Syntactic Properties of Polymorphic Subtyping

Fritz Henglein
DIKU, University of Copenhagen
Universitetsparken 1
2100 Copenhagen
Denmark
Email: henglein@diku.dk

May 13, 1996

Abstract

Subtyping is used in language design, type checking and program analysis. Mitchell and others have studied the foundations of *structural* subtyping (primitive subtype relations only between type constants) in a monomorphic setting; that is, there is no let-construct that admits polymorphic use of variables in their languages.

Both Kaes and Smith investigate (ML-)polymorphic extensions of subtyping, though embedded in a system with overloading. Somewhat surprisingly, polymorphic subtyping has to date not been studied in isolation, and some of its central syntactic properties have not been investigated generally and systematically.

In this paper we study polymorphic subtyping, where the subtyping theory is *not* required to be structural. We observe that type schemes with subtyping qualifications are strictly necessary in order to obtain principal typing. We identify a new instance relation on typing judgements, the *halbstark* relation. It is a hybrid, lying in strength between Mitchell's original instance relation and Fuh and Mishra's lazy instance relation. We present a sound and complete type inference algorithm in the style of Milner's Algorithm W. The significance of the halbstark relation emerges from the fact that the algorithm is *generic* in that it admits replacing typing judgements by *any* halbstark-equivalent judgements at any point. This provides a generalized correctness argument for Algorithm W independent of any particular constraint simplification strategy chosen.

Finally, we show that polymorphic typing judgements are preserved under let-unfolding, let-folding, and η-reduction, but not in general under β-reduction. The latter holds, though, if the subtyping discipline has the *decomposition* property, which says that two function types are in a subtype relation only if their domain and range types are in the appropriate contra-/covariant subtype relation.

1 Introduction

We study the polymorphic generalization of monomorphic subtyping, as studied by Mitchell [Mit84, Mit91] and Fuh and Mishra [FM88, FM89, FM90]. In contrast to them we shall not, however, assume that the subtyping theory is atomic; that is, has only primitive subtyping relations between type constants.

In this introduction we first recall some of the salient properties of monomorphic subtyping, leading us to the observation that a polymorphic generalization must allow for *subtype-qualified* type schemes. In Section 2 we present the basic typing rules for polymorphic subtyping and present a *generic* syntax-directed algorithm, modeled after Algorithm W, that is syntactically sound and complete for the typing rules.

The main novelty of Section 2 is the identification of the importance of a hybrid instance relation on typing judgements, which combines Mitchell's original instance relation [Mit84] with Fuh and Mishra's *lazy* instance relation [FM89]. We call it the *halbstark* instance relation, and the induced equivalence the halbstark-equivalence on typing judgements. Algorithm W computes a typing judgement $C, A \vdash e : \tau$ for given type environment A and expression e (C is a set of subtype constraints, and τ is a simple type). In practice, the constraints (and types) produced by type inference are *simplified* in order to reduce the number of constraints generated and the number of constraints copied for each use of a let-bound variable x [FM89], also in nonstructural subtyping theories [AW93, Pot96]. In principle, any such simplification strategy used has to be subjected to a new proof of correctness when used inside a polymorphic type inference algorithm.

Our Algorithm W is generic in that it admits *arbitrary* transformation of typing judgements as long as the transformed judgement is halbstark-equivalent with the original one. This entails that *any* simplification strategy is correct — that is, preserves soundness *and* completeness — as long as the simplification strategy preserves halbstark-equivalence. The simplification strategies of G-simplification and S-simplification — though somewhat restricted — of Fuh and Mishra preserve halbstark equivalence. So do the simplifications described by Smith. One particular consequence of Algorithm W's correctness is the Principal Type Property.

In Section 3 we show that typing judgements are closed under various transformations of the underlying expression: η-reduction, let-unfolding and let-folding. β-reduction does *not* generally preserve typing judgements; it does, though, if function types are in a subtype relation only if the domain and range types are in the appropriate subtype relation. This property holds not only in structural subtyping systems, but also in many nonstructural ones, such as ones with recursive types. One particular consequence of these subject reduction/expansion results is that typing is invariant under let-folding and unfolding. In particular, we can always find the (principal) type of an expression under given assumptions by first eliminating all let-expressions by unfolding and then applying *monomorphic* subtype inference. This is analogous to the relation of ML (Damas-Milner) to simple (Curry-Hindley) typing, and it has analogous consequences; for example, determining whether a closed expression has a simple type over an atomic subtyping theory where the primitive subyping form a forest of lattices on type constants is DEXPTIME-complete, by application of results of Kanellakis, Mairson, Mitchell [Mai90, KMM91], Kfoury, Tiuryn, Urzyczyn [KTU90] and Tiuryn [Tiu92].

1.1 Principality

Note: The remaining parts of the introduction presuppose some knowledge of monomorphic subtyping. All the relevant terms and notation for this paper are introduced in the following section, however.

There are two commonly known notions of instance relations between typing judgements: the *strong* (Fuh/Mishra, lazy) instance relation and the *weak* (Mitchell) instance relation. A derivable typing judgement $t \equiv (C, A \vdash_P e : \tau)$ is *principal* for a given instance relation $\preceq$ if any other derivable judgement for e is an instance of t. A principal typing is thus a *single* description for *all* the possible typings a given expression can have. In particular, all the other typings for e can be *derived* from a principal one, if we add the admissible rule

$$\text{INST} \quad \frac{C, A \vdash_P e : \tau}{C', A' \vdash_P e : \tau'} \quad (\text{if } (C, A \vdash_P e : \tau) \preceq (C', A' \vdash_P e : \tau'))$$

to our typing rules.[1]

But why should we be interested at all in having a single description for a typing from which all other typings are derivable? Consider an expression e' that has an occurrence of e; that is, $e' \equiv C[e]$ for some context $C[]$. In the process of deriving a typing for e' we have to derive a judgement for e. Which one of the possibly many typings for e is required depends on the context $C[]$; for example, if $e' = ee''$ then the typing for e better be such that e has a function type that is appropriate for application to e''. The critical point, if not the defining property of principal typings, is that we can factor *any* typing for e whatsoever through a principal one. So, we can first "blindly" infer a principal typing judgement for e without any regard to (or need to know) the particular context in which e occurs, and then we can *adapt* that typing by applying the INST-rule.

This is necessary to avoid having to "redo" typing derivations in two typical situations:

1. There is more than one occurrence of an expression e in e'; in this case we can infer a principal typing once and for all for e and then adapt – possibly in different ways! – that typing to the different local contexts in which e occurs. So we avoid having to do type inference for e several times.

2. There is no known context for e at all: Imagine e is a library function that can be called "later" and thus occurs in a priori unknown contexts. We can infer a principal typing for e (and even compile e and throw away the source expression — thus there is no going back to e to do type inference at a later point) and leave it to the later users to adapt that typing to the respective uses. If f is an identifier referring to e then the fact that the

[1]Given an inference system I a (new) rule R is *admissible (in R)* if I extended with R is conservative over I; that is, only judgements that are derivable in I are derivable in the extended system (though, obviously, there may be more derivations for a particular judgement in the extended system).

It is important to distinguish admissible rules from *derivable* rules: a (new) rule R is derivable in I if the conclusion of R is derivable from the premises of R by using the rules from I. Note that a rule that is derivable in R will be derivable in any extension of R whereas a rule admissible in R may not be admissible in some extension of R.

chosen typing for e is principal guarantees that any use of f in context $C[]$ can be typed if and only if $C[e]$ itself can be typed.

To summarize, principal typings make it possible to proceed modularly (sometimes also called by "local" analysis): find out and express all the relevant information about an expression (this is a principal typing in our setting) once and for all and use that information in any relevant context without going back to reanalyze the expression.

1.2 Coercion sets in judgements

We could save ourselves quite a bit of technical complications by restricting (monomorphic) subtying theory to judgements without coercion sets; that is, $A \vdash_P e : \tau$, with a rule

$$[\text{SUB}] \quad \frac{A \vdash_P e : \tau \qquad \vdash_P \tau \leq \tau'}{A \vdash_P e : \tau'}.$$

To be sure, we would still introduce coercions, satisfying substitutions, and other concepts, but the coercions would only exist at the *meta-level* (that is, at the level of us using them to solve the typability problem for subtyping, just as we'd be using any other mathematical objects), not at the *object level* where they occur as part of the formal judgements that enter into derivations.[2]

The sole reason why we have coercion sets is to be able to express principal typings. Fuh and Mishra [FM89] have argued that coercion sets are not only useful to express principal typings, they are also *necessary.* Consider, for example, $P = \{\mathit{integer}, \mathit{real}\}$ ordered by $\mathit{integer} \leq \mathit{real}$. The combinator

$$\text{twice} \equiv \lambda f.\lambda x.f(fx)$$

has only principal typings with nonempty coercion sets; in other words, there is no typing $\vdash_P$ twice : τ that has *all* the typings for twice as (strong) instances. Note for example, that both $\vdash_P$ twice : $(\alpha \to \alpha) \to (\alpha \to \alpha)$ and $\vdash_P$ twice : $(\mathit{real} \to \mathit{integer}) \to (\mathit{real} \to \mathit{integer})$ are derivable, but the least common generalization of $(\alpha \to \alpha) \to (\alpha \to \alpha)$ and $(\mathit{real} \to \mathit{integer}) \to (\mathit{real} \to \mathit{integer})$, which is $(\gamma \to \delta) \to (\gamma \to \delta)$, is not a type of twice.

Indeed Hong and Mitchell [HM95] show that the need for coercions exists for any sound instantiation relation and that the minimum number of required coercions in principal typings grows with the size of the expressions.

1.3 Polymorphism

The purpose of *(predicative) polymorphism* is to describe all the (simple) types an expression has and to derive any particular type from that description. This is critical for modularity in that we do not need to know the points of all uses of some definition to type it. There are actually two requirements we expect to hold of such *type schemes*:

[2]Indeed, in subtyping without coercion sets the notion of *satisfiablity* would be inherently built into the type system. To wit: if $C, A \vdash_P e : \tau$ is principal for e then there exists A', τ' such that $A' \vdash_P e : \tau'$ if and only if C is satisfiable.

1. (Universality) The type scheme has to describe all possible types of the expression in a given environment. In this case the type scheme is *principal* for the expression (in the environment).

2. (Nonemptiness) The type scheme for an expression describes at least one type; that is, it has at least one simple type instance. If this is not the case then we call e untypable.

A third and in practice important requirement is that type schemes should be as "descriptive" and comprehensible as possible to a programmer.

In the following we shall *not* insist on nonemptiness of type schemes in order to separate the question of finding a type scheme from that of determining whether it describes a nonempty set of types. The latter depends on the particular structure of primitive subtyping relations on the given ground types, whereas the former *does not.* So: we shall admit type schemes and typing judgements with unsatisfiable type schemes, that is with type schemes that do not have a single valid simple type instance. We shall sketch at the end however, how satisfiability can be checked in a separate phase after type inference without regard to satisfiability.

Let us consider an example. The instance relation for simple typing, as defined by Hindley, is

$$(A \vdash e : \tau) \preceq_{CH} (A' \vdash e : \tau')$$

if for some substitution S we have

1. $A' \vdash S(A)$ and

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

Hindley [Hin69] and Ben-Yelles [BY79] show that every simply typable expression (without **let**) has a principal typing under $\preceq_{CH}$. ML, which has polymorphically typable let-expressions, can be seen as an "internalization" of the instance relation. In ML type schemes are prenex-quantified types $\forall\alpha_1 \ldots \alpha_n.\tau$. Substitution is captured by substituting types for the α_i. The corresponding typing rule is

$$\frac{A \vdash e : \forall\alpha_1 \ldots \alpha_n.\tau}{A \vdash e : \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}.$$

The explicit quantification expresses which variables may be instantiated and which may *not.* The generalization rule introduces those type variables that may be instantiated:

$$\frac{A \vdash e : \tau}{A \vdash e : \forall\alpha_1 \ldots \alpha_n.\tau} \quad (\text{if } \alpha_1, \ldots, \alpha_n \text{ not free in } A).$$

The reason for the asymmetric treatment of type variables occurring in A and those occurring only in τ is that we think of the types occurring in A as *fixed.* This is to make sure that the principal typing $y : \tau \vdash e : \tau'$ of an expression e with free variable y is not instantiated several times with completely different types for y — after all these are not different y's. This generalization of simple typing to ML type schemes is sufficient for ML since the underlying logic of simple typing is purely equational.

These general considerations are there to emphasize the purpose of principality, and of polymorphism as an internalization of principality. In subtyping it should thus come as no surprise that we cannot simply tack on quantifiers to types and expect to have a suitable system of polymorphism.

1.4 ML polymorphism is not good enough

Consider

$$e \equiv \textbf{let}\ \text{twice} = \lambda f.\lambda x.f(fx)\ \textbf{in}\ (\text{twice floor } 5.0, \text{twice succ } 1)$$

where floor : $\mathit{real} \rightarrow \mathit{integer}$, succ : $\mathit{integer} \rightarrow \mathit{integer}$ with P as before. If we view **let** ... **in** ... as abbreviation for the corresponding β-redex then it is easy to see that e is not typable. Intuitively, this is due to the "stickiness" effect of monomorphic type systems: all the actual arguments to a function are merged inside the function body and flow as a single compound argument (set of arguments) through the function body. The operations in the body must be possible for each component (element) of such a compound argument. In the case above floor and succ together are bound to f and 5.0 and 1 together are bound to x. The problem is now that the application of f to x is not well-typed since the succ component of f cannot be applied to the 5.0 component of x. Clearly in the typing system we have "confused" the two applications of twice, since succ is in actuality only applied to 1 (and floor to 5.0), but that's what the monomorphic typing rule for functions expresses!

Polymorphism helps to disentangle different call contexts of a function. If we add the ML-polymorphic rules of the previous section for introducing and eliminating prenex-quantified type schemes we still cannot type e, however! The reason is that there is no (ML-)type scheme for twice that contains "enough" types for twice as instances to type e. For example, with twice : $\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ the application twice floor is not typable. With twice : $(\mathit{real} \rightarrow \mathit{integer}) \rightarrow (\mathit{real} \rightarrow \mathit{integer})$ the application twice succ is not typable. This is not surprising since we have already seen that twice has no principal typing without coercions, and an ML-polymorphic type scheme describes only such coercion-free typings.

Our conclusion is: We must internalize coercions in the type schemes for subtyping. This is what the theory of polymorphic typing with subtype constraints is about.

2 Unrestricted polymorphic typing with subtype constraints

In this section we present the unrestricted theory of polymorphic typing with subtype constraints. What makes it unrestricted is that we do not restrict the coercions in coercion assumptions to have a particular form — such as atomic — and we do not require that coercion sets be consistent or satisfiable. We do not even require that the subtyping theory be *structural*; that is, that P only relate base types to each other. We shall address these considerations briefly later.

The unrestricted theory is an excellent point of departure for studying restricted variations, since it allows us factor type inference into the inference of a

$$
\begin{array}{ll}
[\text{CONST}] & \dfrac{}{C \vdash_P \tau \leq \tau'} \quad (\text{if } \tau \leq_P \tau') \\[2ex]
[\text{REF}] & \dfrac{}{C \vdash_P \tau \leq \tau} \\[2ex]
[\text{HYP}] & \dfrac{}{C \cup \{\tau \leq \tau'\} \vdash_P \tau \leq \tau'} \\[2ex]
[\text{TRANS}] & \dfrac{C \vdash_P \tau \leq \tau' \qquad C \vdash_P \tau' \leq \tau''}{C \vdash_P \tau \leq \tau''} \\[2ex]
[\text{ARROW}] & \dfrac{C \vdash_P \tau_1' \leq \tau_1 \qquad C \vdash_P \tau_2 \leq \tau_2'}{C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}
\end{array}
$$

Figure 1: Subtype Logic

type scheme, including its simplification, independent of the particular subtyping theory, and the solution of subtyping constraints, which depends critically on the nature of the given subtyping theory.

2.1 Subtype logic

We assume we have a nonempty set of *base types* b, *type variables* α and *(simple) types* τ built from

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

Furthermore, we are given a partial order P on ground types, which defines our subtyping theory. The subtype inference rules are given in Figure 1.

2.2 Typing rules for constrained type schemes

Our language of discourse is defined by the following grammar for *expressions* e:

$$e ::= c \mid x \mid \lambda x.e \mid ee' \mid \mathbf{let}\, x = e \,\mathbf{in}\, e'$$

where c ranges over a given set of *constants* and x over *(program) variables.* λx and $\mathbf{let}\, x$ are binding constructs. Expressions are identified up to α-equivalence. Thus there is always a way of writing an expression such that its free and bound variables are disjoint, and every variable is bound at most once. Henceforth we shall always assume that any expression has this form.

Type schemes are prenex quantified types with constraints on the substitutions:

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n : C.\tau$$

where C is a set of subtype constraints of the form $\tau \leq \tau'$. If $n = 0$ and $C = \emptyset$ then we identify the type scheme with τ. Every constant c is assumed to have a given closed type scheme σ_c. As with expressions we identify α-equivalent type schemes. Substitution of types for type variables in type schemes as well as of expresssions for program variables in expressions is defined as in λ-calculus so as to avoid capture of free variables.[3] We shall use τ, υ for types and σ for type

[3]Note that substitution is well-defined since a substitution applied to two α-equivalent type schemes results in α-equivalent type schemes.

[VAR] $$\frac{}{C,\Gamma \cup \{x:\sigma\} \vdash_P x:\sigma}$$

[BASE] $$\frac{}{C,\Gamma \vdash_P c:\sigma_c}$$

[ABS] $$\frac{C,\Gamma \cup \{x:\tau\} \vdash_P e:\tau'}{C,\Gamma \vdash_P \lambda x.e:\tau \rightarrow \tau'} \quad (\text{if } x \notin \mathit{domain}\,(\Gamma))$$

[APP] $$\frac{C,\Gamma \vdash_P e:\tau \rightarrow \tau' \qquad C,\Gamma \vdash_P e':\tau}{C,\Gamma \vdash ee':\tau'}$$

[LET] $$\frac{C,\Gamma \vdash_P e:\sigma \qquad C,\Gamma \cup \{x:\sigma\} \vdash e':\tau}{C,\Gamma \vdash_P \mathbf{let}\, x = e\, \mathbf{in}\, e':\tau} \quad (\text{if } x \notin \mathit{domain}\,(\Gamma))$$

[SUB] $$\frac{C,\Gamma \vdash_P e:\tau \qquad C,\Gamma \vdash_P \tau \leq \tau'}{C,\Gamma \vdash_P e:\tau'}$$

[GEN] $$\frac{C \cup C',\Gamma \vdash_P e:\tau}{C,\Gamma \vdash_P e:\forall\vec{\alpha}{:}C'.\tau} \quad (\text{if } \vec{\alpha} \text{ not free in } C,\Gamma)$$

[INST] $$\frac{C,\Gamma \vdash_P e:\forall\vec{\alpha}{:}C'.\tau \qquad C \vdash_P C'[\vec{\tau}/\vec{\alpha}]}{C,\Gamma \vdash_P e:\tau[\vec{\tau}/\vec{\alpha}]}$$

Figure 2: Typing rules

schemes.

Typing assumptions Γ are a set of *variable typings* of the form $x:\sigma$, where every variable has at most one typing in Γ. We write $\mathit{domain}\,(\Gamma)$ for $\{x : (\exists\sigma)\ (x:\sigma) \in \Gamma\}$. We write $FTV(X)$ for the set of free type variables occurring in X, where X can be a type scheme, typing assumptions, coercions or any other syntactic object.

The typing rules for expressions are given in Figure 2.

2.3 Closure properties of typing judgements

We now show that judgements are closed in a number of ways.

Lemma 2.1 *(Closure under substitution)*
If $C,\Gamma \vdash_P e:\sigma$ then $S(C), S(\Gamma) \vdash_P e:S(\sigma)$ for all substitutions S.

PROOF (Sketch) By rule induction on the definition of $C,\Gamma \vdash_P e:\sigma$. All rules but [GEN] and [INST] are completely straightforward. □

An immediate consequence of the lemma is that we can always move subtype qualifications in a type scheme to the coercion assumptions of a typing judgement:

Proposition 2.2 *Let $\vec{\alpha}$ not be free in C,Γ. Then:*

$$C,\Gamma \vdash_P e:\forall\vec{\alpha}{:}D.\tau \textit{ if and only if } C \cup D,\Gamma \vdash_P e:\tau.$$

Lemma 2.3 *(Closure under strengthening of coercion assumptions)*
If $D \vdash_P C$ and $C,\Gamma \vdash_P e:\sigma$ then $D,\Gamma \vdash_P e:\sigma$.

PROOF Straightforward, by rule induction on $C, \Gamma \vdash_P e : \sigma$. We replace C by D in every typing judgement of a derivation. In particular, in rule [SUB] we replace premise $C \vdash_P \tau \leq \tau'$ by $D \vdash_P \tau \leq \tau'$. We can do so since $D \vdash_P C$ (by assumption) and $C \vdash_P \tau \leq \tau'$ with transitivity of $\vdash_P$ gives $D \vdash_P \tau \leq \tau'$. □

Definition 2.4 (Subtyping logic for type schemes) Extend the subtyping logic of Figure 1 to type schemes as follows:

1. $C \vdash_P \sigma \leq \tau$ if $C, x : \sigma \vdash_P x : \tau$.
2. $C \vdash_P \sigma \leq \sigma'$ if $FTV(\sigma) \subseteq FTV(C, \sigma')$ and for all D, τ such that $D \vdash_P C$ and $D \vdash_P \sigma' \leq \tau$ we have $D \vdash_P \sigma \leq \tau$.

□

Note that the coercion assumptions in the definition are still of the form $\tau \leq \tau'$ where τ and τ' are simple types. The condition $FTV(\sigma) \subseteq FTV(C, \sigma')$ is necessary to avoid statements with "frivolous" coercion qualifications like $C \vdash_P \forall\alpha{:}\{\beta \leq \beta\}.\alpha \to \alpha \leq \forall\alpha.\alpha \to \alpha$. (Necessary in the proof of Lemma 2.6.)

The following proposition gives another definition of subtyping on type schemes.

Proposition 2.5 *Define*

$$C \vdash_P \sigma \leq' \sigma' \iff C, x : \sigma \vdash_P x : \sigma'$$

and add the following inference rule to the type system of Figure 2:

$$\text{[WEAK]} \quad \frac{C, \Gamma \vdash_P e : \sigma}{C \cup C', \Gamma \vdash_P e : \sigma}.$$

(Note that [WEAK] is admissible by Lemma 2.3.)

Then $C \vdash_P \sigma \leq \sigma'$ if and only if $C \vdash_P \sigma \leq' \sigma'$.

Lemma 2.6 *(Closure under subtyping)*

Let $C \vdash_P \sigma \leq \sigma'$. Then:

1. *If $C, \Gamma \vdash_P e : \sigma$ then $C, \Gamma \vdash_P e : \sigma'$.*
2. *If $C, \Gamma \cup \{x : \sigma'\} \vdash_P e : \sigma''$ then $C, \Gamma \cup \{x : \sigma\} \vdash_P e : \sigma''$.*

PROOF

1. Let $\sigma \equiv \forall\vec{\alpha}{:}D.\tau$ and $\sigma' \equiv \forall\vec{\beta}{:}D'.\tau'$ with $\vec{\beta}$ not free in C or Γ.

   By definition of $C \vdash_P \sigma \leq \sigma'$ we have $C \cup D' \vdash_P \sigma \leq \tau'$ since $C \cup D' \vdash_P C \cup \{\sigma' \leq \tau'\}$. This means that $C \cup D' \vdash_P D[\vec{\tau}/\vec{\alpha}] \cup \{\tau[\vec{\tau}/\vec{\alpha}] \leq \tau'\}$ for some vector of types $\vec{\tau}$.

   Assume $C, \Gamma \vdash_P e : \sigma$. We have to show that $C, \Gamma \vdash_P e : \sigma'$. By Lemma 2.3 we also have $C \cup D', \Gamma \vdash_P e : \sigma$, and we can build the following derivation:

$$\cfrac{\cfrac{\cfrac{C \cup D', \Gamma \vdash_P e : \sigma \qquad C \cup D' \vdash_P D[\vec{\tau}/\vec{\alpha}]}{C \cup D', \Gamma \vdash_P e : \tau[\vec{\tau}/\vec{\alpha}]} \qquad C \cup D' \vdash_P \tau[\vec{\tau}/\vec{\alpha}] \leq \tau'}{C \cup D', \Gamma \vdash_P e : \tau'}}{C, \Gamma \vdash_P e : \forall\vec{\beta}{:}D'.\tau' (= \sigma')}$$

2. By induction on the derivation of $C, \Gamma \cup \{x : \sigma'\} \vdash_P e : \sigma''$. The only two interesting cases are [VAR] and [GEN]. All other cases are straightforward.

$\square$

2.4 Instance relation

In this section we define a new instance relation on typing judgements, one that, as we shall see, is particularly suited to computing principal typings where the typing assumptions for free variables are *fixed*. The instance relation lies in strength between the strong and the weak instance relation. As a consequence we shall refer to it as the *halbstark* (or *halbschwach*) instance relation.[4]

2.4.1 Strong and weak instance relations

Let us recall the definitions of strong and weak instance.

Definition 2.7 (Strong and weak instance relation)
Let $t \equiv C, A \vdash_P e : \tau$ and $t' \equiv C', A' \vdash_P e : \tau'$.

1. We say that t' is an *(strong) instance* of t if there exists substitution S such that:
   (a) $C' \vdash_P S(C)$
   (b) $C', A' \vdash_P S(A)$
   (c) $C' \vdash_P S(\tau) \leq \tau'$
2. We say that t' is a *weak instance* of t if there exists substitution S such that:
   (a) $C' \vdash_P S(C)$
   (b) $A'|_{domain\,(A)} = S(A)$
   (c) $S(\tau) = \tau'$

$\square$

Here we have used the notation $C', A' \vdash_P S(A)$ to express that every $(x : \pi) \in S(A)$ is derivable from C', A'. If all the assumptions in A' and A are monomorphic then this is equivalent to the previously used condition $C' \vdash_P A'(x) \leq S(A(x))$ for all $x \in domain\,(A)$.

The weak instance relation is *weak* (does not relate a lot of judgements) because it demands that, after substitution, both expression type and typing assumptions of the compared judgements must be *equal*. The strong instance relation is *strong* (relates quite a lot of judgements) because it admits, after substitution, subtyping steps to be applied both to the free variables in the expression and to the type of the whole expression.

[4]German: *halb* = half, *stark* = strong, and *schwach* = weak.

2.4.2 The halbstark instance relation

Definition 2.8 (Halbstark instance relation)

1. We say $t' \equiv (C', \Gamma' \vdash_P e : \tau')$ is a *halbstark instance* of $t \equiv (C, \Gamma \vdash_P e : \tau)$ and write $t \preceq_{hs} t'$ if there exists a substitution S such that
   (a) $C' \vdash_P S(C)$,
   (b) $\Gamma' = S(\Gamma)$, and
   (c) $C' \vdash S(\tau) \leq \tau'$.
2. Let $\vec{\beta}$ not be free in C', Γ', and $\vec{\alpha}$ not free in C, Γ. We say $t' \equiv (C', \Gamma' \vdash_P e : \forall\vec{\beta}{:}D'.\tau')$ is a *halbstark instance* of $t \equiv (C, \Gamma \vdash_P e : \forall\vec{\alpha}{:}D.\tau)$ and write $t \preceq_{hs} t'$ if
$$(C \cup D, \Gamma \vdash_P e : \tau) \preceq_{hs} (C' \cup D', \Gamma' \vdash_P e : \tau').$$

$\square$

Note that, in contrast to monomorphic subtyping, Γ and Γ' here may contain assumptions of the form $x : \sigma$ where σ is a type scheme. The extension of the halbstark instance relation to type schemes expresses that the instances of a type scheme must be in the expected subset relation. Two characterizations of the halbstark instance relation are expressed in the following two propositions.

Proposition 2.9 *(Alternative definition of halbstark instance relation)*
Let $t \equiv (C, \Gamma \vdash_P e : \sigma)$ and $t' \equiv (C', \Gamma' \vdash_P e : \sigma')$. Define $t \preceq'_{hs} t'$ if forall D, υ such that $D \vdash_P C' \cup \{\sigma' \leq \upsilon\}$ there exists a substitution S such that $D \vdash_P S(C) \cup \{S(\sigma) \leq \upsilon\}$. Then: $t \preceq_{hs} t'$ if and only if $t \preceq'_{hs} t'$.

Proposition 2.10 *Let $t \equiv (\Gamma \vdash e : \sigma)$ and $t' \equiv (C', \Gamma' \vdash e : \sigma')$. We have $t \preceq_{hs} t'$ if and only if there exists S such that*

1. *$\Gamma' = S(\Gamma)$ and*
2. *$C' \vdash_P S(\sigma) \leq \sigma'$.*

It is easy to check that the halbstark instance relation is transitively closed:

Proposition 2.11 *(Closure under transitivity)*
If $t \preceq_{hs} t'$ and $t' \preceq_{hs} t''$ then $t \preceq_{hs} t''$.

In particular, if $t \preceq_{hs} t'$ via substitution S and $t' \preceq_{hs} t''$ via S' then $t \preceq_{hs} t''$ via $S' \circ S$. This shows that $\preceq_{hs}$ defines a preorder on typing judgements. We write $t \approx_{hs} t'$ if $t \preceq_{hs} t'$ and $t' \preceq_{hs} t$.

The halbstark instance relation is a *sound* instance relation in the sense of Hoang and Mitchell [HM95]. In particular, if t is derivable and t' is a halbstark instance of t then t' is also derivable.

$$\begin{array}{rcl}
W(\Gamma, x) & = & \textbf{if}\,(x : \forall\vec{\alpha}{:}C.\tau) \in \Gamma\,\textbf{then} \\
& & (C[\vec{\beta}/\vec{\alpha}], \Gamma \vdash x : \tau[\vec{\beta}/\vec{\alpha}]) \text{ with } \vec{\beta} \text{ fresh} \\
& & \textbf{else} \text{ fail} \\
\\
W(\Gamma, \lambda x.e') & = & \textbf{let}\,(C', \Gamma' \vdash e' : \tau') = W(\Gamma \cup \{x : \alpha\}, e') \\
& & \qquad \text{with } \alpha \text{ fresh} \\
& & \textbf{in}\,(C', \Gamma' \vdash \lambda x.e' : \alpha \to \tau') \\
\\
W(\Gamma, e'e'') & = & \textbf{let}\,(C', \Gamma' \vdash e' : \tau') = W(\Gamma, e')\,\textbf{in} \\
& & \textbf{let}\,(C'', \Gamma'' \vdash e'' : \tau'') = W(\Gamma', e'')\,\textbf{in} \\
& & (C' \cup C'' \cup \{\tau' \leq (\tau'' \to \alpha)\}, \Gamma'' \vdash e'e'' : \alpha) \\
& & \qquad \text{with } \alpha \text{ fresh} \\
W(\Gamma, \textbf{let}\, x = e'\,\textbf{in}\, e'') & = & \textbf{let}\,(C', \Gamma' \vdash e' : \tau') = W(\Gamma, e')\,\textbf{in} \\
& & \textbf{let}\,\vec{\alpha} = FTV(C', \tau') - FTV(\Gamma')\,\textbf{in} \\
& & \textbf{let}\,(C'', \Gamma'' \vdash e'' : \tau'') = W(\Gamma' \cup \{x : \forall\vec{\alpha}{:}C'.\tau'\} \\
& & \textbf{in}\,(C'', \Gamma'' \vdash \textbf{let}\, x = e'\,\textbf{in}\, e'' : \tau')
\end{array}$$

Figure 3: Algorithm W for unrestricted subtyping

2.5 Algorithm W

We shall now present an algorithm for computing a typing judgment for a given expression and a given environment. We call it *Algorithm W* since its functionality and purpose is completely analogous to Milner's famed Algorithm W [Mil78] (see also Damas's thesis [Dam84]).

Intuitively, Algorithm W returns for Γ and e a typing judgement $C, \Gamma \vdash e : \tau$ such that $C, \Gamma \vdash_P e : \tau$ (soundness), and any other typing $C', \Gamma' \vdash_P e : \tau'$, where Γ' is substitution instance of Γ, will have stronger coercions and a weaker type: $C' \vdash_P S(C)$ and $C' \vdash_P S(\tau) \leq \tau'$ for some substitution S (completeness). The formulation of Algorithm W we give here for unrestricted subtyping is particularly simple since we avoid having to return substitutions that have to be carefully composed. Instead, these substitutions are already applied in the result of a call to W.[5] Algorithm W is given in Figure 3. We shall always assume that the arguments to W are named apart. In particular, *domain*(Γ) is disjoint from the bound variables of e. It is easy to see that $W(\Gamma, e)$ fails if and only if there is no assumption in Γ for a free variable of e.

Proposition 2.12 *$W(A, e)$ fails if and only if there is a free variable in e for which A contains no assumption.*

Note that Algorithm W is completely independent of P. Furthermore, if $(C', \Gamma' \vdash e' : \tau') = W(A, e)$ then $e \equiv e'$ and $\Gamma = \Gamma'$. This means that we could have restricted ourselves to returning only the pair (C', τ'). The chosen formulation emphasizes that W returns a typing judgement and makes this easier to "see". Furthermore, Algorithm W as given serves as a basis for variations on

[5]For unrestricted subtyping we actually do not need to return a substitution even with the "classical" formulation of Algorithm W since the identity substitution is good enough. For atomic subtyping this is not the case, however.

subtyping. For atomic subtyping Γ' is not necessarily identical to Γ, and e' may be a so-called completion of e. So the available degree of freedom in returning also Γ' and e' and not only C' and τ' comes in handy in other situations.

2.5.1 Soundness of Algorithm W

As suggested by the notation, the results of Algorithm W are derivable typing judgements:

Lemma 2.13 *(Soundness of Algorithm W)*
Let P be any given partial order on ground types (not necessarily only base types). If $W(\Gamma, e)$ succeeds with $(C, \Gamma \vdash e : \tau)$ then $C, \Gamma \vdash_P e : \tau$.

PROOF By structural induction on e. □

2.5.2 Completeness of Algorithm W

Now we show that Algorithm W does not overcommit, but, in a sense, produces a most general typing judgement.

Lemma 2.14 *(Completeness of Algorithm W)*
If $C', S(\Gamma) \vdash_P e : \sigma'$ then $W(\Gamma, e)$ succeeds with $(C, \Gamma \vdash e : \tau)$ and we have $(C, \Gamma \vdash_P e : \tau) \preceq_{hs} (C', S(\Gamma) \vdash_P e : \sigma')$.

PROOF The proof is by induction on the derivation of $C', S(\Gamma) \vdash_P e : \tau'$. W.l.o.g. we assume that e has disjoint bound and free variables, and no variable is bound more than once. We cover only the case where the last rule applied to derive $C', S(\Gamma) \vdash_P e : \tau'$ is rule [LET].

Case [LET]: Let $C', S(\Gamma) \vdash_P \mathbf{let}\, x = e\, \mathbf{in}\, e' : \tau'$ be derived from $C', S(\Gamma) \vdash_P e : \sigma$ and $C', \Gamma \cup \{x : \sigma\} \vdash_P e' : \tau'$. By induction hypothesis we have $(C, \Gamma \vdash e : \tau) = W(\Gamma, e)$ and $(C, \Gamma \vdash_P e : \tau) \preceq (C', S(\Gamma) \vdash e : \sigma)$.

Since, by definition, $\Gamma \vdash_P : e : \forall\vec{\alpha}\!:\!C.\tau \approx_{hs} C, \Gamma \vdash_P e : \tau$ we have $(\Gamma \vdash_P e : \forall\vec{\alpha}\!:\!C.\tau) \preceq (C', S(\Gamma) \vdash e : \sigma)$. This in turn implies $C' \vdash_P S(\forall\vec{\alpha}\!:\!C.\tau) \preceq \sigma$ by Proposition 2.10, part two. In other words, $S(\forall\vec{\alpha}\!:\!C.\tau)$ is a subtype of σ. Since we can strengthen typing assumptions (Lemma 2.6) we get that $C', S(\Gamma) \cup \{x : S(\forall\vec{\alpha}\!:\!C.\tau)\} \vdash_P e' : \tau'$. Since $S(\Gamma) \cup \{x : S(\forall\vec{\alpha}\!:\!C.\tau)\}$ is a substitution instance (via S) of $\Gamma \cup \{x : \forall\vec{\alpha} : C.\tau\}$ we can apply the second induction hypothesis: $(C'', \Gamma \cup \{x : \forall\vec{\alpha} : C.\tau\} \vdash e' : \tau'') = W(\Gamma \cup \{x : \forall\vec{\alpha} : C.\tau\}, e')$ succeeds and $C', S(\Gamma) \cup \{x : S(\forall\vec{\alpha}\!:\!C.\tau\} \vdash_P e' : \tau'$ is an instance of $C'', \Gamma \vdash_P e' : \tau''$. This means there exists S' such that $C' \vdash_P S'(C'')$, $S(\Gamma) = S'(\Gamma)$ and $C' \vdash_P S'(\tau'') \leq \tau'$. But this means that $(C'', \Gamma \vdash \mathbf{let}\, x = e\, \mathbf{in}\, e' : \tau'') \preceq_{hs} (C', S(\Gamma) \vdash \mathbf{let}\, x = e\, \mathbf{in}\, e' : \tau')$. □

2.5.3 Optimizing Algorithm W

Clearly, both soundness (Lemma 2.13) and completeness (Lemma 2.14) of Algorithm W are preserved if we replace the result of a call $W(\Gamma, e)$ by any of its halbstark-equivalent typing judgements. An immediate consequence is that the *generic* version of Algorithm W, given in Figure 4, is correct. This opens the possibility of "optimizing" the results of Algorithm W by transformations that eliminate coercions.

$$
\begin{array}{rcl}
W(\Gamma, x) & = & \textbf{if}\,(x : \forall\vec{\alpha}{:}C.\tau) \in \Gamma\;\textbf{then} \\
 & & (C[\vec{\beta}/\vec{\alpha}], \Gamma \vdash x : \tau[\vec{\beta}/\vec{\alpha}]) \text{ with } \vec{\beta} \text{ fresh} \\
 & & \;\textbf{else}\;\; \text{fail} \\[1em]
W(\Gamma, \lambda x.e') & = & \textbf{let}\;(C', \Gamma' \vdash e' : \tau') \approx_{hs} W(\Gamma \cup \{x : \alpha\}, e') \\
 & & \qquad \text{with } \alpha \text{ fresh} \\
 & & \;\textbf{in}\;(C', \Gamma' \vdash \lambda x.e' : \alpha \to \tau') \\[1em]
W(\Gamma, e'e'') & = & \textbf{let}\;(C', \Gamma' \vdash e' : \tau') \approx_{hs} W(\Gamma, e')\;\textbf{in} \\
 & & \textbf{let}\;(C'', \Gamma'' \vdash e'' : \tau'') \approx_{hs} W(\Gamma', e'')\;\textbf{in} \\
 & & (C' \cup C'' \cup \{\tau' \leq (\tau'' \to \alpha)\}, \Gamma'' \vdash e'e'' : \alpha) \\
 & & \qquad \text{with } \alpha \text{ fresh} \\
W(\Gamma, \textbf{let}\;x = e'\;\textbf{in}\;e'') & = & \textbf{let}\;(C', \Gamma' \vdash e' : \tau') \approx_{hs} W(\Gamma, e')\;\textbf{in} \\
 & & \textbf{let}\;\vec{\alpha} = FTV(C', \tau') - FTV(\Gamma')\;\textbf{in} \\
 & & \textbf{let}\;(C'', \Gamma'' \vdash e'' : \tau'') \approx_{hs} W(\Gamma' \cup \{x : \forall\vec{\alpha}{:}C'.\tau'\} \\
 & & \;\textbf{in}\;(C'', \Gamma'' \vdash \textbf{let}\;x = e'\;\textbf{in}\;e'' : \tau')
\end{array}
$$

Figure 4: Generic Algorithm W for unrestricted subtyping

Proposition 2.15 *The (nondeterministic) generic Algorithm W is sound and complete; that is, both Lemma 2.13 and Lemma 2.14 hold.*

In practice the flexibility of replacing the typing judgement computed by Algorithm W by an (halbstark-)equivalent optimized one is only used in the call for **let**-expressions **let** $x = e'$ **in** e'': By reducing the number of type variables and coercions in the result $C', \Gamma' \vdash e' : \tau'$ we want to make the type scheme $\forall\vec{\alpha}{:}C'.\tau'$ bound to x in the second call to W as "compact" as possible. This is especially important if the variable x has several occurrences in e'', since for every occurrence of x a completely new generic instance of $\forall\vec{\alpha} : C'.\tau'$ is generated — see W when applied to variables: The fewer variables in $\vec{\alpha}$ and the fewer coercions in C' the less work has to be performed in this case.

2.6 The Principal Type Property

We can now show that every expression has, for given typing assumptions for its free variables, a type scheme from which all types can be derived using the "logical" inference rules in our type inference system: Rules [SUB], [GEN] and [INST]. If we throw in rule [WEAK] from Proposition 2.5 then all type *schemes* that an expression has can be derived from a principal type for e by using rules [SUB], [GEN], [INST] and [WEAK].

Theorem 2.16 *(Principal Type Property)*

Let Γ contain an assumption for each free variable of e. Then there is σ such that:

1. $\Gamma \vdash_P e : \sigma$ *and*
2. *for all σ', if $\Gamma \vdash_P e : \sigma'$ then $\vdash_P \sigma \leq \sigma'$.*

PROOF By Proposition 2.12 we know that $W(\Gamma, e)$ succeeds. Let $C, \Gamma \vdash e : \tau$ be the result of $W(\Gamma, e)$. We define $\sigma \equiv \forall\vec{\alpha}{:}C.\tau$ where $\vec{\alpha} = FTV(C, \tau) - FTV(\Gamma)$.

By Lemma 2.13 we have that $C, \Gamma \vdash_P e : \tau$. Since $\vec{\alpha} = FTV(C, \tau) - FTV(\Gamma)$ we can apply rule [GEN], which shows that $\Gamma \vdash_P e : \sigma$.

Assume $\Gamma \vdash_P e : \sigma'$. Clearly $\Gamma = S(\Gamma)$ for S the identity substitution. Thus, by Lemma 2.14 we can conclude that $(C, \Gamma \vdash_P e : \tau) \preceq_{hs} (\Gamma \vdash_P e : \sigma')$ and thus $(\Gamma \vdash_P e : \sigma) \preceq_{hs} (\Gamma \vdash_P e : \sigma')$. By Proposition 2.10 we can conclude that

1. $S'(\Gamma) = \Gamma$ and
2. $\vdash_P S'(\sigma) \leq \sigma'$

for some substitution S'. Since $S'(\Gamma) = \Gamma$, S' is the identity on the free type variables of Γ. Since $FTV(\sigma) = FTV(\Gamma)$ by definition of σ it follows that $S'(\sigma) = \sigma$, and we have $\vdash_P \sigma \leq \sigma'$. □

3 Typing judgements and expression transformation

In the previous section we have studied the structure of typing judgements that have the same expression. In this section we look at the interaction of typing judgements and transformation of expressions. We say derivable typing judgement $C, \Gamma \vdash_P e : \sigma$ is *preserved* under transformation of e to e' if $C, \Gamma \vdash_P e' : \sigma$ is also derivable.[6] We shall see that typing judgements are *preserved* under η-reduction, **let**-unfolding and even **let**-folding, with no particular requirements on P; that is, in unrestricted subtyping. Interestingly we shall see that this does not hold for β-reduction. If, however, P has the decomposition property, viz. $C \vdash_P (\tau \to \upsilon) \leq (\tau' \to \upsilon')$ if and only if $C \vdash_P \tau' \leq \tau$ and $C \vdash_P \upsilon \leq \upsilon'$, then β-reduction also preserves typing judgements. Note that the above property is satisfied by atomic subtyping (where P only relates base types to each other), but that it does not *require* atomic subtyping.

We can think of these results as giving proof that derivability is *semantically robust*: if we apply a local transformation to an expression, such as reducing a β-redex, eliminating an η-redex, unfolding or even folding a **let**-expression, then derivability is *preserved*: any judgement derivable for the original expression can still be derived by the transformed expression (though the transformed expression may actually have *more* derivable judgements than the original one).

3.1 Normalized derivations

It is easy to see that an application of [INST] that immediately follows an application of [GEN] can be eliminated.

To wit: Assume we have

$$\frac{\dfrac{C \cup C', \Gamma \vdash_P e : \tau}{C, \Gamma \vdash_P e : \forall\vec{\alpha}{:}C'.\tau} \qquad C \vdash_P C'[\vec{\tau}/\vec{\alpha}]}{C, \Gamma \vdash_P e : \tau[\vec{\tau}/\vec{\alpha}]}.$$

[6]Because of the historical study of such properties in Curry and Feys' book on combinatory logic [CF58] such properties are often called *subject reduction properties.*

$$\begin{array}{ll}
\text{[VAR/INST]} & \dfrac{C \vdash'_P C'[\vec{\tau}/\vec{\alpha}]}{C, \Gamma \cup \{x : \forall\vec{\alpha}{:}C'.\tau\} \vdash'_P x : \tau[\vec{\tau}/\vec{\alpha}]} \\[2ex]
\text{[BASE]} & \dfrac{}{C, \Gamma \vdash'_P c : \sigma_c} \\[2ex]
\text{[ABS]} & \dfrac{C, \Gamma \cup \{x : \tau\} \vdash'_P e : \tau'}{C, \Gamma \vdash'_P \lambda x.e : \tau \rightarrow \tau'} \quad (\text{if } x \notin \mathit{domain}\,(\Gamma)) \\[2ex]
\text{[APP]} & \dfrac{C, \Gamma \vdash'_P e : \tau \rightarrow \tau' \qquad C, \Gamma \vdash'_P e' : \tau}{C, \Gamma \vdash ee' : \tau'} \\[2ex]
\text{[GEN/LET]} & \dfrac{C \cup C', \Gamma \vdash'_P e : \tau \qquad C, \Gamma \cup \{x : \forall\vec{\alpha}{:}C'.\tau\} \vdash e' : \tau'}{C, \Gamma \vdash'_P \mathbf{let}\, x = e \,\mathbf{in}\, e' : \tau'} \quad (*) \\[2ex]
\text{[SUB]} & \dfrac{C, \Gamma \vdash'_P e : \tau \qquad C, \Gamma \vdash'_P \tau \leq \tau'}{C, \Gamma \vdash'_P e : \tau'}
\end{array}$$

Figure 5: Normalized typing rules

By Lemma 2.1 we can derive

$$C \cup C'[\vec{\tau}/\vec{\alpha}], \Gamma \vdash_P e : \tau[\vec{\tau}/\vec{\alpha}]$$

and, by Lemma 2.3, also

$$C, \Gamma \vdash_P e : \tau[\vec{\tau}/\vec{\alpha}]$$

without this pair of rule applications.

Consequently we can confine applications of rule [GEN] to an expression e where it is not necessarily followed by an application of rule [INST]. This is the case if and only if [GEN] is the last rule applied in a derivation or e occurs in the context $C[] \equiv \mathbf{let}\ x = []\ \mathbf{in}\ e'$. Similarly, [INST] can be confined to application to expressions whose type scheme is polymorphic by another rule than [GEN]. This is the case only for variables. Thus we can "build" the application of [GEN] into rule [LET], and [INST] into rule [VAR]. The resulting typing rules are in Figure 5. The side condition $(*)$ in rule [GEN/LET] is: $x \notin \mathit{domain}\,(\Gamma)$ and $\vec{\alpha}$ not free in C, Γ.

By the considerations above we have not lost any typing judgements by restricting ourselves to the *normalized* typing rules of Figure 5:

Proposition 3.1 *$C, \Gamma \vdash_P e : \tau$ if and only if $C, \Gamma \vdash'_P e : \tau$.*

3.2 Basic properties

Proposition 3.2 *Let x not be free in e. Then:*

$$C, \Gamma \cup \{x : \sigma'\} \vdash_P e : \sigma \textit{ if and only if } C, \Gamma \vdash_P e : \sigma.$$

PROOF Standard. (Both implications by induction on derivation of $C, \Gamma \cup \{x : \sigma'\} \vdash_P e : \sigma$, respectively $C, \Gamma \vdash_P e : \sigma$. The only interesting case is rule [GEN] in the "if" direction.) □

Lemma 3.3 *(Formation Lemma)*

$$(\lambda x.e')e \longrightarrow e'[e/x]$$
$$\lambda x.ex \longrightarrow e \qquad \text{if } x \text{ not free in } e$$
$$\mathbf{let}\ x = e\ \mathbf{in}\ e' \longrightarrow e'[e/x]$$
$$e \longrightarrow e$$
$$\frac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'}$$
$$\frac{e \longrightarrow e' \qquad e'' \longrightarrow e'''}{ee'' \longrightarrow e'e'''}$$
$$\frac{e \longrightarrow e' \qquad e'' \longrightarrow e'''}{\mathbf{let}\ x = e\ \mathbf{in}\ e'' \longrightarrow \mathbf{let}\ x = e'\ \mathbf{in}\ e'''}$$

Figure 6: Definition of (parallel) $\beta\eta$**let**-reduction

1. *$C, \Gamma \vdash_P \lambda x.e : \tau''$ if and only if there exist τ, τ' such that $C, \Gamma \cup \{x : \tau\} \vdash_P e : \tau'$ and $C \vdash_P (\tau \to \tau') \leq \tau''$.*
2. *$C, \Gamma \vdash_P ee' : \tau$ if and only if there exists τ' such that $C, \Gamma \vdash_P e : \tau' \to \tau$ and $C, \Gamma \vdash_P e' : \tau'$.*

Lemma 3.4 *(Substitution lemma)*
If $C, \Gamma \cup \{x : \sigma\} \vdash_P e' : \sigma'$ and $C, \Gamma \vdash_P e : \sigma$ then $C, \Gamma \vdash_P [e/x]e'$.

3.3 Preservation of typing judgements under reduction

We define a Tait-Martin-Löf version of $\beta\eta$**let**-reduction. Define $\longrightarrow$ by the inference system in Figure 6.

Theorem 3.5 *Let P be such that $C \vdash_P (\tau \to \upsilon) \leq (\tau' \to \upsilon')$ only if $C \vdash_P \tau' \leq \tau$ and $C \vdash_P \upsilon \leq \upsilon'$. Then:*
If $C, \Gamma \vdash_P e : \sigma$ and $e \longrightarrow e'$ then $C, \Gamma \vdash_P e : \sigma$.

PROOF (Sketch) The proof is by induction on the definition of $e \longrightarrow e'$. It is easy to see that we can restrict ourselves to the case where σ is a simple type. Note that, by Proposition 3.1, in this case we can assume that all derivations are normalized. The only interesting cases are the axioms defining $\beta-$, $\eta-$ and **let**-reduction, respectively. The other four rules are straightforward.

Case $\boxed{(\lambda x.e')e \longrightarrow e'[e/x]}$: (Here we need the property that $C \vdash_P (\tau \to \upsilon) \leq (\tau' \to \upsilon')$ only if $C \vdash_P \tau' \leq \tau$ and $C \vdash_P \upsilon \leq \upsilon'$.)

Case $\boxed{\lambda x.ex \longrightarrow e}$: (We don't need the property.)

Case $\boxed{\mathbf{let}\ x = e\ \mathbf{in}\ e' \longrightarrow e'[e/x]}$: (We don't need the property.) □

A more careful formulation of the theorem says that η- and **let**-reduction preserve typing judgements without any demands on P whatsoever. Only the case of β-reduction requires that $C \vdash_P (\tau \to \upsilon) \leq (\tau' \to \upsilon')$ only if $C \vdash_P \tau' \leq \tau$ and $C \vdash_P \upsilon \leq \upsilon'$. The following example shows that this requirement is not artificial, but truly required. Let P have base types *bool* and *integer* with (*integer* $\to$ *integer*) $\leq_P$ (*bool* $\to$ *integer* being the only nontrivial inequality.

Assume that add is a constant of type *integer* $\rightarrow$ *integer* $\rightarrow$ *integer* and that *true*, *false* are constants of type *bool*. Then $(\lambda x.\text{add } x\ x)$*true* has type *integer*, but its β-reduct, add *true true*, does not.

3.4 Invariance under let-folding/unfolding

In the previous section we have seen that, under suitable conditions, typing judgements are preserved under $\beta\eta$**let**-reduction. To show this we made no use of the Principal Type Property. In this section we show that typing judgements are also preserved under **let**-*folding*, the *inverse* of **let**-reduction. This is a limited sort of "subject expansion" property of our typing system. For this the Principal Type Property is crucial.

Lemma 3.6 *(Decomposition Lemma) Let e' be an expression where, for $n \geq 0$, each of the variables $x_1, \ldots, x_n$ occurs freely exactly once. Then $C, \Gamma \vdash_P e'[e_1/x_1, \ldots, e_n/x_n] : \sigma'$ only if there exist $\sigma_1, \ldots, \sigma_n$ such that*

1. *$C, \Gamma \vdash_P e_i : \sigma_i$ for $1 \leq i \leq n$ and*
2. *$C, \Gamma \cup \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash_P e' : \sigma'$.*

Theorem 3.7 *(Closure under* **let***-folding) If $e_1 \longrightarrow e_2$ by* **let***-folding, $e'[e/x] \longrightarrow$* **let** $x = e$ **in** e'*, and $C, \Gamma \vdash_P e_1 : \sigma$ then $C, \Gamma \vdash_P e_2 : \sigma$.*

PROOF This follows from the Decomposition Lemma and the Principal Typing Property. □

Corollary 3.8 *Let $=$ be the equational theory generated by*

$$(\textbf{let}\, x = e\, \textbf{in}\, e') = e'[e/x].$$

If $e = e'$ then $C, \Gamma \vdash_P e : \sigma$ if and only if $C, \Gamma \vdash_P e' : \sigma$.

We can consider polymorphic subtyping *complete* under **let**-folding/unfolding. This is a salient and arguably a defining property of (predicative, "ML-style") polymorphism.

Since every expression has a (unique) **let**-free normal form under **let**-unfolding a consequence of this lemma is that we can "reduce" polymorphic typing to monomorphic typing: To type check an expression first unfold all **let**-expressions in it and then perform monomorphic type inference (with a slightly extended inference rule for constants). Thus results from monomorphic subtyping become immediately applicable (see introduction).

We have admitted type schemes that have no simple type instances. Our results carry over to a system where this is forbidden, for example in a system with added premise $C \vdash C'[\vec{\tau}/\vec{\alpha}]$ in rule [GEN] in Figure 2 [Smi91].

4 Conclusion and related work

4.1 Conclusion

We have studied fundamental syntactic properties of polymorphic subtyping with simple types and subtype-qualified type schemes. We have introduced the *halbstark* instance relation on typing judgements, which is a hybrid of Mitchell's weak instance relation and Fuh and Mishra's strong instance relation. We have exhibited a type inference algorithm, modeled on Algorithm W, which is syntactically sound and complete. In particular, its correctness implies the Principal Type Property: all expressions e have a principal type σ in a given type environment; that is, σ is a subtype of any other type σ' of e.

The significance of the halbstark instance relation emerges from the fact that our type inference algorithm is generic in the sense that it allows arbitrary transformations on typing judgements so long as these preserve *halbstark-equivalence*. Examples of such transformations are simplifications that primarily seek to reduce the size and number of coercions in typing judgements and type schemes, such as Fuh and Mishra's G-simplification and a somewhat restricted S-simplification [FM89] as well as the simplifications Smith describes in his thesis [Smi91].

Finally we have shown that typing judgements are closed under η-reduction, let-folding and unfolding, but not generally under β-reduction, unless the subtyping theory obeys the *decomposition property*: $C \vdash (\tau \rightarrow \tau') \leq (\upsilon \rightarrow \upsilon')$ only if $C \vdash \upsilon \leq \tau$ and $C \vdash \tau' \leq \upsilon'$. The invariance of typings under let-folding and unfolding, though arguably a hallmark of let-polymorphism, seems not to have been proved and published before.

4.2 Related work

Syntactic, semantic and algorithmic properties of monomorphic subtyping have been studied extensively in the literature. Less so what we consider its natural extension to (let-)polymorphism. We only survey those works with direct connection and relevance to this paper.

Mitchell introduced what we have called here the weak instance relation. He showed amongst other things that in atomic structural subtyping all typable expressions have principal typing judgements relative to the weak instance relation [Mit84, Mit91]. Fuh and Mishra introduced what we here call the strong instance relation and show how principal typings can be simplified by transforming them to strongly equivalent typings with fewer coercions and type variables [FM89]. Simplifying subtyping constraints is important both in the process of "generating" constraints [AW93, Pot96] and solving or presenting them. Note that, although structural subtype theory does not admit recursive types as solutions, this has only impact on the solvability of subtype constraints, not their generation and simplification.[7]

Computing principal typings reduces the problem of typability over a particular subtype theory to one of deciding satisfiability of subtyping constraints. There has been an array of results [WO89, LM92, Tiu92, Ben93, HM95, PT95] that characterize when solving subtyping constraints is feasible and when not. Subtyping has recently been studied intensively in a setting with recursive types and combinatorially simple primitive subtype theories: Palsberg et al. (e.g. [KPS92, KPS93, Pal94, PWO95]) have worked on efficient type inference algorithms. Pottier

Let-polymorphism with subtyping has been studied by Kaes [Kae92] and

[7]Though it should be noted that having logical type operators such as disjunction or conjunction available does make a difference.

Smith [Smi91, Smi93, Smi94].[8] Kaes presents a qualified type inference system that, in contrast to Jones's [Jon92, Jon94], allows incorporating subtyping steps. He presents a type inference algorithm that computes principal types and applies some simplifications analogous to Fuh and Mishra's application of matching substitutions [FM88, FM90]. His system includes a form of overloading and allows, with some limitations, also recursive types. Smith studies polymorphic subtyping and a very general form of overloading. Simplification of typing judgements is done in his type inference algorithm in combination with generalizing and then binding a type to a let-bound variable. He extracts some properties that are sufficient for the correctness of his type inference algorithm; those are more complex and less general than our correctness guarantee due to halbstark-equivalence. Neither Kaes nor Smith study subjection reduction properties or invariance under let-folding/unfolding.

Aiken, Wimmers and Lakshman [AW93, AWL94] and Eifrig, Smith and Trifonov [EST95] describe let-polymorphic systems with subtyping. Their type inference rule actually allow *only* canonical derivations, essentially those yielded by Algorithm W presented here, and *with no simplifications permitted.* Thus completeness of type inference is a cinch, at the price that, technically, simplifications on derived typing judgements have no meaning inside the type system. (It is best to understand the typing rules in these papers as specifications of a constraint generation algorithm.)

Acknowledgements

This article is written in an attempt to solidify understanding — not least my own — of the fundamental structural properties of parametric (ML-)polymorphism and its interaction with monomorphic subtyping. Jakob Rehof has laid the foundation for this work, both in intellectual and practical terms, since his work on monomorphic subtyping and the many spirited discussions we have had have been instrumental for my own understanding.

References

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Copenhagen, Denmark*, pages 31–41. ACM, ACM Press, June 1993.

[AWL94] Alexander Aiken, Edward 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, ACM Press, January 1994.

[Ben93] Marcin Benke. Efficient type reconstruction in the presence of inheritance. In A. M. Borzyszkowski and S. Sokolowski, editors, *Proc.*

[8]It is important to distinguish generalizing monomorphic subtyping to let-polymorphism from other notions of subtyping in a System F setting such as System $F_{<}$ [CMMS91] and System $F_{\sqsubseteq}$ [LMS95].

Mathematical Foundations of Computer Science (MFCS), pages 272–280. Springer-Verlag, 1993. Lecture Notes in Computer Science (LNCS), Vol. 711.

[BY79] Ch. Ben-Yelles. Type assignment in the lambda-calculus: Syntax and semantics. Technical report, Department of Pure Mathematics, University College of Swansea, September 1979. Author's current address: Univérsite des Sciènces et dé la Technologie Houari Boumediene, Institut D'Informatique, El-Alia B.P. No. 32, Alger, Algeria.

[CF58] H. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[CMMS91] L. Cardelli, S. Martini, J. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A. Meyer, editors, *Proc. Int'l Conf. on Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, pages 750–770. Springer, September 1991. Lecture Notes in Computer Science, Vol. 526.

[Dam84] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984. Technical Report CST-33-85 (1985).

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 11th Conf. on the Mathematical Foundations of Programming Semantics (MFPS)*, April 1995.

[FM88] 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.

[FM89] 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.

[FM90] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science (TCS)*, 73:155–175, 1990.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, December 1969.

[HM95] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proc. 22nd ACM Symp. on Principles of Programming Languages (POPL), San Francisco, California*, pages 176–185, January 1995.

[Jon92] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Brückner, editor, *Proc. 4th European Symposium on Programming (ESOP), Rennes, France*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, February 1992.

[Jon94] Mark Jones. A theory of qualified types. *Science of Computer Programming*, 22:231–256, 1994.

[Kae92] 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.

[KMM91] 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.

[KPS92] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient inference of partial types. Technical Report DAIMI PB-394, DAIMI, Aarhus University, April 1992.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. In *Proc. 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 419–428. ACM, ACM Press, January 1993.

[KTU90] A. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Coll. on Trees in Algebra and Programming (CAAP), Copenhagen, Denmark*, pages 206–220. Springer, May 1990. Lecture Notes in Computer Science, Vol. 431.

[LM92] 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.

[LMS95] Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. A logic of subtyping. In *Proc. IEEE Symp. on Logic in Computer Science (LICS), San Diego, California*. IEEE Computer Society, IEEE Computer Society Press, June 1995.

[Mai90] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. 17th ACM Symp. on Principles of Programming Languages (POPL)*. ACM, January 1990.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mit84] J. Mitchell. Coercion and type inference. In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, 1984.

[Mit91] J. Mitchell. Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–285, July 1991.

[Pal94] Jens Palsberg. Efficient type inference of object types. In *Proc. 9th Annual IEEE Symp. on Logic in Computer Science (LICS), Paris, France*, pages 186–195. IEEE Computer Society Press, July 1994.

[Pot96] François Pottier. Simplifying subtyping constraints. In *To appear at ICFP '96, Philadelphia, Pennsylvania*, May 1996.

[PT95] Vaughan Pratt and Jerzy Tiuryn. Satisfiability of inequalities in a poset. Technical Report 95-15(215), Institute of Informatics, Warsaw University, October 1995.

[PWO95] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. Technical Report RC-95-33, BRICS, Dept. of Computer Science, Aarhus University, April 1995.

[Smi91] Geoffrey Seward Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, August 1991.

[Smi93] 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.

[Smi94] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[Tiu92] 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.

[WO89] 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.