Minimal Typings in Atomic Subtyping

Jakob Rehof
DIKU, Department of Computer Science
University of Copenhagen, Denmark
rehof@diku.dk

Abstract

This paper studies the problem of simplifying typings and the size-complexity of most general typings in typed programming languages with atomic subtyping. We define a notion of minimal typings relating all typings which are equivalent with respect to instantiation. The notion of instance is that of Fuh and Mishra [13], which supports many interesting simplifications. We prove that every typable term has a unique minimal typing, which is the logically most succinct among all equivalent typings. We study completeness properties, with respect to our notion of minimality, of well-known simplification techniques. Drawing upon these results, we prove a tight exponential lower bound for the worst case dag-size of constraint sets as well as of types in most general typings. To the best of our knowledge, the best previously proven lower bound was linear.

1 Introduction

Subtyping is a fundamental idea in type systems for programming languages, which can in principle be integrated into standard type systems and type inference for languages such as, e.g., the simply typed lambda calculus [7], ML [22], Haskell [18], Miranda [33] as well as being a basic notion in typed object oriented languages, see e.g.[15, 11, 1]. Type inference algorithms infer type information from programs without requiring programmers to insert explicit type declarations in the program and discovers many programming errors at compile time. Moreover, systems of subtyping are becoming increasingly used in non-standard ways in type based program analysis, where program properties of a typed language are automatically extracted from the program text using inference algorithms, e.g., [3, 4, 16].

In all cases, subtyping adds expressiveness to a type system by allowing an expression to have several types depending on the context in which it occurs. This is achieved by imposing an order relation (subtype order) on types together with a rule (subsumption) which allows any expression of type τ to have also any type τ' which is larger than τ in the given ordering. For example, an integer of type `int` may be converted to a real of type `real`, which can be modeled by having `int` a subtype of `real`.

Subtypings associate not only a type to an expression but also a set of subtyping *constraints* (coercions) which express assumptions about the subtype order which must hold for the given typing. The presence of such assumptions are necessitated by the desire to have most general (principal) typings, which summarize all possible typings for a given term. A form of subtyping which is logically simple and natural, yet expressive enough to be interesting, is *atomic subtyping* [23, 24, 13, 14] where only subtyping relations between atoms (type variables or constants) can occur in typings. However, even though several type inference algorithms have appeared for atomic subtyping (e.g., [24, 13, 14]), it is generally recognized that major obstacles remain for subtype inference to become practicable in a large scale setting. To quote from [17], "the main problems seem to be that the algorithm is inefficient, and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read". The problem has generated a significant amount of work which aims at *simplifying* constraints in the typings generated by type inference algorithms; works addressing the subtype simplification problem include [13, 10, 19, 30, 11, 26, 32, 12, 5]. As is argued in [5], simplification is beneficial for at least three reasons: first, it may speed up type inference, second, it makes types more readable, and, third, it makes the information content of a typing more explicit.

Contribution of the paper

In this paper we study the subtype simplification problem for atomic subtyping from a theoretical perspective. The motivation for this is as follows. Viewing simplification as an optimization problem, it is natural to ask whether there always exists a unique most simplified typing. A positive answer to this question will tell us that simplification has a well-defined target. Further, it is fundamentally important to know the limits of what can possibly be achieved by simplification techniques, in the worst case. In this paper we give a positive answer to the first question, and we use this result to answer the second question by determining the worst case size of both constraint sets and types in principal typings relative to the instance relation defined by Fuh and Mishra in [13]. More specifically, the paper contains the following results:

1. We define a notion of *minimal* typings for atomic subtyping and show that every typable term has a unique

minimal typing. Intuitively, a minimal typing is a logically most succinct presentation of all possible equivalent typings for a term, where equivalence is defined in terms of the instance relation defined in [13]. Surprisingly, no such notion has, to our knowledge, been studied previously. The instance relation of [13] is important, because it is natural and powerful enough to validate pragmatically important simplifications. The notion of minimality is surprisingly strong, and therefore the result that minimal typings exist can be used to derive non-trivial properties about the type system (see below.)

2. Having an independent notion of minimality we can ask whether known simplification techniques are complete for minimization (computing minimal typings.) We give completeness and incompleteness results for the simplification techniques introduced in [13]. We show that minimization under the instance relation of [13] cannot be computed in polynomial time, unless P = NP. Among other things, this supports the claim that quite powerful simplifications can be captured in the framework studied here.

3. We prove, perhaps the main result of the paper, a tight exponential lower bound for the size of most general typings relative to the instance relation of [13]. This is a non-trivial result, because the instance relation used validates quite powerful simplifications. To the best of our knowledge, the best previous result is the linear lower bound proven in [17] for a whole class of so-called sound instance relations. Our result shows that there is an intrinsic limit to what simplification under instance relations not stronger than that of [13] can ever achieve. The proof uses new techniques and draws in an essential way upon results mentioned under both items 1 and 2 above.

It is perhaps worth stressing that the lower bound result is still interesting, even if one considers stronger instance relations than the one adopted here. In this case, the result can be viewed as a lower bound for a subclass of sound simplifications, namely those which can be validated under the instance relation of [13]. This remark is further clarified in Section 3 below.

The remainder of this paper is organized as follows. Section 2 introduces the type system, which appears in Appendix A, and Section 2 also contains some additional technical preliminaries. In Section 3 we define the notion of instance together with a notion of sound simplifications. The following three sections of the paper then give the technical results. Section 4 contains the results mentioned under item 1 above, Section 5 contains the results mentioned under item 2 above, and Section 6 proves the lower bound result mentioned under item 3 above. Section 7 contains some concluding remarks, and Section 8 ends the paper with a discussion of related work. Longer proofs are placed in Appendix B. Due to space limitations, proofs that are less illuminating had to be left out. They can be found in [29].

2 Preliminaries

We begin with a brief introduction to the system of atomic subtyping, which appears in Appendix A. For a more comprehensive account we refer the reader to [24].

We assume types ranged over by τ and defined by

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

where α ($\beta, \gamma \ldots$) ranges over a denumerable set $\mathcal{V}$ of type variables and b ranges over base types drawn from a partially ordered set $(P, \leq_P)$ of type constants, such as, e.g., `int` (integers) and `real` (reals). An *atomic type* (or, an *atom*, for short) is either a variable or a constant. Atoms are ranged over by A. The term language is the lambda calculus extended with term constants,

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

Here x ($y, z \ldots$) ranges over term variables and c ranges over a given set of constants, such as, e.g., numerals and numerical functions.

We consider *typing judgements* t of the form $t = C, \Gamma \vdash_P M : \tau$ in the standard system of atomic subtyping studied in, e.g., [23], [24], [13], [14]. The type system is a proof system for deriving such judgements, and it is given in Appendix A for reference. In a judgement, C is a *constraint set*, that is a finite set of subtyping hypotheses of the form $\tau \leq \tau'$, and Γ is a finite set of type assumptions $x : \tau$ for the free variables of M with no variable occurring more than once; thus, Γ can be regarded as a finite function from term variables to type expressions.

An *atomic constraint set* is a constraint set in which all inequalities have the form $A \leq A'$, i.e., only hypotheses involving atomic types are allowed. In *atomic subtyping*, we consider a judgement $C, \Gamma \vdash_P M : \tau$ a *derivable atomic judgement*, if and only if it can be derived using the rules of the type system, *and*, moreover, C is an atomic constraint set. Only the derivable atomic judgements define well-typings in atomic subtyping.

The distinctive feature of subtyping systems is the presence of a rule of *subsumption*, which allows a term of type τ to have also any supertype τ'. This is embodied in the rule [*sub*], which appears in Figure 2 in Appendix A. This rule depends on a logic for deriving subtyping relations which is also given in Appendix A (Figure 1). The subtype logic allows derivations of the form $C \vdash_P \tau \leq \tau'$, meaning that under the subtyping hypotheses C, the subtyping relation $\tau \leq \tau'$ is derivable from P. We say that a constraint set C_1 *syntactically entails* constraint set C_2, written $C_1 \vdash_P C_2$, if and only if $C_1 \vdash_P \tau \leq \tau'$ for all $\tau \leq \tau' \in C_2$; we say that C_1 and C_2 are *equivalent*, written $C_1 \sim_P C_2$, if and only if $C_1 \vdash_P C_2$ and $C_2 \vdash_P C_1$.

In the following development *we require C to be consistent with P*, i.e., we consider only judgements $t = C, \Gamma \vdash_P M : \tau$ such that, whenever $b, b' \in P$ and $C \vdash_P b \leq b'$, then $b \leq_P b'$ holds in P. The reason for this restriction is that the theory of minimality for typings which can contain inconsistent constraint sets becomes more complicated, and since we usually have no interest in inconsistent typings, this restriction seems natural.

If $f : A \to B$ is a partial function from A to B then $Dm(f)$ denotes the domain of f. A *type substitution* 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. If $t = C, \Gamma \vdash_P M : \tau$ we let $\mathrm{Var}(C), \mathrm{Var}(\Gamma), \mathrm{Var}(\tau)$ denote, respectively, the type variables appearing in C, Γ, τ. We let $\mathrm{Var}(t) = \mathrm{Var}(C) \cup \mathrm{Var}(\Gamma) \cup \mathrm{Var}(\tau)$. If S is a renaming on $\mathrm{Var}(t)$ we sometimes say just that S is a renaming on t, for short. For constraint set C we write $S(C) = \{S(\tau) \leq S(\tau') \mid \tau \leq \tau' \in C\}$, and for assumption set Γ we write $S(\Gamma) = \{x : S(\tau) \mid x : \tau \in \Gamma\}$. We sometimes write the application of a finite substitution in reverse order, as in $C\{\alpha \mapsto A\}$.

3 Instance relation and sound simplifications

The following definition gives the instance relation as defined by Fuh and Mishra in [13]. There the relation is called *lazy instance*, here it is just called the instance relation. Alternative instance relations are mentioned below.

Definition 3.1 *(Instance relation, principal typing)* *Let $t_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$, $t_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$ be two atomic judgements, and let S be a type substitution. We say that t_2* is an instance of *t_1 under S, written $t_1 \prec_S t_2$, iff*

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

We say that t_2 is an instance of *t_1, written $t_1 \prec t_2$, iff there exists a type substitution S such that $t_1 \prec_S t_2$. We say that t_1 and t_2 are* equivalent, *written $t_1 \approx t_2$, iff $t_1 \prec t_2$ and $t_2 \prec t_1$. Clearly, $\approx$ is an equivalence relation.*

A typing judgement for a term M is called a principal typing for *M if it is derivable in the type system, and it has all other derivable judgements for M as instances.* □

A fundamentally new problem, which occurs in subtyping systems as compared to, say, simply typed lambda calculus ($\lambda^{\rightarrow}$ for short) and ML, is the brake-down of strong uniqueness properties of principal typings. While it is well known that principal types in $\lambda^{\rightarrow}$ are unique up to renaming of type variables, and principal ML types are unique up to renaming of bound variables, reordering of quantifiers and dropping of dummy quantifiers, a similarly simple situation does not hold for subtyping systems. Moreover, the principal type of a simple typed or an ML-typed program is guaranteed to be the syntactically shortest possible type for that program [1] This property, too, brakes down in subtyping systems. These points are illustrated in the following very simple examples [2]

Example 3.2 *Using Definition 3.1, one can show that the following typings are all principal typings for the identity function $\lambda x.x$:*

$$t_1 = \{\alpha \leq \beta, \beta \leq \alpha\}, \emptyset \vdash_P \lambda x.x : \alpha \rightarrow \beta$$

$$t_2 = \{\alpha \leq \gamma, \gamma \leq \beta\}, \emptyset \vdash_P \lambda x.x : \alpha \rightarrow \beta$$

$$t_3 = \{\alpha \rightarrow \beta\}, \emptyset \vdash_P \lambda x.x : \alpha \rightarrow \beta$$

$$t_4 = \emptyset, \emptyset \vdash_P \lambda x.x : \alpha \rightarrow \alpha$$

□

The next example is taken from [13]:

Example 3.3 *Let* comp *be the composition combinator defined by*

$$\mathrm{comp} = \lambda f.\lambda g.\lambda x.f(gx)$$

Then both of the following typings are principal for comp*:*

$$t_1 = \{\delta \leq \alpha, \eta \leq \gamma, \beta \leq \upsilon\}, \emptyset \vdash_P \mathrm{comp} : (\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \delta) \rightarrow (\eta \rightarrow \upsilon)$$

$$t_2 = \emptyset, \emptyset \vdash_P \mathrm{comp} : (\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \alpha) \rightarrow (\gamma \rightarrow \beta)$$

□

In general, the equivalence class, with respect to $\approx$, of typings of a given term contains infinitely many members, which may be related in non-trivial ways. Clearly, we should prefer the principal typings shown last in Example 3.2 and Example 3.3 to the others, since they are smaller, more readable and more informative than the others. It is the task of *subtype simplification* to discover such more desirable representatives among the many equivalent typings.

Simplifications must satisfy certain *soundness* conditions. We might say that simplifications must *preserve the information content of typings*. In our setting, we take the information content of a typing to be the set of all its instances. This leads to the requirement that, if $\mapsto$ is a simplification, viewed as 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.

More subtle notions of instance and soundness are possible, for example such that are given by semantic concepts rather than the syntactic notion of instance used here. Recent works taking this approach to simplification include [3] [32, 26, 5]. 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 [4] As was mentioned in the Introduction, lower bound results obtained using $\prec$ have an independent interest for other, more powerful frameworks as well, because the results remain valid for a certain class of simplifications, namely those that are sound with respect to $\approx$. Moreover, there seems to be good sense in studying restricted simplification frameworks, because even though we may choose more powerful ones, we may not be able to exploit their full power in practice. The reason is that it may be too costly, in terms of computational complexity, to verify that highly subtle soundness conditions are

[1] Because any other type can be produced from the principal one by substitution, and substitution can only increase the size of a type.

[2] The reader should be aware that the examples can be scaled up rather easily to become much more complex. For more examples consult works on subtype simplification, such as [13, 10, 19, 30, 11, 17, 26, 32, 5].

[3] See also [17] where a generic notion of *sound instance relation* is defined, based on syntactic concepts.

[4] We note, for completeness, that [24] uses a less powerful notion of instance (relating fewer typings) which cannot validate many of the simplifications that are sound under $\prec$. See [13, 17] for comparisons. The weaker notion is of independent interest; for example it can be used to prove strong properties of principality and completeness of type inference algorithms.

satisfied, and this is typically something which must be done by a simplification algorithm before a potential simplifying transformation can be carried out.

Before proceeding with the main technical development we recall some standard basic lemmas for atomic subtyping systems (see [24].) We say that two type expressions τ and τ' *match* if they have the same shape. More precisely, τ matches τ' if and only if both are atoms (not necessarily identical) or else $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$ and τ_1 matches τ_1' and τ_2 matches τ_2'.

Lemma 3.4 *(Substitution Lemma)*
If $C \vdash_P \tau \leq \tau'$, then $S(C) \vdash_P S(\tau) \leq S(\tau')$.

Lemma 3.5 *(Match Lemma)*
If C is atomic, and $C \vdash_P \tau \leq \tau'$, then τ and τ' match.

Lemma 3.6 *(Decomposition Lemma)*
If C is atomic, then $C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ if and only if $C \vdash_P \{\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'\}$.

4 Existence and uniqueness of minimal typings

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 prove existence.

In order to define minimality we first need the following notion of *specialization*.

Definition 4.1 *(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 $\Gamma_1(x) = S(\Gamma_2(x))$ for all $x \in Dm(\Gamma_1)$.*

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

Clearly, $\lesssim$ is reflexive and transitive. Note also that, if $t \lesssim_S t'$, then $t \prec_{id} S(t')$.

Example 4.2 *Consider again the typings shown in Example 3.2. With $S = \{\beta \mapsto \alpha, \gamma \mapsto \alpha\}$ we have $t_4 \lesssim_S t_i$ for $i = 1, 2, 3$.*

Consider the typings t_1 and t_2 shown in Example 3.3 and take $S = \{\delta \mapsto \alpha, \eta \mapsto \gamma, \upsilon \mapsto \beta\}$. Then $t_2 \lesssim_S t_1$. □

For a typing judgement t we let $[t]$ denote the equivalence class of t with respect to $\approx$, i.e., $[t] = \{t' \mid t' \approx t\}$.

We now define what it means for a typing judgement to be *minimal*. Intuitively, a minimal typing t is a *most specialized* element within $[t]$. It is therefore an irredundant representative in its equivalence class, where the typing information has been made as explicit as possible.

Definition 4.3 *(Minimality)*
A typing judgement t is called minimal *iff it holds for all $t' \in [t]$ that $t \lesssim t'$.* □

Notice that it can be quite non-trivial to establish that a given typing is minimal, because the definition of minimality quantifies over all the infinitely many members of an equivalence class. In order to reason about minimality it will therefore be necessary to develop some characterizations of this notion. We shall do so in Section 5.2 below. The result (Theorem 5.6) given there will, for example, allow us to conclude very easily that the typing t_4 of Example 3.2 is minimal for $\lambda x.x$ and that t_2 in Example 3.3 is minimal for *comp*.

If $t = C, \Gamma \vdash_P M : \tau$ and S is a type substitution, we write $S(t) = S(C), S(\Gamma) \vdash_P M : S(\tau)$. It is easy to show

Theorem 4.4 *(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 S_1 a renaming on t_1, S_2 a renaming on t_2 and $t_1 = S_2(t_2)$ and $t_2 = S_1(t_1)$.

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 constraint 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 $[t]$.

Definition 4.5 *(Full substitution instance)*
If $t' = S(t)$, then we say that t' is a substitution instance of *t under S. A typing judgement t is called* fully substituted *iff, whenever $S(t) \in [t]$, then S is a renaming on $Var(t)$. A* full substitution instance *of a typing t is a substitution instance of t which is fully substituted.* □

Notice that $t' = S(t)$ implies $t' \lesssim_S t$ but not vice versa.

Lemma 4.6 *If $S(t) \in [t]$, then $S \mid_{Var(t)}$ is an atomic substitution.*

Proof. Let $t = C, \Gamma \vdash_P M : \tau$. By the assumptions, $t' = S(t)$ is an atomic judgement, so $S \mid_{\mathrm{Var}(C)}$ 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 $S \mid_{\mathrm{Var}(\tau) \cup \mathrm{Var}(\Gamma)}$ is an atomic substitution. We have now shown that $S \mid_{\mathrm{Var}(t)}$ is atomic. □

By repeated application of non-renaming atomic substitutions to a typing, Lemma 4.6 entails that for any typing judgement t, there exists a full substitution instance of t within $[t]$. Using the Substitution Lemma and the definition of $\prec_S$ it is not too difficult to show the following useful lemma (also used in Appendix B)

Lemma 4.7 *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 $S_2(t_2) \prec_{S_1} t_2$.*

If $t_1 \prec_{S_1} t_2$ and $t_2 \prec_{S_2} t_1$ where S_1 is a renaming on $\mathrm{Var}(t_1)$ and S_2 is a renaming on $\mathrm{Var}(t_2)$, 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 4.8 *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 4.7 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 substituted, it follows that S_1 is a renaming on $\mathrm{Var}(t_1)$ and since t_2 is fully substituted, it follows that S_2 is a renaming on $\mathrm{Var}(t_2)$. 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 wish to strengthen the conclusion of Lemma 4.8. Before doing so, we must consider cycle elimination in constraint sets. Given constraint set C we can regard $C \cup P$ as a digraph in the obvious way. We say that an atomic set C is *cyclic* if there are atoms $A_1, \ldots, A_n$, $n \geq 2$, with $A_1 = A_n$ and $A_i \leq A_{i+1} \in C \cup P$ for $i = 1 \ldots n-1$. Note that "loops" of the form $A \leq A$ can occur in an acyclic set.

The proof of the following lemma uses that judgements are consistent. One can then show that it is sound to eliminate cycles in an atomic constraint set, by substituting all elements of a cycle to a single atom (compare also [13].) This leads to

Lemma 4.9 *If $t = C, \Gamma \vdash_P M : \tau$ is fully substituted with C consistent, then C is acyclic.*

Define the following operations on atomic constraint sets:

$$\begin{array}{lcl} C^\circ & = & C \setminus (\{A \leq A \mid A \leq A \in C\} \cup \\ & & \{b \leq b' \in C \mid b, b' \in P\}) \\ C^* & = & \text{transitive closure of } C \\ C^- & = & \bigcap\{C' \mid (C')^* = C^*\} \end{array}$$

In case C is acyclic, C^- is the *transitive reduction* of C (see [2]), which uniquely satisfies $(C^-)^* = C^*$ and whenever $(C')^* = C^*$ then $C^- \subseteq C'$. One can show that if C is atomic and consistent with P, then $C \sim_P (C^\circ)^-$, and if, in addition, C is acyclic, then $C' \sim_P C$ implies $(C^\circ)^- \subseteq C'$.

The following proposition is the central technical result in this section. Perhaps surprisingly, the proof is somewhat tricky; it is given in Appendix B.1. The tricky part is to prove properties (*ii*) and (*iii*) of the proposition.

Proposition 4.10 *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* $\Gamma_1(x) = S(\Gamma_2(x))$ *for all* $x \in Dm(\Gamma_1)$.

Theorem 4.11 *(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 4.8 and Proposition 4.10) 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 $S(\Gamma(x)) = R(S'(\Gamma'(x)))$ for all $x \in Dm(\Gamma)$.

Since $S(C)$ must be acyclic (by Lemma 4.9) and $S(C)$ is consistent, we have $((S(C))^\circ)^- \subseteq R(S'(C'))$. We have shown $\tilde{t} \lesssim_{RoS'} t'$, thereby proving the theorem. □

5 Minimization

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 so-called G- and S-transformations defined by Fuh and Mishra in [13].

The main technical result is Theorem 5.6 below, which shows that under certain restricted conditions, S-simplification is complete for minimization. This result is used in a critical way in the lower bound proof in Section 6. Second, we show (Theorem 5.3 below) that, unless P = NP, there can be no polynomial time procedure for computing minimal typings. This underlines the claim made earlier that $\approx$ validates quite powerful simplifications. Third, we show that the G- and S-transformations taken together are incomplete for minimization. This shows that minimality as defined here (Definition 4.3) is not reducible to these simplifications.

We need a few definitions in order to introduce the G- and S-transformations. Given typing $t = C, \Gamma \vdash_P M : \tau$, let the *observable types* in t, denoted $\mathrm{Obv}(t)$, be the constants in P and type variables appearing in Γ or τ, i.e., $\mathrm{Obv}(t) = \mathrm{Var}(\Gamma) \cup \mathrm{Var}(\tau) \cup P$. Let the *internal variables* in t, denoted $\mathrm{Intv}(t)$, be the set $\mathrm{Intv}(t) = \mathrm{Var}(C) \setminus \mathrm{Obv}(t)$. Then G-transformation changes only the internal variables of a typing, whereas S-transformation changes only the observable variables of a typing.

5.1 G-simplification and G-minimization

There are two kinds of G-transformation, one stronger than the other. The weaker one is called *G-simplification*, the stronger one is called *G-minimization*. They both eliminate *internal* variables from constraint sets. To recall from [13] the definition of G-simplification, let

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

and

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

Given variable $\alpha \in \mathrm{Var}(C)$, we say that α is *G-subsumed* by an atom A with respect to C, written $\alpha \leq_{\mathbf{g}} A$, iff $\uparrow_C(\alpha) \setminus \{\alpha\} \subseteq \uparrow_C(A)$ and $\downarrow_C(\alpha) \setminus \{\alpha\} \subseteq \downarrow_C(A)$. The transformation $\mapsto_{\mathbf{g}}$ on typings is then defined as follows. Let $t_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$ and $t_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$. Then $t_1 \mapsto_{\mathbf{g}} t_2$ iff $C_2 = C_1\{\alpha \mapsto A\}$ where $\alpha \in \mathrm{Intv}(C_1)$, $\alpha \neq A$ and $\alpha \leq_{\mathbf{g}} A$ with respect to C_1. Note that $\Gamma_1 = \Gamma_2$ and $\tau_1 = \tau_2$, whenever $t_1 \mapsto_{\mathbf{g}} t_2$. If $t_1 \mapsto_{\mathbf{g}} t_2$, then $t_1 \approx t_2$, as shown in [13].

G-simplification can be regarded as an efficiently computable [5] approximation to a more powerful transformation

[5] A low order polynomial time procedure is in [13].

$\mapsto_{\mathbf{g}^+}$ called *G-minimization*, which can be defined as follows. Given typing $t = C, \Gamma \vdash_P M : \tau$, let Subs($t$) be the set of substitutions S such that

$$S(\alpha) = \begin{cases} \alpha & \text{if } \alpha \notin \text{Intv}(t) \\ A \in \text{Var}(C) \cup P & \text{if } \alpha \in \text{Intv}(t) \end{cases}$$

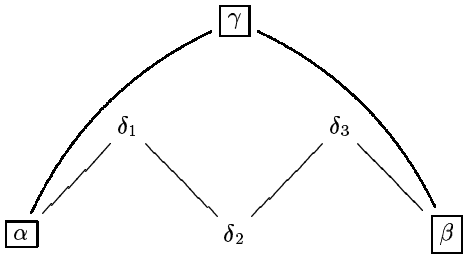
Then $t_1 \mapsto_{\mathbf{g}^+} t_2$ iff there exists $S \in \text{Subs}(t_1)$ such that $C_2 = S(C_1)$, S is not a renaming on $\text{Var}(C_1)$ and $C_1 \vdash_P S(C_1)$.

Observe that both $\mapsto_{\mathbf{g}}$ and $\mapsto_{\mathbf{g}^+}$ are terminating, since the size of Intv(t) shrinks at every reduction step. Normalization with respect to $\mathbf{g}^+$ is accomplished by the procedure *minimize* in [13].

Fuh and Mishra [13] remark that finding a $\mathbf{g}^+$-normal form of a typing t appears to require exhaustive search through Subst(t) in the worst case, which, if true, would mean that G-minimization is exponential in the size of $C \cup P$. We strengthen this remark technically in Theorem 5.3 below.

We first give examples that illustrate the strength of $\mapsto_{\mathbf{g}}$. The first example shows why $\mapsto_{\mathbf{g}}$ is an incomplete approximation to $\mapsto_{\mathbf{g}^+}$. The example shows that, in order to G-minimize, it is not sufficient to consider substitutions with singleton support[6] and this explains, in part, why G-minimization is hard to compute (Theorem 5.3.)

Example 5.1 *Consider constraint set C below, where observable variables are shown inside a box:*



Call a substitution S simple *if Supp(S) is a singleton set. It can be seen that C is G-*simplified *(i.e., in normal form with respect to $\mapsto_{\mathbf{g}}$), because it is not possible to find a simple, non-identity substitution S of the form $\{\delta_i \mapsto A\}$, $A \in \{\alpha, \beta, \gamma, \delta_1, \delta_2, \delta_3\}$, such that $C \vdash_P S(C)$. However, the set C is* not *G-*minimal*, because the substitution*

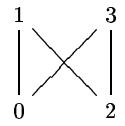
$$S_{min} = \{\delta_1 \mapsto \gamma, \delta_2 \mapsto \gamma, \delta_3 \mapsto \gamma\}$$

satisfies $C \vdash_P S_{min}(C)$. Note that even though S_{min} can be factored into a sequence of simple substitutions, as $S_{min} = \{\delta_1 \mapsto \gamma\} \circ \{\delta_2 \mapsto \gamma\} \circ \{\delta_3 \mapsto \gamma\}$, still there is no simple non-identity substitution S' which satisfies $C \vdash_P S'(C)$. □

The second example introduces an important poset, called 2-*crown*. This poset will play a major rôle both in the proof of Theorem 5.3 below and in the lower bound proof in Section 6. Pratt and Tiuryn [27] studied so-called n-*crowns* to show that, for some finite posets P, the problem P-SAT is NP-complete (P-SAT is: given atomic constraint set C over P, determine if C is satisfiable in P.) This result holds for $P = n$-*crown*, for all $n \geq 2$, and it has been used by several researchers in the study of the complexity of subtype inference [31, 21, 8, 9].

[6] A related phenomenon is noticed by Pottier [26] for simplification with recursive types.

Example 5.2 *(2-crowns)*
A 2-crown is the poset with 4 elements $0, 1, 2, 3$ ordered as shown below:



Let C be the set of internal variables, $C = \{\alpha \leq \gamma, \beta \leq \gamma, \alpha \leq \delta, \beta \leq \delta\}$. Viewing C as a poset, it constitutes a 2-crown of variables. The reader can verify that $C \mapsto_{\mathbf{g}}^* \{\alpha \leq \alpha\}$ by using the successive G-subsumptions $\beta \leq_{\mathbf{g}} \alpha, \gamma \leq_{\mathbf{g}} \alpha, \delta \leq_{\mathbf{g}} \alpha$. □

Let G-minimization be the following problem: given an atomic constraint set C with a subset of Var(C) designated as observable, compute a $\mathbf{g}^+$-normal form of C. The following theorem is proven in Appendix B.2.

Theorem 5.3 *If $P \neq NP$, then G-minimization cannot be computed in polynomial time for any partial order P.*

Theorem 5.3 also holds for minimization (i.e., computing minimal typings in the sense of Definition 4.3), since we can always create situations where no observable variable can be substituted under minimization, by having both positive and negative occurrences of the observable variables in the type of the typing. For such typings minimization degenerates to G-minimization [7]

5.2 S-simplification

S-simplification eliminates observable variables from constraint sets. Given atomic constraint set C, type τ, variable α and atom A, we say that α is *S-subsumed* by A in C and τ, written $\alpha \leq_{\mathbf{s}} A$, iff

1. *either* $C \vdash_P A \leq \alpha$ and α does not occur negatively in τ and $\downarrow_C(\alpha) \setminus \{\alpha\} \subseteq \downarrow_C(A)$
2. *or* $C \vdash_P \alpha \leq A$ and α does not occur positively in τ and $\uparrow_C(\alpha) \setminus \{\alpha\} \subseteq \uparrow_C(A)$

If $t = C, \Gamma \vdash M : \tau$, with $dom(\Gamma) = \{x_1, \ldots, x_n\}$, then we define the type $\text{clos}(t) = \Gamma(x_1) \to \ldots \to \Gamma(x_n) \to \tau$. We then define, following Fuh and Mishra [13], the reduction $\mapsto_{\mathbf{s}}$ on typings: Let $t_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$ and $t_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$; then

$$t_1 \mapsto_{\mathbf{s}} t_2 \text{ iff } \begin{cases} (1) & \alpha \leq_{\mathbf{s}} A \text{ in } C_1 \text{ and clos}(t_1) \\ (2) & C_2 = C_1\{\alpha \mapsto A\} \\ (3) & \Gamma_2 = \Gamma_1\{\alpha \mapsto A\} \\ (4) & \tau_2 = \tau_1\{\alpha \mapsto A\} \\ (5) & \{\alpha \mapsto A\} \text{ not renaming on } t_1 \end{cases}$$

Notice that $\alpha \leq_{\mathbf{s}} A$ implies $\alpha \leq_{\mathbf{g}} A$ but not vice versa. The stronger conditions for $\leq_{\mathbf{s}}$ ensure that the soundness property $t \mapsto_{\mathbf{s}} t' \Rightarrow t \approx t'$ holds.

Example 5.4 *Let C be as in Example 5.2. Then the reader can verify (using the definition of $\leq_{\mathbf{s}}$) that C cannot be S-simplified. The reason for this has to do with the fact that for $\alpha \leq_{\mathbf{s}} A$ to hold, it must be the case that α and A are comparable under the hypotheses in C.* □

[7] We are assuming that any combination of constraint sets and types can be given as input to minimization. Still, our results appear to be indicative of the difficulty of minimization in practice

It turns out that S-simplification has an interesting completeness property, which can be used, under certain conditions, to give easy proofs that typings are minimal. The main technical result is in the following proposition. The proof can be found in Appendix B.3.

Proposition 5.5 *Let $t = C, \Gamma \vdash_P M : \tau$ and $\tau' = clos(t)$. Assume S is not the identity on $Var(t)$ with*

(1) $C \vdash_P S(C)$

(2) $C \vdash_P S(\tau') \leq \tau'$

(3) $Supp(S) \subseteq Obv(t)$

(4) *C is acyclic*

Then there exists $\alpha \in Supp(S)$ such that $\alpha \leq_{\mathbf{s}} S(\alpha)$ with respect to C and τ'.

Using Proposition 5.5, one can prove (the proof is in Appendix B.4)

Theorem 5.6 *(Completeness of S)*
If $t = C, \Gamma \vdash_P M : \tau$ is an S-simplified typing with C acyclic and $Var(C) \subseteq Obv(t)$, then the typing $t_0 = (C^\circ)^-, \Gamma \vdash_P M : \tau$ is minimal.

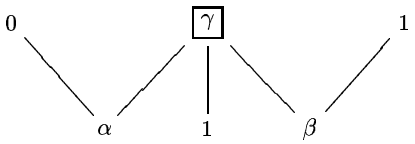
Notice that (as promised in Section 4) Theorem 5.6 immediately implies that the typing t_4 from Example 3.2 is minimal for $\lambda x.x$ and that typing t_2 from Example 3.3 is minimal for *comp*.

Theorem 5.6 is used in Section 6 below, where we prove an exponential lower bound for the size of constraint sets in principal typings.

5.3 Incompleteness of G and S

Theorem 5.6 naturally leads to the question whether *G-minimization* together with S-simplification is complete for minimization in the sense of Definition 4.3. By Example 5.7 below the answer is negative, *independently* of P; the example shows that in minimization one cannot treat optimization of internals and observables separately without loosing completeness.

Example 5.7 (Incompleteness of G-minimization and S-simplification) *Let P be any non-trivial poset (i.e., P not discretely ordered.) 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, C is 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 $Var(C)$, then S cannot satisfy $C \vdash_P S(C)$; moreover, C is also S-simplified, since for no $A \neq \gamma$ do we have $\gamma \leq_{\mathbf{s}} A$. However, with $S_{min} = \{\alpha \mapsto 0, \beta \mapsto 0, \gamma \mapsto 1\}$ we do have $C \vdash_P S_{min}(C)$, because $S_{min}(C) \subseteq P$. Hence, any typing t of the form $t = C, \emptyset \vdash_P M : \tau$ with $Var(\tau) = \{\gamma\}$ and γ occurring positively (and not negatively) in τ, cannot be fully substituted and hence it cannot be minimal, since $S_{min}(t) \prec t$.* □

6 The size of principal typings

We prove a tight worst case exponential lower bound for the dag-size of both constraint sets and types in principal typings (Theorem 6.2 below.) To the best of our knowledge, the best lower bound previously proven is the linear lower bound for a whole class of sound instance relations shown in [17]. The results of Section 4 as well as Theorem 5.6 are important ingredients in the lower bound proof to be presented below.

For $\lambda^{\rightarrow}$ we know (see [20]) that while *textual* type size can be exponential, *dag-size* (see, e.g., [20] and [25]Chapter 11.3) is at most linear. The basic property responsible for this is that the type system of $\lambda^{\rightarrow}$ imposes enough equality constraints, in the sense of [34], that sharing yields exponential succinctness. To take a contrasting example, 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, 25] with further references.) Subtyping is a system based on *in*equalities, and, as shown below, this leads to exponential dag-size of typings (constraint 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 $n \geq 0$, the principal typing of $\mathbf{Q}_n$ generated by a certain standard procedure 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 5.6 and the characterization of minimal typings of Section 4, which are employed in order to prove that the same property in fact holds for *all* principal typings of $\mathbf{Q}_n$, viz. that *any* principal typing must have a number of variables in both constraint set and type which is exponentially dependent on the size of the terms.

6.1 Preliminaries to the construction

For the purpose of the following development we shall first assume that we have a conditional construct 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.

We shall consider the form of principal typings of terms. We use the fact, shown in [24], that a principal typing can always be obtained by the following *standard procedure*:

(1) first 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 [24] for details) is applied to the extracted constraint set (the match-step may fail, but if so then the term has no typing with atomic subtyping at all) and finally

(3) *decompose* the matching constraints into atomic constraints, using the Decomposition Lemma (any matching set can be decomposed)

Once steps (1) through (3) have been performed, we can then apply transformations such as G and S to the typing, 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.

6.2 The construction

We want to construct a series of terms $\mathbf{Q}_n$, such that for all $n \geq 0$, any principal typing of $\mathbf{Q}_n$ must contain an exponential number of distinct type variables in both constraint set and type. It is not difficult to write down a series of lambda terms that will produce types of textual size exponential in the size of the terms. A well known (see [20, 25]) series of terms with this property is $\mathbf{D}^n M$, the n-fold application of the "duplicator" $\mathbf{D} = \lambda x.\langle x, x\rangle$ to M. However, the principal typings of these terms, as derived by the standard procedure in the subtype system, can be simplified (using, in fact, only G-and S-simplification) in such a way that we get back their simple types of linear dag-size. This leads to the idea that, in order to get an exponential blow-up in dag-size of types in *every possible* principal typing, we must somehow produce a series of terms with the following properties

1. The terms must in some uniform way generate exponentially large sets of constraints which cannot be simplified much, and

2. The terms must be such that we can easily tell what their minimal typings look like

The second point is there, because in the absence of this property we should have difficulty *proving* that *all* possible representatives of the principal typings must be of exponential dag-size. To achieve these goals, we construct the terms $\mathbf{Q}_n$ essentially in such a way that the standard procedure will generate an exponential number of 2-*crowns* of *observable* variables in the constraint sets. Observable 2-*crowns* cannot be simplified (recall Example 5.4), and we can then invoke Theorem 5.6 to argue that the *minimal* typing of the $\mathbf{Q}_n$ must have exponentially many distinct variables. From this the lower bound will follow from the definition of minimality.

We proceed to give the construction. Let $cond_{x,y}$ denote the expression with two free variables x and y, given by

$$cond_{x,y} = \mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle x, y\rangle\ \mathit{else}\ \langle y, x\rangle$$

For a term variable f, define the expression $\mathbf{P}_f$ with free variable f as follows:

$$\begin{array}{rcl}\mathbf{P}_f & = & \lambda z.\mathbf{K} \\ & & (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle\rangle \\ & & \mathit{else}\ \langle\langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle, z\rangle) \\ & & (f\ z)\end{array}$$

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 $\lambda f_{[n]}.M = \lambda f_n.\ldots.\lambda f_1.M$ 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

$$\begin{array}{rcl}\mathbf{Q}_1 & = & \lambda f_1.\lambda x.\lambda y.(\lambda z.\mathbf{K} \\ & & (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle\rangle \\ & & \mathit{else}\ \langle\langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle, z\rangle) \\ & & (f_1 z)) \\ & & cond_{x,y}\end{array}$$

To prepare for a proof that the $\mathbf{Q}_n$ behave as claimed, let α and β be two distinct variables and let $\upsilon_0, \upsilon_1, \ldots, \upsilon_k, \ldots$ and $\omega_0, \omega_1, \ldots, \omega_k, \ldots$ be two distinct enumerations of infinitely many type variables (so, all the ω_i are different from all the υ_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

$$\upsilon_m, \upsilon_{m+1}, \ldots, \upsilon_{m+2^n-1}$$

from left to right. So, for instance, T_0^0 is just the variable υ_0, T_1^0 is the type $\upsilon_0 * \upsilon_1$, T_2^0 is the type $(\upsilon_0 * \upsilon_1) * (\upsilon_2 * \upsilon_3)$, 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

$$\begin{array}{rcl}\tau^{[0]} & = & \alpha \to (\beta \to T_1^0), \\ \tau^{[n]}(n > 0) & = & (\sigma_n \to \omega_n) \to \ldots (\sigma_1 \to \omega_1) \to \\ & & \alpha \to \beta \to T_{n+1}^{k+1}\end{array}$$

where σ_i is a renaming of T_i^0 for $i = 1 \ldots n$, using fresh variables which occur nowhere else in the type. We assume that the σ_i use the variables $\omega_1, \omega_2, \ldots, \omega_k$.

We show by induction on $n \geq 0$ that

Lemma 6.1 (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 $C_0, \emptyset \vdash_P \lambda x.\lambda y.cond_{x,y} : \alpha \to (\beta \to \upsilon_0 * \upsilon_1)$ with $C_0 = \{\alpha \le \upsilon_0, \alpha \le \upsilon_1, \beta \le \upsilon_0, \beta \le \upsilon_1\}$. 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:

$$\begin{array}{l}\lambda x : \alpha.\lambda y : \beta.\mathit{if}\ \mathtt{true}\\ \mathit{then}\ \langle \uparrow_\alpha^{\upsilon_0} x, \uparrow_\beta^{\upsilon_1} y\rangle\\ \mathit{else}\ \langle \uparrow_\beta^{\upsilon_0} y, \uparrow_\alpha^{\upsilon_1} x\rangle\end{array}$$

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 5.6 that the typing is the *minimal* principal typing for $\mathbf{Q}_0$, and this typing satisfies the claim of the lemma for the case $n = 0$.

For the inductive case, consider $\mathbf{Q}_{n+1} =$

$$\begin{array}{l}\lambda f_{n+1}.\lambda f_{[n]}.\lambda x.\lambda y.(\lambda z.\mathbf{K}\\ \quad (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle\rangle\\ \quad \mathit{else}\ \langle\langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle, z\rangle)\\ \quad (f_{n+1} z))\\ \quad (\mathbf{P}^n cond_{x,y})\end{array}$$

By induction hypothesis for $\mathbf{Q}_n$, $(\mathbf{P}^n cond_{x,y})$ has a minimal principal typing with type T_{n+1}^m (for some m) and coercion set C_n with at least 2^{n+1} distinct variables, assuming the appropriate types $\sigma_i \to \omega_i$ 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*.) Consider the type which must be assumed for f_{n+1}. Because $(\mathbf{P}^n cond_{x,y})$ is applied to $\lambda z. \ldots$, the type of z must be T_{n+1}^m, and due to the application $(f_{n+1} z)$, the type assumed for f_{n+1} must therefore have the form $\tau_1 * \tau_2 \to \gamma$, where $\tau_1 * \tau_2$ is a renaming of T_{n+1}^m (using fresh variables, and γ fresh.) However, since every variable in $\tau_1 * \tau_2$ occurs only positively in the type of the entire expression $\mathbf{Q}_{n+1}$, it follows by S-simplification that $\tau_1 * \tau_2$ can be identified with T_{n+1}^m, and hence we need apply no coercion to z or f_{n+1}. Using elimination of internal variables by G-simplification, it can be seen that a principal completion of $\mathbf{Q}_{n+1}$ is obtained by inserting coercions as follows (assuming suitable coercions inside $(\mathbf{P}^n cond_{x,y})$)

$$\begin{array}{l}\lambda f_{n+1} : T_{n+1}^m \to \gamma.\lambda f_{[n]} : [\sigma_n'].\lambda x : \alpha.\lambda y : \beta.\\ (\lambda z : T_{n+1}^m.\mathbf{K}\\ \quad (\mathit{if}\ \mathtt{true}\ \mathit{then}\\ \quad \langle \uparrow_{T_1*T_2}^{\theta_1*\theta_2} z, \langle \mathtt{2nd}\ \uparrow_{T_1*T_2}^{T_1*\theta_3} z, \mathtt{1st}\ \uparrow_{T_1*T_2}^{\theta_4*T_2} z\rangle\rangle\\ \quad \mathit{else}\\ \quad \langle\langle \mathtt{2nd}\ \uparrow_{T_1*T_2}^{T_1*\theta_1} z, \mathtt{1st}\ \uparrow_{T_1*T_2}^{\theta_2*T_2} z\rangle, \uparrow_{T_1*T_2}^{\theta_3*\theta_4} z\rangle)\\ \quad (f_{n+1}\ z))\\ \quad (\mathbf{P}^n cond_{x,y})\end{array}$$

Here T_1 and T_2 are such that $T_1 * T_2 = T_{n+1}^m$, 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 $\lambda f_i : \sigma_i \to \omega_i$, $i = 1 \ldots n$. 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:

$$\begin{array}{lcl}C_1 & = & \{T_1 \le \theta_1, T_1 \le \theta_2, T_2 \le \theta_2, T_2 \le \theta_1\}\\ C_2 & = & \{T_1 \le \theta_4, T_1 \le \theta_3, T_2 \le \theta_3, T_2 \le \theta_4\}\end{array}$$

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

$$\begin{array}{l}(T_{n+1}^m \to \gamma) \to \sigma_n' \to \ldots \to \sigma_1' \to\\ \alpha \to \beta \to ((\theta_1 * \theta_2) * (\theta_3 * \theta_4))\end{array}$$

(with $\sigma_i' = \sigma_i \to \omega_i$) which can be renamed to $\tau^{[n+1]}$. Since, by induction, all variables in C_n are observable in $\tau^{[n]}$, 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 5.6 that the typing shown for $\mathbf{Q}_{n+1}$ is the minimal one. This completes the proof of the lemma. □

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.

Eliminating pairing and projections is more subtle. A first attempt might be to use the standard lambda-calculus encodings (see [6]Chapter 6.2), taking an encoded pair of M and N to be $\langle\!|\, M, N \,|\!\rangle$ with the definition

$$\langle\!|\, M, N \,|\!\rangle = \lambda p.(p\ M)\ N$$

and $\mathtt{fst} = \lambda x.\lambda y.x$, $\mathtt{snd} = \lambda x.\lambda y.y$. This will not work, however, because the application of a variable z of encoded pair-type $(\tau \to \sigma \to \theta) \to \theta$ (corresponding to $\tau * \sigma$) to both projections will force $\tau = \sigma$; this is well-known in simple types and the same identification is also made (via valid simplifications) in subtyping. [8] For example, the expression

$$M = (\lambda z.\langle\!|\, z\,(\lambda x.\lambda y.x), z\,(\lambda x.\lambda y.y) \,|\!\rangle)\ \langle\!|\, x, y \,|\!\rangle$$

[8] This can be explained in terms of the Curry-Howard isomorphism, because one cannot define logical conjunction as a derived notion from implication in minimal logic (see, e.g., [28].)

gets principal typing

$$\emptyset, \{x : \alpha, y : \alpha\} \vdash_P M : (\alpha \to \alpha \to \beta) \to \beta$$

This identification of types will render the lower-bound proof invalid for these encodings. However, instead of writing expressions such as

$$\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle\rangle\ \mathit{else}\ \langle\langle \mathtt{2nd}\ z, \mathtt{1st}\ z\rangle, z\rangle$$

we can do without the projections, by using the term

$$\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle\!\langle z, s\rangle\!\rangle, \langle\!\langle z, t\rangle\!\rangle\rangle\!\rangle\ \mathit{else}\ \langle\!\langle\langle\!\langle s, z\rangle\!\rangle, \langle\!\langle t, z\rangle\!\rangle\rangle\!\rangle$$

where s and t are distinct, free variables. Summing up, we use the definitions

$$\begin{aligned}
eqty(M,N) &= \lambda x.\mathbf{K}(xM)(xN)\\
\langle\!\langle M, N\rangle\!\rangle &= \lambda p.(pM)N\\
\mathbf{P}_{f,s,t} &= \lambda z.\mathbf{K}\\
&\quad eqty(\langle\!\langle\langle\!\langle z, s\rangle\!\rangle, \langle\!\langle z, t\rangle\!\rangle\rangle\!\rangle, \langle\!\langle\langle\!\langle s, z\rangle\!\rangle, \langle\!\langle t, z\rangle\!\rangle\rangle\!\rangle)\\
&\quad (f\ z)\\
\mathbf{P}^{n+1}N &= \mathbf{P}_{f_{n+1},s_{n+1},t_{n+1}}(\mathbf{P}^n N)\\
\mathbf{Q}_n &= \lambda f_{[n]}.\lambda s_{[n]}.\lambda t_{[n]}.\lambda x.\lambda y.\mathbf{P}^n\, eqty(x,y)
\end{aligned}$$

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, G- and S-simplification still leaves an exponential number of observable 2-*crowns*, and Theorem 5.6 can therefore be used as before.

We can now prove

Theorem 6.2 *For any poset P of base types it holds for arbitrarily large n, that there exist closed terms of length $O(n)$ such that any principal typing (wrt. $\prec$) for the terms has a constraint set containing $2^{\Omega(n)}$ distinct observable type variables and a type containing $2^{\Omega(n)}$ distinct type variables.*

PROOF. Take the series of terms $\mathbf{Q}_n$; clearly, the size of $\mathbf{Q}_n$ is of the order of n, and Lemma 6.1 shows that $\mathbf{Q}_n$ has a minimal principal typing t with constraint 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 $t \overset{<}{\sim} t'$, by minimality of t. By the definition of $\overset{<}{\sim}$, this entails that t' must have at least as many distinct type variables in both constraint set and type as t. □

The theorem immediately implies that the *dag-size* of constraint 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 [24] and [13] that a principal atomic typing can be obtained by extracting a set C of (possibly non-atomic) constraints 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 6.3 *For any poset P of base types, the dag-size of constraint sets as well as of types in atomic principal typings (wrt. $\prec$) is of the order $2^{\Theta(n)}$ in the worst case.*

7 Concluding remarks

This paper was written as part of a research effort to clarify which subtyping systems may become practicable in larger scale. However, the lower bound result does not by itself answer this question definitively. For example, type inference for ML teaches us to be cautious in drawing conclusions from theoretical lower bounds. ML has a DEXPTIME-complete type inference problem, and yet this appears to be no problem in actual ML-programs (see [25]Chapter 11.3 with further references.) On the other hand, experience does seem to suggest that subtyping systems can be difficult to scale up and that the simplification problem is critical (see references in the Introduction.)

An open problem is to give lower bounds for the size of types and constraint sets for principal typings with respect to notions of equivalence that are more powerful than $\approx$. We are currently investigating this problem, and we conjecture that some of the techniques presented in this paper can be used to prove exponential lower bounds for at least some of the more powerful frameworks one could naturally suggest.

8 Related work

Most closely related to the present work is that of Hoang and Mitchell [17] and that of Fuh and Mishra [13]. In [17], the authors prove a linear lower bound on the size of principal typings for a whole class of sound instance relations. Strong lower bounds for this class of relations are expectedly harder to obtain than corresponding lower bounds for the framework studied here, since less information is available about instance related typings. Also, the techniques used by Kanellakis, Mairson and Mitchell in [20] to analyze type size for simply typed lambda calculus and ML are related to our technique for the lower bound proof in Section 6, but the main part of that proof relies on new techniques tailored for subtyping. The pioneering work by Fuh and Mishra [13] on subtype simplification provided important background in the form of their S- and G-transformations. Fuh and Mishra operate with a notion of minimal typings, but their notion is defined in terms of their transformations, and, as shown by our incompleteness results, this notion is distinct from ours. As far as we know, our notion of minimality is the first purely logical notion, which relates all typings in a given equivalence class, independently of particular transformations. Recent work by Pottier [26], by Trifonov and Smith [32] and by Aiken, Wimmers and Palsberg [5] introduce more powerful notions of simplification for polymorphic constrained types, based on semantic subtyping and entailment relations. Moreover, the paper by Aiken, Wimmers and Palsberg initiates a systematic study of completeness properties of simplification procedures.

Acknowledgements

Foremostly, I am pleased to acknowledge my great debt to Fritz Henglein, and I am grateful for our countless discussions about type theory. Thanks are due to Christian Mossin for a discussion about terms that generate hard constraint sets. Thanks also to Jerzy Tiuryn for helpful comments on [31], to Vaughan Pratt for helpful comments on [27] and to my referees and Sergei Soloviev for helpful comments on this paper.

$$[const] \quad C \vdash_P b \leq b' \text{, provided } b \leq_P b'$$

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

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

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

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

Figure 1: Subtype logic

$$[var] \quad C, \Gamma \cup \{x : \tau\} \vdash_P x : \tau$$

$$[base] \quad C, \Gamma \vdash_P c : \text{TypeOf}(c)$$

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

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

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

Figure 2: Subtype system

A Type system

The subtyping system shown in Figure 2 uses, in the rule [*sub*] for coercion, a standard logic $\vdash_P$, shown in Figure 1, for deriving subtyping relations of the form $\tau \leq \tau'$ between types, given assumptions C of atomic subtyping relations of the form $A \leq A'$. The rules are parametric in $(P, \leq_P)$, which is a given finite poset of base types, containing relations such as, e.g., `int` $\leq$ `real`. The rule [*base*] assumes a function TypeOf assigning types to term constants. A judgement $C, \Gamma \vdash_P M : \tau$ is a *derivable atomic subtyping judgement* iff it is derivable in the proof system of Figure 2 *and* C *is atomic*, i.e., C contains only subtyping hypotheses of the form $A \leq A'$ relating atomic types.

B Proofs

In the sequel we use the following property several times: if C is a constraint 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., $\alpha \in \text{Var}(C)$. 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 [*ref*].

B.1 Proof of Proposition 4.10

We first state two technical lemmas without proof. We then we give a main lemma including its proof, since it is enlightening. Finally, we use the lemmas to prove Proposition 4.10.

Lemma B.1 *Let C_1, C_2 be atomic constraint 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 $Var(C_i)$. Then $C_1 \sim_P S_2(C_2)$ and $C_2 \sim_P S_1(C_1)$.*

Lemma B.2 *(Anti-symmetry)*
Let C be atomic. If $C \cup P$ contains no proper cycle, then $C \vdash_P \tau \leq \tau'$ and $C \vdash_P \tau' \leq \tau$ imply $\tau = \tau'$.

The main lemma follows:

Lemma B.3 *Let S be a substitution and C an atomic constraint set with variable $\alpha \in Var(C)$. Assume*

(i) S is a renaming on $Var(C)$

(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 $\alpha \in \text{Var}(C)$, $S(\alpha) \neq \alpha$, (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

$$D_n \subseteq Var(C)$$

(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 $n > 0$ (full details are in [29].)

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 $V \subseteq \text{Var}(C)$, 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. □

We are now ready to prove

Proposition 4.10 *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* $\Gamma_1(x) = S(\Gamma_2(x))$ *for all* $x \in Dm(\Gamma_1)$.

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 easily from Lemma B.1 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

$$C_1 \vdash_P S_2(S_1(\tau_1)) \leq \tau_1 \tag{1}$$

Now, we know from Lemma 4.7 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 substituted that

$$S_2 \circ S_1 \text{ is a renaming on } \mathrm{Var}(t_1) \tag{2}$$

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

$$C_1 \vdash_P S_2(S_1(C_1)) \tag{3}$$

We now *claim* that in fact we have

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

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

$$S_2(S_1(\tau_1)) \neq \tau_1 \tag{4}$$

Then there is a variable occurrence α in τ_1 such that

$$(S_2 \circ S_1)(\alpha) \neq \alpha \tag{5}$$

Let $A = (S_2 \circ S_1)(\alpha)$. Then, because of (1), the Decomposition Lemma entails that α and A must be comparable under C_1, and hence

$$\text{either } C_1 \vdash_P S_2(S_1(\alpha)) \leq \alpha \text{ or } C_1 \vdash_P \alpha \leq S_2(S_1(\alpha)) \tag{6}$$

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

$$\alpha \in \mathrm{Var}(C_1) \tag{7}$$

since otherwise α and A could not be comparable under C_1. Finally, by Lemma 4.9, we know that

$$C_1 \text{ is acyclic} \tag{8}$$

because t_1 is assumed to be fully substituted. Now, (2), (3), (6), (7) and (8) allow us to apply Lemma B.3 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 (5), and so we must reject the assumption (4), 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

$$C_1 \vdash_P \tau_1 \leq S_2(\tau_2) \tag{9}$$

We know (8) that C_1 is acyclic. But then $C_1 \vdash_P S_2(\tau_2) \leq \tau_1$ (which holds by the assumptions) and (9) show (using Lemma B.2 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. □

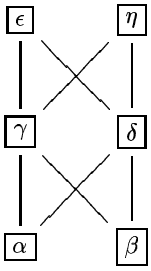


Figure 3: Constraint set D with all variables observable

B.2 Proof of Theorem 5.3

We sketch the proof of Theorem 5.3. The details left out are not so difficult to reconstruct from the sketch, but full details are in [29]. We are to prove:

Theorem 5.3 *If $P \neq NP$, then G-minimization cannot be computed in polynomial time for any partial order P.*

PROOF. Recall from Example 5.2 (and [27]) the poset called *2-crown*. In [27] it is shown that the problem 2-*crown*-SAT (satisfiability of atomic inequalities in 2-*crown*) is NP-complete. We will show that, if G-minimization could be computed in polynomial time, then 2-*crown*-SAT could also be solved in polynomial time, *independently* of the poset P.

Fix the constraint set D shown in Figure 3 with variables $\alpha, \beta, \gamma, \delta, \epsilon, \eta$, all designated as observable. Note that D can be regarded as two copies (in variables) of 2-*crown* which are "spliced together" at γ and δ, and, in particular, the lower part $\{\alpha, \beta, \gamma, \delta\}$ of D is isomorphic to 2-*crown* under the embedding $\pi = \{0 \mapsto \alpha, 1 \mapsto \gamma, 2 \mapsto \beta, 3 \mapsto \delta\}$.

Let C be any atomic constraint set over 2-*crown*, and assume w.l.o.g. that $\mathrm{Var}(C) \cap \mathrm{Var}(D) = \emptyset$. We also assume w.l.o.g. that all constraint sets considered are irreflexive (i.e., they have no loops.) Translate C to $\overline{C}$ with

$$\overline{C} = D \cup \pi(C) \cup \{\alpha' \leq \epsilon, \alpha' \leq \eta \mid \alpha' \in \mathrm{Var}(C)\}$$

with all $\alpha' \in \mathrm{Var}(C)$ designated as internal variables in $\overline{C}$. Let $\widetilde{C}$ be a $\mathbf{g}^+$-normal form of $\overline{C}$. Then one can show that

$(*)$ C is satisfiable in 2-*crown* if and only if $D \vdash_\emptyset \widetilde{C}$

The theorem follows from $(*)$ by NP- completeness of 2-*crown*-SAT, since the condition $D \vdash_\emptyset \widetilde{C}$ can be checked in polynomial time by computing transitive closure (see, e.g., [24].) □

B.3 Proof of Proposition 5.5

We are to prove

Proposition 5.5 *Let $t = C, \Gamma \vdash_P M : \tau$ and $\tau' = clos(t)$. Assume S is not the identity on $Var(t)$ with*

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

(*ii*) $C \vdash_P S(\tau') \leq \tau'$

(*iii*) $Supp(S) \subseteq Obv(t)$

(iv) 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

$$\neg\exists\alpha \in \mathrm{Supp}(S).\ \alpha \leq_s S(\alpha) \text{ wrt. } C \text{ and } \tau' \qquad (10)$$

Pick any $\alpha \in \mathrm{Supp}(S)$ (by assumption $\mathrm{Supp}(S) \neq \emptyset$), so we have $\alpha \neq S(\alpha)$. By (ii) and (iii), α must be comparable to $S(\alpha)$ under C. Using these facts, it is easy to verify (by induction on τ', using (iv) acyclicity of C) that (ii) 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)

$$(*) \qquad \begin{array}{l} C \vdash_P S(\alpha) \leq \alpha \\ \text{with } \alpha \text{ not occurring negatively in } \tau' \end{array}$$

By $C \vdash_P S(\alpha) \leq \alpha$ and $S(\alpha) \neq \alpha$ we get that, if $S(\alpha)$ is not a constant, then $S(\alpha)$ must be a variable occurring in C. Therefore $\{\alpha \mapsto S(\alpha)\}$ cannot be a renaming on t. By (10) we must then 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

$$A_1 \notin \downarrow_C(S(\alpha)) \qquad (11)$$

In particular, we therefore have

$$A_1 \neq S(\alpha), A_1 \neq \alpha \text{ and } C \vdash_P A_1 \leq \alpha \qquad (12)$$

By the Substitution Lemma, we then have $S(C) \vdash_P S(A_1) \leq S(\alpha)$, and so, by (i), we also have

$$C \vdash_P S(A_1) \leq S(\alpha) \qquad (13)$$

If $A_1 = S(A_1)$, then by (13) it would follow that $C \vdash_P A_1 \leq S(\alpha)$, hence $A_1 \in \downarrow_C(S(\alpha))$ in contradiction with (11). Therefore we must have

$$A_1 \neq S(A_1) \qquad (14)$$

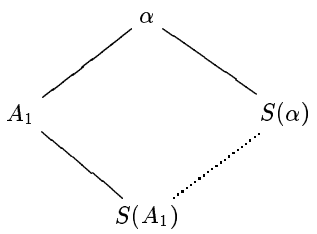
from which it follows that

$$A_1 \text{ is a variable in } \mathrm{Supp}(S) \qquad (15)$$

Now, by (15) and assumption (iii), A_1 is an observable variable, occurring in τ'. Then (ii) implies that A_1 and $S(A_1)$ must be comparable under the hypotheses C. We already know (14) that $A_1 \neq S(A_1)$, and if $C \vdash_P A_1 \leq S(A_1)$, then it would follow by (13) that $C \vdash_P A_1 \leq S(\alpha)$ in contradiction with (11). We must therefore conclude that

$$\begin{array}{l} C \vdash_P S(A_1) \leq A_1 \\ \text{with no negative occurrence of } A_1 \text{ in } \tau' \end{array} \qquad (16)$$

Summing up so far, we now have the situation



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 (16) 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 $A_2 \neq A_1$ 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 (iv). We must therefore reject the assumption (10), thereby proving the Proposition. □

B.4 Proof of Theorem 5.6

We are to prove

Theorem 5.6 *If $t = C, \Gamma \vdash_P M : \tau$ is an S-simplified typing with C acyclic and $Var(C) \subseteq Obv(t)$, then the typing $t_0 = (C^\circ)^-, \Gamma \vdash_P M : \tau$ is minimal.*

PROOF. We will show that any typing t_0 satisfying the conditions of the theorem must be fully substituted. Let $C_0 = (C^\circ)^-$. Now suppose that S is a non-renaming substitution on t_0 such that $S(t_0) \prec t_0$. We can assume w.l.o.g. that S is the identity on variables not appearing in t_0. Since $\mathrm{Var}(C) \subseteq \mathrm{Obv}(t)$ it follows that $\mathrm{Supp}(S) \subseteq \mathrm{Obv}(t_0)$. Since $S(t_0) \prec t_0$, there is a substitution S' such that

1. $C_0 \vdash_P S' \circ S(C_0)$
2. $C_0 \vdash_P S' \circ S(\mathrm{clos}(t_0)) \leq \mathrm{clos}(t_0)$

Since S is not a renaming on t_0, it follows that $S' \circ S$ is not a renaming on t_0, and then Proposition 5.5 shows that t_0 is not S-simplified. This contradicts the assumption that t is S-simplified. Therefore, any substitution S such that $S(t_0) \in [t_0]$ must be a renaming on t_0. This establishes that t_0 is fully substituted. Theorem 4.11 then implies that t_0 is minimal. □

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[2] 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.

[3] A. Aiken and E.L. Wimmers. Type inclusion constraints and type inference. In *Proceedings FPCA '93, Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 31–42, June 1993.

[4] A. Aiken, E.L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon.* ACM Press, January 1994.

[5] A. Aiken, E.L. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. Technical Report UCB/CSD-96-909, University of California, Berkeley, July 1996.

[6] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[7] 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.

[8] Marcin Benke. Efficient type reconstruction in the presence of inheritance. In *Mathematical Foundations of Computer Science (MFCS)*, pages 272–280. Springer Verlag, LNCS 711, 1993.

[9] Marcin Benke. Some complexity bounds for subtype inequalities. Technical Report TR 95-20 (220), Warsaw University, Institute of Informatics, Warsaw University, Poland, December 1995.

[10] P. Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Parc, February 1990.

[11] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA '95*, 1995.

[12] M. Fähndrich and A. Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints, Cambridge MA*, 1996.

[13] 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.

[14] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science (TCS)*, 73:155–175, 1990.

[15] C.A. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[16] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In *Proc. European Symposium on Programming (ESOP), Edinburgh, Scotland*. Springer-Verlag, April 1994. Also DIKU Semantics Report D-198.

[17] 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.

[18] P. Hudak and J. Fasel. A gentle introduction to Haskell. *Sigplan Notices*, 27(5):Section T, 1992.

[19] S. 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 Press, June 1992. also in LISP Pointers, Vol. V, Number 1, January-March 1992.

[20] P. Kanellakis, H. Mairson, and J.C. 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.C. 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] R. Milner, M. Tofte., and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[23] J.C. Mitchell. Coercion and type inference (summary). In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, pages 175–185, 1984.

[24] J.C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[25] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[26] F. Pottier. Simplifying subtyping constraints. In *Proceedings ICFP '96, International Conference on Functional Programming*, pages 122–133. ACM Press, May 1996.

[27] V. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. *Studia Logica (to appear)*.

[28] D. Prawitz. *Natural deduction*. Almquist & Wiksell, Uppsala 1965.

[29] J. Rehof. Minimal typings in atomic subtyping. Technical Report D-278, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark. Available at http://www.diku.dk/research-groups/ topps/personal/rehof/publications.html, 1996.

[30] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[31] J. Tiuryn. Subtype inequalities. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science (LICS), Santa Cruz, California*, pages 308–315. IEEE Computer Society Press, June 1992.

[32] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings SAS '96, Static Analysis Symposium, Aachen, Germany*, pages 349–365. Springer, 1996. Lecture Notes in Computer Science, vol.1145.

[33] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, 1985. Springer. Lecture Notes in Computer Science, Vol. 201.

[34] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.