Type Inference in the Presence of Overloading, Subtyping and Recursive Types

Stefan Kaes

Praktische Informatik
TH Darmstadt
D-6100 Darmstadt, Magdalenenstr. 11c
E-mail: kaes@pi.informatik.th-darmstadt.de

ABSTRACT

We present a unified approach to type inference in the presence of overloading and coercions based on the concept of *constrained types*. We define a generic inference system, show that subtyping and overloading can be treated as a special instance of this system and develop a simple algorithm to compute principal types. We prove the decidability of type inference for the class of *decomposable predicates* and develop a canonical representation for principal types based on *most accurate simplifications* of constraint sets. Finally, we investigate the extension of our techniques to *recursive types*.

1 INTRODUCTION

Parametric polymorphism, as developed by Milner [14, 3], has been used as the basic building block in the design of type systems for various programming languages. A number of extensions have been proposed to increase the expressiveness of the basic scheme. Among them are systems for structural subtyping [15, 5], extended subtyping [24, 11, 17, 19] and overloading [12, 23, 20].

Polymorphism allows us to abstract over the types of function arguments and thus supports the development of reusable small scale software components. The abstraction is uniform, i.e. every instance of the inferred parameter types is allowed. Restrictions on function parameters like "$+ : \alpha \rightarrow \alpha \rightarrow \alpha$ can be applied to integer or real numbers", "$=: \alpha \rightarrow \alpha \rightarrow bool$ can be applied to any first order type" or "$f : \alpha \rightarrow \beta \rightarrow int$ can be applied to any pair of subtypes α, β of int" cannot be expressed with pure parametric polymorphism.

At first sight, overloading may seem as a pure notational convenience, but this is true only for overloaded operators with a finite set of overloading instances. Examples for this kind of overloading are arithmetic operators, which will typically be overloaded for every availabe number type of the language in question.[1]

A more interesting example of an overloaded operator is polymorphic equality as it is supported in the current version of Standard ML [9]. Equality of functions is undecidable in general, hence the type system should limit the application of computable equality to pure first order data. But the type $\forall\alpha.\alpha \rightarrow \alpha \rightarrow bool$ is clearly too general. This problem has been solved in Standard ML by the introduction of a special class of equality type variables, which are instantiable to non function types only. With this extension, computable equality is given the type $\forall\alpha'.\alpha' \rightarrow \alpha' \rightarrow bool$, where the prime on type variable α restricts the set of possible instances to types not built with the function type constructor.

In previous work [12] we have demonstrated that this approach can be generalized to a restricted form of overloading which we have termed *parametric overloading*. The restrictions imposed on overloaded operators are (1) that the type of each overloaded operator can described by a type scheme over *one* distinguished type variable and (2) the possible instances of this type variable can be defined inductively over the structure of type expressions. In this case, type variables can be annotated with sets of overloaded operator names, in order to restrict the set of valid instantiations of such a type variable α_X, to types admissible as arguments of all operators in X. The type of the equality operation would then be $\forall\alpha_{\{=\}}.\alpha_{\{=\}} \rightarrow \alpha_{\{=\}} \rightarrow bool$.

Given the inductive definition of admissable argument types, it is possible to adapt conventional unification algorithms to deal with annotated type variables. In fact, the unification algorithm described in [12] can be interpreted as an unification algorithm for order sorted algebras with overloaded type constructors, where the sort structure forms a complete lattice. The connection between order sorted unification and type systems for overloading has recently been investigated in [16].

In the presence of subtyping, types are expressed relative to a set of subtype constraints. The function $twice \equiv \lambda f.\lambda x.f\ (f\ x)$ for example, would be given the type $\forall\alpha,\beta.(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \mid \beta \lhd \alpha$, which states that *twice* maps functions of type $\alpha \rightarrow \beta$ to functions of type $\alpha \rightarrow \beta$, provided β is a subtype of α. The symbol $\lhd$ can be interpreted as a binary *predicate* on ground types, restricting the possible instantiations of α and β to pairs (s,t) satisfying $s \lhd t$.

Overloaded operators can be treated in a similar way: For each overloaded function symbol o, we introduce a (n-ary) predicate symbol p_o describing admissable arguments for o. The type of o can then be given as a *constrained type scheme* $\forall\alpha_1,\ldots,\alpha_n.\tau \mid p_o(\alpha_1,\ldots,\alpha_n)$. Typical examples

[1] Nevertheless, the possibility to give semantically similar operations a common name must not be undervalued and should be supported by the type system.

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0193...$1.50

are:

$=:\alpha \to \alpha \to bool \mid p_=(\alpha)$	computable equality
$\leq:\alpha \to \alpha \to bool \mid p_\leq(\alpha)$	total order relation
$+:\alpha \to \alpha \to \alpha \mid p_+(\alpha)$	addition
$\in:\alpha \to \beta \to bool \mid p_\in(\alpha,\beta)$	membership test
$\downarrow:\alpha \to \beta \to \gamma \mid p_\downarrow(\alpha,\beta,\gamma)$	data structure indexing

Overloading and coercion constraints can be arbitrarily combined as in

$$inc\ x \equiv x + 1$$
$$inc : \forall\alpha,\beta.\alpha \to \beta \mid int \lhd \beta,\ \alpha \lhd \beta,\ p_+(\beta),$$

which tells us that *inc* applied to values of type α yields values of type β, whenever *int* and α are subtypes of β and β admits an addition operation.

The remainder of this paper is organized as follows: First we define the notion of constrained types precisely and present a generic type system which admits a simple algorithm to compute principal types. However, principal types computed by this algorithm are rather weak: they consist of a type expression and a set of constraints which has to be solved separately in order to prove typability. We introduce the concept of *simplifying substitutions* and show that there exists a *most accurate simplification* which can be used to obtain a *canonical* principal type representation. Based on this concept we modify our inference algorithm, obtaining Milner's original algorithm as a special case. We then introduce the class of *decomposable* predicates, for which constraint solving is decidable and show that most accurate simplifications are computable for the subclass of *structural similarity* enforcing predicates. Finally we sketch an algorithm for constraint solving in the presence of recursive types as a conservative extension of the finite case.

2 CONSTRAINED TYPES

We shall confine our investigation to the usual core-ML language, which corresponds to basic lambda calculus, extended by a let-construct to permit user defined polymorphism. The expressions of core-ML are generated by the grammar

$$M ::= x \mid \lambda x.M_1 \mid M_1\ M_2 \mid \textbf{let}\ x = M_1\ \textbf{in}\ M_2$$

where x ranges over a countable alphabet of syntactic variables. As usual, we will denote the set of free variables of M by $\mathcal{FV}(M)$.

Types are elements of the term algebra $T_F(\mathcal{V})$ generated by a finite n-indexed family of constructors $F = \bigcup_{0\leq n\leq k} F_n$. We define F_+ as $F \setminus F_0$. The set of type variables ocurring in τ is denoted by $\mathcal{V}(\tau)$ and $root(\tau)$ denotes the outermost symbol of τ. Let P be a finite n-indexed family of predicate symbols. The set of predicate constraints over $T_F(\mathcal{V})$ is defined as $\{\, p(\tau_1,\ldots,\tau_n) \mid p \in P_n, \tau_i \in T_F(\mathcal{V}) \,\}$. We use $args(p(\overline{\tau_n}))$ to denote $\{\tau_1,\ldots,\tau_n\}$.

A substitution $S \in \mathcal{V} \to T_F(\mathcal{V})$ is a mapping from type variables to types, such that $S(\alpha) \neq \alpha$ only for finitely many type variables. Substitutions are extended canonically to types and constraint sets and will often be denoted by $\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\}$. Substitutions form a quasi order w.r.t. subsumption, where R subsumes S on $W \subseteq \mathcal{V}$, written $R \leq^W S$, if $\exists R' : \forall\alpha \in W : S(\alpha) = R'(R(\alpha))$. The set of substitutions forms a complete lower semi lattice modulo $\cong$ with the identity substitution as its least element, if we define $R \cong^W S \iff R \leq^W S \wedge S \leq^W R$. A set X of substitutions is minimal, if $S \not\leq R$ for all S, R in X. The greatest lower bound of two substitutions is denoted by $S \wedge R$.

We assume that we are given an interpretation of P, i.e. a family of total computable functions $(\hat{p})_{p\in P}$, such that for $p \in P_n$ $\hat{p} : T_F^n \to \mathbf{2}$. A constraint set C is satisfiable if there exists a substitution S such that $p(\tau_1,\ldots,\tau_n) \in C \Rightarrow \hat{p}(S(\tau_1),\ldots,S(\tau_n))$. Satisfiability will be denoted as $S \models C$.

Definition 2.1 *A* constrained type *is a pair $\tau|C$, consisting of a type expression τ and a set of constraints C. A* constrained type scheme *is of the form $\forall\alpha_1,\ldots,\alpha_n.\tau|C$.*[2]

Type schemes are used to discriminate between type variables that depend on the context, which are called specific, and those that can be instantiated arbitrarily, which are called generic. Variables $\alpha_1,\ldots,\alpha_n$ in type scheme $\sigma = \forall\overline{\alpha_n}.\tau|C$ (an abbreviation of $\forall\alpha_1,\ldots,\alpha_n.\tau|C$) are called generic, whereas any other type variable free in $\tau|C$ is called specific.

Essential for the definition of the instance relation on constrained types is the existence of a notion of *entailment* on constraint sets, written $C_1 \Vdash C_2$, the exact definition of which may depend on the particular predicate system.[3] However, the results of this section can be proved if we require that the entailment relation is closed under substitution and that it satifies $C_1 \Vdash C_2 \Rightarrow \forall L : L \models C_1 \Rightarrow L \models C_2$. We write $C_1 \equiv C_2$ for $C_1 \Vdash C_2 \wedge C_2 \Vdash C_1$.

If σ is a type scheme and S a substitution, then $S(\sigma)$ is the type scheme obtained from σ by replacing each variable $\alpha \in \mathcal{FV}(\sigma)$ by $S(\alpha)$, possibly renaming bound variables in σ to avoid name clashes. A type scheme $\sigma' = \forall\overline{\beta_m}.\tau'|C'$ is a generic instance of type scheme $\sigma = \forall\overline{\alpha_n}.\tau|C$, written $\sigma' \prec \sigma$, if $\tau' = \{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\}\tau$, no β_i is free in $\tau|C$ and $C' \Vdash S(C)$.

Typability of an expression M is expressed as a judgement $C, \Gamma \vdash M : \tau$, called typing, which can be read as "M has type τ under type assumption Γ, provided C is satisfiable". A type assumption Γ is a finite mapping from variables to type schemes. We use $[x_1 : \sigma_1,\ldots,x_n : \sigma_n]$ to denote the type assumption assigning σ_i to x_i for $i = 1..n$ and $\Gamma_1 + \Gamma_2$ for the extension of Γ_1 with the assignments of Γ_2. The closure, or generalisation, of a constrained type $\tau|C$ in the context of a type assumption Γ is denoted by $gen(\Gamma, \tau|C)$; it is the type scheme $\forall\overline{\alpha_n}.\tau|C'$ such that $\{\alpha_1,\ldots,\alpha_n\} = \mathcal{V}(\tau|C) - \mathcal{FV}(\Gamma)$ and $C' = \{\, p(\overline{\tau}) \in C \mid \mathcal{V}(p(\overline{\tau})) \cap \overline{\alpha_n} \neq \emptyset \,\}$.

Definition 2.2 *A typing $C, \Gamma \vdash M : \tau$ is more general than $C', \Gamma' \vdash M : \tau'$, if there exists a substitution S, such that*

$$(1)\quad x \in dom(\Gamma) \Rightarrow \Gamma'(x) \prec S(\Gamma(x))$$
$$(2)\quad gen(\Gamma', \tau'|C') \prec S(gen(\Gamma, \tau|C))$$

One can think of a large number of different inference systems involving constrained types. For example, there are at least 3 different formulations for a pure subtype system: Allow coercions everywhere, only at variable nodes or only at application arguments. In order to avoid reproving typability for each of these systems, we introduce a *generic* inference system for constrained types. The generic system, which is given in figure 1, is parameterized w.r.t. three functions R_{var}, R_{abs} and R_{app}. Each of these functions takes as arguments the types derived for immediate subexpressions of the corresponding syntactical rule, and maps them into a constrained type scheme $\tau|C$. In this type scheme, τ always corresponds to the type of the composite expression, and C relates this type to the subexpression types. Valid specifications for R_{var}, R_{abs} and R_{app} must satisfy

[2] Unless stated to the contrary we shall use the following variable convention: $\sigma, \sigma', \ldots$ denote type schemes, $\tau, \tau', \ldots$ denote types and $t, t', \ldots$ will be used for monotypes.

[3] See e.g. [5]

$$[\text{VAR}]\ \frac{\tau_i|C_i \prec \Gamma(x) \quad \tau_v|C_v \prec R_{var}(\tau_i) \quad C \Vdash C_i \cup C_v}{C,\Gamma \vdash x : \tau_v}$$

$$[\text{ABS}]\ \frac{C,\Gamma + [x:\tau_a] \vdash M : \tau_r \quad \tau_f|C_f \prec R_{abs}(\tau_a,\tau_r) \quad C \Vdash C_f}{C,\Gamma \vdash \lambda x.M : \tau_f}$$

$$[\text{APP}]\ \frac{C,\Gamma \vdash M_1 : \tau_f \quad C,\Gamma \vdash M_2 : \tau_a \quad \tau_r|C_r \prec R_{app}(\tau_f,\tau_a) \quad C \Vdash C_r}{C,\Gamma \vdash M_1 M_2 : \tau_r}$$

$$[\text{LET}]\ \frac{C_1,\Gamma \vdash M_1 : \tau_1 \quad C,\Gamma + [x : gen(\Gamma,\tau_1|C_1)] \vdash M_2 : \tau_2 \quad C \Vdash C_1}{C,\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2}$$

Figure 1: Generic Type Deduction with Constrained Types

$\mathcal{FV}(R_x(\overline{\tau})) = \mathcal{V}(\overline{\tau})$, i.e. the free type variables of the application of such a function must consist of the type variables occuring in the supplied arguments.

The inference rules are fairly obvious, with the exception of rule [LET]. It is the only rule which is not parameterized. It states that expression **let** $x = M_1$ **in** M_2 can be given the constrained type $\tau_2|C$, provided $\tau_1|C_1$ is a constrained type for M_1 in the context Γ, $\tau_2|C$ is a constrained type for M_2 in the context extended with the assumption $x : \sigma$, where σ is the generalisation of $\tau_1|C_1$ in the context Γ. Note that the side condition $C \Vdash C_1$ effectively forbids typable expressions with untypable subexpressions. It is not difficult to see that rule [LET] can be equivalently replaced by

$$\frac{C_1,\Gamma \vdash M_1 : \tau_1 \quad C,\Gamma \vdash M_2[M_1/x] : \tau_2 \quad C \Vdash C_1}{C,\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2}$$

A few examples should suffice to show the generality of our generic system. Milners type deduction system for ML-style polymorphism, or more accurately, its syntax oriented variant, is just a special instance of a constraint based system with equality as the only predicate symbol. It can be obtained by instantiating R_{var}, R_{abs} and R_{app} as

$$\begin{aligned} R_{var}(\tau_i) &= \tau_i|\emptyset \\ R_{abs}(\tau_a,\tau_r) &= \tau_a \to \tau_r|\emptyset \\ R_{app}(\tau_f,\tau_a) &= \forall\alpha.\ \alpha|\{\tau_f{=}\tau_a \to \alpha\} \end{aligned}$$

A subtype system, where coercions occur at function applications only, is given by

$$\begin{aligned} R_{var}(\tau_i) &= \tau_i|\emptyset \\ R_{abs}(\tau_a,\tau_r) &= \tau_a \to \tau_r|\emptyset \\ R_{app}(\tau_f,\tau_a) &= \forall\alpha,\beta.\ \beta|\{\tau_f{=}\alpha \to \beta,\ \tau_a \lhd \alpha\} \end{aligned}$$

whereas

$$\begin{aligned} R_{var}(\tau_i) &= \forall\alpha.\ \alpha|\{\tau_i \lhd \alpha\} \\ R_{abs}(\tau_a,\tau_r) &= \tau_a \to \tau_r|\emptyset \\ R_{app}(\tau_f,\tau_a) &= \forall\alpha.\ \alpha|\{\tau_f{=}\tau_a \to \alpha\} \end{aligned}$$

describes a system where only variables are coerced. On the other hand, if $app(\tau_f,\tau_a,\tau_r)$ states that values of type τ_f appplied to values of type τ_a yield values of type τ_r, then

$$\begin{aligned} R_{var}(\tau_i) &= \tau_i|\emptyset \\ R_{abs}(\tau_a,\tau_r) &= \tau_a \to \tau_r|\emptyset \\ R_{app}(\tau_f,\tau_a) &= \forall\alpha.\ \alpha|\{app(\tau_f,\tau_a,\alpha)\} \end{aligned}$$

gives a system with an overloaded application primitive.

The generic inference system is syntax oriented: there is one inference rule for every syntactic rule. Thus any instance of our generic type system can be turned directly into a type inference algorithm: simply traverse the expression and collect all relevant constraints implied by the inference rules. The algorithm, which we call $\mathcal{C}$, is given in figure 2. It makes use of an auxiliary function *inst*, which takes a constrained type scheme $\forall\overline{\alpha_n}.\tau|C$ and consistently replaces each α_i in $\tau|C$ by a "new" type variable.

$\mathcal{C}[\![x]\!]\Gamma = \tau_1|C_0 \cup C_1$ where
$\tau_0|C_0 = inst(\Gamma(x))$
$\tau_1|C_1 = inst(R_{var}(\tau_0))$.

$\mathcal{C}[\![\lambda x.M]\!]\Gamma = \tau_1|C_0 \cup C_1$
where α "new" and
$\tau_0|C_0 = \mathcal{C}[\![M]\!](\Gamma + [x:\alpha])$
$\tau_1|C_1 = inst(R_{abs}(\alpha,\tau_0))$.

$\mathcal{C}[\![M_1\ M_2]\!]\Gamma = \tau_3|C_1 \cup C_2 \cup C_3$
where
$\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$
$\tau_2|C_2 = \mathcal{C}[\![M_2]\!]\Gamma$
$\tau_3|C_3 = inst(R_{app}(\tau_1,\tau_2))$.

$\mathcal{C}[\![\textbf{let } x = M_1 \textbf{ in } M_2]\!]\Gamma = \tau_2|C_1 \cup C_2$
where
$\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$ and
$\tau_2|C_2 = \mathcal{C}[\![M_2]\!](\Gamma + [x : gen(\Gamma,\tau_1|C_1)])$.

Figure 2: Algorithm $\mathcal{C}$

Theorem 2.3 *Let an instance of a constraint based inference system be given, M be an expression, Γ be a type assumption and $\tau|C = \mathcal{C}[\![M]\!]\Gamma$. If $\Gamma' \prec S(\Gamma)$ and $C',\Gamma' \vdash M : \tau'$ is a valid typing, then $C,\Gamma \vdash M : \tau$ is valid and more general than $C',\Gamma' \vdash M : \tau'$.*

As an immediate consequence we obtain that an expression is typable under Γ iff the constraint set computed by algorithm $\mathcal{C}$ is satisfiable. Moreover, since T_F and the set of substitutions with domain $\mathcal{V}(\tau|C)$ is recursively enumerable, type inference with constrained types is semi-decidable.

3 CANONICAL PRINCIPAL TYPES

Algorithm $\mathcal{C}$ cannot be used in practice, since satisfiability of the resulting constraint set is not guaranteed. But even if this were the case, one would not obtain a particulary compact representation of the principal type. As an example, cosinder the constraint based Damas-Milner system and the expression $\lambda f.\lambda x.f(fx)$ for which algorithm $\mathcal{C}$ computes the principal type $\alpha \to \beta \to \gamma|\{\alpha = \beta \to \epsilon, \alpha = \epsilon \to \gamma\}$. But $(\beta \to \beta) \to \beta \to \beta|\{\}$ is an equivalent, yet shorter representation of the same type. This type can be obtained from the first one through the use of an unification algorithm to solve the constraints, followed by the application of the most general unifier $U = \{\beta/\epsilon, \beta/\gamma, \beta \to \beta/\alpha\}$,

which enables the elimination of the redundant equations $\{\beta \to \beta = \beta \to \beta, \beta \to \beta = \beta \to \beta\}$.

The most general unifier solves the given constraint set: any substitution of types for type variables in $U(C)$ satisfies the original problem.

Definition 3.1 *A substitution S is a* solution *of C, if $S' \circ S \models C$ for all $S' \in \mathcal{V} \to T_F$.*

At this point, most of the definitional machinery of unification in nonempty equational theories [21] can be carried over to our problem. For a given constraint set C, let $\mathcal{L}_P(C)$ denote its set of solutions for a given predicate system P.

Definition 3.2 *A set L of solutions of C is* complete, *if $L \subseteq \mathcal{L}_P(C)$ and $\forall R \in \mathcal{L}_P(C) : \exists S \in L : S \leq^{\mathcal{V}(C)} R$. L is a complete set of* minimal *solutions of C, if each element of L is minimal w.r.t. subsumption on $\mathcal{V}(C)$, i.e., if $S \neq R \in \mathcal{L}_P(C) \Rightarrow S \not\leq^{\mathcal{V}(C)} R$.*

A complete set of minimal solutions need not exist for a given predicate system, but is unique if it does. In correspondence with unification theory, we may classify constraint solving as follows: Let $\mu\mathcal{L}_P(C)$ denote an arbitrary set of minimal solutions. Constraint solving is

unitary	if $\lvert\mu\mathcal{L}_P(C)\rvert \leq 1$ for all finite C
finitary	if $\lvert\mu\mathcal{L}_P(C)\rvert < \infty$ for all finite C
infinitary	if $\lvert\mu\mathcal{L}_P(C)\rvert = \infty$ for some finite C
decidable	if $\mathcal{L}_P(C) = \emptyset$ is decidable for all finite C
nullary	if $\mu\mathcal{L}_P(C)$ does not exist for some finite C

Unfortunately, even for the simple case of structural subtyping, or overloading with unary predicates, complete sets of minimal solutions are infinitary. It is therefore impossible to take complete sets of minimal solutions as the canonical representation of a principal constrained type.

However, there are two mgu properties which can form a basis for a principal type representation: First, every solution of the original constraint set can be represented as the composition of U and a solution to the simplified constraint set (which happens to be empty in the example above). Second, the most general unifier is the best way to simplify the given constraint set: it determines the maximal amount of information common to all solutions.

Definition 3.3 *A substitution S is called* simplification *of C, iff for every solution $L \models C$ there exists a substitution L', such that $L = L' \circ S$. If in addition for every other simplification S' there exists a substitution S'', such that $S = S'' \circ S'$ then S is called* most accurate simplification *of C.*

In general, a large number of simplifying substitutions will exist for a given constraint set C. For example, the identity substitution is a simplification of every constraint set and the same holds for arbitrary variable renamings. The set of simplifications of C is partially ordered w.r.t. subsumption, it is even a lower semilattice with the identity substitution as its least element.

Computability of most accurate simplifications clearly depends on the interpretation of the predicate symbols in C. The notions of most general unifier and most accurate simplification coincide if C consists of equational constraints only.[4]

Repeated simplification is itself a simplification, i.e. $S' \circ S$ simplifies C, iff S simplifies C and S' simplifies $S(C)$. More interesting is the diamond property of simplifications, which allows the application of simplifying substitutions in arbitrary order: Given that S_1 simplifies C and S_2 simplifies C, one can always find simplifications R_1 of $S_1(C)$ and R_2 of $S_2(C)$, such that $R_1 \circ S_1 = R_2 \circ S_2$. In fact, one can show that $R_1 \circ S_1$ is a least upper bound of S_1 and S_2.

Since substitutions already form a lower semilattice w.r.t. subsumption, this implies that the set of simplifications of a solvable constraint set C forms a lattice w.r.t. subsumption on $\mathcal{V}(C)$: since C is solvable, the size of types in the range of a simplification is bounded, which implies that the set of simplifications is finite (modulo $\cong$) and thus has a largest element.

Lemma 3.4 *$\bigwedge\{L \mid L \models C\}$ is a most accurate simplification for every solvable constraint set C.*

In general, a given constraint set will have an infinite number of solutions and thus prohibits the obvious naive method of computing most accurate simplifications. In these cases we can use the following lemma to our advantage.

Lemma 3.5 *Let C be a set of constraints and $S_1, \ldots, S_n$ be substitutions such that (1) $\forall i \in 1..n : \models S_i(C)$ and (2) $\models C \iff \exists i \in 1..n : \models S_i(C)$. If $R_1, \ldots, R_n$ are most accurate simplifications of $S_1(C), \ldots, S_n(C)$ then $\bigwedge_{i=1..n} R_i \circ S_i$ is a most accurate simplification of C.*

Sometimes it is necessary to prove that a given simplification is most accurate. This can be done as follows: First, note that the composition of a most accurate simplification and a simplification is itself most accurate: If S simplifies C, and S' is a most accurate simplification of $S(C)$, then $S' \circ S$ is a most accurate simplification of C. This implies that S is a most accurate simplification of C, if the identity substitution is a most accurate simplification of $S(C)$.

Lemma 3.6 *The identity substitution is a most accurate simplification of C, iff*

(a) $\forall \alpha, \beta \in \mathcal{V}(C) : \exists L \models C : L(\alpha) \neq L(\beta)$
(b) $\forall \alpha \in \mathcal{V}(C) : \exists L_1, L_2 \models C : root(L_1(\alpha)) \neq root(L_2(\alpha))$

Not only can simplifying substitutions be applied to a principal type computed by algorithm $\mathcal{C}$, they also provide a means to improve algorithm $\mathcal{C}$: Let *simplify* be an algorithm, which, given a constrained type $\tau|C$, either detects nonsatisfiability of C or returns a simplifying substitution S and a simplified constrained type $\tau'|C'$, such that $\tau' = S(\tau)$ and $C' \equiv S(C)$. Then algorithm $\mathcal{D}$ given in figure 3 computes a more compact representation of the principal type, if it exists. The main idea used in this algorithm is to simplify constraint sets as soon as new constraints are added. This strategy tends to decrease the number of generic variables and constraints that occur in type schemes of let-bound identifiers, avoiding duplicate work when these types are instantiated at variable usages. Thus, algorithm $\mathcal{D}$ will in most cases be faster then algorithm $\mathcal{C}$ followed by *simplify*.

It is not difficult to see that for the constraint based ML-system of section 2 algorithm $\mathcal{D}$ specializes to Milners original algorithm if $simplify(\tau|C) = (mgu(C), mgu(C)(\tau)|\emptyset)$. Correctness and completeness of $\mathcal{D}$ are derived from the following

Theorem 3.7 *Let $(S, \tau|C) = \mathcal{D}[\![M]\!]\Gamma$ and $\tau_0|C_0 = \mathcal{C}[\![M]\!]\Gamma$, then there exists a simplification S_0 of C_0 such that $\tau = S_0(\tau_0)$ and $C \equiv S_0(C_0)$. If D fails, then M is not typable.*

[4] Provided the equational theory is unitary unifying.

$\mathcal{D}[\![x]\!]\Gamma = (S, \tau|C)$
if there exists a substitution S, such that
$\tau_0|C_0 = inst(\Gamma(x))$
$\tau_1|C_1 = inst(R_{var}(\tau_0))$
$(S, \tau|C) = simplify(\tau_1|C_0 \cup C_1)$.

$\mathcal{D}[\![\lambda x.M]\!]\Gamma = (S_2 \circ S_1, \tau|C)$
if there exist substitutions S_1, S_2 and α "new", such that
$(S_1, \tau_1|C_1) = \mathcal{D}[\![M]\!](\Gamma + [x : \alpha])$
$\tau_2|C_2 = inst(R_{abs}(S_1\alpha, \tau_1))$
$(S_2, \tau|C) = simplify(\tau_2|C_1 \cup C_2)$.

$\mathcal{D}[\![M_1\ M_2]\!]\Gamma = (S_3 \circ S_2 \circ S_1, \tau|C)$
if there exist substitutions S_1, S_2, S_3, such that
$(S_1, \tau_1|C_1) = \mathcal{D}[\![M_1]\!]\Gamma$,
$(S_2, \tau_2|C_2) = \mathcal{D}[\![M_2]\!]S_1\Gamma$ and
$\tau_3|C_3 = inst(R_{app}(S_2\tau_1, \tau_2))$
$(S_3, \tau|C) = simplify(\tau_3|S_2C_1 \cup C_2 \cup C_3)$.

$\mathcal{D}[\![\textbf{let } x = M_1 \textbf{ in } M_2]\!]\Gamma = (S_3 \circ S_2 \circ S_1, \tau|C)$
if there exist substitutions S_1,S_2,S_3, such that
$(S_1, \tau_1|C_1) = \mathcal{D}[\![M_1]\!]\Gamma$ and
$(S_2, \tau_2|C_2) = \mathcal{D}[\![M_2]\!](\Gamma + [x : gen(\Gamma, \tau_1|C_1)])$
$(S_3, \tau|C) = simplify(\tau_2|S_2C_1 \cup C_2)$.

The algorithm fails in all other cases.

Figure 3: Algorithm $\mathcal{D}$

4 DECIDABLE PREDICATE SYSTEMS

In order to get a handle on constraint solving, we must choose a formalism for the specification of validity of predicates on ground terms. In this presentation, we will use constraint set rewrite rules, although a suitable restriction of the first order predicate calulus might serve as well. The choice of formalism is of subsidiary importance, as long as it enables us to isolate interesting classes of predicates for which satisfiability is decidable.

Let us assume that we are given a rewrite relation $\longrightarrow\!\!\!\!\!\bullet$, such that $L \models C \iff L(C) \longrightarrow\!\!\!\!\!\bullet^* \emptyset$. A natural requirement is that $\longrightarrow\!\!\!\!\!\bullet^*$, the reflexive and transitive closure of $\longrightarrow\!\!\!\!\!\bullet$, should be canonical, i.e. confluent and terminating. Entailment can then be defined as $C \Vdash D \iff \exists C' \subseteq C.C' \longleftrightarrow\!\!\!\!\!\bullet^* D$, with $\longleftrightarrow\!\!\!\!\!\bullet^*$ denoting the reflexive, transitive and symmetric closure of $\longrightarrow\!\!\!\!\!\bullet$. This definition implies that entailment and equivalence of constraint sets is decidable and that entailment is indeed substitution closed.

In the sequel, we assume that the rewrite relation is specified by a finite set R of rules $l \longrightarrow\!\!\!\!\!\bullet\ r$, where l and r are sets of constraints and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. The rewrite relation $\longrightarrow\!\!\!\!\!\bullet$ is then defined as the set of pairs $(C \cup D, C \cup S(r))$, such that $D = S(l)$ for some $l \longrightarrow\!\!\!\!\!\bullet\ r \in R$.

Figure 4 gives as an example a set of rewrite rules for subtyping and the overloaded operators mentioned in the introduction. Some of its consequences are: coercion of functions is anti monotonic in argument types, references can not be coerced, functions are not comparable, reference cells can be compared for equality, regardless of their contents and pairs can be ordered if the first component admits equality and both components can be ordered.

The simplest class of constraint set rewrite systems with solvable constraint problems are those corresponding to "traditional" overloading, where we are given a finite set of overloading instances in addition to syntactic term equality. The constraint solving procedure for this class of rewrite systems consists of computing the most general unifier $U(C)$ of all equational constraints in a given constraint set C and enumerating a set of minimal substitutions S such that $S(U(C))$ rewrites to $\emptyset$. By lemma 3.5, the most accurate simplification is then obtained as the composition of U and the join of all these substitutions.

In order to enumerate a set of minimal solutions, we define a transformation relation $\Longrightarrow_1$ on constraint problems, such that any normal form of C under $\Longrightarrow_1$ is either unsolvable or in solved form, which determines a solution of C.

Definition 4.1 *A constraint problem* $C = \{s_1 = t_1, \ldots, s_n = t_n\} \cup \{p_1(\overline{u}_1), \ldots, p_l(\overline{u}_l)\}$ *is in solved atomic form, if*

1. $s_i \in \mathcal{V}$ *for all* i
2. $s_i \neq s_j \wedge s_i \notin \mathcal{V}(t_j)$ *for all* $i \neq j$
3. $s_i \notin \mathcal{V}(u_{jk})$ *for all* i, j, k
4. $u_{jk} \in F_0 \cup \mathcal{V}$ *for all* j, k
5. $\overline{u}_i \cap \mathcal{V} \neq \emptyset$ *for all* i

It is in solved form, if $l = 0$.

Lemma 4.2 *If* C *is in atomic solved form, then* $\vec{C} = \{t_1/s_1, \ldots, t_n/s_n\}$ *is idempotent and* $\vec{C}(C)$ *has the same set of solutions as* $\{p_1(\overline{u}_1), \ldots, p_l(\overline{u}_l)\}$.

For predicates describing the possible instances of overloaded operators, we may assume that the left hand sides of the corresponding rewrite rules consist of a single constraint and that these constraints are pairwise nonunifiable. If all predicates are non-recursive, i.e. for no constraint $p(\overline{s_n})$ exists a set D containing $p(\overline{t_n})$, such that $p(\overline{s_n}) \longrightarrow\!\!\!\!\!\bullet^+ D$, then the transformation relation $\Longrightarrow_1$, given in figure 5, is terminating and yields a complete set of minimal solutions.

Definition 4.3 *A transfomation relation* $\Longrightarrow$ *on constraint problems is*

1. *sound, if* $C \Longrightarrow C' \Rightarrow \mathcal{L}(C') \subseteq \mathcal{L}(C)$,
2. *complete, if* C *not in solved form implies* $\mathcal{L}(C)_{|\mathcal{V}(C)} \subseteq \bigcup_{C \Longrightarrow C'} \mathcal{L}(C')_{|\mathcal{V}(C)}$.

Theorem 4.4 $\Longrightarrow_1$ *is sound and complete. If all predicates are non-recursive, then* $\Longrightarrow_1$ *is terminating. Moreover, any normal form under* $\Longrightarrow_1$ *is either unsolvable or in solved form.*

Proof: Delete, Compose and Eliminate are just the usual rules for syntactic unification. Rule Match is clearly sound, since any solution of $C \cup \{\overline{r_m}\} \cup \{\tau_1 = l_1, \ldots, \tau_n = l_n\}$ is a solution of $C \cup \{p(\overline{\tau_n})\}$ as well, provided $p(\overline{l_n}) \longrightarrow\!\!\!\!\!\bullet\ \overline{r_m}$. Completeness follows from the fact, that all solutions must enable rewriting via some rule.

Since all predicates are non-recursive, there is an upper bound to the number of Match-steps in any transformation sequence, regardless of the size of arguments of constraints in C. Let $B(C)$ denote this bound and take the triple $\langle B(C), U(C), S(C)\rangle$ as a complexity measure for constraint sets, where $U(C)$ is the number of variables which occur not only once on the left hand side of an equation $\alpha = \tau$, and $S(C)$ is the number of symbols of C. Termination then follows by well founded induction over the usual ordering of natural number triples.

The third part of the theorem follows by case analysis of the transformation rules. □

Adding subtyping or parametric overloading requires that predicates can be defined recursively, which implies that

$$
\begin{array}{lll}
int \lhd int \longrightarrow \emptyset & p_{=}(int) \longrightarrow \emptyset & p_{\leq}(int) \longrightarrow \emptyset \\
real \lhd real \longrightarrow \emptyset & p_{=}(real) \longrightarrow \emptyset & p_{\leq}(real) \longrightarrow \emptyset \\
int \lhd real \longrightarrow \emptyset & p_{=}(ref(a)) \longrightarrow \emptyset & p_{\leq}(list(a)) \longrightarrow p_{\leq}(a), p_{=}(a) \\
a \to b \lhd c \to d \longrightarrow c \lhd a, b \lhd d & p_{=}(list(a)) \longrightarrow p_{=}(a) & p_{\leq}(a \times b) \longrightarrow p_{\leq}(a), p_{=}(a), p_{\leq}(b) \\
a \times b \lhd c \times d \longrightarrow a \lhd c, b \lhd d & p_{=}(a \times b) \longrightarrow p_{=}(a), p_{=}(b) & \\
ref(a) \lhd ref(b) \longrightarrow a = b & p_{\in}(a, list(a)) \longrightarrow p_{=}(a) & p_{\downarrow}(list(a), int, a) \longrightarrow \emptyset \\
list(a) \lhd list(b) \longrightarrow a \lhd b & p_{\in}(a, set(a)) \longrightarrow p_{=}(a) & p_{\downarrow}(array(a, b), a, b) \longrightarrow p_{=}(a) \\
 & p_{\in}(b, array(a, b)) \longrightarrow p_{=}(b) & p_{\downarrow}(a \to b, a, b) \longrightarrow \emptyset
\end{array}
$$

Figure 4: A rewrite system for structural subtyping and overloaded operators

Delete	$C \cup \{\tau = \tau\}$	$\Longrightarrow_1$	C
Decompose	$C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}$	$\Longrightarrow_1$	$C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$
Eliminate	$C \cup \{\alpha = \tau\}$	$\Longrightarrow_1$	$\{\tau/\alpha\}(C) \cup \{\alpha = \tau\}$ if $\alpha \in \mathcal{V}(C) - \mathcal{V}(\tau),\ \tau \in \mathcal{V} \Rightarrow \tau \in \mathcal{V}(C)$
Match	$C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_1$	$C \cup \overline{\tau_m} \cup \{\tau_1 = l_1, \ldots, \tau_n = l_n\}$ if $p(\overline{l_n}) \longrightarrow \overline{\tau_m}$ is a rewrite rule away from $\mathcal{V}(C) \cup \mathcal{V}(\overline{\tau_n})$

Figure 5: Solving constraint problems

there may be infinitely many solutions to a given constraint set. In [15, 5] it was shown that for the special case of "structural" subtyping, one can find algorithms to transform any constraint set into an equivalent one containing only type variables and base types and that these "atomic" constraint sets have a solution, iff they have a solution substituting base types for the remaining type variables.

Two properties of structural subtyping are essential for the derivation of this result: that it is *inductively defined* and that it enforces *structural equivalence* on its arguments, where two types are *structurally equivalent* ($\tau \overset{se}{\sim} \tau'$) if they are equal under identification of zero-ary constructor symbols and type variables.

Definition 4.5 *A predicate $p \in P_n$ is inductively defined, if for all $(f_1, \ldots, f_n) \in F_+^n$, either p is not satisfiable for argument vectors of the form $f_1(\overline{t_1}), \ldots, f_n(\overline{t_n})$, or there exists an index set $I(p, f_1, \ldots, f_n)$, such that for all $\overline{t_1}, \ldots, \overline{t_n} \in T_F^m$:*

$$p(f_1(\overline{t_1}), \ldots, f_n(\overline{t_n})) \longrightarrow \bigcup_{i \in I(p, f_1, \ldots, f_n)} p_i(\overline{s_i})$$

where $s_{ij} \in \{t_{11}, \ldots, t_{nm}\}$. We say that p enforces structural equivalence, if for all $\overline{t_n} \in T_F^n$

$$p(t_1, \ldots, t_n) \longrightarrow^* \emptyset \Rightarrow t_1 \overset{se}{\sim} t_2 \overset{se}{\sim} \cdots \overset{se}{\sim} t_n \ .$$

The requirement that p enforces structural equivalence can be slightly generalized to *structural similarity*, where two terms are structurally similar ($\tau \overset{ss}{\sim} \tau'$) if they have the same tree skeleton. This enables the definition of subtyping rules between structured types, such as

$$list(a) \lhd set(b) \longrightarrow a \lhd b, p_{=}(b) \ .$$

which permits the coercion of lists of type a to sets of type b, provided a is a subtype of b and b admits equality.

Note that structural similarity is itself an inductively defined structural similarity enforcing predicate, whereas structural equivalence and syntactic equality enforce structural equivalence.

Theorem 4.6 *Let C be a constraint set over structural similarity enforcing, inductively defined predicates. Then there exists a computable, complete set of minimal structural similarity enforcing substitutions $\mu SS(C)$, such that*

- $\forall S \in \mu SS(C)$: $p(\tau_1, \ldots, \tau_n) \in S(C) \Rightarrow \tau_1 \overset{ss}{\sim} \tau_2 \overset{ss}{\sim} \cdots \overset{ss}{\sim} \tau_n$
- $L \models C \Rightarrow \exists S \in \mu SS(C) : L = L' \circ S$

If all predicates enforce structural equivalence instead, then $\mu SS(C)$ is either empty or consists of a simplifying substitution.

Solvable constraint sets satisfying $p(\overline{\tau}) \in C \Rightarrow \tau_i \overset{ss}{\sim} \tau_j$ can be easily transformed into equivalent atomic constraint sets computing the normal form under $\longrightarrow$. If the normal form contains non-atomic constraints, then C is unsolvable.

Corrollary 4.7 *If P is a system of inductively defined structural equivalence enforcing predicates, then every typable expression M has a principal type with an atomic constraint set component.*

Computing $\mu SS(C)$ followed by rewriting is of course rather inefficient, since many substitutions in $\mu SS(C)$ will lead to unsatisfiable constraint sets. Fortunately, it is possible to interleave the computation of structural similarity enforcing substitutions with rewriting.

Figure 6 presents a transformation relation $\Longrightarrow_2$ for simultanous computation of $\mu SS(C)$ and normal forms under $\longrightarrow$. The definition makes use of the notion of a template for a given type expression τ, which is obtained by replacing each occurence of a type variable or type constant $f \in F_0$ in τ by a fresh type variable and renaming occurences of constructors $f \in F_+$ arbitrarily. Note that rule Match of $\Longrightarrow_1$ has been split into two new rules: Match and Rewrite.

Lemma 4.8 *$\Longrightarrow_2$ is sound and complete. Moreover, if C is a normal form under $\Longrightarrow_2$, then C is either in solved atomic form, or unsolvable.*

Delete	$C \cup \{\tau = \tau\}$	$\Longrightarrow_2$	C
Decompose	$C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}$	$\Longrightarrow_2$	$C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$
Eliminate	$C \cup \{\alpha = \tau\}$	$\Longrightarrow_2$	$\{\tau/\alpha\}(C) \cup \{\alpha = \tau\}$ if $\alpha \in \mathcal{V}(C) - \mathcal{V}(\tau),\ \tau \in \mathcal{V} \Rightarrow \tau \in \mathcal{V}(C)$
Rewrite	$C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\}$	$\Longrightarrow_2$	$C \cup S(\overline{\tau_k})$ if $p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow \overline{\tau_k} \in R$ and $S = \{\tau_{11}/\alpha_{11}, \ldots, \tau_{nm}/\alpha_{nm}\}$
Match	$C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_2$	$\{\tau'/\alpha\}(C \cup \{p(\overline{\tau_n})\}) \cup \{\alpha = \tau'\}$ if $\alpha, \tau \in args(p(\overline{\tau_n}))$, $root(\tau) \in F_+$ where τ' is a template for τ away from $\mathcal{V}(C) \cup \mathcal{V}(\overline{\tau_n})$

Figure 6: Transformation into atomic constraint sets

Proof: Soundness is easily established. For completeness note that rules Rewrite and Match are mutually disjoint and together capture all possible rewriting path's. □

However, $\Longrightarrow_2$ is not terminating, as the following simple example shows. The problem $\alpha \lhd l(\beta),\ \beta \lhd \alpha$ inevitably leads to an infinite transformation sequence, like

$$\begin{aligned}
&\alpha \lhd l(\beta),\ \beta \lhd \alpha \\
&\overset{\text{Match}}{\Longrightarrow} \quad l(\alpha_1) \lhd l(\beta),\ \beta \lhd l(\alpha_1),\ \alpha = l(\alpha_1) \\
&\overset{\text{Rewrite}}{\Longrightarrow} \quad \alpha_1 \lhd \beta,\ \beta \lhd l(\alpha_1),\ \alpha = l(\alpha_1) \\
&\overset{\text{Match}}{\Longrightarrow} \quad \alpha_1 \lhd l(\beta_1),\ l(\beta_1) \lhd l(\alpha_1),\ \alpha = l(\alpha_1),\ \beta = l(\beta_1) \\
&\overset{\text{Rewrite}}{\Longrightarrow} \quad \alpha_1 \lhd l(\beta_1),\ \beta_1 \lhd \alpha_1,\ \alpha = l(\alpha_1),\ \beta = l(\beta_1) \\
&\quad \vdots
\end{aligned}$$

The above problem is unsolvable, since any solution L must satisfy $L(\alpha) \overset{ss}{\sim} L(\beta)$ as well as $L(\alpha) \overset{ss}{\sim} L(l(\beta))$, and by transitivity of $\overset{ss}{\sim}$: $L(\beta) \overset{ss}{\sim} L(l(\beta))$, which is impossible for finite terms.

In order to detect unsolvable constraint problems, one needs to add an equivalent of the normal occur check to the transformation rules.[5] Every constraint problem C over structural similarity enforcing predicates induces an equivalence relation $\overset{C}{\sim}$ on $\mathcal{V}(C)$, defined as the reflexive, transitive and symmetric closure of the relation

$$\begin{aligned}
EQ(C) \quad &= \quad \bigcup\{sim(\tau, \tau') \mid \tau = \tau' \in C\} \\
&\cup \quad \bigcup\{sim(\tau_i, \tau_{i+1}) \mid p(\overline{\tau_n}) \in C,\ i \in 1..n-1\}
\end{aligned}$$

where

$$sim(\tau, \tau') = \begin{cases} \{(\tau, \tau')\} & \text{if } \tau, \tau' \in \mathcal{V} \\ \bigcup_{i=1}^{n} sim(\rho_i, \rho'_i) & \text{if } \tau = f(\overline{\rho_n}), \tau' = g(\overline{\rho'_n}) \\ \emptyset & \text{otherwise} \end{cases}$$

It is easy to see that $S \models C \Rightarrow S(\alpha) \overset{ss}{\sim} S(\beta)$ for all $\alpha \overset{C}{\sim} \beta$. Moreover, we have

Lemma 4.9 *Let $\equiv$ be an equivalence relation such that*

$$\forall \alpha, \beta \in \mathcal{V}(C) : \alpha \equiv \beta \Rightarrow (L \models C \Rightarrow L(\alpha) \overset{ss}{\sim} L(\beta)) \ . \quad (*)$$

Let $[\alpha]_\equiv = \{\beta \mid \alpha \equiv \beta\}$ and $[\tau]^\equiv = \{[\alpha]_\equiv \mid \alpha \in \mathcal{V}(\tau)\}$. If C contains a constraint $p(\overline{\tau_n})$ or $\tau_1 = \tau_2$, such that $[\alpha]_\equiv \in [\tau]^\equiv$ for some $\alpha, \tau \in \{\tau_1, \ldots, \tau_n\}$, then C has no solution.

[5] This has already been observed in previous work on subtype inference. Our presentation differs slightly from [15, 5], due to the generalization to arbitrary predicates.

We are now ready to introduce $\Longrightarrow_3$ (see figure 7). The principal difference between $\Longrightarrow_2$ and $\Longrightarrow_3$, is that we always replace complete structural similarity classes, as indicated by the equivalence class component E[6], instead of a single variable. This is necessary to ensure termination. For simplicity we have split Rule Eliminate in two parts: Eliminate$_1$ replaces variable α by β if both appear in the constraint set C. Eliminate$_2$ leads to a replacement of all variables in the equivalence class of α by a fresh template structurally similar to τ. This turns all variables in the class into solved variables. We let $\mathcal{SS}(\tau, \{\alpha_1, \ldots, \alpha_n\}, W)$ denote a complete set of minimal substitutions away from W, such that for each $S \in \mathcal{SS}(\tau, \{\alpha_1, \ldots, \alpha_n\}, W)$, $S(\alpha_i)$ is a template for τ and $\mathcal{V}(S(\alpha_i)) \neq \mathcal{V}(S(\alpha_j))$ for all $i \neq j \in 1..n$. We have also introduced a special constraint problem $\top$ for the class of all unsolvable constraint problems, in order to speed up the detection of blind alleys in transformation sequences.

Theorem 4.10 *$\Longrightarrow_3$ is sound, complete and terminating if the equivalence relation E satisfies equation *. A normal form under $\Longrightarrow_3$ is either $\top$ or in solved atomic form.*

Proof: Note that if E satisfies equation * and $E, C \Longrightarrow_3 E', C'$, then E' satisfies equation * as well. Soundness is inherited from $\Longrightarrow_2$, since each transformation sequence in $\Longrightarrow_3$ not involving Clash or Cycle rules can be mapped into $\Longrightarrow_2$ by ignoring the equivalence relation component and expanding Eliminate$_2$ and Match rules into sequences of Eliminate and Match rules of $\Longrightarrow_2$. Completeness is based on lemma 4.9. Termination follows by well founded induction over the triple $\langle |E|, U(C), S(C)\rangle$, where $|E|$ is the number of equivalence classes of E, $U(C)$ and $S(C)$ are defined as for $\Longrightarrow_1$. □

Transformation relations $\Longrightarrow_1$ and $\Longrightarrow_3$ can be combined, if the predicate symbols can be separated into two disjoint sets: P_N for non-recursively defined predicates, and P_{SI} for structural similarity enforcing inductively defined predicates. Rewrite rules for predicates in P_N may use predicates in P_{SI} on their right hand side, but not vice versa. Typical examples for such predicates are $p_\in$ and $p_\downarrow$ of figure 4.

Let $\Longrightarrow_{ns}$ be the set of rewrite rules obtained by adding

[6] We use the following notation for equivalence relation operations: E_α is E with the equivalence class of α removed, $\{\beta/\alpha\}E$ is E with all occurrences of α renamed to β, and $E + R$ is the reflexive, transitive and symmetric closure of $E \cup R$.

Delete	$E, C \cup \{\tau = \tau\}$	$\Longrightarrow_3$	E, C
Decompose	$E, C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}$	$\Longrightarrow_3$	$E, C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$
Rewrite	$E, C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\}$	$\Longrightarrow_3$	$E, C \cup S(\overline{r_k})$ if $p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow \overline{r_k} \in R$ and $S = \{\tau_{11}/\alpha_{11}, \ldots, \tau_{nm}/\alpha_{nm}\}$
Eliminate$_1$	$E, C \cup \{\alpha = \beta\}$	$\Longrightarrow_3$	$\{\beta/\alpha\}(E + \{(\alpha, \beta)\}), \{\beta/\alpha\}(C) \cup \{\alpha = \beta\}$ if $\alpha, \beta \in \mathcal{V}(C)$, $\alpha \neq \beta$
Eliminate$_2$	$E, C \cup \{\alpha = \tau\}$	$\Longrightarrow_3$	$E', S \cup (\vec{S} \circ \{\tau/\alpha\})(C) \cup \{\alpha = \tau\}$ if $\alpha \in \mathcal{V}(C)$, $\tau \notin \mathcal{V}$, $[\alpha]_E \notin [\tau]^E$ and $\vec{S} \in \mathcal{SS}(\tau, [\alpha]_E - \{\alpha\}, \mathcal{V}(C \cup \{\alpha = \tau\}))$ where $E' = E_\alpha + EQ(\{\tau = \vec{S}(\beta) \mid \beta \in [\alpha]_E\})$
Match	$E, C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_3$	$E', S \cup \vec{S}(C \cup \{p(\overline{\tau_n})\})$ if $\alpha, \tau \in args(p(\overline{\tau_n}))$, $root(\tau) \in F_+$, $[\alpha]_E \notin [\tau]^E$ and $\vec{S} \in \mathcal{SS}(\tau, [\alpha]_E, \mathcal{V}(C \cup \{p(\overline{\tau_n})\}))$ where $E' = E_\alpha + EQ(\{\tau = \vec{S}(\beta) \mid \beta \in [\alpha]_E\})$
Clash$_1$	$E, C \cup \{f(\overline{\tau_n}) = g(\overline{\tau'_n})\}$	$\Longrightarrow_3$	$E, \top$
Clash$_2$	$E, C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\}$	$\Longrightarrow_3$	$E, \top$ if $\nexists\, p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow \overline{r_m} \in R$
Cycle$_1$	$E, C \cup \{\alpha = \tau\}$	$\Longrightarrow_3$	$E, \top$ if $\tau \notin \mathcal{V}$, $[\alpha]_E \in [\tau]^E$
Cycle$_2$	$E, C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_3$	$E, \top$ if $\exists \alpha, \tau \in args(p(\overline{\tau_n}))$, $\tau \notin \mathcal{V}$, $[\alpha]_E \in [\tau]^E$

Figure 7: Terminating transformation into atomic constraint sets

the rule

$$\text{Match}_n \quad E, C \cup \{p(\overline{\tau_n})\} \Longrightarrow_{ns} E, C \cup \overline{\tau_m} \cup \{\tau_1 = l_1, \ldots, \tau_n = l_n\} \text{ if } p \in P_N,\ p(\overline{l_n}) \longrightarrow \overline{r_m} \text{ is a rewrite rule away from } \mathcal{V}(C) \cup \mathcal{V}(\overline{\tau_n})$$

and restricting the use of $\Longrightarrow_3$-rules to predicates in P_{SI}.

Theorem 4.11 *$\Longrightarrow_{ns}$ is sound, complete and terminating. Normal forms under $\Longrightarrow_{ns}$ are either $\top$ or solved atomic forms.*

Proof: Soundness and completeness are trivial. Termination is established by the complexity measure $\langle B(C), |E|, U(C), S(C)\rangle$. The second part is obvious. □

It remains to be shown that it is decidable whether an atomic constraint set can be solved. First, if C contains unsatisfiable ground constraints, then C has no solution. Solvable variable free constraints can be removed from C without changing the set of possible solutions. Second, note that the equivalence relation $\overset{C}{\sim}$ induces a partition of C into disjoint subsets. We write

$$C = \biguplus_{i=1..n}^{\overset{C}{\sim}} C_i$$

if $C = C_1 \uplus \cdots \uplus C_n$, where $n = |\{[\alpha]_{\overset{C}{\sim}} \mid \alpha \in \mathcal{V}(C)\}|$ and $\mathcal{V}(C_i) = [\alpha]_{\overset{C}{\sim}}$ for some $\alpha \in \mathcal{V}(C)$. Each of the C_i can be solved independently and the solutions can be composed to yield a solution for C. This leaves us with two sorts of constraint sets: those that contain base types and those that don't. Due to structural similarity, the former kind has a solution iff there is a base type solution $L \in \mathcal{V} \to F_0$. The latter kind poses a problem: it may have a solution but not a base type one![7]

As an example for this phenomenon, consider structural subtyping enriched with predicates for two overloaded operators p and q, such that $p(a \times b) \longrightarrow q(a), q(b)$ and $q(int) \longrightarrow \emptyset$. Although the constraint problem $\{\alpha \lhd \beta,\ \beta \lhd \gamma, q(\gamma)\}$ has the solution $\{int \times int/\alpha,\ int \times int/\beta,\ int \times int/\gamma\}$, it has no base type solution in this system.

Due to structural similarity and the inductive nature of predicates, any solution can be described as a composition of *partial solutions.*

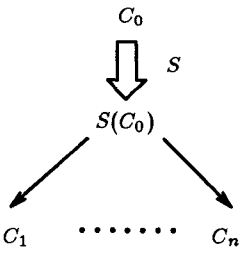
Definition 4.12 *A substitution S is a partial solution of C_0 with root constructors $f_1, \ldots, f_k \in F_n$, iff*

1. $dom(S) = \mathcal{V}(C_0) = \{\alpha_1, \ldots, \alpha_k\}$
2. $\forall \alpha_i : \exists \beta^i_1, \ldots, \beta^i_n : S(\alpha_i) = f_i(\beta^i_1, \ldots, \beta^i_n)$,
3. $\forall p(\alpha_{i_1}, \ldots, \alpha_{i_k}) \in C_0 : p(f_{i_1}(\overline{\gamma}_1), \ldots, f_{i_k}(\overline{\gamma}_k)) \longrightarrow \overline{r_m} \in R$

If S is a partial solution for C_0, then $S(C_0)$ can be rewritten as $\biguplus_{i=1}^{\overset{S}{\sim}} {}_{|} C_i$. Moreover, if there exist solutions $L_1, \ldots, L_l$ for $C_1, \ldots, C_l$ then $S \circ L_1 \circ \cdots \circ L_l$ is a solution for C_0.

Partial solutions can be conveniently drawn as trees consisting of constraint sets and two sorts of branches: substitutions and rewrites. The general scheme is

[7] This distinguishes our system from a pure structural subtype system.



and the tree in figure 8 describes a solution of the example above.

A careful look at this example shows, that with the exception of leaf nodes and those obtained by substitution application, all constraint sets involve exactly the same number of variables! This property can be captured in the following definition:

Definition 4.13 *A system of inductively defined predicates is called decomposable iff satisfiability can expressed with left linear rewrite rules of the form*

$$p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow \overline{r_k}$$

such that the set of equivalence classes of the equivalence relation $\overline{r_k}$ *is* $\{ \{\alpha_{1i}, \ldots, \alpha_{ni}\} \mid i = 1..m \}$, *where* $f_1, \ldots, f_n \in F_m$.

Note that decomposability, in contrast to structural similarity, is a purely syntactic criterion and thus can be enforced by the typechecker, e.g. if user defined overloading and subtyping are allowed.

Theorem 4.14 *Let C be an atomic constraint set, the variables of which form a single $\overset{C}{\sim}$-equivalence class. It is decidable whether C can be solved.*

Proof: Decomposability implies, that the number of constraint sets which can actually occur in a solution tree is finite (modulo variable renamings): any application of a partial solution with root constructors $f_i \in F_m$ to a constraint set C with n variables forming a single $\overset{C}{\sim}$-equivalence class, will lead to m new constraint sets with n variables each. Thus any solution tree for C can be pruned to reduce its height to be less than the number $h(n)$ of constraint sets over n variables. □

It is also decidable whether C has an infinite number of solutions, only base type solutions or whether some substitution is a most accurate simplification of a given constraint set.

Theorem 4.15 *Given a solvable constraint set over decomposable predicates, a most accurate simplification can be effectively computed.*

Proof: By lemma 3.5, lemma 3.6 and the preceding discussion, it suffices to show that most accurate simplifications can be computed for atomic constraint sets C, such that $\mathcal{V}(C)$ forms a single $\overset{C}{\sim}$-equivalence class. Therefore, let C be atomic. If C contains a base type, it has only a finite set of solutions, the join of which is a most accurate simplification for C. If C consists of variable constraints only, we need to check for condition (a) and (b) of lemma 3.6.

If condition (a) is violated, i.e. there exist two variables α and β, such that $L \models C \Rightarrow L(\alpha) = L(\beta)$, then $\{\alpha/\beta\}$ is a simplification for C. This simplification leads to a new constraint set with strictly fewer variables, and thus can be applied at most n times. But how do we check for (a)?

Consider a solution tree for $C = C_1^0$, such that $L(\alpha) \neq L(\beta)$. W.l.o.g. we assume variables in distinct constraint set nodes to be distinct. There must be a shortest path $w = j_1, \ldots, j_k$, for which $root(L(\alpha)/w) \neq root(L(\beta)/w)$. There exist substitutions $S_1, \ldots, S_k$, constraint sets C_j^i for $i \in 1..k$, $j \in 1..n_i$ and solutions $R_j^i \models C_j^i$ for $i \in 1..k$, $j \in 1..n_i$, $j \neq j_i \vee i = k$ such that $L = S_1 \circ \cdots \circ S_k \circ \bigcup_{i, j \neq j_i \vee i=k} R_j^i$ and $S_i(C_{j_{i-1}}^{i-1}) \longrightarrow^* C_1^i \uplus \cdots \uplus C_{n_i}^i$ (see figure 9). The solution trees for C_j^i, $j \neq j_i \vee i = k$ can be pruned to height $h(n)$ and the path from C_1^1 to $C_{j_k}^j$ can be shortened to length $h(n)$ as well. Thus the total height can be constrained to $2h(n)$ which enables us to restrict our search for a violation of (a) to a finite search space.

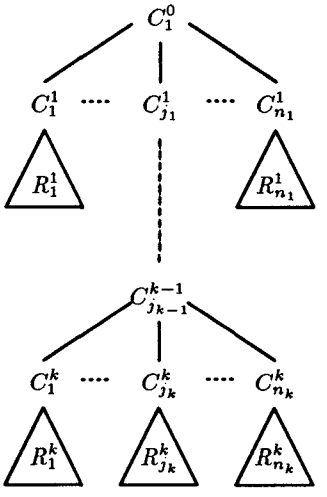


Figure 9: A solution tree violating (b)

Condition (b) is checked for in the following way: For each variable $\alpha \in \mathcal{V}(C)$ and constructor $f \in F_m$ we apply a substitution replacing α by $f(\overline{\beta_m})$, where $\overline{\beta_m}$ are new type variables, and check for solvability of the resulting constraint sets. This enables us to determine a substitution $S = \{f_{i_1}(\overline{\beta_{m_1}})/\alpha_1, \ldots, f_{i_n}(\overline{\beta_{m_1}})/\alpha_n\}$, such that $L \models C \Rightarrow root(L(\alpha_k)) = f_{i_k}$. S is obviously a simplification of C. If S is the identity, then C cannot be simplified further. If the f_i are base types ($m = 0$), then C has only base type solutions and S is already a most accurate simplification of C. Otherwise, if $m \geq 1$, we apply S to C, compute the normal forms of $S(C)$ under $\Longrightarrow_3$ and recursively apply the procedure to the solvable subset $C_1, \ldots, C_l$ of the resulting constraint sets. This gives us most accurate simplifications $A_1, \ldots, A_l$ of $C_1, \ldots, C_l$ and by lemma 3.5, $\bigwedge_{i=1..l} A_i \circ S$ is a most accurate simplification of C.

The whole process must terminate, since solvability of C puts an upper bound on the size of $S(\alpha)$ for any simplification S of C: Let $H(C)$ be the smallest number m_0, such that there exists a solution $L \models C$ satisfying $height(L(\alpha)) \leq m_0$ for all $\alpha \in \mathcal{V}(C)$. If $H(C) = 1$, C must have a base type solution, which implies that the substitution S above, will either be the identity or C has only base type solutions. If $H(C) > 1$, all solvable normal forms C' of $S(C)$ under $\Longrightarrow_3$ satisfy $H(C') \leq H(C) - 1$. □

The complexity of the computation of both solutions and most accurate simplifications of atomic constraint sets seems

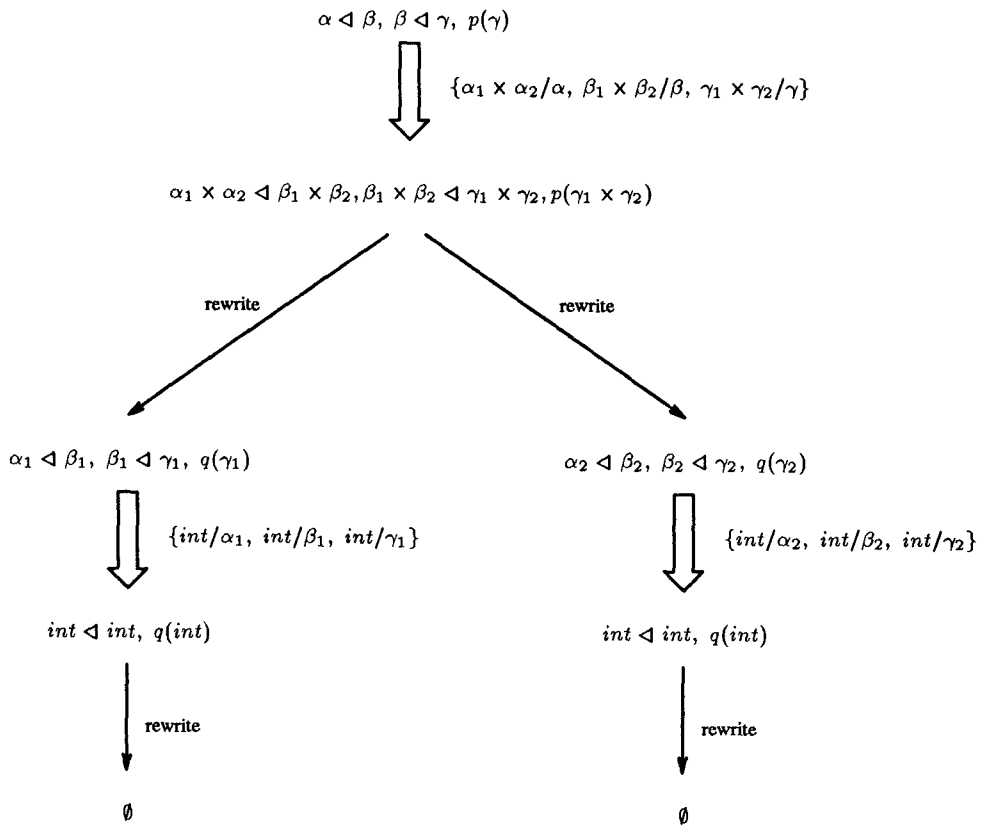


Figure 8: A solution tree for $\{\alpha \lhd \beta,\ \beta \lhd \gamma, q(\gamma)\}$

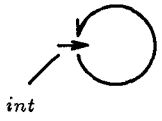
to be prohibitive. However, note that in most cases constraint sets will have base type solutions, which cuts down the search space enormously. Moreover, for special predicate systems, like e.g. parametric overloading or structural subtyping, more efficient algorithms are possible.

5 REGULAR TYPES

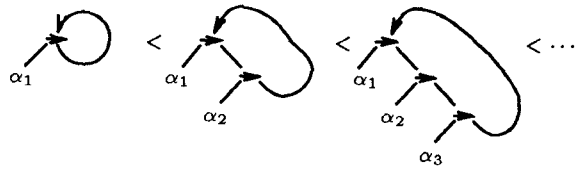
Recursive types arise naturally as solutions of unification problems having no finite solution, for example $\alpha = \tau(\alpha)$ where τ contains α as a proper subterm. If we allow infinite terms then $\{\tau(\tau(\tau(\ldots)))/\alpha\}$ is a unique most general unifier for this problem. There are at least two reasons why we would like to be able to deal with regular types: First, regular types allow the typing of self appplication and therefore enable us do define various fixpoint combinators directly, without resort to special recursive language constructs. Second, recursive types provide direct means to define recursive data structures such as the disjoint union type $tree\ \alpha = \alpha \oplus (tree\ \alpha \times tree\ \alpha)$, without the otherwise necessary introduction of a new type constructor *tree*.

The set of regular trees forms a complete metric space, such that boolean functions on finite trees can be uniquely extended to regular trees (see e.g. [2]), thus the semantics of decomposable predicates on regular trees is completely defined. Since unification and matching of regular trees are decidable too, it is natural to ask whether our constraint solving algorithm can be adapted as well. Unfortunately, the results of the previous sections seem to have no straightforward extension to recursive types. To see why, note that the crucial idea behind the constraint solving algorithm for finite types, is that the set of types structurally equivalent resp. similar to a given term t, can be represented by a single type expression resp. a finite set of type expressions. However, this is no longer true for regular trees!

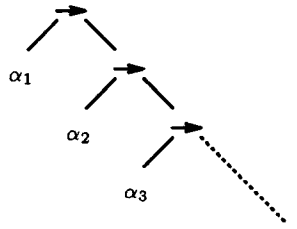
Consider the constraint problem $\alpha \lhd t$, where t is the regular tree



In order to reduce this problem to an equivalent atomic one, we need to replace α by a representation of all trees structurally eqivalent to t. This set contains the strictly increasing tree sequence



which converges to the infinite, non-regular tree t_∞:



Thus any complete constraint solving algorithm would have to deal with matching and unification of non-regular infinite trees, for which no solution is currently known. Moreover, since the reduction to atomic constraint sets is the first step in proving typability, it is far from clear, whether typability is decidable for decomposable predicates and regular trees.

On the other hand, if we don't require completeness, then we can simply select some approximation of t_∞, say the first element of the chain above, replace α with it and obtain, due to contravariance, the atomic constraint set $\alpha \lhd \alpha_1$, $\alpha_1 \lhd \alpha$. This approach, combined with a suitable adaption of constraint set rewriting, leads to a sound constraint solving heuristic.

Lemma 5.1 *Let $t_1 \overset{se}{\sim} \cdots \overset{se}{\sim} t_n$ be infinite trees and p be a decomposable predicate. The constraint $p(t_1, \ldots, t_n)$ is equivalent to an infinite conjunct of atomic constraints $\bigwedge_{i \in I} p_i(a_{i_1}, \ldots, a_{i_{n_i}})$, where the a_i occur as leaves of $t_1, \ldots, t_n$. If the t_i are regular, I is finite.*

Proof: The first part is obvious. The finiteness of I for regular trees follows from the finiteness of P and the fact that a regular tree has only finitely many subtrees. □

Regular trees can be obtained as unique solutions of *extended regular systems*, i.e. sets of equations $E = \langle x_1 = t_1, \ldots, x_n = t_n \rangle$, such that $x_i \neq x_j$ for $i \neq j$, and $t_i \notin \mathcal{V}$ or x_i occurs only once in E. An alternative method for denoting regular trees are rational tree expressions of the form $\mu\alpha.\tau$. In this case the tree is defined as the unique least fixed point of the expansion $\tau[\mu\alpha.\tau/\alpha]$. Yet a third characterization is given by cyclic graphs of nodes labelled with constructor and variable names. In the following presentation of our constraint solving algorithm we will be somewhat informal and use the usual term notation for matching and tree formation to apply to regular trees.

The following two rules specify an abstract algorithm for transformation of matching constraint sets into equivalent atomic ones.

$$H, C \cup \{c\} \Longrightarrow_4 H, C \qquad \text{if } c \in H$$

$$H, C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\} \Longrightarrow_4 H \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\}, C \cup S(\overline{\tau_k})$$
$$\text{if } p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow \overline{\tau_k} \in R$$
$$S = \{\tau_{11}/\alpha_{11}, \ldots, \tau_{nm}/\alpha_{nm}\}$$
$$\text{and } p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n)) \notin H$$

Lemma 5.2 *$\Longrightarrow_4$ is a terminating transformation relation. If $\emptyset, C \Longrightarrow_4 H, C'$, and H, C' is a normal form under $\Longrightarrow_4^*$, then C' is an atomic constraint set equivalent to C or C is unsolvable.*

The complete transformation system is obtained by dropping the Cycle rules and changing rules Decompose, Eliminate$_2$ and Match. Rule Decompose is now

$$H, E, C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\} \Longrightarrow_4 H \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}, E, C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$$
$$\text{if } f(\overline{\tau_n}) = f(\overline{\tau'_n}) \notin H$$

in order to prevent infinite looping. Rule Eliminate$_2$ is changed into

$$H, E, C \cup \{\alpha = \tau\} \Longrightarrow_4 \vec{S}H, E', S \cup \vec{S}(C \cup \{\alpha = \tau\})$$
$$\text{if } \alpha \in \mathcal{V}(C),\ \tau \notin \mathcal{V},$$
$$\text{and } \vec{S} \in \mathcal{SS}(\mu[\alpha]_E.\tau, [\alpha]_E - \{\alpha\}, \mathcal{V}(C \cup \{\alpha = \tau\}))$$
$$\text{where } E' = E_\alpha + EQ(\{\mu[\alpha]_E.\tau = \vec{S}(\beta) \mid \beta \in [\alpha]_E - \{\alpha\}\})$$

and rule Match is replaced by

$$H, E, C \cup \{p(\overline{\tau_n})\} \Longrightarrow_4 \vec{S}H, E', S \cup \vec{S}(C \cup \{p(\overline{\tau_n})\})$$
$$\text{if } \alpha, \tau \in args(p(\overline{\tau_n})),\ root(\tau) \in F_+$$
$$\text{and } \vec{S} \in \mathcal{SS}(\mu[\alpha]_E.\tau, [\alpha]_E, \mathcal{V}(C \cup \{p(\overline{\tau_n})\}))$$
$$\text{where } E' = E_\alpha + EQ(\{\mu[\alpha]_E.\tau = \vec{S}(\beta) \mid \beta \in [\alpha]_E\})$$

We use $\mu S.\tau$ as a shorthand for $\mu\alpha_1.\cdots.\mu\alpha_n.\tau$, where $S = \{\alpha_1, \ldots, \alpha_n\}$ and $\mu\{\}.\tau$ is equivalent to τ. Note that $\mu\alpha.\tau = \tau$ if α is not free in τ. Due to space limitations, we have omitted the algorithms for generating similarity enforcing substitutions and computing $EQ(C)$. Both are simple extensions of the finite case, augmented by appropriate termination conditions.

A small example should help to clarify the interpretation of the rules. Consider the constraint problem $\alpha = f(\beta, \gamma), \beta \lhd \alpha, \delta \lhd \alpha$, which has no finite solution. Suppose that α, β and δ already form an equivalence class in E. An application of the modified rule Eliminate$_2$ will then replace each of α, β and δ by a fresh template structurally similar to the regular tree denoted by $\mu[\alpha]_E.f(\beta, \gamma)$. Since

$$\mu[\alpha]_E.f(\beta, \gamma) = \mu\alpha.\mu\beta.\mu\delta.f(\beta, \gamma) = \mu\beta.f(\beta, \gamma)$$

one possible similarity enforcing substitution is

$$\{\mu\beta.f(\beta, \gamma_1)/\alpha,\ \mu\beta.f(\beta, \gamma_2)/\beta,\ \mu\beta.f(\beta, \gamma_3)/\delta\}$$

The transformational system for decomposable predicates can be efficiently implemented if we restrict ourselves to structural similarity enforcing predicates. In this case, we can use a term representation based on cyclic graphs, which is destructively updated during the transformational process.

6 RELATED WORK

Constrained types have been implicitly used in a number of investigations about extensions of parametric polymorphism. The algorithm to compute structural similarity enforcing substitutions can be seen as an extension of similar algorithms for structural subtyping in [15, 5].

Our approach to overloading is similar in spirit to Haskell, which allows the grouping of related operators into type classes. Type classes can be arranged to form arbitrary non circular subclass hierarchies. This allows the definition of default implementations of overloaded operators based on other operators in the same class or any of its superclasses.

The complexity of type systems for Haskell-style overloading has been investigated in [22], where it was shown to be undecidable for arbitrary recursively defined predicates and NEXPTIME-hard when restricted to parametric overloading.

Semantic foundations of subtyping in the presence of recursive types for structural subtyping extended with a smallest and a largest type can be found in [1]. However, the authors do not treat the question "$\exists S : S(\tau) \lhd S(\tau')$?" but rather the simpler one "$t \lhd t'$?" for $\mathcal{V}(t) = \mathcal{V}(t') = \emptyset$.

7 CONCLUSIONS

We have sketched the first type system incorporating parametric polymorphism, overloading, implicit coercions and recursive types. A typechecker based on this system has been implemented as part of an interactive programming environment [8] for the functional programming language SAMPλE [10]. The current version of the typechecker handles structural subtyping, parametric overloading and recursive types.

Our constraint solving algorithm for the class of decomposable predicates significantly extends previous solutions for subtyping and overloading. Although this algorithm is not complete for recursive types, we have not found this to be of practical importance. The main reason for this is the fact that most programs can be typed either without recursive types or whithout coercions. Our algorithm is complete for this case. Moreover, the prorammer can always guide the inference process using type annotations at the appropriate program subexpressions. Nevertheless, it is important to know whether constraint solving with decomposable predicates of arity ≥ 2 is decidable for recursive types.

Another area which needs further investigation is the representation of typings, which are rather complicated in general. In [6] Fuh and Mishra have presented a simplifaction method for structural subtyping. Their method is based on two observations: In most cases, coercion sets for user defined functions computed by the typechecker contain redundant constraints which can be removed without affecting the set of possible types. Moreover, some of the coercion constraints can be moved from function definitions to function usages. We have adapted their methods to the case of predefined parametric overloading and implemented in our typechecker. They are rather effective and often result in empty coercion sets.

8 ACKNOWLEDGEMENTS

The author would like to thank his colleagues Gregor Snelting and Michael Gloger for many stimulating discussions on type inference and language design. The presentation of section 4 has greatly benefited from reading Tobias Nipkow's course notes on equational reasoning.

REFERENCES

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 104–118, Orlando, Florida, January 1991.

[2] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[3] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[4] *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523 of *Lecture Notes in Computer Science*, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[5] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In Ganzinger [7], pages 94–114.

[6] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *TAPSOFT'89 — Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183, Barcelona, March 1989. Springer-Verlag.

[7] Harald Ganzinger, editor. *ESOP'88, 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, Nancy, France, March 1988. Springer-Verlag.

[8] Michael Gloger, Stefan Kaes, and Christoph Thies. Entwicklung funktionaler Programme in der SAMPλE-Programmierumgebung. Technical Report PI-R3/90, TH Darmstadt, Praktische Informatik, D-6100 Darmstadt, June 1990.

[9] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML Version 2. Technical Report ECS-LFCS-88-62, University of Edinburgh, August 1988.

[10] Michael Jäger, Michael Gloger, and Stefan Kaes. SAMPLE — a functional language. In Robin E. Bloomfield, Lynn S. Marshall, and Roger B. Jones, editors, *Proceedings VDM'88, VDM — The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*, pages 202–217, Dublin, September 1988. Springer-Verlag.

[11] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes. In LFP88 [13], pages 198–211.

[12] Stefan Kaes. Parametric overloading in polymorphic programming languages. In Ganzinger [7], pages 131–144.

[13] *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, Snowbird, July 1988.

[14] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[15] John C. Mitchell. Coercion and type inference. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[16] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In FPCA91 [4], pages 1–14.

[17] Atsushi Ohori. Type inference in a database programming language. In LFP88 [13], pages 174–183.

[18] *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, January 1989.

[19] Didier Rémy. Typechecking records and variants in a natural extension of ML. In POPL89 [18], pages 77–88.

[20] François Rouaix. Safe run-time overloading. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 355–366, San Francisco, California, January 1990.

[21] Jörg H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207–273, February 1989.

[22] Dennis Volpano and Geoffrey S. Smith. On the complexity of ML typability with overloading. In FPCA91 [4], pages 15–28.

[23] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In POPL89 [18], pages 60–76.

[24] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the Second Annual Symposium of Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.