, and the last claim now follows from the first. □

To conclude, we have shown two normal form theorems, corresponding to the two kinds of normal form subtyping proofs exploited by, respectively, Mitchell in [26] and Fuh and Mishra in [12]. As these papers show, the normal form properties can be put to practical use, since they imply that it is enough to generate subtype hypotheses at a few, specific places in the syntax tree. Thus, in [26], an algorithm for atomic subtyping is given, which only extracts constraints at the leaves, and in [12] another algorithm for atomic subtyping is given, which only extracts constraints when an argument is passed to a function. In contrast, algorithm $\mathcal{U}$ generates new constraints at *every* syntactic construction. So, in addition to simplifying the type inference algorithms, the use of normal form typings will in many cases decrease the size of coercion sets.

3.2 Type inference

It turns out that the notion of *matching substitutions* is the key to type inference with atomic subtyping. Let S be a type substitution, and let C be a coercion set. Then S is called a matching substitution for C iff it holds for all $\tau \leq \tau' \in C$ that $S(\tau)$ and $S(\tau')$ match. A matching substitution S is called a *most general matching substitution* for C iff every other matching substitution T for C factors through S, *i.e.*, there exists a substitution V such that $T = V \circ S$.

Assume for the moment that there exists a procedure MATCH (we shall soon show the assumption true) such that MATCH(C) returns a most general matching substitution for C, if any matching substitution exists for C, and otherwise MATCH(C) fails. Then it is not difficult to see that one can obtain type inference algorithms for atomic subtyping, atomic weak subtyping and atomic, strong subtyping by composing algorithm $\mathcal{U}$ with matching. We begin with atomic subtyping.

Atomic subtyping

We define an algorithm $\mathcal{A}$ for atomic subtyping as follows:

```
A[[M]] =  let C, Γ ⊢ M : τ = U[[M]]
          and S = MATCH(C)
          in
             ATOMIC(S(C)), S(Γ) ⊢ M : S(τ)
          end
```

Theorem 3.2.1 *(Soundness of $\mathcal{A}$) If $C, \Gamma \vdash M : \tau = \mathcal{A}[\![M]\!]$, then $C, \Gamma \vdash M : \tau$ is a derivable atomic judgement.*

PROOF Let

$$C', \Gamma' \vdash M : \tau' = \mathcal{U}[\![M]\!]$$

Soundness of $\mathcal{U}$ (Theorem 2.3.6) together with Lemma 2.3.1 shows that the judgement

$$S(C'), S(\Gamma') \vdash M : S(\tau')$$

is derivable (with S, C', Γ', M, τ' as in the algorithm.) But then, by Lemma 2.1.12 and the property that MATCH finds a matching substitution, we have

$$\text{ATOMIC}(S(C')) \vdash_P S(C')$$

and therefore $\text{ATOMIC}(S(C')), S(\Gamma') \vdash M : S(\tau') = \mathcal{A}[\![M]\!]$ must be derivable. □

Theorem 3.2.2 *(Completeness of $\mathcal{A}$) If $C', \Gamma' \vdash M : \tau'$ is a derivable atomic judgement, then $\mathcal{A}[\![M]\!] \neq$ FAIL and $C', \Gamma' \vdash M : \tau'$ is a weak instance of $\mathcal{A}[\![M]\!]$.*

PROOF Under the conditions, $\mathcal{U}[\![M]\!]$ cannot fail, by completeness of $\mathcal{U}$ (Theorem 2.3.7); completeness of $\mathcal{U}$ also entails that, with $C, \Gamma \vdash M : \tau = \mathcal{U}[\![M]\!]$, we have $C', \Gamma' \vdash M : \tau'$ a weak instance of $C, \Gamma \vdash M : \tau$, i.e., there exists a substitution S' such that

1. $C' \vdash_P S'(C)$
2. $S'(\tau) = \tau'$
3. $S'(\Gamma) \subseteq \Gamma'$

Since C' was assumed to be atomic, S' must be a matching substitution for C (by the Matching Lemma, Lemma 2.1.5, together with (1) above.) Therefore, the call to MATCH in $\mathcal{A}[\![M]\!]$ does not fail, so $S = \text{MATCH}(C)$ is a most general matching substitution for C, from which we conclude

$$S' = V \circ S$$

for some substitution V. But then conditions (1)-(3) above show that $C', \Gamma' \vdash M : \tau'$ is a weak instance of $S(C), S(\Gamma) \vdash M : S(\tau)$ under substitution V. Now, Lemma 2.1.12 shows that $\text{ATOMIC}(S(C))$ does not fail, since $S(C)$ is matching, and moreover

$$\text{ATOMIC}(S(C)) \vdash S(C)$$

which entails that $S(C), S(\Gamma) \vdash M : S(\tau)$ is a weak instance of $\text{ATOMIC}(S(C)), S(\Gamma) \vdash M : S(\tau)$. We can conclude that $\mathcal{A}[\![M]\!] \neq \text{FAIL}$ and $C', \Gamma' \vdash M : \tau'$ is a weak instance of $\mathcal{A}[\![M]\!]$. □

The reader should note that, in the litterature on atomic subtyping, inference algorithms are usually not given by composition with $\mathcal{U}$. The main reason for this is probably that one wants to exploit (one of) the normalization theorems (Theorem 3.1.5 and Theorem 3.1.7), so the constraint extraction of $\mathcal{U}$ must be changed accordingly to an optimized version $\mathcal{U}'$. See [26] and [11] for algorithms which exploit normal forms; the former uses leaf-normal forms, the latter uses applicative normal form. Moreover, it is often the case that the steps corresponding to those introduced in $\mathcal{A}$ are combined with those of $\mathcal{U}'$ into a single procedure; one motivation for this is that one may want the MATCH-test to be carried out on the fly, so as to be able to abort with an error message immediately, as soon as a match failure is discovered. However, we have found it more relevant to take the modular approach here, since it makes the correctness proofs modular and easier to understand.

Matching

We now prove that most general matching substitutions can be computed. We say that a type substitution is *variable-to-variable* iff it maps every variable to a variable. First we note that we have

Exercise 16 (Mitchell)
Let C be an arbitary constraint set but *with no type constants* and let

$$E_C = \{(\tau = \tau') \mid \tau \leq \tau' \in C\}$$

Prove that S is a matching substitution for C if and only if there exists a variable-to-variable substitution T such that $T \circ S$ is a unifier for E_C.

Show that the property above no longer holds if type constants are allowed in C. □

If C is a constraint set, we let $\mathrm{FTV}(C)$ denote the set of variables that occur in C.

Lemma 3.2.3 *There is a function* MATCH *from constant free constraint sets to substitutions such that* $\mathrm{MATCH}(C)$ *fails if no matching substitution exists for* C*, and if a matching substitution exists for* C*, then* $\mathrm{MATCH}(C)$ *does not fail and the substitution*

$$\{\alpha \mapsto \mathrm{MATCH}(C)(\alpha)\}_{\alpha \in FTV(C)}$$

is a most general matching substitution for C*;* $\mathrm{MATCH}(C)$ *can be computed in time* $\mathcal{O}(|C|)$*, and, for each* C*,* $\mathrm{MATCH}(C)(\alpha)$ *can be computed in time* $\mathcal{O}(|\mathrm{MATCH}(C)(\alpha)|)$.

PROOF Define procedure MATCH by

```
MATCH($C$) = let $S = \mathrm{UNIFY}(E_C)$
             in
                $\lambda v.$if $m[v] \neq$ UNDEFINED
                   then $m[v]$
                   else
                      let $\tau = \mathrm{FRESH}(S(v))$
                      in
                         $(m[v] := \tau;\ \tau)$
                      end
             end
```

where m is an array of type expressions indexed by type variables, working as a memiozation table; FRESH inserts a distinct, fresh variable for every distinct occurrence of a variable in the argument type expression.

First note that $\mathrm{MATCH}(C)$ fails if and only if E_C is not unifiable; by Exercise 16, this shows that if MATCH fails, then no matching substitution exists for C.

Suppose now that $\mathrm{MATCH}(C)$ does not fail. We wish to show that

$$S' = \{\alpha \mapsto \mathrm{MATCH}(C)(\alpha)\}_{\alpha \in \mathrm{FTV}(C)}$$

is a most general matching substitution for C. First note that S' is clearly a matching substitution for C, since it arises from freshening variables in the output of a unifying substitution S. Now assume that S'' is any other matching substitution for C. By Exercise 16, there exist variable-to-variable substitutions T' and T'' such that $T' \circ S'$ and $T'' \circ S''$ are both unifiers for the set E_C. Since S' arises from freshening variables in a most general unifier for E_C, there evidently must exist a variable-to-variable substitution T_1 such that $T_1 \circ S'$ is a most general unifier for E_C. Hence, there exists a substitution R such that

$$T'' \circ S'' = R \circ T_1 \circ S'$$

It follows that, for all $\alpha \in \mathrm{FTV}(C)$,

$$(*) \quad S''(\alpha) \text{ matches with } R \circ T_1 \circ S'(\alpha)$$

Now consider the substitution $\overline{R}$ given by $\alpha \mapsto \textsc{fresh}(R \circ T_1(\alpha))$. Since no variable occurs more than once in $S'(\alpha)$, it follows that no variable occurs more than once in $\overline{R} \circ S'(\alpha)$. But then, by $(*)$, there must exist a variable-to-variable substitution T_2 such that

$$S'' = T_2 \circ \overline{R} \circ S'$$

which shows that S' is a most general matching substitution for C.

The complexity claims are obvious, since unification can be programmed to run in linear time, using graph representations. □

See [26] for a slightly different proof of the previous lemma. See also [12] for yet a third treatment of matching. It is necessary to be careful about the statement of complexity of matching, because unfolding a most general matching substitution may lead to an exponantial blow-up in the size of the coercion set:

Exercise 17 Show that, if S is a most general matching substitution for C, the size of $S(C)$ can be $2^{\Omega(|C|)}$ in the worst case. □

Assuming that UNIFY produces a graph structure which maintains equivalence classes and which can be suitably accessed, we can deal with type constants as follows. Given constraint set C possibly with constant types, we select a special set of fresh variables, $\{\alpha_b \mid b \text{ constant in } C\}$ and process $C\{b \mapsto \alpha_b\}$ instead of C. We insert the test

$$\textbf{if } \exists b \in C.\, S(\alpha_b) \text{ is not an atom } \textbf{then } \textsc{fail};$$

immediately after the call to UNIFY, where S is the unifying substitution. We can assume that the test can be performed in time $\mathcal{O}(|C|)$, since we need only inspect the "initial" part of $S(\alpha_b)$ in the graph representing S; this is under reasonable assumptions about the implementation of UNIFY (such as, e.g., that every equivalence class representative of a set containing structured types is itself a structured type.) We then "redirect" the equivalence class of each α_b to contain only itself; the variables α_b can then be substituted back to the appropriate constants b.

Note that, in atomic subtyping, derivability is not closed under arbitrary type substitutions. However, we do have

Lemma 3.2.4 *If $C, \Gamma \vdash M : \tau$ is a derivable atomic judgement and S is a matching substitution for C, then* $\textsc{atomic}(S(C)), S(\Gamma) \vdash M : S(\tau)$ *is a derivable atomic judgement.*

PROOF Use Lemma 2.3.1 and Lemma 2.1.12. □

Weak atomic subtyping

We now consider type inference for weak atomic subtyping. The obvious idea is to extend algorithm $\mathcal{A}$ with a consistency check. Accordingly, we define the algorithm $\mathcal{A}^W$ for weak atomic subtyping as follows:

```
$\mathcal{A}^W[\![M]\!]$  =  let $C, \Gamma \vdash M : \tau = \mathcal{A}[\![M]\!]$
          in
             if CONSISTENT$(C)$ then $C, \Gamma \vdash_W M : \tau$
             else FAIL
          end
```

We assume a procedure CONSISTENT which returns *true* if the input constraint set is consistent over P and *false* if not. It is easy to see that such a procedure exists, *e.g.*, using Lemma 2.1.13.

It is trivial to see that $\mathcal{A}^W$ is sound:

Theorem 3.2.5 *(Soundness of $\mathcal{A}^W$) If $C, \Gamma \vdash_W M : \tau = \mathcal{A}^W[\![M]\!]$, then $C, \Gamma \vdash_W M : \tau$ holds with C atomic.*

PROOF Obvious. $\square$

Theorem 3.2.6 *(Completeness of $\mathcal{A}^W$) If $C', \Gamma' \vdash_W M : \tau'$ is a derivable judgement with C' atomic, then $\mathcal{A}^W[\![M]\!] \neq$ FAIL and $C', \Gamma' \vdash_W M : \tau'$ is a weak instance of $\mathcal{A}^W[\![M]\!]$.*

PROOF Let

$$C, \Gamma \vdash M : \tau = \mathcal{A}[\![M]\!]$$

Then, by completeness of $\mathcal{A}$ (Theorem 3.2.2), we get that $C', \Gamma' \vdash_W M : \tau'$ is a weak instance of $C, \Gamma \vdash M : \tau$, so, in particular, there exists a substitution S such that $C' \vdash_P S(C)$. Clearly, we have shown the theorem, if we can show that C is consistent. To see this, assume on the contrary that C were not consistent, *i.e.*, for some $b, b' \in P$ we have $C \vdash_P b \leq b'$ but *not* $\vdash_P b \leq b'$. Then, by Lemma 2.1.4, we get $S(C) \vdash_P b \leq b'$, and therefore also $C' \vdash_P b \leq b'$, since $C' \vdash_P S(C)$; but then C' is inconsistent, which is a contradiction. This demonstrates that C is consistent, hence $\mathcal{A}^W[\![M]\!] \neq$ FAIL and the proof is complete. $\square$

Strong atomic subtyping

Strong atomic subtype inference can be done by composing a satisfiability test with $\mathcal{A}$. We define the algorithm $\mathcal{A}^S$ for strong atomic subtyping as follows.

```
$\mathcal{A}^S[\![M]\!]$ = let $C, \Gamma \vdash M : \tau = \mathcal{A}[\![M]\!]$
           in
              if SATISFIABLE($C$) then $C, \Gamma \vdash_S M : \tau$
              else FAIL
           end
```

We assume a procedure SATISFIABLE which returns *true* if the input constraint set is satisfiable and *false* if not. It is easy to see that the satisfiability predicate is computable, since there are only finitely many valuations that need to be considered (use Exercise 5 together with the fact that there are only finitely many atomic valuations that need to be considered for given C. Note that we are not concerned with complexity issues here.)

It is trivial that $\mathcal{A}^S$ is sound.

Theorem 3.2.7 *(Soundness of $\mathcal{A}^S$) If $C, \Gamma \vdash_S M : \tau = \mathcal{A}^S[\![M]\!]$, then $C, \Gamma \vdash_S M : \tau$ holds with C atomic.*

PROOF Obvious. □

Theorem 3.2.8 *(Completeness of $\mathcal{A}^S$) If $C', \Gamma' \vdash_S M : \tau'$ is a derivable judgement with C' atomic, then $\mathcal{A}^S[\![M]\!] \neq$ FAIL and $C', \Gamma' \vdash_S M : \tau'$ is a weak instance of $\mathcal{A}^S[\![M]\!]$.*

PROOF Let

$$C, \Gamma \vdash M : \tau = \mathcal{A}[\![M]\!]$$

Then, by completeness of $\mathcal{A}$ (Theorem 3.2.2), we get that $C', \Gamma' \vdash_S M : \tau'$ is a weak instance of $C, \Gamma \vdash M : \tau$, so, in particular, there exists a substitution S such that $C' \vdash_P S(C)$. Clearly, we have shown the theorem, if we can show that C is satisfiable. To see that C must be satisfiable, let ρ be a satisfying valuation for C', *i.e.*, $\rho \models_P C'$. By $C' \vdash_P S(C)$ together with Lemma 2.2.1, we then get $\rho \models_P S(C)$, and so $\rho \circ S$ is a satisfying valuation for C. □

Observe that, if C is satisfiable, then there exists a matching substitution for C, because any satisfying valuation ρ for C will also be a matching substitution for C, by the Match Lemma (only matching types are comparable in $(T_P, \leq)$.) Hence, we could say that any derivable judgement $C, \Gamma \vdash M : \tau$ can only fail to be a strong typing if C is unsatisfiable. In contrast, a set can well be consistent and yet have no matching substitution, as for example $C = \{\alpha \leq \alpha \rightarrow \alpha\}$ is consistent but has no matching substitution.

3.3 Simplification

In this section we introduce two kinds of optimization on atomic typings. Both were identified by Fuh and Mishra in [11], and the reader is referred to that paper for further details not given here. Both can be described as typing transformations which preserve the instance equivalence $\approx$.

3.3.1 Eliminating redundant coercions

The first kind of optimization introduced by Fuh and Mishra deals with logically redundant type variables in coercion sets. There are two transformations to consider.

Cycle elimination

The first optimization performs *cycle elimination* in constraint sets. It is assumed but not described or analyzed in [11]. The transformation can be described as a reduction $\mapsto_{ce}$ on typing judgements, as follows:

$$C, \Gamma \vdash M : \tau \mapsto_{ce} C', \Gamma' \vdash M : \tau' \text{ iff } \begin{cases} (1) & C \vdash_P \alpha \leq A \,,\, C \vdash_P A \leq \alpha \,, \alpha \not\equiv A \\ (2) & C' = C\{\alpha \mapsto A\} \\ (3) & \Gamma' = \Gamma\{\alpha \mapsto A\} \\ (4) & \tau' = \tau\{\alpha \mapsto A\} \end{cases}$$

The transformation clearly terminates, since $\mathrm{FTV}(C)$ shrinks at every reduction step; hence, every judgement has a normal form under $\mapsto_{ce}$. The rationale of the transformation is really Exercise 4, which shows that we have

Lemma 3.3.1 *If $t_1 \mapsto_{ce} t_2$ then*

1. *t_2 is a weak instance of t_1*

2. *t_1 is an instance of t_2*

and in particular, $t_1 \approx t_2$.

PROOF Suppose that $t_1 = C, \Gamma \vdash M : \tau$, and $t_1 \mapsto_{ce} t_2$ under substitution $\{\alpha \mapsto A\}$. That t_2 is a weak instance of t_1 is obvious, using the substitution $\{\alpha \mapsto A\}$. To see that $t_2 \prec t_1$, use Exercise 4 which entails that

1. $C \vdash_P C\{\alpha \mapsto A\}$

2. $C \vdash_P \tau\{\alpha \mapsto A\} \leq \tau$

3. $C \vdash_P \Gamma\{\alpha \mapsto A\}(x) \leq \Gamma(x)$

□

Observe that Exercise 12 shows that cycle elimination does *not* work under the weak instance relation, since the transformation does not preserve equivalence of typings under weak instance.

Cycle elimination should not be implemented as a rewrite system. In practice, one would compute the strongly connected components of $C \cup P$ and then collapse each strongly connected component containing a variable into a chosen variable of that component. Strongly connected components can be computed in time $\mathcal{O}(|C|)$ (for graph $G = (V, E)$ in time $\mathcal{O}(V + E)$, see [5]), so cycle elimination can be performed in $\mathcal{O}(|C|)$.

Normally, we shall assume that the coercion set C is at least consistent (otherwise we shall typically reject a typing using C) and therefore we can assume that every proper cycle of $C \cup P$ contains at most one element of P. For such acceptable C cycle elimination will indeed succeed in removing all proper cycles. The resulting set will enjoy the anti-symmetry property of Lemma 2.1.8.

G-minimization and G-simplification

The second optimization is more complicated. It is based on the idea that, if a variable α occurs in the coercion set C of a judgement $C, \Gamma \vdash M : \tau$ but α occurs *neither* in Γ *nor* in τ, then α can be eliminated (in a certain way) from C, *unless* α contributes in an *essential* way to the deductive strength of C. Accordingly, define for typing judgement $t = C, \Gamma \vdash M : \tau$ the set $\mathrm{Obv}(t)$ of *observable type variables* to be the set of type variables which occur in either Γ or in τ; define the set $\mathrm{Intv}(t)$ of *internal type variables* to be $\mathrm{Intv}(t) = \mathrm{FTV}(C) \setminus \mathrm{Obv}(t)$, *i.e.*, the set of type variables of C which are not observable. Intuitively, $\alpha \in C$ contributes in an essential way to the typing only if α constrains some observable variables by "connecting them". We say (following Fuh and Mishra) that

- A coercion set C is called *G-minimal* wrt. a set O of type variables iff whenever substitution S is an identity on O and $C \vdash_P S(C)$, then S is a renaming on type variables in C and O.

  A typing $t = C, \Gamma \vdash M : \tau$ is called *G-minimal* iff C is a G-minimal coercion set wrt. $\mathrm{Obv}(t)$. A typing is called *redundant* iff it is not G-minimal.

To see that it is possible, in principle, to compute G-minimal typings, define $\mathit{Subs}(t)$ for $t = C, \Gamma \vdash M : \tau$ to be the set of substitutions S such that

$$S(\alpha) = \begin{cases} \alpha & \text{if } v \notin \mathrm{Intv}(t) \\ \text{a constant, or a variable in } \mathrm{FTV}(C) & \text{if } \alpha \in \mathrm{Intv}(t) \end{cases}$$

We can now define a program G-MINIMIZE as follows:

```
G-MINIMIZE($t$ as $C, \Gamma \vdash M : \tau$) =
    if $\exists S \in \mathit{Subs}(t).\ C \vdash_P S(C) \wedge S$ not renaming on Intv($t$)
    then G-MINIMIZE($S(t)$)
    else $t$
```

Clearly, for any typing t, the set $\mathit{Subs}(t)$ is finite, since only finitely many variables can occur in t. It follows that G-MINIMIZE always terminates, because the set Intv(t) must shrink at each recursive call. Hence, G-MINIMIZE is certainly computable. Note that we have just given a constructive proof that G-minimal typings exist (any typable term has a G-minimal typing.)

Exercise 18 Show that, for every typing judgement t, t is a weak instance of G-MINIMIZE(t). Conclude that $t \approx$ G-MINIMIZE(t) and that G-MINIMIZE preserves principal typings. □

We now consider the problem of uniqueness of G-minimal typings. Fuh and Mishra [11] claim (Theorem 3 in [11]) that G-minimal typings which are equivalent wrt. $\approx$, are unique up to renaming and the equivalence of coercion sets:

- (Fuh and Mishra [11], Theorem 3)
  If $C_1, \Gamma_1 \vdash M : \tau_1 \approx C_2, \Gamma_2 \vdash M : \tau_2$ are both G-minimal typings, then there exists a renaming S such that

  1. $\tau_1 = S(\tau_2)$
  2. $\Gamma_1 |_{FV(M)} = S(\Gamma_2 |_{FV(M)})$
  3. $C_1 \sim_P S(C_2)$

If this were true, then it would be very interesting, because it would show that minimization fulfills a strong desire we have, namely to be able to pick out an optimal representative of each $\approx$-equivalence class. Unfortunately, Fuh and Mishra's proof of the claim is flawed [1] and, in fact, the claim of

[1]The flaw in the proof is the claim, made at the beginning of the proof, that it follows from the definition of $\approx$ (which is called $\cong$ in [11]) that (under the assumptions given in the statement of [11], Theorem 3, as rendered above) one has: there exist substitutions S_1 and S_2 such that

1. $\tau_1 = S_1(\tau_2)$ and $\tau_2 = S_2(\tau_1)$
2. $\Gamma_1 |_{FV(M)} = S_1(\Gamma_2 |_{FV(M)})$ and $\Gamma_2 |_{FV(M)} = S_2(\Gamma_2 |_{FV(M)})$
3. $C_1 \vdash_P S_1(C_2)$ and $C_2 \vdash_P S_2(C_1)$

The first two claims made here are wrong; they *would* follow, *if* we had assumed the two typings equivalent under *weak instance*, and the proof of Theorem 3 in [11] given there would be valid; but it does not hold under the *lazy* relation $\approx$. Note also that Fuh and Mishra assume coercion sets to be acyclic; however, this still doesn't save the claim (Example 3.3.2 uses acyclic coercion sets.)

Theorem 3 in [11] is *false*. To see this, consider the following example typings (taken, in fact, from [11])

Example 3.3.2 Let $comp \equiv \lambda f.\lambda g.\lambda x.f(gx)$ and let the typing judgements t_1 and t_2 be as in Exercise 14, to recall

$$t_1 = \{\delta \leq \alpha, \eta \leq \gamma, \beta \leq \upsilon\}, \emptyset \vdash \ comp : (\alpha \to \beta) \to (\gamma \to \delta) \to (\eta \to \upsilon)$$

$$t_2 = \emptyset, \emptyset \vdash \ comp : (\alpha \to \beta) \to (\gamma \to \alpha) \to (\gamma \to \beta)$$

Then it is easy to verify that $t_1 \approx t_2$ (cf. Exercise 14) and both t_1 and t_2 are G-minimal typings (in t_1 all variables in the coercion set are observable, and in t_2 the coercion set is empty.) However, there is clearly no renaming which maps the type of one of t_1 and t_2 to the other. □

What *is* true, though, is that the uniqueness property holds in case the two typings are equivalent under *weak* instance (see previous footnote.) Moreover, the following property holds for coercion sets (the proof method is the same as in the relevant parts of the proof of Fuh and Mishra's Theorem 3.)

Lemma 3.3.3 *Let C_1, C_2 be atomic coercion sets which are both G-minimal wrt. the set O. Suppose that there exist substitutions S_1 and S_2 such that*

1. *S_1 and S_2 are identities on O*
2. *$C_1 \vdash_P S_1(C_2)$ and $C_2 \vdash_P S_2(C_1)$*

Then there exists a substitution S which is a renaming on the variables in C_2 and O and identity on O, such that $C_1 \sim_P S(C_2)$.

PROOF By $C_1 \vdash_P S_1(C_2)$ we have $S_2(C_1) \vdash_P S_2(S_1(C_2))$, and hence, by $C_2 \vdash_P S_2(C_1)$, we get $C_2 \vdash_P S_2(S_1(C_2))$. Since C_2 is G-minimal wrt. O and $S_2 \circ S_1$ must be an identity on O, it follows from the definition of G-minimality that $S_2 \circ S_1$ is a renaming on C_2 and O. Therefore, S_1 must be a renaming on C_2 and O (exercise: prove this !) By analogous argument, S_2 must be a renaming on C_1. Since $C_1 \vdash_P S_1(C_2)$, we have $(C_1 \cup P)^* \supseteq (S_1(C_2) \cup P)^*$, so that $|(C_1 \cup P)^*| \geq |(S_1(C_2) \cup P)^*|$; in the same way we also get $|(C_2 \cup P)^*| \geq |(S_2(C_1) \cup P)^*|$. Since S_1 is a renaming on C_2 and S_2 renaming on C_1, it must be the case that $|(C_1 \cup P)^*| = |(S_2(C_1) \cup P)^*|$ and also $|(C_2 \cup P)^*| = |(S_1(C_2) \cup P)^*|$. In total we have

$$|(C_1 \cup P)^*| = |(S_2(C_1) \cup P)^*| \leq |(C_2 \cup P)^*| = |(S_1(C_2) \cup P)^*|$$

By $(C_1 \cup P)^* \supseteq (S_1(C_2) \cup P)^*$ we then get $(C_1 \cup P)^* = (S_1(C_2) \cup P)^*$, and therefore $C_1 \sim_P S_1(C_2)$. Now take $S = S_1$ and the proof is complete. □

Lemma 3.3.3 shows that minimizations of the same coercion set C wrt. O are unique up to renaming and the equivalence of coercion sets: if $C \vdash_P S_1(C)$

and $C \vdash_P S_2(C)$ with S_1, S_2 identities on O and $S_1(C)$, $S_2(C)$ G-minimal, then $S_1(C) \sim_P R(S_2(C))$ for some renaming R on $S_2(C)$ and O with R identity on O.

Having presented the procedure G-MINIMIZE above, Fuh and Mishra [11] go on to remark that this is not a practically interesting procedure, since, for given $t = C, \Gamma \vdash M : \tau$, it appears to require *exhaustive* checking of the condition $C \vdash_P S(C)$ for all $S \in \mathit{Subs}(t)$. If this is true, then certainly minimization is intractable; in the worst case we have $\mathit{Subs}(t) = \mathrm{FTV}(C) \to (\mathrm{FTV}(C) \cup P)$, and hence $|\mathit{Subs}(t)| = (|\mathrm{FTV}(C)| + |P|)^{|\mathrm{FTV}(C)|}$, which would mean an exponential lower bound on minimization. Let us try to understand in more detail what is difficult about minimization. Viewing procedure G-MINIMIZE as a specification of what it means to G-minimize a set, clearly, the difficulty has to do with the non-deterministic choice expressed by the existential quantification in procedure G-MINIMIZE. It seems that an efficient implementation of the specification G-MINIMIZE must somehow discipline the search through $\mathit{Subst}(t)$ in such a way that not all of the search space is processed. The obvious idea would be to try to build up a minimizing substitution from smaller pieces, as is done in the following procedure MIN:

```
MIN($t$ as $C, \Gamma \vdash M : \tau$) =
    let $(\mathit{chosen}, S)$ = CHOOSE($C$, Intv($t$))
    in
        if $\mathit{chosen}$ then MIN($S(t)$)
        else return $t$
    end
```

with auxiliary procedure CHOOSE

```
CHOOSE($C, I$) =
    for $(\alpha, A) \in I \times (P \cup \mathrm{FTV}(C))$ do
        if $\alpha \not\equiv A \wedge C \vdash_P C\{\alpha \mapsto A\}$
         then return $(\mathit{true}, \{\alpha \mapsto A\})$
    end
    return $(\mathit{false}, \{\})$
```

Rather than implementing the existential quantification of procedure G-MINIMIZE by an exhaustive search through $\mathit{Subs}(t)$, procedure MIN chooses (in procedure CHOOSE) a single piece ($\{\alpha \mapsto A\}$) of a substitution, at each recursive call. Procedure MIN can get into at most $|\mathrm{FTV}(C)|$ recursive calls over a complete execution of MIN(t), since Intv(t) shrinks after each call. Considering now procedure CHOOSE, we see that the **for**-loop can be executed at most $(|\mathrm{FTV}(C)| + |P|)^2$ times, at each call; moreover, the body of the **for**-loop can be computed in time $\mathcal{O}(|C|^4)$, using Lemma 2.1.13. In total, procedure MIN transforms typings in time $\mathcal{O}(|C|^7)$, for a fixed order P.

Procedure MIN tries to exploit the idea that simultaneous substitutions can be *sequentialized*, in that it attempts to build a minimizing substitution from *simple* substitutions of the form $\{\alpha \mapsto A\}$. More technically, if S is a substitution, let $\mathit{Supp}(S)$ (the support of S) be the set of variables not mapped to themselves under S, *i.e.*, $\mathit{Supp}(S) = \{\alpha \mid S(\alpha) \not\equiv \alpha\}$; then S is simple iff $\mathit{Supp}(S)$ is a singleton set. In general, it is possible to factor a substitution S with finite support into a series of simple substitutions $S_1, \ldots, S_k$ such that $\forall \alpha \in \mathit{Supp}(S).\, S(\alpha) = S_k \circ \ldots \circ S_1(\alpha)$. Such a factoring amounts to sequentializing the simultaneous substitution S. That such factoring is always possible can be shown by induction on the size of $\mathit{Supp}(S)$. Note that, in general, such factoring may have to introduce "new variables"; for example, the renaming $S = \{\alpha \mapsto \beta, \beta \mapsto \alpha\}$ cannot be factored into simple substitutions without using a new variable γ, as in $\{\gamma \mapsto \beta\} \circ \{\beta \mapsto \alpha\} \circ \{\alpha \mapsto \gamma\}$. Note that the range of one of the factoring substitutions, $\{\alpha \mapsto \gamma\}$, contains an element, γ, which is not in the range of S. This might perhaps lead to a problem in MIN, since minimization requires the range of the substitutions used to be contained in $\mathrm{FTV}(C)$.

However, procedure MIN fails to compute exact G-minimal typings for another, more tangible reason, which also has to do with a sequentialization problem. To see this, consider the following example.

Example 3.3.4 Consider coercion set C over the partial order P as shown below:

C :

$$\begin{tikzcd}
 & \alpha \arrow[dl, dash] \arrow[dr, dash] & & \beta \arrow[dl, dash] \arrow[dr, dash] & \\
b_1 & & \gamma & & b_2
\end{tikzcd}$$

P :

$$\begin{tikzcd}
 & b_3 \arrow[dl, dash] \arrow[dr, dash] & \\
b_1 & & b_2
\end{tikzcd}$$

Suppose C is part of a typing t in which all the variables of C are internal. It can be seen that C is in normal form wrt. MIN (*i.e.*, $\mathrm{MIN}(t) = t$) because it is not possible to find a simple, non-identity substitution S of the form $\{\delta \mapsto A\}$, $\delta \in \{\alpha, \beta, \gamma\}$, $A \in \{\alpha, \beta, \gamma, b_1, b_2, b_3\}$, such that $C \vdash_P S(C)$. However, the set C is *not* G-minimal, because the substitution

$$S_{min} = \{\alpha \mapsto b_3, \beta \mapsto b_3, \gamma \mapsto b_3\}$$

satisfies $C \vdash_P S_{min}(C)$. Note that S_{min} can be factored as $S_{min} = \{\gamma \mapsto b_3\} \circ \{\beta \mapsto b_3\} \circ \{\alpha \mapsto b_3\}$. But there is no *simple* non-identity substitution S_{simple} which satisfies $C \vdash_P S_{simple}(C)$. In other words, the existence of a minimizing simultaneous substitution S does not guarantee that there exists a factoring of S into simple substitutions S_i such that the condition $C \vdash_P S_i(C)$ is satisfied for all i, even though it is the case that $C \vdash_P S(C)$. □

Exercise 19 In Example 3.3.4 we saw a coercion set C with constants from P which cannot be transformed by MIN and yet C is not G-minimal. Is there a set C' which mentions *no constants* such that C' is not G-minimal and yet C' cannot be transformed by MIN? □

We conclude that the strategy underlying procedure MIN is not adequate. However, it is *sound* in the sense that, for all typing judgements t, we have $t \approx \text{MIN}(t)$; only, it is not *complete* in the sense that there exist typing judgements t such that $\text{MIN}(t) = t$, and yet t is not G-minimal. The *approximate* minimization method introduced by Fuh and Mishra in [11] can be viewed as a (quite beautiful) optimization of procedure MIN. While not computing exact G-minimal typings, their algorithm runs in time $\mathcal{O}(|C|^4)$ and appears to produce good approximations to G-minimal typings. The main idea of the approximate method is to consider, for atomic coercion set C and atom A in C, the sets $above_C(A)$ and $below_C(A)$, defined as follows:

$$above_C(A) = \{A' \mid C \vdash_P A \leq A'\}$$

$$below_C(A) = \{A' \mid C \vdash_P A' \leq A\}$$

One then defines a binary relation $\leq_G$ (called *G-subsumption*) on $\text{FTV}(C) \times (\text{FTV}(C) \cup P)$ by taking $\alpha \leq_G A$ wrt. C iff

1. $above_C(\alpha) \setminus \{\alpha\} \subseteq above_C(A)$, and
2. $below_C(\alpha) \setminus \{\alpha\} \subseteq below_C(A)$

One can then show

Lemma 3.3.5 *If $\alpha \leq_G A$ wrt. C and $C' = C\{\alpha \mapsto A\}$, then*

$$C \vdash_P A' \leq A'' \wedge A' \not\equiv \alpha \wedge A'' \not\equiv \alpha \Leftrightarrow C' \vdash_P A' \leq A''$$

PROOF The implication $\Rightarrow$ follows from the Substitution Lemma, Lemma 2.1.4. To see the implication $\Leftarrow$, let

$$C^- = \{A' \leq A'' \in C \mid A' \equiv \alpha \text{ or } A'' \equiv \alpha\}$$

$$C^+ = \{A' \leq A \mid A' \leq \alpha \in C\} \cup \{A \leq A' \mid \alpha \leq A' \in C\}$$

Then $C' = (C \cup C^+) \setminus C^-$. By $\alpha \leq_G A$ one has $C \vdash_P C^+$; for instance, with $A''' \leq A \in C^+$, one has $A''' \leq \alpha \in C$, hence (assuming w.l.o.g. $A''' \not\equiv \alpha$) we have $A''' \in below_C(\alpha) \setminus \{\alpha\}$, hence (by $\alpha \leq_G A$) $A''' \in below_C(A)$, hence $C \vdash_P A''' \leq A$. This shows the implication. □

The property mentioned in Lemma 3.3.5 shows that one can identify α with A in C, if $\alpha \leq_G A$ wrt. C; as shown in the lemma, this identification will not weaken the deductive strength of C wrt. consequences not manifestly involving α. Hence, if α is not observable, we can eliminate α from the coercion set. This is formalized in the notion of G-reduction, defined by the reduction $\mapsto_G$ on typings:

$$C, \Gamma \vdash M : \tau \mapsto_G C', \Gamma \vdash M : \tau \text{ iff } \begin{cases} (1) & \alpha \leq_G A \text{ wrt. } C \\ (2) & \alpha \text{ not in } \Gamma, \alpha \text{ not in } \tau \\ (3) & C' = C\{\alpha \mapsto A\} \end{cases}$$

This reduction takes a typing to an equivalent one:

Exercise 20 Show that, if j_1 and j_2 are two atomic judgements, then $j_1 \mapsto_G j_2$ implies $j_1 \approx j_2$. □

It can be shown (see [11] for details) that $\mapsto_G$ is locally confluent up to the equivalence ($\sim_P$) of coercion sets and also terminating; consequently all typings have G-normal forms which, in addition, are unique up to variable renaming and equivalence of coercion sets.

Exercise 21 (Fuh and Mishra)
Let $C = \{\gamma_1 \leq \alpha, \gamma_2 \leq \alpha, \gamma_1 \leq \beta, \gamma_2 \leq \beta, \gamma_2 \leq \gamma_4, \gamma_3 \leq \gamma_1\}$ with α, β observable. What is the G-normal form of C? Is the normal form also G-minimal?
□

Fuh and Mishra give the following example to show that non-observable variables may be necessary in actual typings:

Exercise 22 (Fuh and Mishra)
Consider the typing:

$$\{\beta \leq \alpha, \beta \leq \delta, \gamma \leq \delta\}, \emptyset \vdash \lambda f.\lambda x.fst(fx, \lambda g.(f(gx), g(fx))) : (\alpha \to \gamma) \to (\beta \to \gamma)$$

Show that the typing is in G-normal form. Is it also G-minimal? □

Exercise 23 Show that the coercion set C in Example 3.3.4 is in G-normal form. □

From Exercise 23 we can conclude that some judgements in G-normal form are not G-minimal, and so we see that G-normalization really is only an approximation.

A key to an efficient implementation of G-reduction is the following property, exploited by Fuh and Mishra:

Lemma 3.3.6 *If $\alpha \leq_G A$ wrt. C and $C' = C\{\alpha \mapsto A\}$, then $above_{C'}(A') = above_C(A') \setminus \{\alpha\}$ and $below_{C'}(A') = below_C(A') \setminus \{\alpha\}$ for all $A' \in C'$.*

PROOF We show just the first equation, the other is shown in analogous way. Let $A' \in C'$. To see that $above_{C'}(A') \subseteq above_C(A') \setminus \{\alpha\}$, suppose $A'' \in above_{C'}(A')$. Then, since $A' \not\equiv \alpha$, we have $A'' \not\equiv \alpha$ and $C' \vdash_P A' \leq A''$. But then, since $C \vdash_P C'$ by Lemma 3.3.5, we get $C \vdash_P A' \leq A''$, hence $A'' \in above_C(A')\setminus\{\alpha\}$. To see that $above_{C'}(A') \supseteq above_C(A')\setminus\{\alpha\}$, suppose $A'' \in above_C(A')\setminus\{\alpha\}$. Then $C \vdash_P A' \leq A''$, hence $C\{\alpha \mapsto A\} \vdash_P A' \leq A''$, because $\alpha \not\equiv A'$ by $A' \in C'$. □

This property shows that one can maintain the sets *above* and *below* "incrementally" across substitutions, without recomputing the sets from scratch.

G-normalization can be performed in time, $\mathcal{O}(|C|^4)$. We first compute, for each atom A in C, the sets $above_C(A)$ and $below_C(A)$; these sets can be computed in time $\mathcal{O}(|C|^3)$, from the transitive closure $(C \cup P)^*$. We assume arrays *above* and *below* indexed by atoms in C; we further assume that each set $below[A]$ and $above[A]$ is itself an array (a bitvector) indexed by atoms in C, with $below[A][A'] = 1$ iff $A' \in below_C(A)$. These arrays are initialized as just described. Let Types(C) denote the set of types appearing in C.

We then perform reduction, as follows:

```
G-SIMPLIFY($C$) =
    let ($chosen$, ($\alpha$, $A$)) = G-CHOOSE($C$)
    in
        if $chosen$ then
            for $A' \in$ Types($C$) do
                $above[A'] := above[A'] \setminus \{\alpha\}$;
                $below[A'] := below[A'] \setminus \{\alpha\}$
            end; (* for *)
            G-SIMPLIFY($C\{\alpha \mapsto A\}$)
        else return $C$
    end
```

with procedure G-CHOOSE defined as follows:

```
G-CHOOSE($C$) =
    for $(\alpha, A) \in (\mathrm{Intv} \times (P \cup \mathrm{FTV}(C))$ do
        if $above[\alpha] \setminus \{\alpha\} \subseteq above[A] \wedge$
           $below[\alpha] \setminus \{\alpha\} \subseteq below[A] \wedge$
           $\alpha \not\equiv A$
        then return ($true$, $(\alpha, A)$)
    end; (* for *)
    return ($false$, $(\_, \_)$)
```

Here the variable Intv is supposed to hold the set Intv(t) where t is the judgement being processed.

Procedure G-SIMPLIFY can be executed recursively at most $\mathcal{O}(|C|)$ times on input C, since FTV(C) shrinks at each call; the **for**-loop of G-SIMPLIFY can be computed in time $\mathcal{O}(|C|)$. Procedure G-CHOOSE loops $\mathcal{O}(|C|^2)$ times with a conditional test taking $\mathcal{O}(|C|)$ time. Procedure G-SIMPLIFY, then, controls $\mathcal{O}(|C|)$ executions of procedure G-CHOOSE each of the order $\mathcal{O}(|C|^3)$. In total, therefore, the algorithm runs in $\Theta(|C|^4)$. Algorithm G-SIMPLIFY is essentially the algorithm sketched by Fuh and Mishra [11], yet they claim that G-normalization can be performed in cubic time; this is due to the fact that Fuh and Mishra assume a bit-string implementation of the sets *above* and *below* such that testing the incusions in the conditional of procedure G-CHOOSE becomes a constant time operation. This technique will not, however, scale up and, in any case, it is not a legitimate assumption for the purposes of assessing asymptotic complexity.

It was claimed earlier that the method of Fuh and Mishra can be seen as an optimization of the algorithm MIN. That this is so is a consequence of the following property, which shows that the two algorithms are equivalent:

Lemma 3.3.7

$$C \vdash_P C\{\alpha \mapsto A\} \Leftrightarrow \alpha \leq_G A \textit{ wrt. } C$$

PROOF ($\Leftarrow$) Assume $\alpha \leq_G A$ and $A' \leq A'' \in C\{\alpha \mapsto A\}$. Then $C \vdash_P A' \leq A''$ follows from Lemma 3.3.5.

($\Rightarrow$) Assume $C \vdash_P C\{\alpha \mapsto A\}$. Suppose that $A' \in \mathit{below}_C(\alpha) \setminus \{\alpha\}$, so we have $C \vdash_P A' \leq \alpha$. Then $C\{\alpha \mapsto A\} \vdash_P A' \leq A$ (since $A' \not\equiv \alpha$), hence $C \vdash_P A' \leq A$ (because $C \vdash_P C\{\alpha \mapsto A\}$), hence $A' \in \mathit{below}_C(A)$. We have shown $\mathit{below}_C(\alpha) \setminus \{\alpha\} \subseteq \mathit{below}_C(A)$. In the same way one sees that $\mathit{above}_C(\alpha) \setminus \{\alpha\} \subseteq \mathit{above}_C(A)$, and we have $\alpha \leq_G A$ wrt. C. □

We end this section by considering the complexity of G-minimization. The following theorem, which is proven in [32] and [33], strengthens the suspicion of Fuh and Mishra [11] that G-minimization requires exponential search procedures. In more detail, the theorem below shows that, unless $\mathbf{P} = \mathbf{NP}$, then there can be no polynomial time procedure for G-minimization, *regardless of the partial order P of base types.* The basic idea of the proof is to exploit the result by Pratt and Tiuryn [29] (see also [38]) that the SSI-problem for *flat* inequalities over the fixed 4-element poset K_2 called *2-crown* [2] is **NP**-complete. This result is exploited by showing that, when given any set C of flat inequalities over K_2, one can construct, in polynomial time, a constant-free atomic coercion set D such that C is satisfiable in K_2 if and only if the condition $D \vdash_\emptyset \widetilde{C}$ holds, where $\widetilde{C}$ is the G-minimization of C. Since the condition $D \vdash_\emptyset \widetilde{C}$ can be computed in polynomial time (using Lemmalem:subtype-logic-cubic), the theorem follows from **NP**-completeness of the satisfiablity problem for K_2. The reader is referred to [32] and [33] for full details.

[2]The 2-crown is the poset with 4 elements 0,1,2,3 ordered by $0 < 1$, $2 < 1$, $0 < 3$, $2 < 3$

Theorem 3.3.8 *If* **P** $\neq$ **NP**, *then G-minimization cannot be computed in polynomial time for any partial order P.*

3.3.2 Deferring coercions

A coercion (occurrence) which occurs applied at the root of a term is called a *head coercion* (occurrence.) Lemma 3.1.9 shows that one can always remove subtype hypotheses which are only used to head-coerce a term without loosing principality under $\prec$. One way of thinking of this is that head-coercions can always be *deferred to the context of use*; the relation $\prec$ captures this by allowing the type of an expression to be *adapted* via the subtype logic (the second defining property for $\prec$.) Note that this is in contrast to the notion of weak instance.

This suggests a strategy for proof-transformations which help to *minimize* coercion sets: move as many coercions as possible to head-position and then eliminate those coercions which are only used at head-position. It may be instructive to see that one can transform the first typing of Exercise 14 into the second one by such a process, using the equational system of Figure 3.2.

Example 3.3.9 Consider the first typing of Exercise 14,

$$\{\delta \leq \alpha, \eta \leq \gamma, \beta \leq \upsilon\}, \emptyset \vdash \mathit{comp} : (\alpha \to \beta) \to (\gamma \to \delta) \to (\eta \to \upsilon)$$

The typing proof can be translated into the completion $\overline{\mathit{comp}}$:

$$\overline{\mathit{comp}} \equiv \lambda f.^{\alpha\to\beta}\lambda g.^{\gamma\to\delta}\lambda x.^{\eta}[\kappa^{\upsilon}_{\beta}](f\ \ [\kappa^{\alpha}_{\delta}](g\ \ [\kappa^{\gamma}_{\eta}]x))$$

Then, using equations of Figure 3.2, we have

$$\begin{array}{rcl}
\overline{\mathit{comp}} & = & \lambda f.^{\alpha\to\beta}\lambda g.^{\gamma\to\delta}\lambda x.^{\eta}([\kappa^{\alpha}_{\delta} \to \kappa^{\upsilon}_{\beta}]f\ \ (g\ \ [\kappa^{\gamma}_{\eta}]x)) \\
 & = & [(\kappa^{\alpha}_{\delta} \to \kappa^{\upsilon}_{\beta}) \to id](\lambda f.^{\delta\to\upsilon}\lambda g.^{\gamma\to\delta}\lambda x.^{\eta}(f\ \ (g\ \ [\kappa^{\gamma}_{\eta}]x))) \\
 & = & [(\kappa^{\alpha}_{\delta} \to \kappa^{\upsilon}_{\beta}) \to id](\lambda f.^{\delta\to\upsilon}\lambda g.^{\gamma\to\delta}[\kappa^{\gamma}_{\eta} \to id](\lambda x.^{\gamma}(f\ \ (g\ \ x)))) \\
 & = & [(\kappa^{\alpha}_{\delta} \to \kappa^{\upsilon}_{\beta}) \to id](\lambda f.^{\delta\to\upsilon}[id \to (\kappa^{\gamma}_{\eta} \to id)](\lambda g.^{\gamma\to\delta}\lambda x.^{\gamma}f\ \ (g\ \ x))) \\
 & = & [(\kappa^{\alpha}_{\delta} \to \kappa^{\upsilon}_{\beta}) \to id][id \to (id \to (\kappa^{\gamma}_{\eta} \to id))](\lambda f.^{\delta\to\upsilon}\lambda g.^{\gamma\to\delta}\lambda x.^{\gamma}f\ \ (g\ \ x))
\end{array}$$

Removing the head coercion, we get the typing

$$\emptyset, \emptyset \vdash \lambda f.^{\delta\to\upsilon}\lambda g.^{\gamma\to\delta}\lambda x.^{\gamma}f\ \ (g\ \ x) : (\delta \to \upsilon) \to (\gamma \to \delta) \to (\gamma \to \upsilon)$$

which is indeed a renaming of the second typing shown for *comp* in Exercise 14. $\square$

Observe that Exercise 14 and Example 3.3.9 show that Lemma 3.1.9 does *not* hold, when weak instance is used in the notion of principality. Hence, the suggested proof transformation for minimization does not work under the weak instance relation, since the transformation will not, in general, preserve principality under weak instance.

Fuh and Mishra [11] describe a transformation on typing judgements to eliminate coercions, which exploits the power of $\prec$; the idea is that some coercions can be "deferred" by using the second and third defining properties of $\prec$. The transformation is based on a study of the effect on a type variable α occurring in τ when τ is coerced to a supertype. This is formalized in the notion of *polarity*, defined as follows. For α, τ we define sets $p^+(\alpha,\tau)$, $p^-(\alpha,\tau) \subseteq \{\uparrow,\downarrow\}$, called the *expand polarity* and the *contract polarity*, respectively, of α wrt. τ. The idea is that, if $\uparrow\in p^+(\alpha,\tau)$ then α may have to be coerced to a supertype, whenever τ is coerced to a supertype; and if $\downarrow\in p^+(\alpha,\tau)$ then α may have to be coerced to a subtype, whenever τ is coerced to a supertype. The polarities are defined by mutual recursion, as follows

$$p^+(\alpha,\tau) = \begin{cases} \{\uparrow\} & \text{if } \alpha \equiv \tau \\ p^-(\alpha,\tau_1) \cup p^+(\alpha,\tau_2) & \text{if } \tau \equiv \tau_1 \to \tau_2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$p^-(\alpha,\tau) = \begin{cases} \{\downarrow\} & \text{if } \alpha \equiv \tau \\ p^+(\alpha,\tau_1) \cup p^-(\alpha,\tau_2) & \text{if } \tau \equiv \tau_1 \to \tau_2 \\ \emptyset & \text{otherwise} \end{cases}$$

We say that a variable α *occurs positively (negatively)* in a type τ if α can be reached from the root of τ by moving to the left of an arrow ($\to$) an even (odd) number of times. Note, then, that $\uparrow\in p^+(\alpha,\tau)$ iff α occurs positively in τ and $\downarrow\in p^-(\alpha,\tau)$ iff α occurs negatively in τ, $\downarrow\in p^+(\alpha,\tau)$ iff α occurs negatively in τ and $\uparrow\in p^-(\alpha,\tau)$ iff α occurs negatively in τ.

Given atomic coercion set C, type τ and variables α,β, we say that α is *S-subsumed* by β in C and τ, written $\alpha \leq_S \beta$, iff

1. *either* $C \vdash_P \beta \leq \alpha$ and $p^+(\alpha,\tau) = \{\uparrow\}$ and $below_C(\alpha) \setminus \{\alpha\} \subseteq below_C(\beta)$
2. *or* $C \vdash_P \alpha \leq \beta$ and $p^+(\alpha,\tau) = \{\downarrow\}$ and $above_C(\alpha) \setminus \{\alpha\} \subseteq above_C(\beta)$

Observe that, if $\alpha \leq_S \beta$, then $\alpha \leq_G \beta$, and so Lemma 3.3.5 holds for $\leq_S$ also.

If $t = C,\Gamma \vdash M : \tau$, with $dom(\Gamma) = \{x_1,\ldots,x_n\}$, then we define the type $type\_closure(t) = \Gamma(x_1) \to \ldots \to \Gamma(x_n) \to \tau$. We then define, following Fuh and Mishra [11], the reduction $\mapsto_S$ on typings:

$$C,\Gamma \vdash M : \tau \mapsto_S C',\Gamma' \vdash M : \tau' \text{ iff } \begin{cases} (1) & \alpha \leq_S \beta \text{ in } C \text{ and } type\_closure(C,\Gamma \vdash M : \tau) \\ (2) & C' = C\{\alpha \mapsto \beta\} \\ (3) & \Gamma' = \Gamma\{\alpha \mapsto \beta\} \\ (4) & \tau' = \tau\{\alpha \mapsto \beta\} \end{cases}$$

The soundness of the reduction $\mapsto_S$ comes from the following technical properties:

Lemma 3.3.10 *If $C \vdash_P \beta \leq \alpha$ then*

1. $\downarrow\notin p^+(\alpha,\tau) \Rightarrow C \vdash_P \tau\{\alpha \mapsto \beta\} \leq \tau$
2. $\downarrow\notin p^-(\alpha,\tau) \Rightarrow C \vdash_P \tau \leq \tau\{\alpha \mapsto \beta\}$

and if $C \vdash_P \alpha \leq \beta$ then

1. $\uparrow\notin p^+(\alpha,\tau) \Rightarrow C \vdash_P \tau\{\alpha \mapsto \beta\} \leq \tau$
2. $\uparrow\notin p^-(\alpha,\tau) \Rightarrow C \vdash_P \tau \leq \tau\{\alpha \mapsto \beta\}$

PROOF Proof is by induction on τ. We prove only the first part, since the proof of the second part is analogous. So assume $C \vdash_P \beta \leq \alpha$, $\downarrow\notin p^+(\alpha,\tau)$ and $\downarrow\notin p^-(\alpha,\tau)$. If $\tau \equiv \alpha$, then (1) holds, because we do have $\downarrow\notin p^+(\alpha,\alpha)$, but also $C \vdash_P \beta \leq \alpha$. And (2) holds trivially, because we have $\downarrow\in p^-(\alpha,\alpha)$.

If $\tau \equiv A$, $A \not\equiv \alpha$, the claim holds because $\tau\{\alpha \mapsto \beta\} \equiv \tau$.

If $\tau \equiv \tau_1 \to \tau_2$, we get from $\downarrow\notin p^+(\alpha,\tau)$ that $\downarrow\notin p^-(\alpha,\tau_1)$ and $\downarrow\notin p^+(\alpha,\tau_2)$. Then, by induction, we get $C \vdash_P \tau_1 \leq \tau_1\{\alpha \mapsto \beta\}$ and $C \vdash_P \tau_2\{\alpha \mapsto \beta\} \leq \tau_2$, from which claim (1) follows. The second claim (2) follows in analogous manner. □

Lemma 3.3.11 *Let $\tau \equiv \sigma_1 \to \ldots \sigma_n \to \sigma$, for a given type σ. If $\downarrow\notin p^+(\alpha,\tau)$ then*

1. $\downarrow\notin p^-(\alpha,\sigma_i)$ *for all* $i = 1 \ldots n$
2. $\downarrow\notin p^+(\alpha,\sigma)$

and if $\uparrow\notin p^+(\alpha,\tau)$ then

1. $\uparrow\notin p^-(\alpha,\sigma_i)$ *for all* $i = 1 \ldots n$
2. $\uparrow\notin p^+(\alpha,\sigma)$

PROOF The proof is by induction on n, and only the first part of the lemma is proven here, since the second part is analogous. For $n = 0$, we have $p^+(\alpha,\tau) = p^+(\alpha,\sigma)$, and the lemma holds. Assume now that $\tau \equiv \sigma_1 \to \tau_1$ with $\tau_1 \equiv \sigma_2 \to \ldots \to \sigma_n \to \sigma$. Then $p^+(\alpha,\tau) = p^-(\alpha,\sigma_1) \cup p^+(\alpha,\tau_1)$, hence $\downarrow\notin p^+(\alpha,\tau)$ implies $\downarrow\notin p^-(\alpha,\sigma_1)$ and $\downarrow\notin p^+(\alpha,\tau_1)$. By $\downarrow\notin p^+(\alpha,\tau_1)$, induction hypothesis applied to τ_1 gives that $\downarrow\notin p^-(\alpha,\sigma_i)$ for all $i = 2 \ldots n$ and $\downarrow\notin p^+(\alpha,\sigma)$. Since we already have $\downarrow\notin p^-(\alpha,\sigma_1)$, we have shown the claim. □

Now we can prove

Proposition 3.3.12 *If $t_1 = C, \Gamma \vdash M : \tau \mapsto_S C, \Gamma' \vdash M : \tau' = t_2$ then $t_1 \approx t_2$.*

PROOF We show the claim in the case where $t_1 \mapsto_S t_2$ by $\alpha \leq_S \beta$ in C and $\sigma \equiv \mathit{type\_closure}(t_1)$ because $C \vdash_P \beta \leq \alpha$ and $p^+(\alpha, \sigma) = \{\uparrow\}$; the alternative case is similar and left out here. By Lemma 3.3.11 we get $\downarrow \notin p^-(\alpha, \Gamma(x))$ for all $x \in \mathit{dom}(\Gamma)$. Since $C \vdash_P \beta \leq \alpha$ we then get $C \vdash_P \Gamma(x) \leq \Gamma(x)\{\alpha \mapsto \beta\}$, by Lemma 3.3.10. Similarly, Lemma 3.3.11 shows that $\downarrow \notin p^+(\alpha, \tau)$, from which we get $C \vdash_P \tau\{\alpha \mapsto \beta\} \leq \tau$ by Lemma 3.3.10. Since Lemma 3.3.5 holds for $\leq_S$, we have $C \vdash_P C\{\alpha \mapsto \beta\}$. We have shown that $t_2 \prec t_1$ (t_1 is an instance of t_2.) On the other hand, $t_1 \prec t_2$ is obvious, so we have shown $t_1 \approx t_2$ as desired. □

Example 3.3.13 A typing t of the form (written as completion)

$$t = K, \emptyset \vdash \lambda x_1^{\tau_1} \ldots \lambda x_n^{\tau_n}. [\kappa] M\{x_i \mapsto [\kappa_i] x_i\} : \tau_1 \to \ldots \to \tau_n \to \tau$$

with

$$K = \{\kappa_i : \tau_i \leadsto \tau_i\{\alpha \mapsto \beta\}\}_{i=1\ldots n} \cup \{\kappa : \tau\{\alpha \mapsto \beta\} \leadsto \tau\}$$

might (under suitable assumptions) be transformed under $\mapsto_S$ into the typing

$$\emptyset, \emptyset \vdash \lambda x_1^{\tau_1\{\alpha \mapsto \beta\}} \ldots \lambda x_n^{\tau_n\{\alpha \mapsto \beta\}}.M : \tau_1\{\alpha \mapsto \beta\} \to \ldots \to \tau_n\{\alpha \mapsto \beta\} \to \tau\{\alpha \mapsto \beta\}$$

Compare with Example 3.3.9. □

It is possible to show that every typing has a unique normal form under $\mapsto_S$, up to renaming and the equivalence ($\sim_P$) of coercion sets. This is done in [11], Theorem 5, to which the reader is referred for the details. The S-reduction can be implemented as G-reduction, only one needs to add a polarity check when searching for S-subsumable variables.

Exercise 24 Find the S-normal form of the first typing of *comp* shown in Exercise 14. Compare with Example 3.3.9. □

Looking at the equational theory in Figure 3.2 and Example 3.3.9 one may perhaps get the idea that all coercions can be moved to the head of every *linear completion*; a linear completion is a completion M such that $M^\diamond$ is a *linear lambda term*, which, in turn, is a lambda term in which every lambda-bound variable has *exactly one* occurrence in the body of the binding abstraction. One nice thing, wrt. the equational theory, about linear terms is that the non-linear rewrite rule [E3] in Figure 3.2 is always applicable to any abstraction in a linear completion. However, it appears to be impossible to find a rewrite system in the equational theory which would yield the result by a normal form property, and the matter requires some technical work. We devote the remainder of this section to the issue.

We prove that every linear, closed pure lambda term with a (principal) atomic typing has a (principal) atomic typing with *no* subtype hypotheses at all. The main idea of the proof is that we first prove the result for terms in β-normal form; the result will then follow from subject expansion (see below) for linear terms together with our previous results.

If κ is a coercion, define for $n \geq 0$ the coercion $\kappa^{(n)}$ by $\kappa^{(0)} = \kappa, \kappa^{(n+1)} = id \rightarrow \kappa^{(n)}$. If $R_1, \ldots, R_{n+1}$ is a series of terms and M is a term, define the term $(M\ R^{(n)})$, $n \geq 0$, by $(M\ R^{(0)}) = M, (M\ R^{(n+1)}) = ((M\ R^{(n)})\ R_{n+1})$. Then one has

Lemma 3.3.14 $\kappa^{(n+1)} = id \rightarrow \kappa^{(n)} = (id \rightarrow \kappa)^{(n)}$

PROOF By induction on $n \geq 0$, and for $n = 0$ we have $\kappa^{(1)} = id \rightarrow \kappa^{(0)} = (id \rightarrow \kappa)^{(0)}$, by definition. For the induction step, $n > 0$, we have $(id \rightarrow \kappa)^{(n)} = id \rightarrow (id \rightarrow \kappa)^{(n-1)} = id \rightarrow \kappa^{(n)} = \kappa^{(n+1)}$, where induction hypothesis was used at the second equation. □

Lemma 3.3.15 *1.* $E \vdash [\kappa](M\ R^{(n)}) = (([\kappa^{(n)}]M)\ R^{(n)})$

2. $E \vdash ((M\ R^{(n-1)})\ [\kappa]R_n) = (((([(\kappa \rightarrow id)^{(n-1)}]M)\ R^{(n-1)})\ R_n)$

PROOF The first part is shown by induction on $n \geq 0$, and for $n = 0$ we have $[\kappa](M\ R^{(0)}) = [\kappa^{(0)}]M = (([\kappa^{(0)}]M)\ R^{(0)})$, by definition. For the induction step, $n > 0$, we have

$$\begin{array}{rcll}
[\kappa](M\ R^{(n+1)}) & = & [\kappa]((M\ R^{(n)})\ R_{n+1}) & \\
 & = & [\kappa]((M\ R^{(n)})\ [id]R_{n+1}) & \\
 & = & (([id \rightarrow \kappa](M\ R^{(n)}))\ R_{n+1}) & \\
 & = & ((([(id \rightarrow \kappa)^{(n)}]M)\ R^{(n)})\ R_{n+1}) & \text{(by induction)} \\
 & = & ((([\kappa^{(n+1)}]M)\ R^{(n)})\ R_{n+1}) & \text{(by Lemma 3.3.14)} \\
 & = & (([\kappa^{(n+1)}]M)\ R^{(n+1)}) &
\end{array}$$

The second part follows from the first, since we have

$$E \vdash ((M\ R^{(n-1)})\ [\kappa]R_n) = (([\kappa \rightarrow id](M\ R^{(n-1)}))\ R_n\ )$$

□

Recall (from,e.g., [17], p.14) that the set βNF of lambda terms in β-normal form can be defined inductively, as follows:

1. every term variable x is in βNF
2. if $M_1, \ldots, M_n$ are in βNF, then so is $(x\ M^{(n)})$ for any variable x
3. if M is in βNF, then so is $\lambda x.M$

Call a completion M *simple* iff every coercion in M is either a head-coercion of M or is applied to a free variable of M. Then we have

Lemma 3.3.16 *Let M be a linear completion with $M^\diamond$ in βNF. Then there exists a simple completion M' such that $E \vdash M = M'$.*

PROOF By induction on $M^\diamond \in \beta NF$. If $M^\diamond$ is a variable, the matter is clear.

If $M^\diamond \equiv \lambda x.N$, we have, by induction, $E \vdash M = [\kappa](\lambda x.[\kappa']N')$, where we can assume that N' is head-coercion free and simple. Write $\lambda x.[\kappa']N' \equiv \lambda x.[\kappa']N''\{x \mapsto [\kappa'']x\}$, where we can assume w.l.o.g. that $[\kappa'']x$ is not coerced. Since $M^\diamond$ is *linear*, x has just a single occcurrence in N'', and therefore we have

$$E \vdash M = [\kappa][\kappa'' \to \kappa'](\lambda x.N'')$$

The completion $[\kappa][\kappa'' \to \kappa'](\lambda x.N'')$ is simple, and this shows the claim when $M^\diamond$ is an abstraction.

If $M^\diamond \in \beta NF$ is an application, we have

$$E \vdash M = ((((\,[\kappa_x]x)\ [\kappa_1]N_1)\ [\kappa_2]N_2) \ldots [\kappa_k]N_k)$$

by induction, where we can assume that all the N_i are simple. But then, using Lemma 3.3.15 (second part) repeatedly, we have

$$E \vdash M = (((([(\kappa_1 \to id)^{(0)}][(\kappa_2 \to id)^{(1)}] \ldots [(\kappa_k \to id)^{(k-1)}][\kappa_x]x)N_1)N_2) \ldots N_k)$$

which must be simple. □

Corollary 3.3.17 *Every closed, pure linear term in β-normal form which has a (principal) typing (under $\prec$) has a (principal) typing with empty coercion set.*

PROOF Let M be pure, linear, closed β-normal form with (principal) typing. By Lemma 3.3.16 any typing of a closed, linear term in β-normal form can be transformed into a typing which is provable by using only head-coercions, *i.e.*, there is a typing proof which uses rule [SUB] only at the root of the term. By Lemma 3.1.9 (first part), there is then a typing proof which uses no coercions at all, and the typing is principal if the original typing was principal. □

A type system has the *subject expansion* property (wrt. β-reduction) iff typings are closed under β-expansion,*i.e.*, iff, whenever M has a typing and $M' \to_\beta^* M$, then M' has the same typing. It is a property converse to the subject reduction property (Lemma 2.3.4) but unlike that property it does not hold in general.

Exercise 25 1. Show that atomic subtyping systems do not have the subject expansion property.

2. Are there terms M, M' with $M' \twoheadrightarrow_\beta^* M$, which both have a typing but no common typing?

$\square$

Subject expansion does hold, when we restrict to linear terms:

Lemma 3.3.18 *(Subject expansion for linear terms) If $C, \Gamma \vdash M : \tau$ and $M' \twoheadrightarrow_\beta^* M$ with M' linear, then $C, \Gamma \vdash M' : \tau$.*

PROOF We prove the subject expansion property with $M' \twoheadrightarrow_\beta M$ in a single step. The lemma follows from this by obvious induction.

To see that the lemma holds in the case of a single reduction step, assume first that M' is the redex which gets contracted, *i.e.*, $M' \equiv (\lambda x.N)P$ and $M \equiv N\{x \mapsto P\}$. Now, it is easy to verify that the subtype systems have the *subterm property* that, for any typing proof of a term, the proof must contain a subproof of a typing for every subterm of the given term. Since x has a *single* occurrence in M (by linearity of M'), there has to be (by subterm property) a *single* type τ' such that the typing proof of $C, \Gamma \vdash M : \tau$ assigns the type τ' to P; moreover, this can be done under assumptions C (since no subtype hypotheses can be discharged) and Γ (since no variable of P is bound in $N\{x \mapsto P\}$ by an abstraction in N.) But then we have

$$C, \Gamma \cup \{x : \tau'\} \vdash N : \tau$$

and

$$C, \Gamma \vdash P : \tau'$$

and hence

$$C, \Gamma \vdash (\lambda x.N)P : \tau$$

It is now easy to prove that the lemma holds for $M' \equiv C[R]$ with C context and R the contracted redex, by induction on the context. $\square$

Theorem 3.3.19 *Every closed, pure linear lambda term which has a (principal) typing has a (principal) typing with empty coercion set.*

PROOF Let M be closed, pure linear term with (principal) typing. By Lemma 3.0.3, M has a β-normal form M'. Since M is closed, linear, so is M', and since M has a (principal) typing, so has M' by the Subject Reduction lemma (Lemma 2.3.4.) It follows by Corollary 3.3.17 that M' has a (principal) typing with empty coercion set; the result now follows by Subject Expansion, Lemma 3.3.18. $\square$

3.4 Complexity of type inference

In this section we outline the the major results known about the computational complexity of type inference for atomic subtyping. It is beyond the scope of this introduction to give detailed proofs of these results but we do give an outline of the methods used, and the reader is referred to the literature cited for full details.

Throughout this section we assume *strong subtyping*, *i.e.*, only derivable judgements $C, \Gamma \vdash_P M : \tau$ with C *satisfiable* in P are considered to define well-typings. If $\Sigma = (P, B_P)$ is a signature and M is a term in the language of $\lambda_{\leq}(\Sigma)$, then we say that M is *typable over* Σ iff there exists a context Γ and a type τ such that $\emptyset, \Gamma \vdash_P M : \tau$ is a derivable judgement in the type system for $\lambda_{\leq}(\Sigma)$. Note that, by Lemma 2.4.1, M is typable over Σ if and only if there are C, Γ, τ with $C, \Gamma \vdash_P M : \tau$ derivable and C satisfiable in P.

The assumption of strong subtyping is motivated by the fact that this is the variant which has been most studied in the literature, and, in particular, the most significant results concerning the complexity of inference are based on the strong variant. This explains why, as we shall see, the complexity of the satisfiablity problems POSAT and SSI mentioned in Section 2.2 play such a prominent rôle in the complexity theoretic analysis of subtype inference.

Let us begin by clarifying what the computational problem is. Obviously, we are interested in the *type inference problem*: given a term M, compute a principal typing for M, or, if no such typing exists, report failure. However, as is often the case in complexity analysis, we prefer to work with *decision problems* which require a "yes" or "no" answer (see [27] for background information), and in our case this leads to consideration of the *typability problem*, abbreviated TYPABILITY and defined as follows:

- (TYPABILITY) *Given a signature* $\Sigma = (P, B_P)$ *and a* $\lambda_{\leq}(\Sigma)$ *term* M, *is* M *typable over* Σ*?*

There is a closely related decision problem where we not only supply a term M but also a type τ and ask whether there exists a context Γ such that $\emptyset, \Gamma \vdash_P M : \tau$ is derivable. However, even though this problem might seem easier than the problem TYPABILITY, it is in fact not so, since (as is easily seen) M is typable if and only if the term $\lambda x.\mathbf{K}xM$ can be given the type $\alpha \to \alpha$ (here $\mathbf{K}$ is the combinator $\lambda x.\lambda y.x$)

Let us also state from the outset what the major known results concerning the complexity of the TYPABILITY problem are:

1. The problem TYPABILITY is **PSPACE**-hard
2. The problem TYPABILITY is in **DEXPTIME**
3. The problem TYPABILITY is in **PTIME**, when the order P is required to be a lattice

From items 1 and 2 we can see that the TYPABILITY problem has not yet been completely classified, and it is an interesting open problem how to close the gap between the lower bound of item 1 and the upper bound of item 2.

These results have been established via results for the satisfiability problems (SSI) for subtype inequalities together with the *equivalence theorem*, established by Hoang and Mitchell [18], which states that the problems SSI and the problem TYPABILITY are polynomial time equivalent. This latter result is important, because it shows that the complexity of the typability problems can be studied as abstract combinatorial problems (the SSI problems), which mention nothing about lambda terms or type systems at all.

To see how the equivalence theorem is proved, first note that one direction is by now obvious, in that the TYPABILITY problem is evidently polynomial time reducible to the SSI problem; the proof of this is just algorithm $\mathcal{U}$, which produces, in time linear in the size of the term M, a constraint set C_M such that M is typable if and only if C_M is satisfiable: if M is typable, then, by completeness of $\mathcal{U}$, it is easy to show [3] that C_M is satisfiable in P, when C_M is given by $\mathcal{U}[\![M]\!] = C_M, \Gamma \vdash_P M : \tau$. This shows that upper bounds for the SSI problem yield upper bounds for the typability problem.

The other direction of the equivalence theorem, which is needed to transfer lower bounds from SSI to TYPABILITY is more complicated. We must show that, when given an arbitrary set C of subtype inequalities of the form $\tau \leq \tau'$, we can construct, in polynomial time, a lambda term M_C over some signature Σ such that C is satisfiable in P if and only if M is typable over Σ. We now give the construction of the terms M_C. Given a finite poset P we define the signature $\Sigma(P) = (P, B_P)$ with $B_P = \{c_b^f : b \to b \mid b \in P\} \cup \{c_b : b \mid b \in P\}$. For any type σ in $T_P(\mathcal{V})$, we simultaneously define a term M_σ and a context E_σ (a context E being a term with a hole, $[]$, in it) by induction in σ. If $FTV(\sigma) = \{\alpha_1, \ldots, \alpha_n\}$ then M_σ and E_σ will be built using free variables $u_1, v_1, \ldots, u_n, v_n$, as follows:

$$\begin{array}{lcl} M_{\alpha_i} & = & (v_i\ u_i) \\ E_{\alpha_i} & = & (v_i\ []) \\ \\ M_b & = & c_b \\ E_b & = & (c_b^f\ []) \\ \\ M_{\sigma\to\tau} & = & \lambda x.K M_\tau E_\sigma[x] \\ E_{\sigma\to\tau} & = & E_\tau[([]\ M_\sigma)] \end{array}$$

Here K is the usual combinator $K = \lambda x.\lambda y.x$. Given an arbitrary inequality $\zeta \leq \zeta'$ over P we define the term $[\zeta \leq \zeta']$ by

$$[\zeta \leq \zeta'] = E_{\zeta'}[M_\zeta]$$

[3]The proof can be obtained by composing the proofs of Theorem 2.3.7 (completeness of $\mathcal{U}$), Theorem 3.2.2 (completeness of $\mathcal{A}$) and Theorem 3.2.8 (completeness of $\mathcal{A}^S$.)

Given an arbitrary set C of subtype inequalities over P with $C = \{\zeta_i \leq \zeta_i'\}_{i=1\ldots m}$ and $FTV(C) = \{\alpha_1, \ldots, \alpha_n\}$ we construct the term M_C as follows:

$$\begin{aligned} M_C \;=\; & \lambda u_1 \ldots \lambda u_n. \\ & (\lambda v_1 \ldots \lambda v_n. \\ & \quad (\lambda x.[\zeta_1 \leq \zeta_1'])((\lambda x.[\zeta_2 \leq \zeta_2'])(\ldots((\lambda x.[\zeta_{m-1} \leq \zeta_{m-1}'])[\zeta_m \leq \zeta_m']) \ldots)) \\ &)I_1 \ldots I_n \end{aligned}$$

where $I_1, \ldots, I_n$ are n copies of $\lambda x.x$. One can show that M_C is typable over $\Sigma(P)$ if and only if C is satisfiable in P. The hardest part is to show that M_C typable implies C satisfiable; the reader interested in more details is referred to the paper [18] by Hoang and Mitchell.

Now, to see some of the basic features of the techniques used to establish super-polynomial lower bounds for SSI (which, to recall, become lower bounds for TYPABILITY via the equivalence theorem), let us first note that even atomic inequalities can give rise to very complicated computational problems. The first results concerning SSI-problems were given by O'Keefe and Wand in [41], where they showed that the problem POSAT is **NP**-complete, by reduction from SAT (satisfiability of propositional formulae in conjunctive normal form, see [13] and [27] for background information.) Recall that POSAT is the problem

- (POSAT) *Given a finite poset P and a set C of atomic inequalities over P, is C satifiable in P?*

However, it turns out (by results given by Pratt and Tiuryn in [29]) that even the seemingly less difficult problem P-SAT defined below, where a specific partial order P is held fixed, can become **NP**-complete if the poset P is complicated enough:

- (P-SAT) *Given a set C of atomic inequalities over the fixed finite poset P, is C satisfiable in P?*

Pratt and Tiuryn [29] have shown that one can find surprisingly small posets P such that the problem P-SAT becomes **NP**-hard; so "complicated" above does not refer to the size of P but rather to the order relation of P. In fact, the smallest poset P for which P-SAT is **NP**-hard has only 4 elements! The intuitive reason why such a small poset can give rise to a complicated problem is that 4 elements are just enough to build a "non-deterministic structure" into the order relation of the poset, and this in combination with the expressive power of atomic inequalities yields a hard problem.

Before giving more details of the results of [29], let us first try to appreciate the expressive power of order relations and inequalities. We do this by showing that, if we allow the poset P to be sufficiently "odd", then it is really not too difficult to see that P-SAT problems can become **NP**-hard.

The idea here is that we can design a poset P in such a way that a simple reduction to show **NP**-hardness of P-SAT becomes possible. While this yields a particularly simple proof of **NP**-hardness, the drawback is that the poset P is not very "natural", because it is taylored to make the reduction work smoothely.

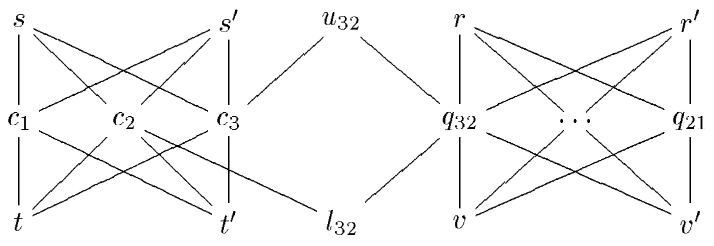
We define our "odd" poset P and show that P-SAT is **NP**-hard for this poset by reduction from the 3-colorability problem (see [13] for background information):

- (3-COL) *Given a graph $G = (V, E)$, is G 3-colorable, i.e., does there exist a "coloring function" $c : V \to \{1, 2, 3\}$ such that $c(u) \neq c(v)$ whenever $(u, v) \in V$?*

In order to reduce from the 3-colorability problem, we must show that, when given a graph $G = (V, E)$, we can construct, in polynomial time, an atomic constraint set C_G such that C_G is satisfiable in P if and only if G is 3-colorable. All we need to do so is to be able to express, for two arbitrary variables α, β, that α and β may take arbitrary *different* values from a set of three elements representing the three colors. Given the ability to express this by a constraint set $\Delta(\alpha, \beta)$, we can encode the 3-colorability problem for a graph $G = (V, E)$ by taking the constraint set

$$C_G = \bigcup_{(\alpha,\beta)\in E} \Delta(\alpha, \beta)$$

Let P be the following poset:



The elements c_1, c_2, c_3 will represent the 3 colors. The elements shown as $q_{32} \ldots q_{21}$ are a list of 6 "choice codes", q_{ij} for $i, j \in \{1, 2, 3\}$, $i \neq j$. For each element q_{ij} there are elements u_{ij} and l_{ij} ordered by $q_{ij} \leq u_{ij}$, $c_i \leq u_{ij}$, $l_{ij} \leq q_{ij}$, $l_{ij} \leq c_j$ (only the elements u_{32}, l_{32} are shown in the figure, the others were left out for readability.) Finally, the elements s, s', t, t' and r, r', v, v' are there to express set membership in the sets $\{c_1, c_2, c_3\}$ and $\{q_{ij}\}$, respectively. We are now ready to define the constraint set $\Delta(\alpha, \beta)$. It has three parts $\Delta_1(\alpha, \beta), \Delta_2(\alpha, \beta), \Delta_3(\alpha, \beta)$, as follows:

$$\Delta_1(\alpha, \beta) = \{\alpha \leq s, \alpha \leq s', \alpha \geq t, \alpha \geq t', \beta \leq s, \beta \leq s', \beta \geq t, \beta \geq t'\}$$

Clearly, $\Delta_1(\alpha, \beta)$ is satisfied, just in case α and β take values in the set of colors $\{c_1, c_2, c_3\}$. The next set, $\Delta_2(\alpha, \beta)$, uses a fresh variable, γ, which is constrained to take its value among the color codes q_{ij}:

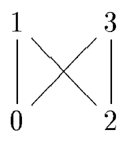
$$\Delta_2(\alpha, \beta) = \{\gamma \leq r, \gamma \leq r', \gamma \geq v, \gamma \geq v'\}$$

Finally, we impose the constraint that there must be an upper bound, δ (fresh variable) for α and γ and a lower bound, ϵ (fresh variable) for β and γ, as follows:

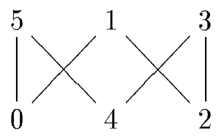
$$\Delta_3(\alpha, \beta) = \{\alpha \leq \delta, \gamma \leq \delta, \beta \geq \epsilon, \gamma \geq \epsilon\}$$

Taking $\Delta(\alpha, \beta) = \bigcup_{i=1}^{3} \Delta_i(\alpha, \beta)$, we see that the desired effect is obtained: the variable γ can take any value among the q_{ij}, but once the value of γ is chosen, $\gamma = q_{ij}$, the values of α and β get fixed to be $\alpha = c_i$, $\beta = c_j$. This is so, because δ being an upper bound of $\alpha \in \{c_1, c_2, c_3\}$ and $\gamma = q_{ij}$ can only be chosen to be $\delta = u_{ij}$ and similarly $\epsilon = l_{ij}$ is forced by the choice of $\gamma = q_{ij}$; hence $\alpha \leq u_{ij}$ and $\beta \geq l_{ij}$ must hold, and this forces $\alpha = c_i$, $\beta = c_j$. Since γ can be chosen arbitrarily among the color codes q_{ij} $(i \neq j)$, it follows that α and β can be valuated to *any* of two *different* colors c_i, c_j. Since C_G is evidently polynomial time constructible, this finishes the reduction.

We now turn to the results of Pratt and Tiuryn (shown in [29]) which were mentioned earlier. They concern a kind of posets called *n-crowns*. For each $n \geq 2$, an n-crown is a poset with $2n$ elements $0, 1, \ldots, 2n-1$ ordered by setting $2i \leq 2i+1$ and $2i \leq 2i-1$, where counting is modulo $2n$ (so, for instance, -1 denotes $2n-1$ and $2n-1 \leq 0$ holds.) To illustrate, the 2-crown can be drawn like this:



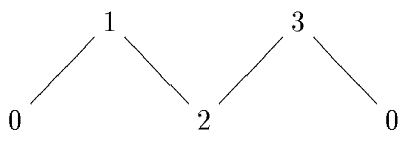
and the 3-crown can be drawn like this:



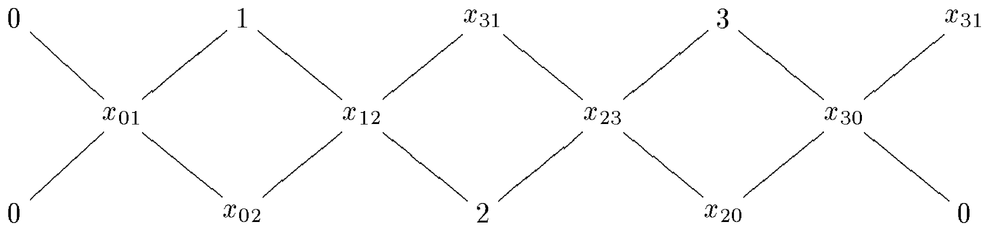
The remarkable result shown in [29] concerning this kind of posets is that for *every* $n \geq 2$, the problem *n-crown*-SAT is **NP**-complete; in particular, this holds for the 4 element poset 2-crown. The result is proven by reductions from SAT. We sketch the reduction for the case $n = 2$, and we refer the reader to [29] for the other cases. [4]

[4]The case $n = 2$ is treated slightly differently and in a more simple manner than the other cases; however, underlying all the reductions is a common pattern, which explains why it is possible to prove hardness for infinitely many n.

For the purpose of giving the reduction for $n = 2$ it is helpful to present the 2-crown in "unrolled" form, with the element 0 repeated, as shown below:



An instance of SAT is a set of *clauses*, each of the form $\xi_1 \vee \xi_2 \vee \xi_3$ where the ξ_i are *literals*, *i.e.*, either a propositional variable x or a negated variable, $\neg x$. To each variable x we associate a constraint set C^x over 2-crown, as follows:



where the x_{pq} are variables in the constraint set, associated with propositional variable x. Interestingly, any constraint set C^x must be *bivalent* in the sense that each variable x_{pq} in it can take exactly one of two values, either p or q, under a satisfing valuation of C^x in 2-crown. In fact, one can show by case analysis:

Exercise 26 Show that the set C^x has exactly two solutions in 2-crown, namely

1. the valuation ρ with $\rho(x_{pq}) = p$ for all x_{pq}, and
2. the valuation ρ with $\rho(x_{pq}) = q$ for all x_{pq}

$\square$

Now, for a given clause $\xi_1 \vee \xi_2 \vee \xi_3$ we translate the literal ξ_i to variable $[\![\xi_i]\!]$ as follows:

$$[\![\xi_1]\!] = \begin{cases} x_{12} \text{ if } \xi_1 = x \\ x_{01} \text{ if } \xi_1 = \neg x \end{cases}$$

$$[\![\xi_2]\!] = \begin{cases} x_{30} \text{ if } \xi_2 = x \\ x_{23} \text{ if } \xi_2 = \neg x \end{cases}$$

$$[\![\xi_3]\!] = \begin{cases} x_{20} \text{ if } \xi_3 = x \\ x_{02} \text{ if } \xi_3 = \neg x \end{cases}$$

We then add to the crowns C^x the additional constraints for each clause $\xi_1 \vee \xi_2 \vee \xi_3$:

$$\begin{tikzcd}
 & u \arrow[dl, dash] \arrow[dr, dash] & & v \arrow[dl, dash] \arrow[dr, dash] & \\
\llbracket \xi_1 \rrbracket & & w \arrow[dl, dash] \arrow[dr, dash] & & \llbracket \xi_2 \rrbracket \\
 & 0 & & \llbracket \xi_3 \rrbracket &
\end{tikzcd}$$

Let us call variables of the form x_{12}, x_{30}, x_{20} *positive constraint variables* and variables of the form x_{01}, x_{23}, x_{02} *negative constraint variables.* Given a valuation ρ of the variables x_{pq} in 2-crown, we interpret ρ as a truth valuation of the literals as follows: if x_{pq} is associated with literal ξ, then we set up the correspondence: for positive variables, we set

$$\begin{array}{lcl} \rho(x_{pq}) = p & \Leftrightarrow & \xi = \text{false} \\ \rho(x_{pq}) = q & \Leftrightarrow & \xi = \text{true} \end{array}$$

and for negative variables we set

$$\begin{array}{lcl} \rho(x_{pq}) = p & \Leftrightarrow & \xi = \text{true} \\ \rho(x_{pq}) = q & \Leftrightarrow & \xi = \text{false} \end{array}$$

One can then verify, by case analysis, that a valuation ρ, which solves the constraint set associated with an instance of SAT, defines a satisfying truth assignment for that instance, via the correspondence shown above; and, conversely, any truth assignment for the instance of SAT defines a valuation satisfying the constraints associated with the instance of SAT, via the same correpondence. For instance, if ξ_1 and ξ_2 are false, then $u = 1$ and $v = 3$ becomes forced, which in turn forces $w = 0$, hence $\llbracket \xi_3 \rrbracket$ is forced to take the value 0, which means that ξ_3 is true. On the other hand, if ξ_1 is true, then $u = v = w = 3$ is possible, and this removes all constraints on ξ_2 and ξ_3.

The constraint set associated with a set of clauses has size polynomially bounded by the size of the set. This proves **NP**-hardness of the problem 2-*crown*-SAT.

These results concerning satisfiability over n-crowns are exploited by Tiuryn in [38] to show **PSPACE**-hardness of the SSI-problems where the poset is fixed to be any n-crown. The proof is a complicated reduction from the problem QBF, satisfiability of quantified boolean formulae (see [13].) The reduction exploits the logical interpretation of constraint sets over n-crowns sketched above. A key idea of the proof is to represent quantifier elimination by expansion of constraint sets under most general matching substitutions; this leads to the possibility of succinctly representing, in a constraint set of subtype inequalities, a complete valuation tree, over the domain $\{0, 1\}$, of exponential size for a boolean formula. This, in turn, leads to the possibility of representing boolean quantifiers. For more details we refer to [38].

The proof that the SSI problem is in **DEXPTIME** is non-trivial and was given by Tiuryn and Wand [39]. The result holds even in the presence of recursive types. The proof is by reducing the satisfiability problem to the emptyness problem for Büchi atomata.

Finally, the important result that the typability problem is in **PTIME** when P is a lattice was shown by Tiuryn in [38] and exploits Lemma 2.2.7. The key step is to show that a property similar to Lemma 2.2.7 holds for non-atomic inequalities: by closing C under transitivity, uses of axioms and the rule

$$\frac{C \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2}{C \vdash \sigma_1 \leq \tau_1, \tau_2 \leq \sigma_2}$$

we obtain a set C^*, and one can show that C is satisfiable in a lattice P, if and only if it is unifiable and, moreover, whenever $b \leq b'$ is in C^*, then $b \leq b'$ holds in P.

Chapter 4

Minimal Typings

We have seen, in Section 2.5, how subtyping raises the issue of optimizing the presentation of typings. We have followed up this problem, in Section 3.3, by studying various typing transformations aimed at simplifying typings; in particular, we considered cycle elimination and the G- and S-transformations defined by Fuh and Mishra in [11]. In this chapter we take a more theoretical look at the problem of simplification. Our motivation will be the following two fundamental questions:

1. *What is an optimal presentation of a typing?*
2. *How big can principal typings become in the worst case, no matter how clever we are at simplifying them?*

As for the first question, agreeing that optimization of the presentation of subtypings is important, we have studied techniques for simplification. However, we have not answered a very fundamental question: *is the problem of optimizing typings mathematically well defined?* As in most other optimization problems, we should wish to have a well-defined notion of optimum, and we would like to be sure that an optimal solution is guaranteed to exist, and preferably a unique one. In the presence of such a notion, at least we know what we are ultimately looking for, and even though it may turn out to be infeasible (for reasons of computational complexity, decidability etc.) to always attain the optimum, we have a chance to measure our tools and results against an immutable yardstick. But in fact, we haven't even defined what it means for a typing to be *optimal*, and we have so much the less given any guarantee that optimal typings always exist.

As for the second question, given that we can find a suitable notion of optimality, what do optimal solutions look like? Here, as in many other branches of computing theory, we are interested in establishing *lower bounds* which tell us how pessimistic (or optimistic) we should be in the worst case. In our setting, this translates to the second question mentioned above.

The remainder of this chapter is based on [32] and [33] to which the reader is referred for reference. The reader interested in the material here should also consult [18], which gave a linear lower bound for the size of constraint sets in most general typings associated with *any* sound instance relation.

4.1 Existence and uniqueness of minimal typings

4.1.1 Soundness of typing transformations

The goal of subtype simplification, as considered in this report, is to transform judgements into more desirable forms. As we have seen, a key technique here is to eliminate variables from coercion sets. However, we require that such transformations *preserve the information content of typings.* One way, which we have adopted throughout here, of defining the information content of a typing, is by defining a notion of *instance* and taking the information content of a typing to be the set of its instances. In our setting, this leads to the requirement that, if $\mapsto$ is a transformation on judgements and t and t' are judgements such that $t \mapsto t'$, then $t \approx t'$ must hold. The property $t \mapsto t' \Rightarrow t \approx t'$ can be regarded as a soundness property for $\mapsto$, which guarantees that the transformation looses no typing power; in particular, a sound transformation will preserve principality of typings.

Other notions of soundness are possible. For instance, one can imagine more powerful notions of soundness based on semantic concepts rather than the syntactic notion of instance used here [1] By "more powerful" we mean here "more admissible" in the sense that more typings can be regarded as equivalent with respect to information content. The instance relation $\prec$ is interesting as an object of study, though, because it is natural and powerful enough to validate many non-trivial typing transformations. Its definition fully exploits the *logical* [2] rules of the type system. This is in contrast with the notion of instance used in [26], which uses the more restrictive conditions $S(\tau_1) = \tau_2$ (instead of condition (2) of Definition ??) and $\Gamma_2(x) = S(\Gamma_1(x))$ (instead of condition (3) of Definition ??.) To recall from Section 2.5, using this notion of instance we cannot validate quite simple typing transformations such as, e.g., cycle elimination in coercion sets. For example, it is natural to consider the typing $\{\alpha \leq \beta, \beta \leq \alpha\}, \emptyset \vdash_P \lambda x.x : \alpha \to \beta$ equivalent to (the more desirable) $\emptyset, \emptyset \vdash_P \lambda x.x : \alpha \to \alpha$, but since $\alpha \to \beta$ is not a substitution instance of $\alpha \to \alpha$, the two typings are not equivalent with

[1]See also [18] where a *generic* concept of instance is defined and studied. The introduction of this concept is an attempt to capture the properties which should be satisfied by any notion of instance.

[2]A typing rule is logical if it *non-structural*, that is, it can be applied to a term independently of the syntactic structure of the term. The rule [*sub*] is the only logical typing rule of the subtype system considered here.

respect to the more restrictive notion of instance.

In the remainder of this section we first define what it means for a typing judgement to be *minimal*. We then prove that minimal typings (if they exist) are unique up to renaming substitutions. We then prove that minimal typing judgements always exist. The hard part is to show existence.

4.1.2 Minimality and uniqueness of minimal typings

For a typing judgement t we let $[t]$ denote the $\approx$-equivalence class of t, i.e., $[t] = \{t' \mid t' \approx t\}$. Recall also, from Section 2.3, the notation $t_1 \prec_S t_2$, meaning that t_2 is an instance of t_1 under substitution S.

Our definition of minimality is based on the following natural relation between typing judgements, called *specialization*.

Definition 4.1.1 (Specialization)
Let $t_1 = C_1, \Gamma_1 \vdash_P M : \tau_1, t_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$. We say that t_1 *is a specialization of* t_2, written $t_1 \lesssim t_2$, iff there exists a substitution S such that

1. $C_1 \subseteq S(C_2)$
2. $\tau_1 = S(\tau_2)$
3. $Dm(\Gamma_1) \subseteq Dm(\Gamma_2)$ and $\forall x \in Dm(\Gamma_1).\ \Gamma_1(x) = S(\Gamma_2(x))$

We may write $t_1 \lesssim_S t_2$ to signify that $t_1 \lesssim t_2$ under substitution S. □

Exercise 27 Show that, if $t \lesssim_S t'$, then $t \prec_{id} S(t')$. □

We now define what it means for a typing judgement t to be *minimal*. Intuitively, a minimal typing t is a *most specialized* element within $[t]$. Therefore, a minimal typing is a most *succinct* representative in its equivalence class.

Definition 4.1.2 (Minimality)
A typing judgement t is called *minimal* iff it holds for all $t' \in [t]$ that $t \lesssim t'$.
□

If $t = C, \Gamma \vdash_P M : \tau$ and S is a type substitution, we write $S(t) = S(C), S(\Gamma) \vdash_P M : S(\tau)$. If $t_1 = S(t_2)$ with S a renaming on $FTV(t_2)$, then we say that t_1 and t_2 are *congruent*, written $t_1 \cong t_2$.

Theorem 4.1.3 *(Uniqueness of minimal typings)*
The relation $\lesssim$ is a partial order up to renaming substitutions: if $t_1 \lesssim_{S_2} t_2$ and $t_2 \lesssim_{S_1} t_1$ then $t_1 \cong t_2$ with S_1 a renaming on t_1, S_2 a renaming on t_2 and $t_1 = S_2(t_2)$ and $t_2 = S_1(t_1)$.

PROOF Let $t_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$ and $t_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$. If $t_1 \lesssim_{S_2} t_2$ and $t_2 \lesssim_{S_1} t_1$ then, in particular, $C_1 \subseteq S_2(C_2)$ and $C_2 \subseteq S_1(C_1)$. Then we have $|S_2(C_2)| \leq |C_2| \leq |S_1(C_1)| \leq |C_1|$ (using the second inclusion) and hence (by the first inclusion) we have $C_1 = S_2(C_2)$; that $C_2 = S_1(C_1)$ follows by an analogous argument. But then, by composing substitutions, we get from $t_1 \lesssim_{S_2} t_2$ and $t_2 \lesssim_{S_1} t_1$ that

(1) $C_1 = S_2(S_1(C_1))$

(2) $\tau_1 = S_2(S_1(\tau_1))$

(3) $\forall x \in Dm(\Gamma_1).\, \Gamma_1(x) = S_2(S_1(\Gamma_1(x)))$

This shows that $S_2 \circ S_1$ is a renaming on t_1, hence S_1 is a renaming on t_1 and $t_2 = S_1(t_1)$; by a similar argument, S_2 is a renaming on t_2 with $t_1 = S_2(t_2)$. $\square$

4.1.3 Existence of minimal typings

The hard part is to show that minimal typings exist in every equivalence class $[t]$. Basically, it will turn out that we shall always find a minimal typing in $[t]$ by taking a *full substitution instance* (see definition below) of t within $[t]$ and then optimizing the representation of the coercion set of the full substitution instance. The fact that a judgement obtained in this way is minimal will follow from a uniqueness property of fully substituted judgements within the equivalence class $[t]$. Perhaps surprisingly, this property is rather difficult to establish, and it requires a good deal of technical work. The technical result we are aiming for is Proposition 4.1.10 below, which is the cornerstone of the existence proof.

Definition 4.1.4 If $t' = S(t)$, then we say that t' is a *substitution instance of t under S*, written $t \overset{S}{\mapsto} t'$. A typing judgement t is called *fully substituted* iff, whenever $S(t) \in [t]$, then S is a renaming on $FTV(t)$. A *full substitution instance* of a typing t is a substitution instance of t which is fully substituted. $\square$

Exercise 28 Let $t = C, \Gamma \vdash_P M : \tau$ and let S be an atomic substitution such that $S(C)$ is consistent with P. Show that $S(t) \prec t$ implies $S(t) \approx t$ under these conditions. $\square$

Lemma 4.1.5 *If $t \overset{S}{\mapsto} t'$ and $t' \in [t]$, then $S \mid_{FTV(t)}$ is an atomic substitution.*

PROOF Let $t = C, \Gamma \vdash_P M : \tau$. By the assumptions, $t' = S(t)$ is an atomic judgement, so $S \mid_{FTV(C)}$ is an atomic substitution. Since $t' \in [t]$, we have

$t \approx t'$, which entails (via the Match Lemma and the definition of $\approx$) that $S(\tau)$ matches τ and that $\Gamma(x)$ matches $S(\Gamma(x))$ for all $x \in Dm(\Gamma)$. It follows that $S\mid_{FTV(\tau)\cup FTV(\Gamma)}$ is an atomic substitution. We have now shown that $S\mid_{FTV(t)}$ is atomic. □

Exercise 29 Show that, for any typing judgement t, there exists a full substitution instance of t within $[t]$. (*Hint:* consider repeated application of non-renaming atomic substitutions to a typing, and apply Lemma 4.1.5.) □

Using the Substitution Lemma and the definition of $\prec_S$ it is not too difficult to show the following very useful lemma.

Lemma 4.1.6 *If* $t_1 \prec_{S_1} t_2$ *and* $t_2 \prec_{S_2} t_1$, *then* $S_1(t_1) \prec_{S_2} t_1$ *and* $S_2(t_2) \prec_{S_1} t_2$.

PROOF Let $t_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$, $t_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$. By the assumption, we have

(i) $C_2 \vdash_P S_1(C_1)$

(ii) $C_2 \vdash_P S_1(\tau_1) \leq \tau_2$

(iii) $Dm(\Gamma_1) \subseteq Dm(\Gamma_2)$ and $\forall x \in Dm(\Gamma_1).\ C_2 \vdash_P \Gamma_2(x) \leq S_1(\Gamma_1(x))$

and

(iv) $C_1 \vdash_P S_2(C_2)$

(v) $C_1 \vdash_P S_2(\tau_2) \leq \tau_1$

(vi) $Dm(\Gamma_2) \subseteq Dm(\Gamma_1)$ and $\forall x \in Dm(\Gamma_2).\ C_1 \vdash_P \Gamma_1(x) \leq S_2(\Gamma_2(x))$

By (i) and the Substitution Lemma we have

$$S_2(C_2) \vdash_P S_2(S_1(C_1)) \tag{4.1}$$

and so, by (iv) and (4.1), we get

$$C_1 \vdash_P S_2(S_1(C_1)) \tag{4.2}$$

Moreover, by (ii) and the Substitution Lemma we have

$$S_2(C_2) \vdash_P S_2(S_1(\tau_1)) \leq S_2(\tau_2) \tag{4.3}$$

hence, by (iv) and (4.3) we get

$$C_1 \vdash_P S_2(S_1(\tau_1)) \leq S_2(\tau_2) \tag{4.4}$$

and then, by (v) and (4.4)

$$C_1 \vdash_P S_2(S_1(\tau_1)) \leq \tau_1 \tag{4.5}$$

In analogous manner one obtains

$$C_1 \vdash_P \Gamma_1(x) \leq S_2(S_1(\Gamma_1(x))) \tag{4.6}$$

Then (4.2), (4.5) and (4.6) show that $S_1(t_1) \prec_{S_2} t_1$. The proof that $S_2(t_2) \prec_{S_1} t_2$ is symmetric and left out. □

Exercise 30 Show that, if $t_1 \prec_{S_1} t_2$ and $t_2 \prec_{S_2} t_1$, then $S_2(S_1(t_1)) \prec_{id} t_1$ and $S_1(S_2(t_2)) \prec_{id} t_2$. □

If $t_1 \prec_{S_1} t_2$ and $t_2 \prec_{S_2} t_1$ where S_1 is a renaming on $FTV(t_1)$ and S_2 is a renaming on $FTV(t_2)$, then we say that t_1 and t_2 are *equivalent under renaming* and we write $t_1 \approx^\bullet t_2$ in this case.

Lemma 4.1.7 *If $t_1 \approx t_2$ and t_1 and t_2 are fully substituted, then $t_1 \approx^\bullet t_2$.*

PROOF By $t_1 \approx t_2$ we have $t_1 \prec_{S_1} t_2$ and $t_2 \prec_{S_2} t_1$ for some substitutions S_1, S_2. By Lemma 4.1.6 we then have $S_1(t_1) \prec_{S_2} t_1$ and $S_2(t_2) \prec_{S_1} t_2$, hence $S_1(t_1) \in [t_1]$ and $S_2(t_2) \in [t_2]$. Since t_1 is fully susbtituted, it follows that S_1 is a renaming on $FTV(t_1)$ and since t_2 is fully substituted, it follows that S_2 is a renaming on $FTV(t_2)$. Then $t_1 \prec_{S_1} t_2$ and $t_2 \prec_{S_2} t_1$ establish that $t_1 \approx^\bullet t_2$. □

We now aim at strengthening the conclusion of Lemma 4.1.7. The driving motivation behind the following development is that we wish to answer the question: what can we say about typings t_1 and t_2 if we know that both are fully substituted and $t_1 \approx^\bullet t_2$? Answering this question is the main technical part of the result in this section, and it turns out to be somewhat subtle.

We say that an atomic coercion set C is *cyclic* if there are atoms $A_1, \ldots, A_n$, $n \geq 2$, with $A_1 = A_n$ and $A_1 \leq A_2, \ldots, A_{n-1} \leq A_n \in C \cup P$ and with $A_i \neq A_j$ for some $i, j \in \{1, \ldots, n\}$. The sequence $A_1, \ldots, A_n$ is called a *proper cycle*. Note that inequalities like $A \leq A$ (i.e., a "loop") can occur in an acyclic coercion set.

Exercise 31 Show that, if $t = C, \Gamma \vdash_P M : \tau$ is fully substituted with C consistent, then C is acyclic. □

The last two technical lemmas needed for proving Proposition 4.1.10 follows. The proof of the first one is a little tricky but tedious, so we leave it out (the reader interested in the details can look at [32].) The ptoof of the second one is illuminating, however, so we include it here.

Lemma 4.1.8 *Let C_1, C_2 be atomic coercion sets, and assume that $C_1 \vdash_P S_2(C_2)$ and $C_2 \vdash_P S_1(C_1)$ where S_i is a renaming on $FTV(C_i)$. Then $C_1 \sim_P S_2(C_2)$ and $C_2 \sim_P S_1(C_1)$.*

PROOF See [32]. □

The proof of our last technical lemma has a simple abstract combinatorial core, which it may be instructive to sketch before giving the proof in detail. Suppose $f : A \rightarrow A$ is a bijection of a finite set A onto itself, and suppose that $\leq$ is any binary relation on A (we use a very suggestive notation, $\leq$, but it could be any binary relation) such that f is "monotone" with respect to $\leq$, i.e., $a \leq a' \Rightarrow f(a) \leq f(a')$. Then it will be the case, for any $a \in A$, that if $f(a) \neq a$ and either $f(a) \leq a$ or $a \leq f(a)$, then A must contain a proper cycle with respect to $\leq$, i.e., there is a sequence of elements $a_1 < \ldots < a_n$ with $a_1 = a_n$, where $a < a'$ means that $a \leq a'$ and $a \neq a'$. To see this, consider that f injective and $f(a) \neq a$ imply $f^n(a) \neq f^{n-1}(a)$ for all n, and, moreover, assuming $f(a) \leq a$ (the other case where $a \leq f(a)$ is similar) we also have $f^n(a) \leq f^{n-1}(a)$ for all n, by "monotonicity" of f; in total we have $f^n(a) < f^{n-1}(a)$ for all n. Now consider the set $V = \{f^n(a) \mid n \geq 0\}$. Since $V \subseteq A$ is finite, there must be $i < j$ with $f^j(a) = f^i(a)$. Then $f^j(a) < f^{j-1}(a) < \ldots < f^i(a)$ constitutes a proper cycle in A.

In the sequel we use the following property several times: if C is a coercion set such that $C \vdash_P \alpha \leq A$ or $C \vdash_P A \leq \alpha$, where α is a variable and $\alpha \neq A$, then α must be mentioned in C, i.e., $\alpha \in FTV(C)$. Note that the assumption $\alpha \neq A$ cannot be dropped here, since $C \vdash_P \alpha \leq \alpha$ for all C and α by the reflexivity rule. We are now ready to prove:

Lemma 4.1.9 *Let S be a substitution and C an atomic coercion set with variable $\alpha \in FTV(C)$. Assume*

(*i*) *S is a renaming on $FTV(C)$*

(*ii*) $C \vdash_P S(C)$

(*iii*) *C is acyclic*

(*iv*) *either $C \vdash_P S(\alpha) \leq \alpha$ or $C \vdash_P \alpha \leq S(\alpha)$*

Then $S(\alpha) = \alpha$.

PROOF We prove that $S(\alpha) \neq \alpha$ together with (*i*), (*ii*) and (*iv*) imply that C is cyclic. We show the implication under the assumption that $C \vdash_P S(\alpha) \leq \alpha$; the proof of the implication under the alternative assumption $C \vdash_P \alpha \leq S(\alpha)$ is similar and left out.

So assume $\alpha \in FTV(C)$, $S(\alpha) \neq \alpha$, (*i*), (*ii*) and $C \vdash_P S(\alpha) \leq \alpha$. We must show that C is cyclic. We first show the following *claim*:

For all $n > 0$ one has:

(*a*) *The set $D_n = \{S^k(\alpha) \mid 0 \leq k \leq n\}$ satisfies $D_n \subseteq FTV(C)$*

(*b*) $S^n(\alpha) \neq S^{n-1}(\alpha)$

(*c*) $C \vdash_P S^n(\alpha) \leq S^{n-1}(\alpha)$

All three items are proven simultaneously by induction on $n > 0$; doing this is left as an exercise for the reader.

Now, to prove the lemma, consider the set $V = \{S^n(\alpha) \mid n \geq 0\}$. By property (*a*) of our *claim* above, we have $V \subseteq FTV(C)$, and therefore V is a finite set of variables with S an injection of V into itself (and, by finiteness of V, S is therefore a bijection of V onto itself.) Hence, by finiteness of V, there must exist $i < j$ such that $S^j(\alpha) = S^i(\alpha)$. By property (*b*) of the *claim* we have $S^j(\alpha) \neq S^{j-1}(\alpha)$, and by property (*c*) one easily sees that $C \vdash_P S^j \leq S^i$ whenever $i \leq j$. We have therefore established that, under the subtype assumptions C, we have

$$S^j(\alpha) < S^{j-1}(\alpha) < \ldots < S^i(\alpha)$$

with $S^i(\alpha) = S^j(\alpha)$ (and $S^j(\alpha) < S^{j-1}(\alpha)$ shorthand for $C \vdash_P S^j(\alpha) \leq S^{j-1}(\alpha)$ with $S^j(\alpha) \neq S^{j-1}(\alpha)$.) The sequence shown establishes that there is a proper cycle in C. □

Exercise 32 Prove the *claim* stated in the proof of Lemma 4.1.9. □

We are finally ready to prove the following important property. Again, the proof is illuminating an we give it in detail.

Proposition 4.1.10 *If* $t_1 \approx^\bullet t_2$ *and* t_1, t_2 *are both fully substituted, then there is a renaming* S *on* t_2 *such that*

(*i*) $C_1 \sim_P S(C_2)$

(*ii*) $\tau_1 = S(\tau_2)$

(*iii*) $Dm(\Gamma_1) = Dm(\Gamma_2)$ *and* $\forall x \in Dm(\Gamma_1).\ \Gamma_1(x) = S(\Gamma_2(x))$

PROOF By $t_1 \approx^\bullet t_2$, we know that $t_1 \prec_{S_1} t_2$ and $t_2 \prec_{S_2} t_1$ with S_1 a renaming on t_1 and S_2 a renaming on t_2. We claim that the proposition holds with $S = S_2$. Part (*i*) follows directly from Lemma 4.1.8 and the assumptions. Part (*ii*) and (*iii*) are similar, so we consider only part (*ii*).

To see that (*ii*) holds with $S = S_2$, one first shows, using Substitution Lemma and assumptions, that

$$C_1 \vdash_P S_2(S_1(\tau_1)) \leq \tau_1 \tag{4.7}$$

Now, we know from Lemma 4.1.6 that $S_2(S_1(t_1)) \prec_{id} t_1$, hence we have $S_2(S_1(t_1)) \approx t_1$. It therefore follows from the assumption that t_1 is fully susbtituted that

$$S_2 \circ S_1 \text{ is a renaming on } FTV(t_1) \tag{4.8}$$

Moreover, by $S_2(S_1(t_1)) \prec_{id} t_1$, we also have

$$C_1 \vdash_P S_2(S_1(C_1)) \tag{4.9}$$

We now *claim* that in fact we have

$$(*) \quad S_2(S_1(\tau_1)) = \tau_1$$

To see that $(*)$ is true, assume that

$$S_2(S_1(\tau_1)) \neq \tau_1 \tag{4.10}$$

Then there is a variable occurrence α in τ_1 such that

$$(S_2 \circ S_1)(\alpha) \neq \alpha \tag{4.11}$$

Let $(S_2 \circ S_1)(\alpha) = A$. Then, because of (4.7), the Decomposition Lemma entails that α and A must be comparable under C_1, and hence

$$\text{either } C_1 \vdash_P S_2(S_1(\alpha)) \leq \alpha \text{ or } C_1 \vdash_P \alpha \leq S_2(S_1(\alpha)) \tag{4.12}$$

Since α is a variable and $A \neq \alpha$, it must be the case that

$$\alpha \in FTV(C_1) \tag{4.13}$$

since otherwise α and A could not be comparable under C_1. Finally, by Exercise 31, we know that

$$C_1 \text{ is acyclic} \tag{4.14}$$

because t_1 is assumed to be fully substituted. Now, (4.8), (4.9), (4.12), (4.13) and (4.14) allow us to apply Lemma 4.1.9 to the substitution $S_2 \circ S_1$ on C_1, and the Lemma shows that we must have $S_2(S_1(\alpha)) = \alpha$. This contradicts (4.11), and so we must reject the assumption (4.10), and $(*)$ is thereby established.

Now, by $(*)$ together with $C_1 \vdash_P S_2(S_1(\tau_1)) \leq S_2(\tau_2)$ (which follows from assumptions and Substitution Lemma) we get that

$$C_1 \vdash_P \tau_1 \leq S_2(\tau_2) \tag{4.15}$$

We know (4.14) that C_1 is acyclic. But then $C_1 \vdash_P S_2(\tau_2) \leq \tau_1$ (which holds by the assumptions) and (4.15) show (using Lemma 2.1.8 and C_1 acyclic) that $\tau_1 = S_2(\tau_2)$, as desired. We have now shown property (*ii*) of the proposition. Property (*iii*) follows by the same reasoning. □

Having Proposition 4.1.10 it is now a short step to proving the existence theorem. The final technical preparation is the following:

Definition 4.1.11 (Operations on coercion sets)
Let C be an atomic coercion set. Then we define the following operations on C:

$$C^r = \{A \leq A \mid A \leq A \in C\}$$

$$C^P = \{b \leq b' \in C \mid b, b' \in P\}$$

$$C^* = \text{ the transitive closure of } C$$

$$C^\circ = C \setminus (C^r \cup C^P)$$

$$C^- = \bigcap\{C' \mid (C')^* = C^*\}$$

□

Recall that we may consider P as a coercion set, or equivalently, as a digraph. In case C is acyclic, the set C^- is called the *transitive reduction* of C (see [1] for a full treatment of transitive reduction of a digraph) and C^- satisfies $(C^-)^* = C^*$ and whenever $(C')^* = C^*$ then $C^- \subseteq C'$, hence C^- is the unique, minimal set which generates the transitive closure of C (see [1] for more details.) Observe also that, by the Decomposition Lemma, we have $C_1 \sim_P C_2$ for any two atomic coercion sets C_1, C_2 if and only if C_1 and C_2 prove the same set of atomic inequalities $A \leq A'$. Hence, comparing two atomic sets with respect to $\vdash_P$ reduces to comparing the sets with respect to atomic inequalities.

Exercise 33 Prove the following statements.

(1) If C_1 and C_2 are atomic and $(C_1)^r = (C_2)^r$, then $C_1 \sim_P C_2 \Leftrightarrow (C_1 \cup P)^* = (C_2 \cup P)^*$

(2) If C_1 and C_2 are atomic, then $C_1 \sim_P C_2 \Leftrightarrow C_1 \setminus (C_1)^r \sim_P C_2 \setminus (C_2)^r$

(3) If C_1 and C_2 are both atomic and consistent with P, then $C_1 \sim_P C_2 \Leftrightarrow (C_1)^\circ \sim_P (C_2)^\circ$

(4) If C_1 and C_2 are both atomic and consistent with P, then $C_1 \sim_P C_2 \Leftrightarrow ((C_1)^\circ)^* = ((C_2)^\circ)^*$

(5) If C is atomic, consistent with P and acyclic, then $C' \sim_P C \Rightarrow (C^\circ)^- \subseteq C'$

(6) If C is atomic and consistent with P, then $C \sim_P (C^\circ)^-$

□

Theorem 4.1.12 *(Existence of minimal typings)*
Let $t = C, \Gamma \vdash_P M : \tau$ be any atomic, consistent judgement, and let $S(t)$ be a full (atomic, consistent) substitution instance of t within $[t]$. Define $\tilde{t}$ by

$$\tilde{t} = ((S(C))^\circ)^-, S(\Gamma) \vdash_P M : S(\tau)$$

Then $\tilde{t}$ is a minimal judgement in $[t]$.

PROOF Clearly, $\tilde{t}$ is fully substituted, and moreover $\tilde{t} \approx t$ because $((S(C))^\circ)^- \sim_P S(C)$ by consistency of $S(C)$. Now let t' be an arbitrary atomic, consistent judgement in $[t]$, and write $t' = C', \Gamma' \vdash_P M : \tau'$. We must show that $\tilde{t} \lesssim t'$. Let $S'(t')$ be a full (atomic, consistent) substitution instance of t' within $[t'] = [t]$. Then $S(t) \approx S'(t')$, and hence (by Lemma 4.1.7 and Proposition 4.1.10) there exists a renaming R on $S'(t')$ such that

(1) $S(C) \sim_P R(S'(C'))$

(2) $S(\tau) = R(S'(\tau'))$

(3) $Dm(\Gamma) = Dm(\Gamma')$ and $\forall x \in Dm(\Gamma).\ S(\Gamma(x)) = R(S'(\Gamma'(x)))$

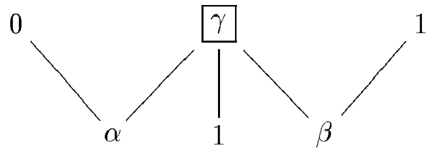
Since $S(C)$ must be acyclic (by Lemma 31) and $S(C)$ is consistent, we have $((S(C))^\circ)^- \subseteq R(S'(C'))$. We have shown $\tilde{t} \lesssim_{R \circ S'} t'$, thereby proving the theorem. □

4.2 Incompleteness of the G- and S-transformations

Since we now have an independent notion of minimal typings, we can meaningfully discuss completeness properties of transformations which are aimed at simplifying typings. We analyze the transformations defined by Fuh and Mishra in [11]. It turns out that they are incomplete with respect to minimization (computing minimal typings in the sense of this chapter.) However, we shall see that S-simplification *is* complete with respect to minimization when all variables in the coercion set are observable. This is an important property, which we shall exploit to prove a tight exponential lower bound on the size of principal typings in the final section of this chapter.

First we give an example which shows that G-*minimization* together with S-simplification are incomplete for *any* non-trivial poset P (we regard a poset trivial if it is discrete.)

Example 4.2.1 *(Incompleteness of G-minimization and S-simplification)* Let P be any non-discrete poset. Then there are two distinct elements $0, 1 \in P$ with $0 < 1$. Let C be the constraint set over P:



Here γ is observable and α, β are internal. Now, it is easy (and left for the reader) to verify that C is indeed *G-minimal*, since if S is any substitution on the internal variables of C, *i.e.*, with Supp$(S) \subseteq \{\alpha, \beta\}$, and S is not the identity on $FTV(C)$, then S cannot satisfy $C \vdash_P S(C)$; moreover, it is easy to verify that C is also S-simplified, since for no $A \neq \gamma$ do we have $\gamma \leq_S A$. However, with $S_{min} = \{\alpha \mapsto 0, \beta \mapsto 0, \gamma \mapsto 1\}$ we do have [3] $C \vdash_P S_{min}(C)$, because $S_{min}(C) = \{0, 1\}$ Hence, any typing t of the form $t = C, \emptyset \vdash_P M : \tau$ with $FTV(\tau) = \{\gamma\}$ and γ occurring positively (and not negatively) in τ, cannot be fully substituted and hence it cannot be minimal, since $S_{min}(t) \prec t$. $\square$

One lesson we can learn from Example 4.2.1 is that in minimization one cannot treat optimization of internals and observables seperately without loosing completeness. It may be instructive to see that we can easily realize, in a simple program, the situation described by the example. Assume we have typed functional constants $\texttt{succ} : \texttt{int} \to \texttt{int}$ and $\texttt{sqrt} : \texttt{real} \to \texttt{real}$; moreover, assume tuples of the form $\langle M_1, \ldots, M_k \rangle$ and projections $\#i$ which pick out the i'th component of a tuple. Let M be the term

$$\begin{aligned}\lambda f.\quad \#4\ \langle &\lambda x.\langle f\, x, \texttt{succ}\, x\rangle,\\ &\lambda y.\langle f\, y, \texttt{sqrt}\, y\rangle,\\ &\qquad\qquad f\, 1,\\ &\qquad\qquad\ 1\rangle\end{aligned}$$

Then we have $C, \emptyset \vdash_P M : (\alpha \to \beta) \to \texttt{int}$ with C as in Example 4.2.1, taking $0 = \texttt{int}$ and $1 = \texttt{real}$.

We now show a partial completeness result for S-simplification. In general, a typing transformation $\mapsto$ must be *sound* in the sense that we must have $t \approx t'$ whenever $t \mapsto t'$, and so, in order for a substitution S to define a sound transformation of a typing $t = C, \Gamma \vdash_P M : \tau$, we must require that $S(t) \prec t$. This implies in particular that $C \vdash_P S(\tau) \leq \tau$ (and a similar condition must hold for $\Gamma(x)$ and $S(\Gamma(x))$), hence if $\alpha \in \mathrm{Supp}(S) \cap \mathrm{Obv}(t)$, then α must be comparable to $S(\alpha)$ under C. This is a quite strong condition, and one may well ask whether S-simplification captures all sound transformations based on substituting into observable variables. In fact, S-simplification has an interesting completeness property which is established in the following proposition:

Proposition 4.2.2 *(Completeness of S-simplification wrt. observable variables)*
Let $t = C, \Gamma \vdash_P M : \tau$ and $\tau' = type\_closure(t)$. Assume S is not a renaming on $FTV(C)$ with

(1) $C \vdash_P S(C)$

(2) $C \vdash_P S(\tau') \leq \tau'$

(3) *Supp*$(S) \subseteq$ *Obv*(t)

[3] We could also take $\beta \mapsto 1$.

(4) *C is acyclic*

Then there exists $\alpha \in$ Supp(S) such that $\alpha \leq_S S(\alpha)$ with respect to C and τ'.

PROOF The proof is by contradiction, so suppose, under the assumptions of the proposition, that

$$\text{there is no } \alpha \in \text{Supp}(S) \text{ such that } \alpha \leq_S S(\alpha) \text{ with respect to } C \text{ and } \tau' \tag{4.16}$$

Pick any $\alpha \in \text{Supp}(S)$ (by assumption $\text{Supp}(S) \neq \emptyset$), so we have $\alpha \neq S(\alpha)$. By (2) and (3), α must be comparable to $S(\alpha)$ under C. Using these facts, it is easy to verify (by induction on τ', using (4) acyclicity of C) that (2) implies either $C \vdash_P S(\alpha) \leq \alpha$ with α not occurring negatively in τ', or else $C \vdash_P \alpha \leq S(\alpha)$ with α not occurring positively in τ'. Assume that we have (the alternative case is similar)

$$(*) \qquad C \vdash_P S(\alpha) \leq \alpha \text{ with } \alpha \text{ not occurring negatively in } \tau'$$

By (4.16) we must have $\downarrow_C(\alpha) \setminus \{\alpha\} \not\subseteq \downarrow_C(S(\alpha))$, since otherwise we should have $\alpha \leq_S S(\alpha)$. So there must be an atomic type $A_1 \in \downarrow_C(\alpha) \setminus \{\alpha\}$ such that

$$A_1 \notin \downarrow_C(S(\alpha)) \tag{4.17}$$

In particular, we therefore have

$$A_1 \neq S(\alpha), A_1 \neq \alpha \text{ and } C \vdash_P A_1 \leq \alpha \tag{4.18}$$

By the Substitution Lemma, we then have $S(C) \vdash_P S(A_1) \leq S(\alpha)$, and so, by (1), we also have

$$C \vdash_P S(A_1) \leq S(\alpha) \tag{4.19}$$

If $A_1 = S(A_1)$, then by (4.19) it would follow that $C \vdash_P A_1 \leq S(\alpha)$, hence $A_1 \in \downarrow_C(S(\alpha))$ in contradiction with (4.17). Therefore we must have

$$A_1 \neq S(A_1) \tag{4.20}$$

from which it follows that

$$A_1 \text{ is a variable in Supp}(S) \tag{4.21}$$

Now, by (4.21) and assumption (3), A_1 is an observable variable, occurring in τ'. Then (2) implies that A_1 and $S(A_1)$ must be comparable under the hypotheses C. We already know (4.20) that $A_1 \neq S(A_1)$, and if $C \vdash_P A_1 \leq S(A_1)$, then it would follow by (4.19) that $C \vdash_P A_1 \leq S(\alpha)$ in contradiction with (4.17). We must therefore conclude that

$$C \vdash_P S(A_1) \leq A_1 \text{ with no negative occurrence of } A_1 \text{ in } \tau' \tag{4.22}$$

Summing up so far, we now have the situation

$$\begin{tikzcd}
 & \alpha \arrow[dl, dash] \arrow[dr, dash] & \\
A_1 \arrow[dr, dash] & & S(\alpha) \arrow[dl, dash, dotted] \\
 & S(A_1) &
\end{tikzcd}$$

where the dotted line means "less than or equal to" and the full lines mean "strictly less than" with respect to C. But this, together with (4.22) shows that the situation described in $(*)$ now holds with A_1 and $S(A_1)$ in place of α and $S(\alpha)$, and all the reasoning starting from $(*)$ can therefore be repeated for A_1 and $S(A_1)$, leading, in particular, to the existence of an atom A_2 with $C \vdash_P A_2 \leq A_1$ and $A_2 \not\approx A_1$ etc. Hence, by repetition of the argument starting at $(*)$, we obtain an arbitrarily long strictly decreasing chain (with respect to C)

$$\alpha > A_1 > A_2 > \ldots$$

implying the existence of a proper cycle in $C \cup P$ and hence contradicting (4). We must therefore reject the assumption (4.16), thereby proving the Proposition. □

Proposition 4.2.2 implies that, whenever $S(t) \prec t$ with S not renaming on t and $\text{Supp}(S) \subseteq \text{Obv}(t)$, then t can be S-simplified. Hence, for a typing t which is normalized under S-simplification there exists no strict substitution instance $S(t) \approx t$ such that S affects only observable variables in t. Consequently, if $t = C, \Gamma \vdash_P M : \tau$ with $FTV(C) \subseteq \text{Obv}(t)$, and if t is S-simplified, then t must be fully substituted; by Theorem 4.1.12 we therefore have

Theorem 4.2.3 *If $t = C, \Gamma \vdash_P M : \tau$ is an S-simplified typing with $FTV(C) \subseteq Obv(t)$, then the typing $(C^\circ)^-, \Gamma \vdash_P M : \tau$ is minimal.*

Theorem 4.2.3 has a number of interesting applications. An important example occurs in Section 4.3 below, where we prove an exponential lower bound for the size of coercion sets in principal typings.

4.3 The size of principal typings

We prove a tight worst case exponential lower bound for the dag-size of coercion sets as well as of types in principal typings. To the best of our knowledge, the best lower bound previously proven is the linear lower bound for a generic instance relation shown in [18]. The basic idea of enforcing a blow-up in textual type size described for simply typed lambda calculus without subtyping ($\lambda^{\rightarrow}$) in [20] is present in our proof. However, the matter is much more complicated in subtyping, and our proof uses new techniques; moreover, the results of Section 4.1 as well as Theorem 4.2.3 are important ingredients.

For $\lambda^{\rightarrow}$ we know (see [20]) that while *textual* type size can be exponential, *dag-size* is at most linear. The basic property responsible for this is that $\lambda^{\rightarrow}$ is a purely equational type system, in the sense of [40], so that enough sharing becomes possible. ML is not purely equational, and with its universal quantifier and its succinct `let`-expressions, we get doubly exponential textual size of types and exponential dag-size, in the worst case (see [20] with further references.) Subtyping is a system based on *in*equalities, and, as shown below, this leads to the impossibility of sharing, resulting in exponential dag-size of typings (coercion sets as well as of types) due to the fact that exponentially many distinct variables may have to be present in a principal typing. Among other things, this result shows an intrinsic limit as to how much type-simplification techniques can possibly achieve for atomic subtyping, at least with instance relations not stronger than $\prec$.

The exponential lower bound proof has two core parts. One is the construction of a series of terms $\mathbf{Q}_n$, with the intention that, for all $n \geq 0$, any principal typing of $\mathbf{Q}_n$ has the form $C_n, \emptyset \vdash_P \mathbf{Q}_n : \tau_n$, where C_n contains more than 2^n distinct type variables and τ_n contains more than 2^n distinct type variables. The second main ingredient in the proof is Theorem 4.2.3 and the characterization of minimal typings of Section 4.1, which are employed in order to prove that the intended property in fact holds for the $\mathbf{Q}_n$, viz. that *any* principal typing of the terms $\mathbf{Q}_n$ must have a number of variables in both coercion set and type which is exponentially dependent on the size of the terms.

4.3.1 Preliminaries to the construction

For the purpose of the following development we shall first assume that we have a conditional construct *if* ... *then* ... *else* ... with the typing rule

$$[\mathit{if}]\quad \frac{C, \Gamma \vdash_P M : \mathtt{bool} \quad C, \Gamma \vdash_P N : \tau \quad C, \Gamma \vdash_P Q : \tau}{C, \Gamma \vdash_P \mathit{if}\ M\ \mathit{then}\ N\ \mathit{else}\ Q : \tau}$$

Moreover, we shall assume pairs $\langle M, N \rangle$ and pair-types $\tau * \tau'$ with the usual typing rule

$$[\mathit{pair}]\quad \frac{C, \Gamma \vdash_P M_1 : \tau_1 \quad C, \Gamma \vdash_P M_2 : \tau_2}{C, \Gamma \vdash_P \langle M_1, M_2 \rangle : \tau_1 * \tau_2}$$

together with projections `1st` and `2nd` using the standard typing rules

$$[\mathit{proj1}]\quad \frac{C, \Gamma \vdash_P M : \tau_1 * \tau_2}{C, \Gamma \vdash_P (\mathtt{1st}\ M) : \tau_1} \qquad [\mathit{proj2}]\quad \frac{C, \Gamma \vdash_P M : \tau_1 * \tau_2}{C, \Gamma \vdash_P (\mathtt{2nd}\ M) : \tau_2}$$

The subtype ordering is lifted co-variantly to pairs in the usual way, such that $\tau_1 * \tau_2 \leq \tau_1' * \tau_2'$ holds if and only if $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$ both hold.

These additional constructs are introduced here only because it is easier to understand the essence of the constructions to be given below using these constructs. Once we have completed the constructions, we shall show how to encode them in the "pure" lambda calculus with no additional constructs.

In the sequel, we shall often carefully consider the form of principal typings of terms. We use the fact, shown in [26], that a principal typing can always be obtained by the following *standard procedure*: first (1) extract subtyping constraints only at the *leaves* of the term (*i.e.*, coercions are applied to variables and constants only), and then (2) perform a *match-step* in which a most general matching substitution (see [26] for details) is applied to the extracted constraint set, and finally (3) *decompose* the matching constraints into atomic constraints (any matching set can be decomposed, and if the match-step fails, then the term has no typing with atomic subtyping at all.) Once steps (1) through (3) have been performed, we can then apply transformations such as G-minimization and S-simplification to simplify the constraint set, without loosing principality. We shall sometimes indicate the form of a typing derivation by *completions* which are terms with explicit subtyping coercions and type assumptions for bound variables. A coercion from τ to τ' applied to a term M will be written as $\uparrow_{\tau}^{\tau'} M$, so, for instance, the completion $\lambda x : \alpha.\ \uparrow_{\alpha}^{\beta} x$ encodes the typing judgement $\{\alpha \leq \beta\}, \emptyset \vdash_P \lambda x.x : \alpha \to \beta$ as well as a derivation of that judgement.

We know (see, e.g., [20]) that, in the simply typed lambda calculus without subtyping, we can create an exponential explosion in the *textual size* of the types of terms. The way to do this is quite obvious, since we can simply copy the argument of a function to the output, by using, say, the combinator $\mathbf{C} = \lambda z.\langle z, z\rangle$; if $\mathbf{C}^n M$ is the n-fold application of $\mathbf{C}$ to argument M, then (for $n \geq 1$) the type of $\mathbf{C}^n M$ will be the product type $\tau * \ldots * \tau$ with 2^n occurrences of τ, assuming that M has type τ. We see, though, that while textual type size can thus become exponentially dependent upon the size of terms, *dag* size is still linear, because the type consists of repetitions of the same type, so sharing introduces exponential succinctness in the dag-representation of the type. The proof that every principal typing in simply typed lambda calculus produces types of size linearly dependent on the size of terms is just the standard unification based implementation of simply typed lambda calculus, reflecting the fact that all type constraints are equational (see [40].) With subtyping, however, the matter gets more complicated, because subtyping is not an equational system, being based, as it is, on *in*equality constraints. In fact, it is easy (and left for the reader) to see that the trick used above for simple types will not yield anything new in the presence of subtyping, since G- and S-simplification will eliminate all variables from the constraint set, so that we get the principal typing $\emptyset, \emptyset \vdash_P \mathbf{C}^n M : \tau * \ldots * \tau$ with subtyping, assuming that τ is a principal type

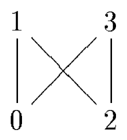


Figure 4.1: A 2-crown

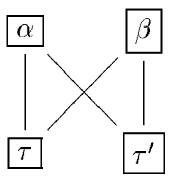


Figure 4.2: Minimal constraint set for $cond_{x,y}$

for M. Hence, we need some new ideas to prove exponential dag-size for subtyping.

In order to illustrate some of the principles that will be appealed to in the construction to be given later, we shall now consider some expressions and their principal typings. Similar expressions will be part of the construction to be given later. The first expression we shall consider is the expression $cond_{x,y}$ with two free variables x and y and defined by

$$cond_{x,y} = \mathit{if}\,\texttt{true}\ \mathit{then}\ \langle x, y\rangle\ \mathit{else}\ \langle y, x\rangle$$

Suppose this occurs in a context where the type τ has been assumed for x and the type τ' has been assumed for y. In case τ and τ' are atoms, it is easy to see (using the *standard procedure*) that the following completion represents a principal typing for $cond_{x,y}$:

$$\mathit{if}\,\texttt{true}\ \mathit{then}\ \langle \uparrow^{\alpha}_{\tau} x, \uparrow^{\beta}_{\tau'} y\rangle\ \mathit{else}\ \langle \uparrow^{\alpha}_{\tau'} y, \uparrow^{\beta}_{\tau} x\rangle$$

at type $\alpha * \beta$. This yields the atomic coercion set in Figure 4.2 (we show *observable variables* inside a box, as usual.) An interesting property of this constraint set is that it constitutes a *2-crown* of variables; recall (from Section 3.3) that a 2-crown is 4-element the poset with elements $0, 1, 2, 3$ ordered as shown in Figure 4.1. Now, 2-crowns are interesting, because any constraint set consisting of 2-crowns built from distinct observable types is easily seen to be S-simplified, and hence, by Theorem 4.2.3, any such set is minimal. In particular, the constraint set of Figure 4.2 is minimal with respect to the term $cond_{x,y}$ by this reasoning.

Suppose now that either of τ or τ' is not an atom but a type built from pairing and atoms. Then the constraint set in Figure 4.2 must be expanded

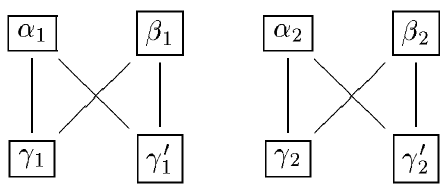


Figure 4.3: Constraint set of Figure 4.2 with $\tau = \gamma_1 * \gamma_2$, $\tau' = \gamma'_1 * \gamma'_2$ after matching expansion and decomposition

into a matching set, where both τ and τ' match both α and β (hence, in particular, τ and τ' must match each other.) This expansion composed with decomposition into atomic constraints will result in a number of 2-crowns of observable atoms, where the number of crowns equals the number of components in the pair-type τ (or, equivalently, in τ'.) For instance, if $\tau = \gamma_1 * \gamma_2$ and $\tau' = \gamma'_1 * \gamma'_2$, then the set of Figure 4.2 will expand into the two 2-crowns shown in Figure 4.3 under the matching substitution which maps α to $\alpha_1 * \alpha_2$ and β to $\beta_1 * \beta_2$ followed by decomposition into atomic constraints.

Now consider the term M defined by

$$M = \lambda x.\mathit{if}\, \mathit{true}\ \mathit{then}\ \langle x, \langle \texttt{2nd}\, x, \texttt{1st}\, x\rangle\rangle\ \mathit{else}\ \langle\langle \texttt{2nd}\, x, \texttt{1st}\, x\rangle, x\rangle$$

It is easy to verify that the following completion represents a principal typing of M:

$$\begin{array}{l}
\lambda x : \alpha * \beta.\mathit{if}\, \mathit{true} \\
\mathit{then}\ \langle \uparrow^{\alpha_1 * \beta_1}_{\alpha * \beta}\ x, \langle \uparrow^{\beta'_2}_{\beta_2} (\texttt{2nd}\ \uparrow^{\alpha_2 * \beta_2}_{\alpha * \beta}\ x), \uparrow^{\alpha'_3}_{\alpha_3} (\texttt{1st}\ \uparrow^{\alpha_3 * \beta_3}_{\alpha * \beta}\ x)\rangle\rangle \\
\mathit{else}\ \langle\langle \uparrow^{\alpha_1}_{\alpha'_1} (\texttt{2nd}\ \uparrow^{\alpha_4 * \alpha'_1}_{\alpha * \beta}\ x), \uparrow^{\beta_1}_{\alpha_5} (\texttt{1st}\ \uparrow^{\alpha_5 * \beta_4}_{\alpha * \beta}\ x)\rangle, \uparrow^{\beta'_2 * \alpha'_3}_{\alpha * \beta}\ x\rangle
\end{array}$$

at the type $\alpha * \beta \rightarrow (\alpha_1 * \beta_1) * (\beta'_2 * \alpha'_3)$. The coercion set corresponding to the coercions shown in the term G-simplifies into the double crown shown in Figure 4.4, using substitutions which map $\alpha_2, \alpha_3, \alpha_4$ and α_5 to α, α'_1 to α_1 and $\beta_2, \beta_3, \beta_4$ to β. Verification of this is easy and left for the reader. Furthermore, it is immediate that the constraint set shown in Figure 4.4 is S-simplified, and since everything is observable, it follows from Theorem 4.2.3 that the completion M' shown below, using the constraints of Figure 4.4, represents a minimal, principal typing of M:

$$\begin{array}{rcl}
M' & = & \lambda x : \alpha * \beta.\mathit{if}\, \mathit{true} \\
 & & \mathit{then}\ \langle \uparrow^{\alpha_1 * \beta_1}_{\alpha * \beta}\ x, \langle \texttt{2nd}\ \uparrow^{\alpha * \beta'_2}_{\alpha * \beta}\ x, \texttt{1st}\ \uparrow^{\alpha'_3 * \beta}_{\alpha * \beta}\ x\rangle\rangle \\
 & & \mathit{else}\ \langle\langle \texttt{2nd}\ \uparrow^{\alpha * \alpha_1}_{\alpha * \beta}\ x, \texttt{1st}\ \uparrow^{\beta_1 * \beta}_{\alpha * \beta}\ x\rangle, \uparrow^{\beta'_2 * \alpha'_3}_{\alpha * \beta}\ x\rangle
\end{array}$$

Now, in analogy with the constraint set for $\mathit{cond}_{x,y}$ shown in Figure 4.3, we see that if a type of the form $\tau_1 * \tau_2$ must be assumed for the variable x

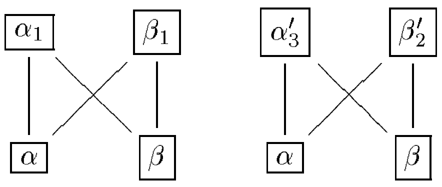


Figure 4.4: Constraint set in principal typing of M after G-simplification

in M (instead of $\alpha * \beta$), with τ_1, τ_2 recursively built by pairing of atoms, then the constraint set shown in Figure 4.4 with τ_1 substituted for α and τ_2 substituted for β will be valid for the principal typing of M, and, moreover, in that coercion set each crown of Figure 4.4 will expand to n_i new 2-crowns of observable atomic types once a matching substitution has been used and decomposition into atomic constraints has been performed, where n_i measures the number of atoms occurring in τ_i, $i \in \{1, 2\}$. If τ_1 and τ_2 are built by pairing from distinct variables, each with just a single occurrence in the type $\tau_1 * \tau_2$, the expanded 2-crowns will be S-simplified, and so it follows by the usual argument from S-normal form and Theorem 4.2.3 that the resulting, expanded coercion set will be valid for the *minimal* principal typing of M in a context in which the type $\tau_1 * \tau_2$ must be assumed for the variable x in M.

4.3.2 The construction

We proceed to give the construction. Let $cond_{x,y}$ denote the expression with two free variables x and y, given by

$$cond_{x,y} = \mathit{if}\,\texttt{true}\ \mathit{then}\ \langle x, y\rangle\ \mathit{else}\ \langle y, x\rangle$$

For a term variable f, define the expression $\mathbf{P}_f$ with free variable f as follows:

$$\begin{array}{rcl} \mathbf{P}_f & = & \lambda z.\mathbf{K} \\ & & (\mathit{if}\,\texttt{true}\ \mathit{then}\ \langle z, \langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle\rangle \\ & & \mathit{else}\ \langle\langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle, z\rangle) \\ & & (f\ z) \end{array}$$

where $\mathbf{K}$ is the combinator $\lambda x.\lambda y.x$. Let $f_1, f_2, \ldots$ be an enumeration of infinitely many distinct term variables, and let N be any expression; then define, for $n \geq 0$, the expression $\mathbf{P}^n N$ recursively by setting

$$\begin{array}{rcl} \mathbf{P}^0 N & = & N, \\ \mathbf{P}^{n+1} N & = & \mathbf{P}_{f_{n+1}}(\mathbf{P}^n N) \end{array}$$

Write $\lambda f_{[n]}.M = \lambda f_n.\dots.\lambda f_1.M$ for $n \geq 1$ and $\lambda f_{[0]}.M = M$. Now we can define the series of terms $\mathbf{Q}_n$ for $n \geq 0$ by setting

$$\mathbf{Q}_n = \lambda f_{[n]}.\lambda x.\lambda y.\mathbf{P}^n cond_{x,y}$$

So, for instance, we have

$$\mathbf{Q}_0 = \lambda x.\lambda y.cond_{x,y}$$

and

$$\begin{array}{rcl} \mathbf{Q}_1 & = & \lambda f_1.\lambda x.\lambda y.(\lambda z.\mathbf{K} \\ & & (\textit{if}\,\texttt{true}\ \textit{then}\ \langle z, \langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle\rangle \\ & & \textit{else}\ \langle\langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle, z\rangle) \\ & & (f_1 z)) \\ & & cond_{x,y} \end{array}$$

To prepare for a proof that the $\mathbf{Q}_n$ behave as claimed, let α and β be two distinct variables and let $u_0, u_1, \dots, u_k, \dots$ and $v_0, v_1, \dots, v_k, \dots$ be two enumerations of infinitely many distinct type variables (so, all the v_i are different from all the u_j and all of these are distinct from α and β.) Let T_n^m denote the pair-type constructed as the fully balanced binary tree of height n with 2^n leaf nodes, with internal nodes labeled by $*$ and leaf nodes labeled by variables $u_m, u_{m+1}, \dots, u_{m+2^n-1}$, from left to right. So, for instance, T_0^0 is just the variable u_0, T_1^0 is the type $u_0 * u_1$, T_2^0 is the type $(u_0 * u_1) * (u_2 * u_3)$, etc. Let us say that a type τ *has the shape of* T_n^m if τ is built from $*$ and variables only and, moreover, τ matches T_n^m; so, in this case, τ differs from T_n^m only by having possibly other variables than T_n^m at the leaves.

Define the type $\tau^{[n]}$ for $n \geq 0$ by setting

$$\begin{array}{rcl} \tau^{[0]} & = & \alpha \to (\beta \to T_1^0), \\ \tau^{[n]} & = & (\sigma_n \to v_n) \to \dots (\sigma_1 \to v_1) \to \\ & & \alpha \to \beta \to T_{n+1}^{k+1} \end{array}$$

where σ_i is a renaming of T_i^0 for $i = 1 \dots n$, using fresh variables which occur nowhere else in the type. We assume that the σ_i use the variables $v_1, v_2, \dots, v_k$.

We show by induction on $n \geq 0$ that

Lemma 4.3.1 (Main Lemma) *The minimal principal typing of* $\mathbf{Q}_n$ *has the form* $C_n, \emptyset \vdash_P \mathbf{Q}_n : \tau^{[n]}$ *where all variables in* C_n *are observable, and* C_n *contains at least* 2^{n+1} *distinct variables.*

PROOF To prove the lemma, consider the term $\mathbf{Q}_0$ for the base case. It is easy to see that a principal typing of $\mathbf{Q}_0$ has the form $C_0, \emptyset \vdash_P \lambda x.\lambda y.cond_{x,y} : \alpha \to (\beta \to u_0 * u_1)$ with $C_0 = \{\alpha \leq u_0, \alpha \leq u_1, \beta \leq u_0, \beta \leq u_1\}$. This typing is derived by the *standard procedure* described earlier, where G-simplification has been performed to eliminate internal variables. The resulting typing derivation is given by the completion:

$$\begin{array}{l}\lambda x : \alpha.\lambda y : \beta.\mathit{if}\,\texttt{true}\\ \mathit{then}\ \langle \uparrow_{\alpha}^{u_0} x, \uparrow_{\beta}^{u_1} y\rangle\\ \mathit{else}\ \langle \uparrow_{\beta}^{u_0} y, \uparrow_{\alpha}^{u_1} x\rangle\end{array}$$

All variables in C_0 are observable, and, moreover, C_0 is a 2-crown, from which it easily follows that the typing shown is S-simplified. It then follows from Theorem 4.2.3 that the typing is the *minimal* principal typing for $\mathbf{Q}_0$, and this typing satisfies the claim $(*)$ for the case $n = 0$.

For the inductive case, consider $\mathbf{Q}_{n+1} =$

$$\begin{array}{l}\lambda f_{n+1}.\lambda f_{[n]}.\lambda x.\lambda y.(\lambda z.\mathbf{K}\\ \quad(\mathit{if}\,\texttt{true}\ \mathit{then}\ \langle z, \langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle\rangle\\ \quad\mathit{else}\ \langle\langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle, z\rangle)\\ \quad(f_{n+1} z))\\ \quad(\mathbf{P}^n \mathit{cond}_{x,y})\end{array}$$

By induction hypothesis for $\mathbf{Q}_n$, $(\mathbf{P}^n \mathit{cond}_{x,y})$ has a minimal principal typing with type T^m_{n+1} (for some m) and coercion set C_n with at least 2^{n+1} distinct variables, assuming the appropriate types $\sigma_i \to v_i$ for the f_i $(i = 1 \ldots n)$, α for x and β for y. Now imagine that we have inserted coercions at the leaves in the remaining part of $\mathbf{Q}_{n+1}$ (cf. the *standard procedure*.) Then, due to the application $(f_{n+1} z)$, the type assumed for f_{n+1} must have the form $\tau_1 * \tau_2 \to \gamma$ where $\tau_1 * \tau_2$ is a renaming of T^m_{n+1} (using fresh variables, and γ fresh), because the type of z must be T^m_{n+1} (because $(\mathbf{P}^n \mathit{cond}_{x,y})$ is applied to $\lambda z.\ldots$) However, since every variable in $\tau_1 * \tau_2$ occurs only positively in the type of the entire expression $\mathbf{Q}_{n+1}$, it easily follows by S-simplification that $\tau_1 * \tau_2$ can be identified with T^m_{n+1}, and hence we need apply no coercion to z or f_{n+1}. Using elimination of internal variables by G-simplification, it is then straight-forward to see that a principal completion of $\mathbf{Q}_{n+1}$ is obtained by inserting coercions as follows (assuming suitable coercions inside $(\mathbf{P}^n \mathit{cond}_{x,y})$)

$$\begin{array}{l}\lambda f_{n+1} : T^m_{n+1} \to \gamma.\lambda f_{[n]} : [\sigma'_n].\lambda x : \alpha.\lambda y : \beta.\\ (\lambda z : T^m_{n+1}.\mathbf{K}\\ \quad(\mathit{if}\,\texttt{true}\ \mathit{then}\\ \quad\langle \uparrow^{\theta_1 * \theta_2}_{T_1 * T_2} z, \langle \texttt{2nd}\ \uparrow^{T_1 * \theta_3}_{T_1 * T_2} z, \texttt{1st}\ \uparrow^{\theta_4 * T_2}_{T_1 * T_2} z\rangle\rangle\\ \quad\mathit{else}\\ \quad\langle\langle \texttt{2nd}\ \uparrow^{T_1 * \theta_1}_{T_1 * T_2} z, \texttt{1st}\ \uparrow^{\theta_2 * T_2}_{T_1 * T_2} z\rangle, \uparrow^{\theta_3 * \theta_4}_{T_1 * T_2} z\rangle)\\ \quad(f_{n+1}\ z))\\ \quad(\mathbf{P}^n \mathit{cond}_{x,y})\end{array}$$

Here T_1 and T_2 are such that $T_1 * T_2 = T^m_{n+1}$, so T_1 and T_2 match each other and each has 2^n distinct variables, and $\lambda f_{[n]} : [\sigma'_n]$ abbreviates the list of typed parameters $\lambda f_i : \sigma_i \to v_i$, $i = 1 \ldots n$. The types θ_i are renamings, using fresh variables, of T_1 (or, equivalently, of T_2.) This shows that a principal typing for $\mathbf{Q}_{n+1}$ can be obtained by adding to C_n the two 2-crowns C_1, C_2 of coercions shown in the completion above:

$$\begin{array}{rcl} C_1 & = & \{T_1 \leq \theta_1, T_1 \leq \theta_2, T_2 \leq \theta_2, T_2 \leq \theta_1\} \\ C_2 & = & \{T_1 \leq \theta_4, T_1 \leq \theta_3, T_2 \leq \theta_3, T_2 \leq \theta_4\} \end{array}$$

Now, these crowns are not atomic, since the types in them are all of the shape of T_n^m, hence to add them to C_n we need to decompose them. It is easy to see that each C_i decomposes into a set C_i' of 2^n atomic 2-crowns (since T_n^m has 2^n leaves) resulting in a total of $2 \cdot 2^n = 2^{n+1}$ new crowns; since the θ_i have fresh variables, each C_i contains $2 \cdot 2^n = 2^{n+1}$ new, distinct variables, and since, by induction, C_n already has at least 2^{n+1} distinct variables, it follows that the number of distinct variables in $C_{n+1} = C_n \cup C_1' \cup C_2'$ is at least $2^{n+1} + 2^{n+1} = 2^{n+2}$.

The type of $\mathbf{Q}_{n+1}$ under the principal typing shown has the form

$$\begin{array}{l} (T_{n+1}^m \to \gamma) \to \sigma_1' \to \ldots \to \sigma_n' \to \\ \alpha \to \beta \to ((\theta_1 * \theta_2) * (\theta_3 * \theta_4)) \end{array}$$

which can be renamed to $\tau^{[n+1]}$. Since, by induction, all variables in C_n are observable, and since all new variables in C_{n+1} are in the θ_i, it follows that all variables in C_{n+1} are observable in the typing of $\mathbf{Q}_{n+1}$.

It remains to show that the typing shown for $\mathbf{Q}_{n+1}$ is *minimal.* By induction hypothesis, C_n is minimal as part of a minimal typing of $\mathbf{Q}_n$, hence, in particular, it is S-simplified. By the shape of 2-crowns, it is easy to verify that adding the crowns of C_1' and C_2' preserves this property, so C_{n+1} is S-simplified also. It then follows from Theorem 4.2.3 that the typing shown for $\mathbf{Q}_{n+1}$ is the minimal one. This completes the proof of the lemma. □

A critical part of the construction given above is that the principal typings of the terms $\mathbf{Q}_n$ have constraints sets with all variables observable. This is what allows us to use Theorem 4.2.3 in order to conclude, in the proof of the preceeding lemma, that typings are *minimal.* The rôle of the application $(f\ z)$ in the terms $\mathbf{P}_f$ is to make the type of the argument z observable in the typing of $\mathbf{Q}_n$; this greatly simplifies the correctness argument for the construction, since it makes it easier to apply Theorem 4.2.3.

The terms $\mathbf{Q}_n$ are simple enough to enable S-simplification and *G-simplification* to do the job of transforming a typing extracted by the *standard procedure* into the minimal one. We have implemented G- and S-simplification and used the system to generate the minimal typings for the first few terms $\mathbf{Q}_n$. Figure 4.5 shows the output for $n = 5$ (showing actually the term $\mathbf{P}^5 cond_{xy}$), and the reader may discern an exponential number of 2-crowns in the coercion set.

$\{\alpha_0 < \alpha_3, \alpha_0 < \alpha_4, \alpha_1 < \alpha_4, \alpha_1 < \alpha_3, \alpha_3 < \alpha_8, \alpha_3 < \alpha_7, \alpha_3 < \alpha_9, \alpha_3 < \alpha_6, \alpha_4 < \alpha_9, \alpha_4 < \alpha_6, \alpha_4 < \alpha_8, \alpha_4 < \alpha_7, \alpha_6 < \alpha_{15}, \alpha_6 < \alpha_{13}, \alpha_6 < \alpha_{17}, \alpha_6 < \alpha_{11}, \alpha_7 < \alpha_{16}, \alpha_7 < \alpha_{14}, \alpha_7 < \alpha_{18}, \alpha_7 < \alpha_{12}, \alpha_8 < \alpha_{17}, \alpha_8 < \alpha_{11}, \alpha_8 < \alpha_{15}, \alpha_8 < \alpha_{13}, \alpha_9 < \alpha_{18}, \alpha_9 < \alpha_{12}, \alpha_9 < \alpha_{16}, \alpha_9 < \alpha_{14}, \alpha_{11} < \alpha_{28}, \alpha_{11} < \alpha_{24}, \alpha_{11} < \alpha_{32}, \alpha_{11} < \alpha_{20}, \alpha_{12} < \alpha_{29}, \alpha_{12} < \alpha_{25}, \alpha_{12} < \alpha_{33}, \alpha_{12} < \alpha_{21}, \alpha_{13} < \alpha_{30}, \alpha_{13} < \alpha_{26}, \alpha_{13} < \alpha_{34}, \alpha_{13} < \alpha_{22}, \alpha_{14} < \alpha_{31}, \alpha_{14} < \alpha_{27}, \alpha_{14} < \alpha_{35}, \alpha_{14} < \alpha_{23}, \alpha_{15} < \alpha_{32}, \alpha_{15} < \alpha_{20}, \alpha_{15} < \alpha_{28}, \alpha_{15} < \alpha_{24}, \alpha_{16} < \alpha_{33}, \alpha_{16} < \alpha_{21}, \alpha_{16} < \alpha_{29}, \alpha_{16} < \alpha_{25}, \alpha_{17} < \alpha_{34}, \alpha_{17} < \alpha_{22}, \alpha_{17} < \alpha_{30}, \alpha_{17} < \alpha_{26}, \alpha_{18} < \alpha_{35}, \alpha_{18} < \alpha_{23}, \alpha_{18} < \alpha_{31}, \alpha_{18} < \alpha_{27}, \alpha_{20} < \alpha_{53}, \alpha_{20} < \alpha_{45}, \alpha_{20} < \alpha_{61}, \alpha_{20} < \alpha_{37}, \alpha_{21} < \alpha_{54}, \alpha_{21} < \alpha_{46}, \alpha_{21} < \alpha_{62}, \alpha_{21} < \alpha_{38}, \alpha_{22} < \alpha_{55}, \alpha_{22} < \alpha_{47}, \alpha_{22} < \alpha_{63}, \alpha_{22} < \alpha_{39}, \alpha_{23} < \alpha_{56}, \alpha_{23} < \alpha_{48}, \alpha_{23} < \alpha_{64}, \alpha_{23} < \alpha_{40}, \alpha_{24} < \alpha_{57}, \alpha_{24} < \alpha_{49}, \alpha_{24} < \alpha_{65}, \alpha_{24} < \alpha_{41}, \alpha_{25} < \alpha_{58}, \alpha_{25} < \alpha_{50}, \alpha_{25} < \alpha_{66}, \alpha_{25} < \alpha_{42}, \alpha_{26} < \alpha_{59}, \alpha_{26} < \alpha_{51}, \alpha_{26} < \alpha_{67}, \alpha_{26} < \alpha_{43}, \alpha_{27} < \alpha_{60}, \alpha_{27} < \alpha_{52}, \alpha_{27} < \alpha_{68}, \alpha_{27} < \alpha_{44}, \alpha_{28} < \alpha_{61}, \alpha_{28} < \alpha_{37}, \alpha_{28} < \alpha_{53}, \alpha_{28} < \alpha_{45}, \alpha_{29} < \alpha_{62}, \alpha_{29} < \alpha_{38}, \alpha_{29} < \alpha_{54}, \alpha_{29} < \alpha_{46}, \alpha_{30} < \alpha_{63}, \alpha_{30} < \alpha_{39}, \alpha_{30} < \alpha_{55}, \alpha_{30} < \alpha_{47}, \alpha_{31} < \alpha_{64}, \alpha_{31} < \alpha_{40}, \alpha_{31} < \alpha_{56}, \alpha_{31} < \alpha_{48}, \alpha_{32} < \alpha_{65}, \alpha_{32} < \alpha_{41}, \alpha_{32} < \alpha_{57}, \alpha_{32} < \alpha_{49}, \alpha_{33} < \alpha_{66}, \alpha_{33} < \alpha_{42}, \alpha_{33} < \alpha_{58}, \alpha_{33} < \alpha_{50}, \alpha_{34} < \alpha_{67}, \alpha_{34} < \alpha_{43}, \alpha_{34} < \alpha_{59}, \alpha_{34} < \alpha_{51}, \alpha_{35} < \alpha_{68}, \alpha_{35} < \alpha_{44}, \alpha_{35} < \alpha_{60}, \alpha_{35} < \alpha_{52}, \alpha_{37} < \alpha_{102}, \alpha_{37} < \alpha_{86}, \alpha_{37} < \alpha_{118}, \alpha_{37} < \alpha_{70}, \alpha_{38} < \alpha_{103}, \alpha_{38} < \alpha_{87}, \alpha_{38} < \alpha_{119}, \alpha_{38} < \alpha_{71}, \alpha_{39} < \alpha_{104}, \alpha_{39} < \alpha_{88}, \alpha_{39} < \alpha_{120}, \alpha_{39} < \alpha_{72}, \alpha_{40} < \alpha_{105}, \alpha_{40} < \alpha_{89}, \alpha_{40} < \alpha_{121}, \alpha_{40} < \alpha_{73}, \alpha_{41} < \alpha_{106}, \alpha_{41} < \alpha_{90}, \alpha_{41} < \alpha_{122}, \alpha_{41} < \alpha_{74}, \alpha_{42} < \alpha_{107}, \alpha_{42} < \alpha_{91}, \alpha_{42} < \alpha_{123}, \alpha_{42} < \alpha_{75}, \alpha_{43} < \alpha_{108}, \alpha_{43} < \alpha_{92}, \alpha_{43} < \alpha_{124}, \alpha_{43} < \alpha_{76}, \alpha_{44} < \alpha_{109}, \alpha_{44} < \alpha_{93}, \alpha_{44} < \alpha_{125}, \alpha_{44} < \alpha_{77}, \alpha_{45} < \alpha_{110}, \alpha_{45} < \alpha_{94}, \alpha_{45} < \alpha_{126}, \alpha_{45} < \alpha_{78}, \alpha_{46} < \alpha_{111}, \alpha_{46} < \alpha_{95}, \alpha_{46} < \alpha_{127}, \alpha_{46} < \alpha_{79}, \alpha_{47} < \alpha_{112}, \alpha_{47} < \alpha_{96}, \alpha_{47} < \alpha_{128}, \alpha_{47} < \alpha_{80}, \alpha_{48} < \alpha_{113}, \alpha_{48} < \alpha_{97}, \alpha_{48} < \alpha_{129}, \alpha_{48} < \alpha_{81}, \alpha_{49} < \alpha_{114}, \alpha_{49} < \alpha_{98}, \alpha_{49} < \alpha_{130}, \alpha_{49} < \alpha_{82}, \alpha_{50} < \alpha_{115}, \alpha_{50} < \alpha_{99}, \alpha_{50} < \alpha_{131}, \alpha_{50} < \alpha_{83}, \alpha_{51} < \alpha_{116}, \alpha_{51} < \alpha_{100}, \alpha_{51} < \alpha_{132}, \alpha_{51} < \alpha_{84}, \alpha_{52} < \alpha_{117}, \alpha_{52} < \alpha_{101}, \alpha_{52} < \alpha_{133}, \alpha_{52} < \alpha_{85}, \alpha_{53} < \alpha_{118}, \alpha_{53} < \alpha_{70}, \alpha_{53} < \alpha_{102}, \alpha_{53} < \alpha_{86}, \alpha_{54} < \alpha_{119}, \alpha_{54} < \alpha_{71}, \alpha_{54} < \alpha_{103}, \alpha_{54} < \alpha_{87}, \alpha_{55} < \alpha_{120}, \alpha_{55} < \alpha_{72}, \alpha_{55} < \alpha_{104}, \alpha_{55} < \alpha_{88}, \alpha_{56} < \alpha_{121}, \alpha_{56} < \alpha_{73}, \alpha_{56} < \alpha_{105}, \alpha_{56} < \alpha_{89}, \alpha_{57} < \alpha_{122}, \alpha_{57} < \alpha_{74}, \alpha_{57} < \alpha_{106}, \alpha_{57} < \alpha_{90}, \alpha_{58} < \alpha_{123}, \alpha_{58} < \alpha_{75}, \alpha_{58} < \alpha_{107}, \alpha_{58} < \alpha_{91}, \alpha_{59} < \alpha_{124}, \alpha_{59} < \alpha_{76}, \alpha_{59} < \alpha_{108}, \alpha_{59} < \alpha_{92}, \alpha_{60} < \alpha_{125}, \alpha_{60} < \alpha_{77}, \alpha_{60} < \alpha_{109}, \alpha_{60} < \alpha_{93}, \alpha_{61} < \alpha_{126}, \alpha_{61} < \alpha_{78}, \alpha_{61} < \alpha_{110}, \alpha_{61} < \alpha_{94}, \alpha_{62} < \alpha_{127}, \alpha_{62} < \alpha_{79}, \alpha_{62} < \alpha_{111}, \alpha_{62} < \alpha_{95}, \alpha_{63} < \alpha_{128}, \alpha_{63} < \alpha_{80}, \alpha_{63} < \alpha_{112}, \alpha_{63} < \alpha_{96}, \alpha_{64} < \alpha_{129}, \alpha_{64} < \alpha_{81}, \alpha_{64} < \alpha_{113}, \alpha_{64} < \alpha_{97}, \alpha_{65} < \alpha_{130}, \alpha_{65} < \alpha_{82}, \alpha_{65} < \alpha_{114}, \alpha_{65} < \alpha_{98}, \alpha_{66} < \alpha_{131}, \alpha_{66} < \alpha_{83}, \alpha_{66} < \alpha_{115}, \alpha_{66} < \alpha_{99}, \alpha_{67} < \alpha_{132}, \alpha_{67} < \alpha_{84}, \alpha_{67} < \alpha_{116}, \alpha_{67} < \alpha_{100}, \alpha_{68} < \alpha_{133}, \alpha_{68} < \alpha_{85}, \alpha_{68} < \alpha_{117}, \alpha_{68} < \alpha_{101}\}$,

$\{f_5 : ((((((\alpha_{68} * \alpha_{67}) * (\alpha_{66} * \alpha_{65})) * ((\alpha_{64} * \alpha_{63}) * (\alpha_{62} * \alpha_{61}))) * (((\alpha_{60} * \alpha_{59}) * (\alpha_{58} * \alpha_{57})) * ((\alpha_{56} * \alpha_{55}) * (\alpha_{54} * \alpha_{53})))) * ((((\alpha_{52} * \alpha_{51}) * (\alpha_{50} * \alpha_{49})) * ((\alpha_{48} * \alpha_{47}) * (\alpha_{46} * \alpha_{45}))) * (((\alpha_{44} * \alpha_{43}) * (\alpha_{42} * \alpha_{41})) * ((\alpha_{40} * \alpha_{39}) * (\alpha_{38} * \alpha_{37}))))) \rightarrow \alpha_{36}), f_4 : (((((\alpha_{35} * \alpha_{34}) * (\alpha_{33} * \alpha_{32})) * ((\alpha_{31} * \alpha_{30}) * (\alpha_{29} * \alpha_{28}))) * (((\alpha_{27} * \alpha_{26}) * (\alpha_{25} * \alpha_{24})) * ((\alpha_{23} * \alpha_{22}) * (\alpha_{21} * \alpha_{20})))) \rightarrow \alpha_{19}), f_3 : ((((\alpha_{18} * \alpha_{17}) * (\alpha_{16} * \alpha_{15})) * ((\alpha_{14} * \alpha_{13}) * (\alpha_{12} * \alpha_{11}))) \rightarrow \alpha_{10}), f_2 : (((\alpha_9 * \alpha_8) * (\alpha_7 * \alpha_6)) \rightarrow \alpha_5), f_1 : ((\alpha_4 * \alpha_3) \rightarrow \alpha_2), y : \alpha_0, x : \alpha_1\} \vdash$

$((\lambda z.((((\lambda x_1.\lambda x_2.x_1)\mathtt{if\ true\ then}\ < z, < (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >>\ \mathtt{else}\ << (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >, z >))((f_5)z)))((\lambda z.((((\lambda x_1.\lambda x_2.x_1)\mathtt{if\ true\ then}\ < z, < (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >>\ \mathtt{else}\ << (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >, z >))((f_4)z)))((\lambda z.((((\lambda x_1.\lambda x_2.x_1)\ \mathtt{if\ true\ then}\ < z, < (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >>\ \mathtt{else}\ << (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >, z >))((f_3)z)))((\lambda z.((((\lambda x_1.\lambda x_2.x_1)\ \mathtt{if\ true\ then}\ < z, < (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >>\ \mathtt{else}\ << (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >, z >))((f_2)z)))((\lambda z.((((\lambda x_1.\lambda x_2.x_1)\ \mathtt{if\ true\ then}\ < z, < (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >>\ \mathtt{else}\ << (\mathtt{2nd}\ z), (\mathtt{1st}\ z) >, z >))((f_1)z)))\ \mathtt{if\ true\ then}\ < x, y >\ \mathtt{else}\ < y, x >))))) :$

$((((((\alpha_{133} * \alpha_{132}) * (\alpha_{131} * \alpha_{130})) * ((\alpha_{129} * \alpha_{128}) * (\alpha_{127} * \alpha_{126}))) * (((\alpha_{125} * \alpha_{124}) * (\alpha_{123} * \alpha_{122})) * ((\alpha_{121} * \alpha_{120}) * (\alpha_{119} * \alpha_{118})))) * ((((\alpha_{117} * \alpha_{116}) * (\alpha_{115} * \alpha_{114})) * ((\alpha_{113} * \alpha_{112}) * (\alpha_{111} * \alpha_{110}))) * (((\alpha_{109} * \alpha_{108}) * (\alpha_{107} * \alpha_{106})) * ((\alpha_{105} * \alpha_{104}) * (\alpha_{103} * \alpha_{102}))))) * (((((\alpha_{101} * \alpha_{100}) * (\alpha_{99} * \alpha_{98})) * ((\alpha_{97} * \alpha_{96}) * (\alpha_{95} * \alpha_{94}))) * (((\alpha_{93} * \alpha_{92}) * (\alpha_{91} * \alpha_{90})) * ((\alpha_{89} * \alpha_{88}) * (\alpha_{87} * \alpha_{86})))) * ((((\alpha_{85} * \alpha_{84}) * (\alpha_{83} * \alpha_{82})) * ((\alpha_{81} * \alpha_{80}) * (\alpha_{79} * \alpha_{78}))) * (((\alpha_{77} * \alpha_{76}) * (\alpha_{75} * \alpha_{74})) * ((\alpha_{73} * \alpha_{72}) * (\alpha_{71} * \alpha_{70})))))$

Figure 4.5: Minimal typing for $\mathbf{P}^5 cond_{xy}$

4.3.3 Encoding the construction in pure lambda calculus

We now show that the conditional construct, the pairing construct and the projection functions can be eliminated from the construction.

First consider the conditional. The only property of the conditional used in the construction is that it requires the types of both of its branches, say M_1 and M_2, to be coerced to a common supertype. This can be effected without the conditional by placing M_1 and M_2 in the context

$$\lambda x.\mathbf{K}(x\, M_1)(x\, M_2)$$

which requires the types of M_1 and M_2 to be coerced to the domain type of x. This eliminates the need for the conditional.

As for the pairing construction with projections, we might be tempted to use the standard lambda-calculus encodings, but this will not work here, since the simply typed lambda-calculus has no type-correct equivalent of pairing with projection. This follows, via the Curry-Howard isomorphism, from the fact that one cannot define logical conjunction as a derived notion from implication in minimal logic (see [30]) However, our construction does not need the full power of pairing and projections, since the projections are only used to permute a pair, in order to force the type of $\langle M, N\rangle$ to be equal to the type of $\langle N, M\rangle$. At the same time, the technique used to achieve the exponential blow-up requires us to type an abstraction $\lambda z. \ldots$, which expects a pair z and copies it into new pairs that are double the size of the original pair [4] However, instead of writing expressions such as

$$\textit{if}\,\texttt{true}\ \textit{then}\ \langle z, \langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle\rangle\ \textit{else}\ \langle\langle \texttt{2nd}\, z, \texttt{1st}\, z\rangle, z\rangle$$

we can obtain a similar effect on typings by writing the term

$$\textit{if}\,\texttt{true}\ \textit{then}\ \langle\!|\, \langle\!|\, z, s \,|\!\rangle, \langle\!|\, z, t \,|\!\rangle \,|\!\rangle\ \textit{else}\ \langle\!|\, \langle\!|\, s, z \,|\!\rangle, \langle\!|\, t, z \,|\!\rangle \,|\!\rangle$$

where s and t are free variables and where $\langle\!|\, M, N \,|\!\rangle = \lambda p.(p\, M)\, N$ is just the standard encoding of pairing.

[4]This is in fact where the *standard* encodings of pairing and projection will fail to type if used to encode the present construction directly; the problem is that we cannot find a common (standard, encoded) type for all of the occurrences of z in the expressions $\mathbf{P}_f$, even using subtyping.

Summing up, we use the definitions

$$\begin{array}{lcl}
\mathit{eqty}(M,N) & = & \lambda x.\mathbf{K}(xM)(xN) \\
\langle\!|\, M, N \,|\!\rangle & = & \lambda p.(pM)N \\
\mathbf{P}_{f,s,t} & = & \lambda z.\mathbf{K} \\
 & & \mathit{eqty}(\langle\!|\, \langle\!|\, z,s \,|\!\rangle, \langle\!|\, z,t \,|\!\rangle \,|\!\rangle, \langle\!|\, \langle\!|\, s,z \,|\!\rangle, \langle\!|\, t,z \,|\!\rangle \,|\!\rangle) \\
 & & (f\ z) \\
\mathbf{P}^{n+1}N & = & \mathbf{P}_{f_{n+1},s_{n+1},t_{n+1}}(\mathbf{P}^n N) \\
\mathbf{Q}_n & = & \lambda f_{[n]}.\lambda s_{[n]}.\lambda t_{[n]}.\lambda x.\lambda y.\mathbf{P}^n \mathit{eqty}(x,y)
\end{array}$$

It is tedious but not difficult to see that all the relevant effects on coercions which is exploited in our construction above are also present under this encoding, using the encoded pair-type $(\tau_1 \to \tau_2 \to \sigma) \to \sigma$ (for arbitrary type σ) to encode the type $\tau_1 * \tau_2$. In particular, it is still possible to eliminate all internal variables using G- and S-simplification on the constraint sets extracted by the standard procedure, and Theorem 4.2.3 can therefore be used as before. As a result, the construction can be carried out for an arbitrary poset of ground types, because the encodings eliminate reference to constants. We therefore have

Theorem 4.3.2 *For any poset P of base types it holds for arbitrarily large n, that there exist closed expressions of length n such that any principal typing for the expressions has a coercion set containing $2^{\Omega(n)}$ distinct type variables and a type containing $2^{\Omega(n)}$ distinct type variables.*

PROOF Take the series of terms $\mathbf{Q}_n$ (using encodings as shown, to eliminate assumptions on P); clearly, the size of $\mathbf{Q}_n$ is of the order of n, and the main lemma (Lemma 4.3.1) shows that $\mathbf{Q}_n$ has a minimal principal typing t with coercion set containing more than 2^n distinct variables and a type containing more than 2^n distinct variables. Any other principal typing t' satisfies $t \approx t'$, and hence t' must have at least as many distinct type variables in both coercion set and type as t, because $t \lesssim t'$ by the definition of minimality. □

The theorem immediately implies that the *dag-size* of coercion sets as well as of types in principal typings are of the order $2^{\Omega(n)}$ in the worst case. Since it follows from standard type inference algorithms such as those of [26] and [11] that a principal atomic typing can be obtained by extracting a set C of (possibly non-atomic) coercions of size linear in the size of the term followed by a match-step which expands and decomposes C to atomic constraints under at most an exponential blow-up, we have

Corollary 4.3.3 *The dag-size of coercion sets as well as of types in atomic principal typings is of the order $2^{\Theta(n)}$ in the worst case.*

Bibliography

[1] A.V. Aho, R. Garey, and J.D. Ullman. The transitive reduction of a directed graph. *SIAM Journal of Computing*, 1(2):131–137, June 1972.

[2] A. Aiken, E.L. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. Technical Report UCB/CSD-96-909, University of California, Berkely, July 1996.

[3] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[4] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford University Press, 1992.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, eleventh printing, 1994.

[6] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Parc, February 1990.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 1990.

[8] N. Dershowitz and J-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume II, pages 243–320. Elsevier, 1990.

[9] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA '95*, 1995.

[10] Y. Fuh and P. Mishra. Type inference with subtypes. In *Proc. 2nd European Symp. on Programming*, pages 94–114. Springer-Verlag, 1988. Lecture Notes in Computer Science 300.

[11] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. Int'l J't Conf. on Theory and Practice of*

Software Development, pages 167–183, Barcelona, Spain, March 1989. Springer-Verlag.

[12] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science (TCS)*, 73:155–175, 1990.

[13] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[14] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[15] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming (SCP)*, 22(3):197–230, 1994.

[16] J.R. Hindley. *Basic Simple Type Theory.* Draft, forthcoming.

[17] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[18] M. Hoang and J.C. Mitchell. Lower bounds on type inference with subtypes. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 176–185. ACM Press, 1995.

[19] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), San Francisco, California*, pages 193–204. ACM, ACM Press, June 1992. also in LISP Pointers, Vol. V, Number 1, January-March 1992.

[20] P. Kanellakis, H. Mairson, and J. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic — Essays in Honor of Alan Robinson.* MIT Press, 1991.

[21] P. Lincoln and J. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proc. 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programmin Languages (POPL), Albuquerque, New Mexico*, pages 293–304. ACM Press, January 1992.

[22] A. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[23] R. Milner, M. Tofte., and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[24] J. Mitchell. Coercion and type inference (summary). In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, pages 175–185, 1984.

[25] J. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[26] J. Mitchell. Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–285, July 1991.

[27] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[28] F. Pottier. Simplifying subtyping constraints. In *Proceedings ICFP '96, International Conference on Functional Programming*, pages 122–133. ACM, ACM Press, May 1996.

[29] Vaughan Pratt and Jerzy Tiuryn. Satisfiability of inequalities in a poset. *Studia Logica, Helene Rasiowa memorial issue (to appear)*, 1996.

[30] Dag Prawitz. *Natural deduction*. Almquist & Wiksell, Uppsala 1965.

[31] Jakob Rehof. Deterministic satisfiability problems in a poset. Technical report, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark, 1995.

[32] Jakob Rehof. Minimal typings in atomic subtyping. Technical report, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark. Available at http://www.diku.dk/research-groups/topps/personal/rehof/publications.html, 1996.

[33] Jakob Rehof. Minimal typings in atomic subtyping. Accepted for publication, to appear in proceedings POPL '97, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 1997.

[34] J. Reynolds. Using category theory to design conversions and generic operators. In *Proc. Semantics-Directed Compiler Generation*, pages 211–258. Springer-Verlag, 1980. Lecture Notes in Computer Science, Vol. 94.

[35] Robert Sedgewick. *Algorithms*. Addison Wesley, Second edition, 1988.

[36] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[37] Geoffrey Smith. Polymorphic type inference with overloading and subtyping. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT), Orsay, France*, volume 668 of *Lecture Notes in Computer Science*, pages 671–685. Springer-Verlag, April 1993.

[38] J. Tiuryn. Subtype inequalities. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science (LICS), Santa Cruz, California*, pages 308–315. IEEE, IEEE Computer Society Press, June 1992.

[39] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT), Orsay, France*, volume 668 of *Lecture Notes in Computer Science*, pages 686–701. Springer-Verlag, April 1993.

[40] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.

[41] M. Wand and P. O'Keefe. On the complexity of type inference with coercion. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, 1989.