Subtyping Constrained Types

Valery Trifonov* ** and Scott Smith*

Department of Computer Science, Johns Hopkins University
Baltimore, MD 21218, USA
http://www.cs.jhu.edu/hog/

Abstract. A *constrained type* is a type that comes with a set of subtyping constraints on variables occurring in the type. Constrained type inference systems are a natural generalization of Hindley/Milner type inference to languages with subtyping. This paper develops several subtyping relations on polymorphic constrained types of a general form that allows recursive constraints and multiple bounds on type variables. Subtyping constrained types has applications to signature matching and to constrained type simplification.

1 Introduction

A constrained type intuitively is a simple type together with a set of subtyping constraints on its type variables. An example is $t \to \mathbf{int} \setminus \{t \leq \mathbf{int}\}$, a type of functions whose argument type t is constrained to be a subtype of $\mathbf{int}$. It is possible to perform let-polymorphic type inference for constrained types, producing polymorphic types ("type schemes") of the form $\forall t_1, \ldots, t_n. \tau \setminus C$, which generalize the type schemes produced by the Hindley/Milner unification algorithm; constrained type inference is strictly more general than unification-based type inference. The idea of including subtyping constraints as part of typing judgements was first developed by Mitchell [17, 18]. His constraint sets were restricted to be atomic, allowing coercions between type variables only; numerous other systems with restricted forms of constraint inference have been introduced since, including [16, 12, 14, 4].

A type inference algorithm for polymorphic constrained types of the form studied here was first discovered by Curtis [8], and later independently discovered in somewhat different form, and first proven sound, by Aiken and Wimmers [1]. These constraint systems are less restrictive than the previously cited formulations: they allow recursive constraints such as $t \leq t \to \mathbf{int}$, and thus subsume recursive types. Additionally, both upper- and lower-bound constraints on variables are legal, and multiple bounds may be placed on a single variable (multiple bounds such as $\{t \leq \tau_1, t \leq \tau_2\}$ are expressed equivalently as $t \leq \tau_1 \cap \tau_2$, where $\cap$ is the type intersection operator of [1]). This extra flexibility allowed in the constraint sets gives a more powerful, but computationally more complex, inference algorithm.

Constrained types are particularly appropriate for object-oriented programming languages [10, 9]: these types incorporate subtyping and recursive dependence, both critical

* Partially supported by NSF grant CCR-9301340
** Partially supported by AFOSR grant F49620-93-1-0169

in an object-oriented setting, and their greater flexibility gives a reasonable solution to the binary methods problem [5].

The objective of this paper is to address the problem of subtyping between polymorphic constrained types:

$$\forall t_1, \ldots, t_n.\tau \setminus C \ \leq^{\forall} \ \forall t_1, \ldots, t_n.\tau' \setminus C'$$

Considering the generality of these types, the relation $\leq^{\forall}$ should be expected to subsume both the "more general than" relation between type schemes in the Hindley/Milner system, and the subtyping relation on recursive types of Amadio/Cardelli [3].

This relation, which has not received the deserved attention in the literature, has at least two important applications. The first is separate compilation via modules and functors, and the subsequent need for signature matching. To a first approximation, in a module system based on Hindley/Milner style of type inference the program specifies a polymorphic type (signature) S for the parameter of a functor F to allow uses of the parameter at different types (which is not possible for parameters of functions); the polymorphic signature S' inferred for the actual argument M is then matched against S at the point of application $F(M)$. In a constrained type setting S and S' are polymorphic constrained types; "matching the signature" thus requires verifying that the $S' \leq^{\forall} S$. Another application is the justification of simplification operations on constrained polymorphic types. In this case a proof is required that the simplified type is equivalent to the original (*i.e.* both a subtype and a supertype).

In this paper we define several variants of $\leq^{\forall}$. The "optimal" form $\leq^{\forall}_{obs}$ may be characterized observationally by analogy with Morris/Plotkin contextual expression equivalence, replacing expression contexts with proof contexts. We then define a semantic form $\leq^{\forall}_{sem}$ based on a regular tree interpretation, and prove it is a good model by showing it is exactly $\leq^{\forall}_{obs}$, a *full type abstraction* property. The ideal model [15] may also be used as a basis of a constrained type ordering $\leq^{\forall}_{ideal}$ [2], but it is not fully abstract. The relation $\leq^{\forall}_{sem}$ is surprisingly complex: we leave open the question whether it is decidable, and develop a powerful decidable approximation $\leq^{\forall}_{dec}$.

In the process of defining these subtyping relations, other results of independent interest are derived. First, an entailment relation $C \ \vdash \ \tau \leq \tau'$ over simple types τ is axiomatized. The set C is a system of arbitrary type constraints, thus generalizing the system of [3] which only allows one upper, non-recursive bound of each variable in C. We define two reduced forms of constraint sets, *constraint maps* (*kernels*) and *canonical* maps, which are also of use as more compact representations of constraint sets in algorithms such as type inference. Next, the subtyping relations on constrained types are defined, and their relationship is established. Finally, soundness of a system of typing rules with constrained types is proved by a simple method of reduction to soundness of a system without constrained types. A principal typing property is established for our type inference algorithm.

In this paper we work over a simple language with only function, top, and bottom types to reduce clutter. However, previous work [10, 9] shows how state, records, variants, classes, and objects all may be incorporated in a constrained type framework, and we explicitly avoid semantic tools (such as the ideal model of types [15]) which lack a strong potential to generalize to such constructs.

2 Simple and Constrained Types

We illustrate our ideas by studying an extension of the call-by-name λ-calculus with constants and let-binding. The abstract syntax of the expressions in the language is

$$e \ ::= \ x \mid \lambda x.\, e \mid e\, e' \mid X \mid \mathsf{let}\ X = e\ \mathsf{in}\ e'$$

To simplify the presentation we assume that the λ-bound and let-bound variables are in different syntactic classes, that λ-bound variables are not re-bound, and that constants are a special case of let-variables, bound in the initial environment. We write $\lambda\_.\, e$ for $\lambda x.\, e$ where x is not free in e, and $(e;\, e')$ for $(\lambda\_.\, e')\, e$.

The *simple types* are

$$\mathit{Typ} \ni \tau \ ::= \ t \mid \bot \mid \top \mid \tau \to \tau' \mid \ \ldots$$

where t ranges over the set *TyVar* of type variables, $\bot$ and $\top$ are "minimal" and "maximal" types. In addition to the function types there may be a set of basic types which we leave unspecified. We call $\bot$, $\top$, $\to$, etc. *type constructors*, and all simple types in $\mathit{Typ} - \mathit{TyVar}$ *constructed*.

A constrained type κ has the form $\forall \bar{t}.\, A \Rightarrow \tau \setminus C$, where the *context* A is a finite map from variables to simple types, written $\langle \overline{x_i : \tau_i} \rangle$, the *root type* τ is a simple type, and the *constraint set* C is a set of *subtyping constraints* on simple types, each of the form $\tau' \leq \tau''$.[2] We use this new notion for constrained types in order to more appropriately present the binding structure of type variables. The context A represents the assumptions about the types of λ-bound variables free in the term, and C is the set of subtyping constraints (a.k.a. coercions) under which the term is typable; they are both part of the type itself instead of the environment. Thus all constrained types in a type sequent are closed, so we can compare constrained types with different sets of type variables, and avoid giving meaning to constrained types with constrained but free type variables.

Definition 1. A constraint set C is *closed* if it is closed under transitivity (*i.e.* $\{\tau \leq \tau',\, \tau' \leq \tau''\} \subseteq C$ entails $\tau \leq \tau'' \in C$) and decomposition ($\tau_1 \to \tau_1' \leq \tau_2 \to \tau_2' \in C$ entails $\{\tau_2 \leq \tau_1,\, \tau_1' \leq \tau_2'\} \subseteq C$).

We denote by $\mathit{Cl}(C)$ the least closed superset of C. Thus, for example, if $C = \{t \to t \leq (\bot \to t) \to \top \to \top\}$, then $\mathit{Cl}(C) = C \cup \{\bot \to t \leq t,\, t \leq \top \to \top,\, \bot \to t \leq \top \to \top,\, \top \leq \bot,\, t \leq \top,\, \bot \to t \leq \top\}$.

A constraint set is *consistent* if for each constraint $\tau \leq \tau'$ in it at least one of the following is true: $\tau = \bot$, $\tau' = \top$, both sides have the same outermost type constructor, or one of them is a type variable.

[2] We only consider closed constrained types, for which $\{\bar{t}\} \supseteq \mathit{FTV}(A) \cup \mathit{FTV}(\tau) \cup \mathit{FTV}(C)$, where $\mathit{FTV}(\tau)$ as usual denotes the set of type variables free in τ. Constrained types are considered identical under α-renaming of bound type variables.

3 Primitive Subtyping

In order to define notions of $\leq^{\forall}$, a theory of primitive subtyping under a set of subtyping constraints, $C \models \tau \leq \tau'$, and its decidable axiomatization are developed. Due to space limitations most of the proofs have been elided; currently they can be found on the World Wide Web at URL `http://www.cs.jhu.edu/hog/subcon.ps.gz`.

3.1 Regular Tree Semantics of Constraints

Sequents $C \models \tau \leq \tau'$ may be defined as valid if they hold for all instantiations of type variables in C, τ, and τ'. There are many possibilities for the notion of "instance." The simplest is to allow instances to range over the variable-free types constructed from $\top$, $\bot$, and $\rightarrow$. However, for our purposes this does not give enough points in the space of instances, *e.g.* when typing binary methods [5] we have to work with recursive constraint sets such as $\{t \rightarrow t \leq t,\ t \leq t \rightarrow \top\}$, which have no solutions in this space. This is an example where differences arise when recursive constraint sets are allowed—if recursive constraint sets were not allowed, the simple type basis would have been appropriate. Another candidate is the ideal model [15] used in [2], but conversely it has too many points, allowing polymorphic types such as $\forall t.t \rightarrow t$ to be substituted for type variables; since our system is "shallow" these points are superfluous in our framework. It turns out that the addition of the solutions of recursive type equations to the ground types gives just enough points to define an appropriate semantics. In the next section a theorem will be proven which rigorously demonstrates this fact. We use the convenient notion of regular trees [7, 3] to model solutions of recursive type equations.

We present the semantics of constraint sets in terms of regular trees over a ranked alphabet. Let us review some definitions and results from [7, 3]. Given a ranked alphabet L, a *tree* φ is a partial function from finite sequences of natural numbers $\mathbb{N}^*$ (*paths*) to L such that $\mathit{Dom}(\varphi)$ is prefix-closed and for each $\pi \in \mathbb{N}^*$ we have $\{k \mid \pi k \in \mathit{Dom}(\varphi)\} = \{0, \ldots \mathit{rank}_L(\varphi(\pi)) - 1\}$. The *subtree at* $\pi \in \mathit{Dom}(\varphi)$ is the function $\lambda\pi'.\varphi(\pi\pi')$; $|\pi|$ is the *level* of that subtree. A tree is *regular* if the set of its subtrees is finite.

Define $\mathbb{T}$ as the set of regular trees over the ranked alphabet L_τ of type constructors in our language, the nullary $\bot$ and $\top$ and the binary $\rightarrow$; we reuse the syntax of types as a notation for trees. A *regular system* in this context is a set of equations of the form $\overline{t_i = \tau_i}$ between type variables $t_i \in \mathit{TyVar}$ and simple types τ_i (*i.e.* finite trees over $L_\tau \cup \mathit{TyVar}$ where the type variables are nullary); a regular system is *contractive* if it has no subset of the form $\{t_0 = t_1, \ldots, t_{n-1} = t_n, t_n = t_0\}$. An *assignment* ρ on $V \subset \mathit{TyVar}$ is a total map in $V \rightarrow \mathbb{T}$; it is homomorphically extended on simple types τ with $\mathit{FTV}(\tau) \subseteq V$: $\rho(\bot) = \bot$, $\rho(\top) = \top$, and $\rho(\tau \rightarrow \tau') = \rho(\tau) \rightarrow \rho(\tau')$. An assignment ρ on $\overline{t_i}$ is a *solution* of the regular system $\overline{t_i = \tau_i}$ if $\overline{\rho t_i = \rho\tau_i}$.

Proposition 2. *Each contractive regular system has a unique solution, and each regular tree is the image of some variable in a solution of a contractive regular system.*

A *level-k cut* $\varphi|_k$ of $\varphi \in \mathbb{T}$ for $k \in \mathbb{N}$ is defined as the (finite) tree obtained by replacing all subtrees at level k of φ (if any) by $\top$.

A partial order over L, in which $\bot$ is the minimal element and $\top$ is the maximal, together with variance specifications for the arguments of non-nullary constructors (in this case, contravariance in the domain and covariance in the range of $\rightarrow$) induce a partial order $\leq_{tree}$ over $\mathbb{T}$ as follows: $\bot \leq_{tree} \varphi$ and $\varphi \leq_{tree} \top$ for each finite tree φ, and $\varphi_1 \rightarrow \varphi_1' \leq_{tree} \varphi_2 \rightarrow \varphi_2'$ if $\varphi_2 \leq_{tree} \varphi_1$ and $\varphi_1' \leq_{tree} \varphi_2'$; finally, $\varphi \leq_{tree} \varphi'$ if $\varphi|_k \leq_{tree} \varphi'|_k$ for each $k \in \mathbb{N}$.

Returning to our type system, within a set of constraints we model simple types by regular trees satisfying these constraints:

Definition 3. (i) An assignment ρ *k-satisfies* a constraint $\tau \leq \tau'$, written $\rho \rhd_k \tau \leq \tau'$, if $\rho(\tau)|_k \leq_{tree} \rho(\tau')|_k$.
(ii) ρ *satisfies* $\tau \leq \tau'$ ($\rho \rhd \tau \leq \tau'$, also ρ is a *solution* of $\tau \leq \tau'$) if $\rho \rhd_k \tau \leq \tau'$ for each $k \in \mathbb{N}$.
(iii) The above properties are extended over a set of constraints C if they hold for each constraint in the set.

Regular trees may now be used to define the theory $C \models \tau \leq \tau'$.

Definition 4. (i) $C \models \tau \leq \tau'$ if for every assignment ρ on *FTV*$(C \cup \{\tau \leq \tau'\})$, if $\rho \rhd C$, then $\rho \rhd \tau \leq \tau'$.
(ii) $C \models C'$ if *FTV*$(C') \subseteq$ *FTV*(C) and for each solution ρ of C there is a solution ρ' of C' which agrees with ρ on *FTV*(C').
(iii) C and C' are *equivalent* if $C \models C'$ and $C' \models C$.

Lemma 5. *C and Cl(C) are equivalent. Thus, if C is satisfiable, then Cl(C) is consistent.*

We leave open the decidability of $C \models \tau \leq \tau'$, and in the sequel we develop decidable approximations to it.

3.2 Constraint Map Representation

We now define *constraint maps* as an equivalent form for representing consistent closed constraint sets. Closed constraint sets may contain significant redundant information. To a constraint set $\{\tau_1 \rightarrow \tau_1' \leq \tau_2 \rightarrow \tau_2'\}$ the closure adds $\{\tau_2 \leq \tau_1, \tau_1' \leq \tau_2'\}$, and the original constraint between functions is completely captured by the additions and can be removed. Constraint maps in fact do not allow constraints between two constructed types, since they can always be expressed by an equivalent family of constraints, provided the constraint set was consistent to begin with. We reuse some of the notation previously defined for constraint sets C on constraint maps K.

Definition 6. A *constraint map* is a finite map $K \in$ *TyVar* $\rightarrow (2^{Typ})^2$, assigning sets of upper and lower bounds to each type variable in its domain; we use the more intuitive notation $t \leq \tau \in K$ and $t \geq \tau' \in K$ for $\tau \in \pi_1(K(t))$ and $\tau' \in \pi_2(K(t))$, respectively, and $(K,\, t \leq \tau)$, $(K,\, t \geq \tau)$ for the maps extending the sets $\pi_1(K(t))$, respectively $\pi_2(K(t))$, to contain τ. We write $t \geq \tau$ instead of $\tau \leq t$ since K is not required to be antisymmetric, *i.e.* $t' \in \pi_1(K(t))$ does not imply $t \in \pi_2(K(t'))$; we use $t = \tau \in K$ for the pair $t \leq \tau \in K$, $t \geq \tau \in K$.

$$(\bot)\quad K \vdash \bot \le \tau \qquad\qquad (\top)\quad K \vdash \tau \le \top$$

$$(=)\quad K \vdash t \le t, \quad t \in \mathit{Dom}(K) \qquad\qquad (\to)\quad \frac{K \vdash \tau_1' \le \tau_1 \quad K \vdash \tau_2 \le \tau_2'}{K \vdash \tau_1 \to \tau_2 \le \tau_1' \to \tau_2'}$$

$$(\uparrow)\quad \frac{(K, t \le \tau, t \le \tau') \vdash \tau \le \tau'}{(K, t \le \tau) \vdash t \le \tau'} \text{ if } (K, t \le \tau, t \le \tau') \text{ is contractive}$$

$$(\downarrow)\quad \frac{(K, t \ge \tau, t \ge \tau') \vdash \tau' \le \tau}{(K, t \ge \tau) \vdash \tau' \le t} \text{ if } (K, t \ge \tau, t \ge \tau') \text{ is contractive}$$

Fig. 1. Rules for primitive subtyping

Definition 7. The *kernel* $\mathit{Ker}(C)$ of a constraint set C is the constraint map defined by the set of constraints $\{\tau \le \tau' \in \mathit{Cl}(C) \mid \{\tau, \tau'\} \cap \mathit{TyVar} \neq \emptyset\}$; a constraint of the form $t \le t'$ sets the appropriate bounds on both variables.

For example, since the closure of the constraint set $C = \{(\top \to t) \to t \le t \to t \to \bot\}$ is $\mathit{Cl}(C) = \{(\top \to t) \to t \le t \to t \to \bot,\ t \le \top \to t,\ t \le t \to \bot\}$, the kernel of C is $\mathit{Ker}(C) = (t \le \top \to t,\ t \le t \to \bot)$.

Proposition 8. *For consistent constraint set C, $\mathit{Ker}(C)$ and C are equivalent.*

The kernel form of a constraint set has significant advantages from an implementation perspective: a type inference algorithm may maintain constraint sets in their equivalent kernel form, which is considerably more compact than the closure.

3.3 Rules for Primitive Subtyping

We take advantage of the equivalent constraint map representation K of a constraint set C, and with the rules in Fig. 1 define a decidable sound approximation $K \vdash \tau \le \tau'$ of the theory $C \models \tau \le \tau'$.

An implicit requirement for $K \vdash \tau \le \tau'$ is $\mathit{FTV}(\tau) \cup \mathit{FTV}(\tau') \subseteq \mathit{Dom}(K)$. As is usually the case in the presence of recursive types, a notion of contractiveness plays an important role in detecting ill-defined constraint maps.

Definition 9. A constraint map K is *contractive* if $\pi_1 \circ K$ and $\pi_2 \circ K$ as relations on *TyVar* have no cycles, *i.e.* if there do not exist variables $\{t_1, \ldots, t_n\} \subseteq \mathit{Dom}(K)$ such that $t_n = t_1$ and $t_i \le t_{i+1} \in K$ (respectively $t_i \ge t_{i+1} \in K$) for each $i \in \{1, \ldots, n-1\}$.

For instance, neither $(t \le t)$ nor $(t \ge t',\ t' \ge t)$ is contractive, while $(t \ge t',\ t \le t')$ is (recall a constraint map is not necessarily symmetric on type variables). However note that contractiveness of constraint maps, as opposed to regular systems, does not entail satisfiability, *e.g.* $(t \le \bot,\ t \ge \top)$ is a contractive map with no solutions. Contractiveness is required in order to ensure soundness by disallowing proofs in which *e.g.* a constraint introduced to K by one rule $(\uparrow)$ is used in another $(\uparrow)$ with no intervening uses of $(\to)$.

Rules $(\bot)$, $(\top)$, $(\to)$, and $(=)$ for reflexivity of the relation on type variables are standard. The novel rules $(\uparrow)$ and $(\downarrow)$ provide the only access to constraints in K; in fact, were the constraint map in its premise identical to the one in its conclusion, rule $(\uparrow)$ would have been just the standard rule for proving an upper bound on a type variable

in a system of rules with eliminated transitivity. With the extra assumption these are induction rules, similar to the (FIX) rule of [11].

Some standard subtyping rules [3] are derivable from those given in Fig. 1 and thus omited, for instance general reflexivity ($K \vdash \tau \leq \tau$ is always provable for *FTV*$(\tau) \subseteq$ *Dom*(K), and $(K,\, t \leq \tau) \vdash t \leq \tau$ is provable by $(\uparrow)$ and general reflexivity. In contrast, transitivity only holds for consistent constraint maps.

Definition 10. A constraint map K is *consistent* if for each t, τ, and τ', if $t \geq \tau \in K$ and $t \leq \tau' \in K$, then $K \vdash \tau \leq \tau'$.

The following lemma shows that a kernel contains all of the information of a consistent set C, and that computing the closure of C is equivalent to the construction of a constraint map K such that $K \vdash C$.

Lemma 11. *If Cl(C) is a consistent constraint set, then Ker(C) is a consistent constraint map, and Ker(C) $\vdash$ Cl(C).*

3.4 Soundness and Decidability of Primitive Subtyping

Next we establish soundness of the proof system $\vdash$ with respect to the relation $\models$; the main idea is to show that all assignments which approximate solutions of a constraint map K also approximate solutions of all subtyping constraints provable from K.

Lemma 12. *If K is contractive, $K \vdash \tau \leq \tau'$ has a proof, and the assignment ρ k-satisfies K, then*

(i) if the proof of $K \vdash \tau \leq \tau'$ has an instance of a rule other than $(\uparrow)$ or $(\downarrow)$ at its root, then $\rho \rhd_{k+1} \tau \leq \tau'$;
(ii) if there is an instance of $(\uparrow)$ or $(\downarrow)$ at the root of the proof of $K \vdash \tau \leq \tau'$, then $\rho \rhd_k \tau \leq \tau'$.

Theorem 13 (Soundness). *If K is contractive and $K \vdash \tau \leq \tau'$, then $K \models \tau \leq \tau'$.*

The system $\vdash$ is incomplete with respect to the relation $\models$; it is not even possible to prove that $(t \leq t \to \bot,\, t \leq \top \to t) \vdash t \leq \top \to \bot$ since the bound we need is stronger than each of the two given. However $\vdash$ is useful because of the following property.

Lemma 14. *The relation $K \vdash \tau \leq \tau'$ is decidable.*

3.5 Canonical Constraint Maps

We can obtain a stronger proof system if we place the constraints in an equivalent *canonical* form that has pre-computed least upper and greatest lower bounds. In a canonical constraint map each variable has exactly one constructed upper and one constructed lower bound. For instance, a canonical equivalent of $(t \leq t \to \bot,\, t \leq \top \to t)$ is $K = (t \geq \bot,\, t \leq \top \to \bot)$. The upper bounds $t \to \bot$ and $\top \to t$ on t have the lub $\top \to \bot$ computed for them. For this set we can indeed prove $K \vdash t \leq \top \to \bot$. The canonicalization process also has potential as an implementation technique.

Definition 15. A constraint map K is *canonical* if

- K assigns exactly one upper and one lower constructed bound (*canonical* bounds) to each type variable in its domain (with no restriction on the number of variable bounds);
- if $t \leq t' \in K$ and $t' \leq t'' \in K$, then $t \leq t'' \in K$, and
- for each $(t \leq t', t \leq \tau, t' \leq \tau') \subseteq K$, where $\{\tau, \tau'\} \cap \mathit{TyVar} = \emptyset$, we have $K \vdash \tau \leq \tau'$, and similarly for the lower bounds.

Clearly if K and K' are equivalent on $\mathit{FTV}(\tau) \cup \mathit{FTV}(\tau')$ then $K \models \tau \leq \tau'$ if and only if $K' \models \tau \leq \tau'$. This allows us to upgrade our system by converting each map K to an equivalent canonical map $\mathit{Can}(K)$ and then proving $\mathit{Can}(K) \vdash \tau \leq \tau'$ instead of the original $K \vdash \tau \leq \tau'$. Here we provide an algorithm for computing an equivalent canonical form $\mathit{Can}(K)$ of a map K.

Algorithm 16 *$\mathit{Can}(K)$ is computed as follows.*

Start with $K' = K$, and for some $t \in \mathit{FTV}(K')$ let V be the least set satisfying $V = \{t\} \cup \{t' \mid \exists t'' \in V. t'' \leq t' \in K'\}$, i.e. *the set of upper bounds on t in the reflexive transitive closure of K' on TyVar; the case of lower bounds is similar. Let $B = \{\tau \in \mathit{Typ} - \mathit{TyVar} \mid \exists t' \in V. t' \leq \tau \in K'\}$, the set of constructed upper bounds on t. We compute the canonical upper bound τ of t as the greatest lower bound of the elements of B, as follows.*

> *If $B \subseteq \{\top\}$, then $\tau = \top$; if $\bot \in B$, then $\tau = \bot$. Otherwise let $\{\overline{\tau_i \to \tau_i'}\}$ be the set of all function types in B, and let $\tau = t^{\wedge}_{T} \to t^{\vee}_{T'}$, where $T = \{\overline{\tau_i}\}$, $T' = \{\overline{\tau_i'}\}$, and $t^{\wedge}_{T}$ and $t^{\vee}_{T'}$ are in general auxiliary type variables we associate with the respective sets of type terms; in the cases when T is a singleton set $\{t'\}$ we let $t^{\wedge}_{T} = t^{\vee}_{T} = t'$ to ensure termination. Add the bounds $(\overline{t^{\wedge}_{T} \geq \tau_j})$ (and similarly for $t^{\vee}_{T'}$) to K'.*

Replace the constructed upper bounds of t by $(t \leq \tau)$; thus the new bounds on t, namely $(t \leq \tau)$ and $(t \leq t')$ for each $t' \in V$, are in canonical form. Continue until all variables in K' are processed. (Adding also all bounds of the form $(t^{\wedge}_{S} \leq t^{\wedge}_{T})$ and $(t^{\vee}_{T'} \geq t^{\vee}_{T})$ if $T \subset S$, when those auxiliary variables appear in the domain of the map, produces an even stronger, with respect to $\vdash$, yet equivalent form.) The resulting K' is the value of $\mathit{Can}(K)$.

The following lemma proves the correctness of this algorithm.

Lemma 17. *For each constraint map K there exists a canonical equivalent $\mathit{Can}(K)$.*

For example computing the canonical equivalent of $K = (t \leq t \to \bot,\ t \leq \top \to t)$ starts by introducing $t_1 = t^{\vee}_{\{t,\, \top\}}$ and $t_2 = t^{\wedge}_{\{\bot,\, t\}}$, and transforming K into $K' = (t \geq \bot,\ t \leq t_1 \to t_2,\ t_1 \geq \top,\ t_1 \leq \top,\ t_1 \geq t,\ t_2 \geq \bot,\ t_2 \leq \bot,\ t_2 \leq t)$. This map is already in canonical form, and it is possible to prove $K' \vdash t \leq \top \to \bot$. However, even when $\vdash$ is used on to canonical equivalents it still does not provide proofs for some valid relations; for instance, $(t \geq \bot,\ t \leq t \to \top) \models t \leq (\top \to \bot) \to \top$.

In the general case, the algorithm implied by Lemma 14 may attempt comparing τ against each upper bound on t currently in K in the process of searching for a proof of $K \vdash t \leq \tau$. In this process, it may have to backtrack if it fails to find a proof using a particular bound. However, in the case of canonical maps a more efficient implementation is possible which has time complexity of $O(n^2)$, where n is the size of $K \cup \{\tau \leq \tau'\}$. This algorithm only compares new bounds on a variable against its canonical bounds (which can be shown to suffice) and uses a form of memoisation to detect looping; details are omited for lack of space.

A parallel can be drawn between our system of subtyping rules and the system $\vdash_{AC}$ of Amadio and Cardelli [3], which is based on a relation of equivalence between recursive types, and on the inductive rule

$$(\mu) \quad \frac{C,\, t \leq t' \vdash_{AC} \tau \leq \tau'}{C \vdash_{AC} \mu t.\, \tau \leq \mu t'.\, \tau'}$$

Since recursive types can be encoded as type variables with identical upper and lower bounds, the corresponding rule for simple types with constraints is

$$\frac{K,\, t = \tau,\, t' = \tau',\, t \leq t' \vdash \tau \leq \tau'}{K,\, t = \tau,\, t' = \tau' \vdash t \leq t'}$$

which is indeed derivable in $\vdash$ in a stronger version by successive applications of $(\uparrow)$ and $(\downarrow)$; furthermore, the steps of the proof of $K,\, t = \tau,\, t' = \tau' \vdash t \leq t'$ follow closely the steps of the algorithm for computing $C \vdash_{AC} \mu t.\, \tau \leq \mu t'.\, \tau'$ presented in [3], which also effectively constructs the type contexts necessary in order to establish type equivalences. Amadio and Cardelli show that their system is complete with respect to the regular tree model of recursive types under certain conditions on C, τ, and τ'; in particular the constraints in C may not be recursive, and no variable may occur in both τ and τ'. An attempt to directly apply the system to prove sequents violating these conditions shows that it is incomplete in the more general setting considered in this paper, *e.g.* $t \leq \top \to t \nvdash_{AC} t \leq \mu t'.\, \top \to t'$. Our system, while still incomplete with respect to the model we present, is capable of proving the corresponding forms of all sequents provable in [3], in addition allowing multiple recursive upper and lower bounds on type variables, *e.g.* $t \leq \top \to t,\, t' = \top \to t' \vdash t \leq t'$.

3.6 Satisfiability of Canonical Constraint Maps

A constrained type only has meaning if its constraints describe a non-empty set of instances, and hence the satisfiability of a constraint map is an important property. In this section we provide a connection between consistency and satisfiability of canonical constraint maps. This connection also plays a role in establishing the relationship between various notions of subtyping on constrained types in Sect. 4.

Definition 18. The canonical map K' is a *submap* of a canonical map K if $K' \subseteq K$. Note that constraints on variables in $\mathit{Dom}(K) - \mathit{Dom}(K')$ may involve variables in $\mathit{Dom}(K')$, but $\mathit{FTV}(\mathit{Codom}(K')) \subseteq \mathit{Dom}(K')$.

Lemma 19. *If K' is a submap of K, and K is consistent and canonical, then every solution of K' can be extended to a solution of K. Thus, considering the special case of $K' = \emptyset$, every consistent canonical constraint map K is satisfiable.*

Combining these results with canonicalization and soundness of $\vdash$ with respect to $\models$, we can reason about canonical maps instead of their equivalent constraint sets.

Definition 20. $C \vdash \tau \leq \tau'$ if $\mathit{Can}(\mathit{Ker}(C)) \vdash \tau \leq \tau'$.

4 Subtyping Constrained Types

In this section we define three concrete $\leq^\forall$ relations of subtyping on constrained types: $\leq^\forall_{obs}$, $\leq^\forall_{sem}$, and $\leq^\forall_{dec}$, as promised in the introduction.

4.1 Operational Subtyping

For an initial definition of $\leq^\forall$ we rely on operational notions as a basis. The basic idea is simple, but we could not find any precedent for it in the literature. Expressions of constrained type $\forall\overline{t}.\, A \Rightarrow \tau \setminus C$ are also of type $\forall\overline{t'}.\, A' \Rightarrow \tau' \setminus C'$ if for all possible uses of expressions of the latter type that are consistent, use of the former type is also consistent. Relation $\leq^\forall_{obs}$ is defined by this means. The difficult issue is how a "use" of a type should be defined. Informally, each use is a typing proof context, in analogy with Morris/Plotkin expression contexts. We give a particular version of typing proof context which is one of many reasonable and equivalent notions: a "use" is a set of constraints of the form that could be added by the inference rules. The constraints added by the inference rules may only introduce upper bounds on the root types, and dually only lower bounds on the types in the context. As a consequence one may obtain a valid typing derivation after replacing a subterm by another term whose constrained type yields a consistent system when those bounds are added. This leads us to the following observational definition of a subtyping relation on constrained types. (We let $A \leq A'$ abbreviate the set of constraints $\{A(x) \leq A'(x) \mid x \in \mathit{Dom}(A')\}$, defined only when $\mathit{Dom}(A) \supseteq \mathit{Dom}(A')$.[3])

Definition 21. For closed constrained types, $\forall\overline{t'}.\, A' \Rightarrow \tau' \setminus C' \leq^\forall_{obs} \forall\overline{t''}.\, A'' \Rightarrow \tau'' \setminus C''$ if for each $\forall\overline{t}.\, A \Rightarrow \tau \setminus C$ such that $\{\overline{t}\}$ is disjoint from $\{\overline{t'}\}$ and $\{\overline{t''}\}$, if $\mathit{Cl}(C \cup C'' \cup (A \leq A'') \cup \{\tau'' \leq \tau\})$ is consistent, then $\mathit{Cl}(C \cup C' \cup (A \leq A') \cup \{\tau' \leq \tau\})$ is consistent.

[3] This "subtyping rule" for contexts is similar to standard record subtyping [6]; the closure conversion $[\![x]\!] = \lambda E.\, E.x$, $[\![\lambda x.\, e]\!] = \lambda E.\, \lambda x.\, [\![e]\!]\,\{\,\overline{x_i = E.x_i}\,|_{x_i \in \mathit{FV}(e)-\{x\}},\ x = x\,\}$, and $[\![e\, e']\!] = \lambda E.\, [\![e]\!]\, E\, ([\![e']\!]\, E)$ makes the environment explicit and maps terms of type $\forall\overline{t}.\, \langle\overline{x_i : \tau_i}\rangle \Rightarrow \tau \setminus C$ to closed terms of type $\forall\overline{t}.\, \langle\rangle \Rightarrow \{\,\overline{x_i : \tau_i}\,\} \rightarrow \tau \setminus C$.

4.2 Semantic Subtyping

Next, a semantic notion $\leq^{\forall}_{sem}$ is defined, via the regular tree model: two polymorphic constrained types are ordered if their sets of regular tree instances are ordered.

The context component A of a constrained type corresponds to a finite map Φ from variables to regular trees; the relation $\leq_{tree}$ can be extended on such maps as follows: $\Phi \leq_{tree} \Phi'$ if $\mathit{Dom}(\Phi) \supseteq \mathit{Dom}(\Phi')$ and $\Phi(x) \leq_{tree} \Phi'(x)$ for each $x \in \mathit{Dom}(\Phi')$. An *instance* of the constrained type $\kappa = \forall \overline{t}.\, A \Rightarrow \tau \setminus C$ is a pair written $\Phi \Rightarrow \varphi$ where $\Phi = \rho \circ A$ and $\varphi = \rho\tau$ for some assignment ρ on $\{\overline{t}\}$ that satisfies C. The set of instances of κ is $\mathit{Inst}(\kappa)$. As in the definition of $\leq^{\forall}_{obs}$, the natural order on instances is $\Phi \Rightarrow \varphi \leq_{tree} \Phi' \Rightarrow \varphi'$ if $\Phi' \leq_{tree} \Phi$ and $\varphi \leq_{tree} \varphi'$. We can now define a semantical notion of subtyping on constrained types.

Definition 22. $\kappa' \leq^{\forall}_{sem} \kappa''$ if for each instance of κ'' there is a smaller instance of κ'.

We may now prove the equivalence of $\leq^{\forall}_{sem}$ and $\leq^{\forall}_{obs}$, demonstrating the appropriateness of the regular tree interpretation.

Theorem 23 (Full Type Abstraction). *The relations $\leq^{\forall}_{sem}$ and $\leq^{\forall}_{obs}$ agree.*

To contrast $\leq^{\forall}_{sem}$ with the ideal model ordering $\leq^{\forall}_{ideal}$, consider the following example, in which we omit contexts and quantifiers when empty. In the regular tree model the only solution of $C = \{\top \to \bot \leq t,\, t \leq \top \to \top,\, t \leq \bot \to \bot\}$ is $\rho = [t \mapsto \top \to \bot]$, which satisfies also $t \leq \top \to \bot$; hence $(\top \to \bot) \to \top \to \bot \setminus \emptyset \leq^{\forall}_{sem} \forall t.\, t \to t \setminus C$. But this fails for $\leq^{\forall}_{ideal}$ since in the ideal model *e.g.* $[\, t \mapsto \forall t'.\, t' \to t' \,]$ is a solution of C which does not satisfy $t \leq \top \to \bot$. As a consequence the ideal model ordering does not offer full type abstraction with respect to the operational subtyping $\leq^{\forall}_{obs}$.

4.3 Decidable Subtyping

The question of decidability of $\leq^{\forall}_{sem}$ is open; we show how it may be approximated by a powerful decidable relation. The material of the previous section is used to define this decidable relation.

The informal idea leading to the decidable relation is simple: observe that adding constraints to a set may only shrink the set of its solutions. For constrained types $\kappa' = \forall \overline{t'}.\, A' \Rightarrow \tau' \setminus C'$ and $\kappa'' = \forall \overline{t''}.\, A'' \Rightarrow \tau'' \setminus C''$, Definition 22 states that $\kappa' \leq^{\forall}_{sem} \kappa''$ if a certain relation holds for *each* instance of κ'' (that is, for the unrestricted set of solutions of C'') and *some* corresponding instance of κ' (that is, an element of a possibly restricted subset of solutions of C'). Thus, assumming that $\{\overline{t'}\}$ and $\{\overline{t''}\}$ are disjoint, we would have a proof of $\kappa' \leq \kappa''$ if we could show that the relations $\tau' \leq \tau''$ and $(A'' \leq A')$ hold under C'' and C' together with some set C of constraints which do not "constrain further" the type variables $\overline{t''}$ (but possibly add constraints on $\overline{t'}$).

Applying the machinery developed in Sect. 3, these ideas are formalized in the following definition of a relation approximating $\leq^{\forall}_{sem}$.

Definition 24. $\kappa' \leq^{\forall}_{dec} \kappa''$ if $\kappa' = \forall \overline{t'}.\, A' \Rightarrow \tau' \setminus C'$ and $\kappa'' = \forall \overline{t''}.\, A'' \Rightarrow \tau'' \setminus C''$ for some $\{\overline{t'}\} \cap \{\overline{t''}\} = \emptyset$, and there exists a consistent canonical map K such that $K \vdash C' \cup (A'' \leq A') \cup \{\tau' \leq \tau''\}$ and $\mathit{Can}(\mathit{Ker}(C''))$ is a submap of K.

Here the map K represents the union of C'', C', and C of our informal discussion: it has *Can*(*Ker*(C'')) as a submap (meaning $\{\overline{t'}\}$ are not further constrained), it entails C', and its constraints on $\{\overline{t'}\}$ may be stronger that those in C' in order to ensure that the relations between root types and contexts hold. The following theorem shows that $\leq^{\forall}_{dec}$ is indeed an approximation to $\leq^{\forall}_{sem}$.

Theorem 25. *If* $\kappa' \leq^{\forall}_{dec} \kappa''$, *then* $\kappa' \leq^{\forall}_{sem} \kappa''$.

Although the incompleteness of $\vdash$ with respect to $\models$ implies incompleteness of $\leq^{\forall}_{dec}$ with respect to $\leq^{\forall}_{sem}$, the relation $\leq^{\forall}_{dec}$ is still quite powerful: it subsumes the relation of instantiation between type schemes in the Hindley/Milner system, the subtyping relation between recursive types in the Amadio/Cardelli system, and their union on recursive polymorphic types in shallow (prenex) form. Consider Hindley/Milner subtyping in more detail. The type scheme $\forall\overline{t''}.\,\tau''$ is an instance of $\forall\overline{t'}.\,\tau'$ if $\tau'' = \sigma\tau'$, where σ is a simple type substitution on $\{\overline{t'}\} = \mathit{FTV}(\tau')$, and $\{\overline{t''}\} = \mathit{FTV}(\tau'')$; then we have $\forall\overline{t'}.\,\langle\rangle \Rightarrow \tau' \setminus \emptyset \leq^{\forall}_{dec} \forall\overline{t''}.\,\langle\rangle \Rightarrow \tau'' \setminus \emptyset$ by Definition 24, as evidenced by the canonical constraint map $K = (\overline{t' \leq \sigma(t')},\ \overline{t' \geq \sigma(t')})$, which entails $\tau' \leq \sigma(\tau')$ and is obviously consistent. Closed recursive types can also be represented as constrained types, with the constraint set effectively encoding a regular system; when restricted to these types, $\leq^{\forall}_{dec}$ is equivalent to the system of Amadio and Cardelli.

Furthermore, $\leq^{\forall}_{dec}$ is sufficiently strong to allow proving correctness of many useful simplifications of types inferred by the system. For example, $\leq^{\forall}_{dec}$ can be used to show the soundness of the constraint set simplification "garbage collection" of [9], which allows the removal of "unreachable" constraints.

Definition 26. Given a constrained type $\forall\overline{t}.\,A \Rightarrow \tau \setminus K$, where K is a canonical constraint map, a type variable t is *positively reachable* if t occurs positively in τ, or negatively in A, or positively in the canonical lower bound in K of some positively reachable t', or negatively in the constructed upper bound in K of some negatively reachable t''; *negative reachability* is defined symmetrically. (Recall that an occurrence of a variable in a simple type is *positive* (resp. *negative*) if it occurs inside an even (odd) number of type subterms in argument position of $\rightarrow$.)

A constraint $t \leq \tau' \in K$ (resp. $t \geq \tau' \in K$) is *reachable* if t is negatively (positively) reachable; $t \leq t' \in K$ is reachable if t is negatively and t' is positively reachable.

This notion of reachability is motivated by the type rules (Sect. 5, Fig. 2), which only set upper bounds on types of subterms. Thus for instance a type variable t in the type κ of a term e can only obtain new upper bounds (when e is used as a subterm) if t is positively reachable in κ; in this case t's lower bounds may be the source of an inconsistency (via transitivity). Conversely, however, if t is not reachable positively, its lower bounds are not going to cause inconsistency in any use of e—hence they may safely be ignored; for example,

$$\forall t.\,\langle\rangle \Rightarrow t \setminus \{\top\rightarrow t \leq t,\ t \leq t\rightarrow\top\} \ \leq^{\forall}_{dec}\ \forall t.\,\langle\rangle \Rightarrow t \setminus \{\top\rightarrow t \leq t\}.$$

Proposition 27. *If $GC(\kappa)$ is the constrained type obtained by removing the unreachable constraints in κ, then $GC(\kappa) \leq^{\forall}_{dec} \kappa$ and $\kappa \leq^{\forall}_{dec} GC(\kappa)$.*

Pottier [20] offers an alternative definition of reachability which ignores the polarity of the occurences of type variables. Our experience with applications of constrained type systems to object-oriented languages [10, 9] shows that keeping track of polarity makes a significant difference when simplifying types inferred for new objects (which are fixed points of classes). Type variables associated with objects have upper bounds inherited from the class definition (before taking the fixed point); they are often unreachable by our definition but not by Pottier's. Additionally, removing more constraints often enables other simplifications, *e.g.* unifying a type variable with its bound.

We present an outline of an algorithm for computing $\forall\overline{t'}.\, A' \Rightarrow \tau' \setminus C' \leq^{\forall}_{dec} \forall\overline{t''}.\, A'' \Rightarrow \tau'' \setminus C''$. The algorithm either fails, if the subtyping does not hold, or it produces a set of constraints C which only put bounds on the type variables in $\{\overline{t'}\}$; the constraint map K required by Definition 24 can then be obtained by extending *Can*(C'') with *Can*(*Ker*$(C' \cup C)$). The algorithm is very similar to the one for computing closure of a constraint set; in fact it is a generalization of the latter.

Algorithm 28 *$\forall\overline{t'}.\, A' \Rightarrow \tau' \setminus C' \leq^{\forall}_{dec} \forall\overline{t''}.\, A'' \Rightarrow \tau'' \setminus C''$ is computed as follows.*

We start by computing $K'' =$ Can(C'') and with an initially empty set C_0 of new constraints on variables in $\{\overline{t''}\}$ which are "pending proof," and proceed as in computing the closure of $C' \cup \{\tau' \leq \tau''\} \cup (A'' \leq A')$, namely, failing on inconsistent constraints, and reducing consistent ones between constructed types to constraints on variables, of the form $t \leq \tau$ (or $\tau \leq t$). When $t \in \{\overline{t'}\}$, if the constraint is already in $C' \cup C$, the search succeeds; otherwise we add these constraints to C and continue as in the closure computation by searching for a proof of $\tau_L \leq \tau$ (resp. $\tau \leq \tau_U$), where τ_L and τ_U represent the lower and upper bound(s) on t in $C' \cup C$ so far. However, when $t \in \{\overline{t''}\}$, we instead attempt to prove *that this constraint is implied (by the rules for primitive subtyping) by the constraints on t in K''. The proof search goes much as described in Sect. 3.4: if τ is already an upper (lower) bound of t in $K'' \cup C_0$, it succeeds, otherwise the new constraint is added to C_0, and we search for a proof of the constraint $\tau_U \leq \tau$ (resp. $\tau \leq \tau_L$), where τ_U and τ_L are the canonical upper and lower bounds on t in K''.*

Thus, the algorithm treats variables in $\{\overline{t'}\}$ and $\{\overline{t''}\}$ differently, but symmetrically: it compares new upper bounds on a t' with its old lower bounds, but new upper bounds on a t'' with its old (canonical) upper bound. (The reader may have noticed that converting C' to a canonical constraint map is not necessary for this algorithm; however it may improve its performance.)

Theorem 29. *The relation $\leq^{\forall}_{dec}$ is decidable.*

5 Soundness of the Type System and Completeness of Inference

The typing rules shown in Fig. 2 infer sequents of the form $\Gamma \vdash^{\tau} e : \kappa$; the type environment Γ only assigns constrained types to let-bound variables, while the types

$$\text{(VAR)}\ \Gamma \vdash^{\mathrm{T}} x : \forall \bar{t}.\, \langle \overline{x_i : \tau_i},\, x : \tau \rangle \Rightarrow \tau \setminus C, \qquad \{\bar{t}\} \supseteq \bigcup_i \mathit{FTV}(\tau_i, \tau, C)$$

$$\text{(ABS)}\ \frac{\Gamma \vdash^{\mathrm{T}} e : \forall \bar{t}.\, \langle \overline{x_i : \tau_i},\, x : \tau \rangle \Rightarrow \tau' \setminus C}{\Gamma \vdash^{\mathrm{T}} \lambda x.\, e : \forall \bar{t}.\, \langle \overline{x_i : \tau_i} \rangle \Rightarrow \tau \to \tau' \setminus C}$$

$$\text{(APP)}\ \frac{\Gamma \vdash^{\mathrm{T}} e' : \forall \bar{t}.\, A \Rightarrow \tau'' \to \tau \setminus C \qquad \Gamma \vdash^{\mathrm{T}} e'' : \forall \bar{t}.\, A \Rightarrow \tau'' \setminus C}{\Gamma \vdash^{\mathrm{T}} e'\, e'' : \forall \bar{t}.\, A \Rightarrow \tau \setminus C}$$

$$\text{(LETVAR)}\ \Gamma \vdash^{\mathrm{T}} X : \Gamma(X), \qquad X \in \mathit{Dom}(\Gamma)$$

$$\text{(LET)}\ \frac{\Gamma \vdash^{\mathrm{T}} e : \kappa \qquad \Gamma, X : \kappa \vdash^{\mathrm{T}} e' : \kappa'}{\Gamma \vdash^{\mathrm{T}} \mathsf{let}\ X = e\ \mathsf{in}\ e' : \kappa'}$$

$$\text{(SUB)}\ \frac{\Gamma \vdash^{\mathrm{T}} e : \kappa \qquad \kappa \leq^{\forall} \kappa'}{\Gamma \vdash^{\mathrm{T}} e : \kappa'}$$

Note: the closures of the constraint sets in all conclusions must be consistent.

Fig. 2. Typing rules.

of λ-bound variables are included in the context component of κ. Each rule has the implicit side condition that the closure of the constraint set in the constrained type in its conclusion is consistent. Rule (APP) requires the types of the subterms to share the context A, constraint set C, and set of bound variables $\{\bar{t}\}$. Rule (LET) is sound with respect to the call-by-name semantics;[4] constraints on types of variables free in e need not be reflected in κ' unless X occurs free in e'. Finally the subsumption rule (SUB) replaces the constrained type of a term by a supertype; it is thus the only rule that may allow the constraint set in the type of a term to be taken into account or modified. The rules are parametric in the choice of $\leq^{\forall}$, for which we considered a number of different possibilities; the notation $\vdash^{\mathrm{T}}$ represents the rules with abstract $\leq^{\forall}$, and $\vdash^{\mathrm{T}}_{sem}$ for instance represents $\vdash^{\mathrm{T}}$ with $\leq^{\forall}$ defined as the concrete relation $\leq^{\forall}_{sem}$.

Rules for type inference are presented[5] in Fig. 3; there is no rule for subsumption, and the let-related rules are the same as in $\vdash^{\mathrm{T}}$ and hence omitted.

We may now establish soundness of the typing rules of Fig. 2. In our previous proofs of soundness of constrained typing systems [10], a direct subject reduction argument was used. Recent observations concerning the close relation between constrained type systems and simple type systems [19] allow us to establish soundness based on soundness of a simple type system. We believe this direct approach to type soundness of constrained type systems should be applicable to other constrained type languages.

Amadio and Cardelli [3] present a type system $\vdash^{\mu}$ with recursive types (modeled by regular trees) and a subtyping relation on them equivalent to $\leq_{tree}$. This system can be applied to the let-free fragment of our language to produce sequents of the form $\Phi \vdash^{\mu} e : \varphi$, where Φ is a finite map from variables to regular trees whose role in our type system $\vdash^{\mathrm{T}}$ is played by a context A.

[4] A version sound with respect to call-by-value can be obtained by defining $\mathsf{let}_v\ X = e\ \mathsf{in}\ e'$ as $\mathsf{let}\ X = e\ \mathsf{in}\ (X;\, e')$ for type-checking purposes.

[5] We write the inference rules with a top-down propagation of the contexts; a bottom-up presentation with synthesized context components is also possible.

$$(\textsc{Var}^{\textsc{i}})\ \Gamma \vdash^{\textsc{i}} x : \forall\overline{t_i}.\, t.\, \langle \overline{x_i : t_i},\, x : t\rangle \Rightarrow t \setminus \emptyset$$

$$(\textsc{Abs}^{\textsc{i}})\ \frac{\Gamma \vdash^{\textsc{i}} e : \forall\overline{t_i}.\, t.\, \langle \overline{x_i : t_i},\, x : t\rangle \Rightarrow \tau \setminus C}{\Gamma \vdash^{\textsc{i}} \lambda x.\, e : \forall\overline{t_i}.\, t.\, \langle \overline{x_i : t_i}\rangle \Rightarrow t{\to}\tau \setminus C},\ \{\overline{x_i}\} = \mathit{FV}(\lambda x.\, e)$$

$$(\textsc{App}^{\textsc{i}})\ \frac{\Gamma \vdash^{\textsc{i}} e' : \forall\overline{t'}.\, A \Rightarrow \tau' \setminus C' \qquad \Gamma \vdash^{\textsc{i}} e'' : \forall\overline{t''}.\, A \Rightarrow \tau'' \setminus C''}{\Gamma \vdash^{\textsc{i}} e'\, e'' : \forall\overline{t'},\, \overline{t''},\, t.\, A \Rightarrow t \setminus C' \cup C'' \cup \{\tau' \leq \tau''{\to}t\}}$$

where $\{\overline{t'}\} - \mathit{FTV}(A)$, $\{\overline{t''}\} - \mathit{FTV}(A)$, $\mathit{FTV}(A)$, and $\{t\}$ are all disjoint

Fig. 3. Typing rules modified for type inference.

We now establish that a typing derivation in $\vdash^{\textsc{t}}_{sem}$ can be viewed as a family of derivations in $\vdash^{\mu}$.

Definition 30. The let-*expansion* $\mathit{LE}(e)$ of a term e is defined as the homomorphic extension of $\mathit{LE}(\mathsf{let}\ X = e'\ \mathsf{in}\ e'') = (\mathit{LE}(e');\, \mathit{LE}(e'')[\mathit{LE}(e')/X])$, where the postfix $[/]$ denotes capture-free substituton.

Theorem 31. *If* $\emptyset \vdash^{\textsc{t}}_{sem} e : \kappa$, *then* $\emptyset \vdash^{\mu} \mathit{LE}(e) : \varphi$ *for each* $\varphi \in \mathit{Inst}(\kappa)$. *If* $\emptyset \vdash^{\mu} \mathit{LE}(e) : \varphi$, *then* $\emptyset \vdash^{\textsc{t}}_{sem} e : \kappa$ *for some* κ *such that* $\varphi \in \mathit{Inst}(\kappa)$.

Corollary 32. *The type system* $\vdash^{\textsc{t}}_{sem}$ *is sound.*

Proof. Implied by the soundness of $\vdash^{\mu}$ [3]: the typability of a term e under $\vdash^{\textsc{t}}_{sem}$ implies the typability of $\mathit{LE}(e)$ under $\vdash^{\mu}$, which by soundness of $\vdash^{\mu}$ implies that the evaluation of $\mathit{LE}(e)$ will not cause a run-time error. Since the let-expansion of e is observationally equivalent to e, this implies that the evaluation of e is free of run-time errors.

Corollary 33. *The type system* $\vdash^{\textsc{t}}_{dec}$ *is sound.*

Theorem 34. *The inference system* $\vdash^{\textsc{i}}$ *is complete with respect to* $\vdash^{\textsc{t}}_{sem}$.

6 Related Work

Pottier [20] has independently derived results that are related to some results of this paper. He defines a syntactic and a semantic notions of entailment on constraint sets, shows they are equivalent, and presents a type system with subsumption based on this entailment. He also provides an algorithm for an approximation to the entailment relation, which appears equivalent to $K \vdash \tau \leq \tau'$ for canonical K; finally, the theory is used as a basis for proving the soundness of a number of constrained type simplifications. However the entailment relations do not take into account reachability of type variables, which depends on the polarity of their occurrences and hence on the root type; in particular his syntactic entailment $C' \vdash C''$ requires $C'' \cup C$ to be consistent whenever $C' \cup C$ is, for *any* constraint set C, including sets that put bounds on unreachable type variables, which is not possible during type inference. As a consequence both the relation between

constrained types, implied by his subsumption rule, and its decidable approximation are strictly less powerful than ours.

Jim [13] also defines a notion of $\leq^{\forall}$ that relates fewer types than ours but is still powerful enough to prove some principal typing properties for constrained type systems.

Previous researchers [21, 4] have addressed the problem of subtyping constrained types in the context of a system where recursive constraints are not allowed. The choice of whether to allow or disallow recursive constraints greatly changes the theory.

7 Conclusions

This paper establishes a foundation for constrained type theory, in particular via a powerful characterization of subtyping on constrained types. We introduce two natural notions of subtyping, observational $\leq^{\forall}_{obs}$ and semantic $\leq^{\forall}_{sem}$, and prove that they are equivalent; we further give a decidable approximation $\leq^{\forall}_{dec}$ to these relations. Both results represent improvements over recent work on subtyping of constrained types with recursive constraints [9, 20, 13]. We also introduce a novel closed form of constraint types with contexts, which eliminates the problems associated with free type variables. Finally, we present a type system with principal constrained types, and establish its soundness via reduction to the system of Amadio and Cardelli.

The most generous relations $\leq^{\forall}_{sem}$ and $\leq^{\forall}_{obs}$ may be undecidable, but we believe that $\leq^{\forall}_{dec}$ is powerful enough to be useful in practice for signature matching and constraint simplification. Our confidence in the system stems from the fact that $\leq^{\forall}_{dec}$ subsumes the Amadio/Cardelli subtyping of recursive types, the type scheme instantiation in the Hindley/Milner system, and the subtyping relation of [20]. Additionally, it turns out that the known simplifications of constraint sets do not test the limits of the system based on $\leq^{\forall}_{dec}$; we have shown in this paper that $\leq^{\forall}_{dec}$ can be used to demonstrate the correctness of simplifications not included in other systems. Similarly, functor signatures may generally be produced by starting with an inferred constrained type and transforming it in regular ways, thus avoiding constrained types which $\leq^{\forall}_{dec}$ does not relate to the inferred type. We have yet to find a realistic subtyping example which is semantically sound but is not derivable using $\leq^{\forall}_{dec}$, but most convincing would be the performance of a system that uses it for signature matching and simplifications in practice on real code; we are in the process of constructing an implementation for this purpose.

Acknowledgements We wish to thank Simon Marlow, François Pottier, Didier Rémy, Philip Wadler, and the anonymous referees for many helpful comments and suggestions.

References

1. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.
2. A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

3. R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993. Extended abstract in POPL 1991.
4. François Bourdoncle and Stephan Merz. On the integration of functional programming, class-based object-oriented programming, and multi-methods. Technical Report 26, Centre des Mathématiques Appliquées, Ecole des Mines de Paris, 1996. Available at `http://www.ensmp.fr/~bourdonc/`.
5. Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.
6. L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.
7. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
8. Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, XEROX Palo Alto Research Center, `CSLPubs.parc@xerox.com`, 1990.
9. J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95*, pages 169–184, 1995.
10. J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. `http://www.elsevier.nl/locate/entcs/volume1.html`.
11. J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, 1995.
12. Y.-C. Fuh and P. Mishra. Type inference with subtypes. In *European Symposium on Programming*, 1988.
13. Trevor Jim. *Principal typings and type inference*. PhD thesis, MIT, 1996. (to appear).
14. S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *ACM Conference on Lisp and Functional Programming*, pages 193–204, 1992.
15. D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.
16. P. Mishra and U. Reddy. Declaration-free type checking. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21, 1985.
17. John C. Mitchell. Coercion and type inference (summary). In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984.
18. John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.
19. Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *TOPLAS*, 18(5), September 1996.
20. François Pottier. Simplifying subtyping constraints. In *First International Conference on Functional Programming*, pages 122–133, 1996.
21. Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23, 1994.