A Type is a Type is a Type[⋆]

Martin Müller Joachim Niehren

Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
{mmueller,niehren}@dfki.uni-sb.de

Abstract. We present an incremental constraint solver as the nucleus of a soft type checker for a higher-order concurrent constraint language. Designed as a variation of rational tree unification, our algorithm additionally decides satisfiability of weak subtype constraints of the form $x{\subseteq}y$. It allows for a number of extensions such as record types, sorts, union types, and type declarations, which we discuss by example. Hard disjunctive constraints are handeled as incomplete propagators which incrementally make as many simple constraints explicit as feasible. These extensions let our algorithm become suitable for type checking of a full-fledged programming language.

1 Introduction

We present an incremental constraint solver as the nucleus of a static type checker for a dynamically typed language. Our particular interest is to provide types for a higher-order concurrent constraint language such as Oz [Smo95, HSW95, SSW94]. We aim at rejecting as many ill-typed programs as feasible, but not to prove the accepted programs well-typed. While strong type checking strives to maximize the number of programs for which well-typedness is provable, our approach can be characterized as "soft" [CF91, WC94].

Our algorithm decides satisfiability of equational constraints $x{=}y$ and $x{=}f(\overline{y})$, and containment constraints $x{\subseteq}y$ modelling a weak subtype relation. The semantics of containment is axiomatized by the first-order formulae:

$$\overline{u}{\subseteq}\overline{v} \leftrightarrow f(\overline{u}) \subseteq f(\overline{v}) \qquad \text{and} \qquad x{\subseteq}f(\overline{u}) \rightarrow \exists\overline{w}(x{=}f(\overline{w}))$$

This formalizes the intuition that $x{\subseteq}y$ states "x has at least the structure of y".

Satisfiability of constraints can be decided in one of three canonical models: One is based on weak subsumption of feature trees, a second one interprets types as rational constructor trees with holes, and the third one takes a type to be a *non-empty* set and interprets $x{\subseteq}y$ as set inclusion. Our containment relation is

[⋆] A preliminary version of this paper was presented at CLNLP'95 [MN95b]. The research reported here has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), the Esprit Working Group CCL (EP 6028), and a fellowship of the Graduiertenkolleg 'Kognition' at the Univ. des Saarlandes of the first author.

weak in not being able to express coreference information. For instance, interpreted in the model of rational trees with holes $x{=}f(a\ b) \wedge y{=}f(x\ x)$ implies $x{\subseteq}y$, even if $a \neq b$.

The algorithm is designed to extend the standard rule system for rational unification, and it is just as close to its implementation as the latter. Being fully incremental, our algorithm is suited for type checking an interactive language. The algorithm can be extended to a full-fledged type diagnosis system [MN95a], covering record types, sorts, disjunctive (i.e., union) types, recursive data type declarations, and others. Record types are modelled by feature constraints [ST94], and type declarations by membership constraints following [NPT93]. For these extensions, the set based semantics is most flexible and carries furthest.

For efficieny reasons, the treatment of hard disjunctive constraints is restricted by an incomplete solver: We shall consider disjunctive constraints as "propagators" which only emit as many simple constraints as feasible. Our setting allows us to draw from experience in the constraint programming field, where traditionally hard constraints are only approximated. Concepts like that of a propagator have been developed for (finite domain) constraint programming (cf., "cardinality constraints" [HD91, MPSW94]). Note that these propagation techniques are impossible to apply to a non-incremental algorithm.

By design, the whole algorithm is suitable for integration into a concurrent constraint language like Oz [Smo95] or AKL [JH91] and allows us to understand type checking as a application of constraint programming.

Related Work. The containment relation turns out to be equivalent to weak subsumption which was introduced by Dörre [Dör94] as an approximation of the undecidable (strong) subsumption [DR90] between feature trees. Dörre drew his motivation from a linguistic problem in constraint based grammar formalisms. He proved decidability of weak subsumption [Dör94]; by reference to the feature tree model, this also settles decidability of containment. However, [Dör94] does not give an incremental algorithm as we do, and he does not consider extensibility of his algorithm. One relationship between type checking and linguistics via the undecidable semi-unification problem [KTU93] is well-known, since semi-unification is equivalent to both type checking polymorphic recursion [Myc84, Hen88] and (strong) subsumption [DR90] on feature trees. We extend this correlation by showing how weak subsumption relates to soft typing.

In recent years, there has been increasing interest in type analysis for untyped languages (soft typing) [Tha90, AW93, AWL94, CF91, WC94]. By far most of the typing literature is for functional languages. The close relationship between constraint solving and type inference ([Wan87, AW93, PS94] and many others) or more general program analysis (e.g., [Hei92, PS94]) is well-established. Among the abounding literature, the work of Aiken, Wimmers et al. seems closest in some respects [AW93, AWL94]:

In this work, a rich type language containing union, intersection, complement functional and conditional types is considered. For a large class of constraints (so-called inductive systems) they give a complete decision algorithm. For efficiency reasons, their algorithm is not implemented completely, but these *pragmatics* of type inference are not focussed on. In contrast, drawing intuitions from concurrent constraint programming, our focus lies exactly on the problem of how to specify incompletely implemented constraints (propagators).

Our algorithm exhibits failure as soon as any variable is proved to denote the empty set. In contrast, the empty set in Aiken and Wimmer's system denotes the type of a non-terminating yet perfectly well-typed expression. Thirdly, type inference influenced by constraint programming allows to recast conditional types [AWL94] in an operational manner and express more general "constraint propagators" such as overloaded types.

Plan of the Paper. Section 2 illustrates the intended form of type diagnosis by example. In Section 3 we present our constraints and their semantics along with necessary notation. Section 4 gives the rules of the full algorithm and a number of examples. Section 5 sketches the correctness proof, and Section 6 some of the intended extensions. Section 7 summarizes.

2 A Type Diagnosis Example

Type checking a program consists in three steps: (*i*) every program variable is mapped to a type variable, (*ii*) the program itself is mapped to a constraint ϕ over these variables, and (*iii*) inconsistency of ϕ is interpreted as a type error. Consider the following program written in the ρ-calculus [NM95]. The ρ calculus can be considered as relational λ-calculus [Nie94] or as concurrent π-calculus [MPW92] with logic variables. Furthermore, it is a foundation for concurrent constraint programming [Smo94] in the style of Oz.

$$\exists x \exists y \exists z \exists p \quad p{:}uv/v{=}cons(x\ u) \wedge pyy \wedge x{=}f(y\ z)$$

This program declares four variables x, y, z, and p. It defines a relational abstraction p, which states that its two arguments u and v are related through the equation $v = cons(x\ u)$.[2] Furthermore, it states the equality $x{=}f(y\ z)$ and applies p to yy. This application pyy reduces to a copy of the abstraction p with the actual arguments yy replaced for the formal ones uv:

$$\begin{aligned} &\exists x \exists y \exists z \exists p \quad p{:}uv/v{=}cons(x\ u) \wedge pyy \wedge x{=}f(y\ z) \\ \to\ &\exists x \exists y \exists z \exists p \quad p{:}uv/v{=}cons(x\ u) \wedge y{=}cons(x\ y) \wedge x{=}f(y\ z) \end{aligned}$$

Observe how the abstraction p is defined by reference to the global variable x, while the value of x is defined through an application of p in $pyy \wedge x{=}f(y\ z)$. Such a cycle is specific to Oz-like calculi, since no other language offers explicit declaration of logic variables global to an abstraction. The types of the variables involved are described by the following constraint, where – for ease of reading –

[2] The expression $p{:}uv/v{=}cons(x\ u)$ is a relational variant of the functional abstraction $p = \lambda u.cons(x\ u)$. It can also be considered as Prolog predicate `p(u,v) :- v=cons(x u).` extended with externally bound variables such as x.

the type variables are picked identical to the corresponding object variables and declarations are dropped:

$$p{=}\langle u\ v\rangle \wedge v{=}cons(x\ u) \wedge y{\subseteq}u \wedge y{\subseteq}v \wedge x{=}f(y\ z)$$

$\langle u\ v\rangle$ is the relational type of p, and the application gives rise to the constraint $y{\subseteq}u \wedge y{\subseteq}v$, which says that y is constrained by both formal arguments of the procedure p. The subconstraint $x{=}f(y\ z) \wedge y{\subseteq}v \wedge v{=}cons(x\ u)$ reflects the cyclic dependency between x and p. It says that y be contained in the type v which depends through $v{=}cons(x\ u)$ on x, and that x should be exactly $f(y\ z)$.

The use of containment constraints to type applications corresponds to the instantiation of polymorphic types in a functional language with polymorphic recursion. While some polymorphic flavour is preserved, we abandon full parametric polymorphism: For example, the polymorphic type $\forall x.x \to \mathsf{int}$ of *length* is expressible,[3] while the coreference in the type of the identity function $\forall x.x \to x$ is lost due to the weak semantics of $\subseteq$. Here one can see that our constraints have soft typing character.[4]

3 Constraints and Semantics

We assume a signature Σ of function symbols with at least two elements, ranged over by f, g, h, a, b, c, and an infinite set of *base variables* BV.

We build constraints over a set of *variables* V, ranged over by x, y, z, u, v, w, which contain at least the base variable ($BV \subseteq V$). Sequences of variables are written as $\overline{x}$, $\overline{y}$, ... The abstract syntax of our *constraints* ϕ, ψ is as follows:

$$\phi, \psi ::= x{=}y \mid x{=}f(\overline{y}) \mid x{\subseteq}y \mid \phi \wedge \psi$$

As *atomic constraints* we consider equations $x{=}y$ or $x{=}f(\overline{y})$ and containment constraints $x{\subseteq}y$. Constraints are atomic constraints closed under conjunction. A main contribution of this paper is an incremental algorithm deciding the satisfiability of constraints in some model of the axioms Ax in Figure 1.

$$\begin{array}{rll} (\textit{Decom}) & \forall x \forall \overline{y} \forall \overline{z}\ ((x{=}f(\overline{y}) \wedge x{=}f(\overline{z})) \leftrightarrow \overline{y}{=}\overline{z}) & \\ (\textit{Clash}) & \forall x \forall \overline{y} \forall \overline{z}\ ((x{=}f(\overline{y}) \wedge x{=}g(\overline{z})) \leftrightarrow \bot) & \text{if } f \neq g \\ (\textit{Descend}) & \forall x \forall \overline{v}\ (\exists z (x{\subseteq}z \wedge z{=}f(\overline{v}))) \leftrightarrow \exists \overline{u} (x{=}f(\overline{u}) \wedge \overline{u}{\subseteq}\overline{v})) & \end{array}$$

Fig. 1. The Axiom Scheme Ax

The axioms (*Decom*) and (*Clash*) are well known from unification of infinite trees. The axiom (*Descend*) combined with (*Decomp*) implies the monotonicity

[3] In our setting, this type is just $\langle x\ \mathsf{int}\rangle$.

[4] For a more detailed discussion of type diagnosis along these lines the reader is referred to the forthcoming report [MN95a].

of constructor application with respect to $\subseteq$. In order to formalize this, we introduce some syntactic sugar for first-order formulae over constraints. Let s and t denote terms over Σ with variables. If $\overline{u} = (u_i)_{i=1}^n$ and $\overline{s} = (s_i)_{i=1}^n$, then we write:

$$s{\subseteq}t \models\!\mid \exists x \exists y (x{\subseteq}y \wedge x{=}s \wedge y{=}t) \qquad x{=}f(\overline{s}) \models\!\mid \exists \overline{u}(x{=}f(\overline{u}) \wedge \overline{u}{=}\overline{s})$$
$$\overline{u}{=}\overline{s} \models\!\mid u_1{=}s_1 \wedge \ldots \wedge u_n{=}s_n$$

Proposition 1. *In all models of (Decomp), axiom (Descend) is equivalent to the conjunction of the following two schemes:*

$$(Monoton) \quad \overline{u}{\subseteq}\overline{v} \leftrightarrow f(\overline{u}){\subseteq}f(\overline{v}) \qquad (Constr) \quad x{\subseteq}f(\overline{v}) \rightarrow \exists \overline{w}(x{=}f(\overline{w}))$$

Proof. We first assume a model $\mathcal{A}$ of (*Decomp*) and (*Descend*). Obviously, (*Constr*) is implied by (*Descend*). The validity of (*Monoton*) follows from:

$$\begin{array}{rlr} f(\overline{u}){\subseteq}f(\overline{v}) & \models\!\mid \ \exists x(x{=}f(\overline{u}) \wedge x{\subseteq}f(\overline{v})) & \\ & \models\!\mid_{\mathcal{A}} \exists x(x{=}f(\overline{u}) \wedge \exists \overline{w}(x{=}f(\overline{w})) \wedge \overline{w}{\subseteq}\overline{v}) & (Descend) \\ & \models\!\mid_{\mathcal{A}} \exists x \exists \overline{w}(x{=}f(\overline{u}) \wedge \overline{u}{=}\overline{w} \wedge \overline{w}{\subseteq}\overline{v}) & (Decomp) \\ & \models\!\mid \ \overline{u}{\subseteq}\overline{v} & \end{array}$$

For the converse, we consider a model $\mathcal{A}'$ of (*Monoton*) and (*Constr*). We can establish (*Descend*) as follows:

$$\begin{array}{rlr} x{\subseteq}f(\overline{v}) & \models\!\mid_{\mathcal{A}'} x{\subseteq}f(\overline{v}) \wedge \exists \overline{u}(x{=}f(\overline{u})) & (Constr) \\ & \models\!\mid \ \exists \overline{u}(f(\overline{u}){\subseteq}f(\overline{v}) \wedge x{=}f(\overline{u})) & \\ & \models\!\mid_{\mathcal{A}'} \exists \overline{u}(\overline{u}{\subseteq}\overline{v} \wedge x{=}f(\overline{u})) & (Monoton) \end{array}$$

There exists several models of Ax with distinct first order-theories. These have been investigated in rather independent research areas: Subset constraints on sets of trees have proven useful for various program analysis problems such as type inference (e.g., [Hei92, AW93, AWL94]) while weak subsumption constraints on trees have been considered in computational linguistics [Dör94].

In the sequel, we assume a set H of *holes*. We write IT_H for the set of all finite or infinite *trees* over $\Sigma \cup H$, where holes are treated as additional constants. Trees are ranged over by s and t. The symbol IT stands for $\mathsf{IT}_\emptyset$.

Nonempty sets: The domain of Sets consists of all *nonempty* subsets of IT. Constructors are interpreted elementwise and $\subseteq$ as subset relation. Intuitions from type inference may justify the restriction to nonempty sets of trees as types. Note that the validity of all axioms of Ax depends on the nonemptiness assumption. For instance, $f^{\mathsf{Sets}}(\emptyset) = g^{\mathsf{Sets}}(\emptyset)$ holds for all f and g, even if $f \neq g$.

Weak instances or weak subsumption: The domain of IT_H is the set IT_H. Constructors are interpreted as tree constructors. For two trees s and t, we define $s{\subseteq}^{\mathsf{IT}_H}t$ by $s \in \mathsf{Inst}_H(t)$, where the set $\mathsf{Inst}_H(s)$ of *weak instances of* s is defined as the greatest fixed point of the following set-valued equation:

$$\mathsf{Inst}_H(s) = \begin{cases} \mathsf{IT}_H & \text{if } t \text{ is a hole, } t \in H \\ f(\overline{\mathsf{Inst}_H(s)}) & \text{if } t = f(\overline{s}) \text{ for some } \overline{s} \end{cases}$$

There exists an equivalent definition for $\subseteq^{\mathsf{IT}_H}$ in terms of weak subsumption [Dör94, MN95b]. Let $t \downarrow p$ denote the subtree of t at position p, and $label(t)$ the constructor of t. Then we say that s is weakly subsumed by t, if for all paths p and all constructors f: $label(t \downarrow p) = f$ implies $label(s \downarrow p) = f$.

Theorem 2. *Let ϕ be a constraint and $H \neq \emptyset$ a nonempty set of holes. Then the following statements are equivalent:*

1) ϕ is satisfiable in some model of Ax. *2) ϕ is satisfiable in* Sets.
3) ϕ is satisfiable in IT_H. *4) ϕ is satisfiable in $\mathsf{IT}_{\{\bullet\}}$.*

We need at least one hole to prove satisfiability of containment constraints. E.g., $a \subseteq x \wedge b \subseteq x$ is satisfiable over IT_H if and only if $H \neq \emptyset$, provided $a \neq b$.

Proof. Since Sets, $\mathsf{IT}_{\{\bullet\}}$and IT_H are models of Ax, 2) $\Rightarrow$ 1), 3) $\Rightarrow$ 1), and 4) $\Rightarrow$ 1) hold. Since $\mathsf{IT}_{\{\bullet\}}$ is a substructure of Sets and of IT_H, satisfiablity in $\mathsf{IT}_{\{\bullet\}}$ implies satisfiability in Sets and IT_H. Hence, 4) $\Rightarrow$ 2) and 4) $\Rightarrow$ 3) hold. The embedding of $\mathsf{IT}_{\{\bullet\}}$ to Sets is given by mapping $s \mapsto s[\mathsf{IT}/\bullet]$. An embedding of $\mathsf{IT}_{\{\bullet\}}$ to IT_H can be obtained by mapping $\bullet$ to an arbitrary element of H (which exists since $H \neq \emptyset$) and homomorphic extension. It is sufficient to establish 1) $\Rightarrow$ 4). This can be done by standard coinductive arguments.

In presence of negation this equivalence does no longer hold: Let $\Phi_1 = x \subseteq y \wedge y \subseteq x \rightarrow x = y$ and $\Phi_2 = \exists x (a \subseteq x \wedge b \subseteq x \wedge \neg c \subseteq x)$ where $a \neq b, a \neq c, b \neq c$, and observe that Φ_1 is valid in $\mathsf{IT}_{\{\bullet\}}$ and Sets but not in IT_H, while Φ_2 is valid in Sets but not in IT_H nor in $\mathsf{IT}_{\{\bullet\}}$.

Notice that the definition of weak instances implies $f(a\ b) \in \mathsf{Inst}_H(f(x\ x))$, even if $a \neq b$. The set of *strong instances of s* is defined by $\mathsf{Inst'}_H(s) = \{\sigma(s) \mid \sigma : \mathcal{V}(s) \rightarrow H \text{ is a substitution}\}$. Note that $\mathsf{Inst'}_H(s) \subseteq \mathsf{Inst}_H(s)$, and that $f(a\ b) \notin \mathsf{Inst'}_H(f(x\ x))$ if $a \neq b$. Using models of $\mathsf{Inst'}_H(s)$ instead of $\mathsf{Inst}_H(s)$ would make satisfiability of our constraints equivalent to semi-unification and undecidable [KTU90, DR90].

4 The Algorithm

At first sight, the satisfiablity problem seems to be a not too difficult extension of rational unification. We could simply add a directed version of (*Descend*):

$$(Descend) \quad \frac{x \subseteq y \wedge \phi}{x = f(\overline{u}) \wedge \overline{u} \subseteq \overline{z} \wedge \phi} \quad \overline{u} \text{ fresh}, y = f(\overline{z}) \text{ in } \phi.$$

In the above application condition and in the sequel we make use of the following notation: We define $\equiv$ to be the least equivalence relation on constraints such that $\wedge$ is associative and commutative in $\equiv$. Furthermore we write:

$$\phi \text{ in } \psi \qquad \text{iff} \quad \text{exists } \phi' \text{ with } \phi \wedge \phi' \equiv \psi$$

The (*Descend*) rule above is doomed because the introduction of fresh variables induces non-termination. Consider, for instance, a constraint with cyles such as $x \subseteq y \wedge y = f(x)$. On the other hand, (*Descend*) is needed in order to detect inconsistencies such as in: $y = f(u) \wedge u = a \wedge z = f(x) \wedge x \subseteq y \wedge x \subseteq z \wedge \phi$.

The key idea for a terminating algorithm is to add new constraints which avoid the explicit introduction of fresh variables. These can be motivated in the model Sets: To verify satisfiability of $x \subseteq y \wedge x \subseteq z$ in some context ϕ, we have to show that y and z have a nonempty intersection. We define the new class of *intersection variables* as follows:

$$X, Y, Z ::= x \mid X \cap Y$$

Equality on intersection variables is associative, commutative, and idempotent:

$$X \cap Y \equiv Y \cap X, \quad (X \cap Y) \cap Z \equiv X \cap (Y \cap Z), \quad X \cap X \equiv X.$$

We call X a *component* of Y, if there exists Z such that $X \equiv Y \cap Z$. The *set of components* of X is denoted by $\mathcal{C}(X)$, and the set of variables in ϕ by $\mathcal{V}(\phi)$. Note that $x \cap Y \in \mathcal{V}(\phi)$ implies $x \in \mathcal{C}(\mathcal{V}(\phi))$ but in general not $x \in \mathcal{V}(\phi)$. As new constraints we introduce $X \subseteq f(\overline{Y})$ and $x \subseteq Y$. That is, our algorithm actually operates on the following constraints:

$$\phi, \psi ::= x = y \mid x = f(\overline{y}) \mid x \subseteq Y \mid X \subseteq f(\overline{Y}) \mid \phi \wedge \psi$$

The standard model-theoretic semantics still applies when intersection variables are treated just as base variables. However, since it ignores the internal structure of intersection variables, it is incomplete. This will be fixed in Section 5.

Let us call a variable X *determined in* ϕ, if there exists f and $\overline{U}$, such that $X \leq_\phi f(\overline{U})$ is derivable with the following rules:

$$\frac{x = f(\overline{u})}{x \lhd f(\overline{u})} \qquad \frac{X \subseteq f(\overline{U})}{X \lhd f(\overline{U})} \qquad \frac{X \lhd f(\overline{U}) \text{ in } \phi}{X \leq_\phi f(\overline{U})} \qquad \frac{x \subseteq Y \cap Z \qquad Y \lhd f(\overline{U}) \text{ in } \phi}{x \leq_\phi f(\overline{U})}$$

We define the application of an operator $[y/x]$ to intersection variables componentwise. If $Z \equiv (z_1 \cap \ldots \cap z_n)$, then we set:

$$Z[y/x] \equiv z_1[y/x] \cap \ldots \cap z_n[y/x] .$$

Observe that in general $\mathcal{V}(x = y \wedge \phi) \neq \mathcal{V}(x = y \wedge \phi[y/x])$. For instance, if $\phi = z \subseteq x \cap y$, the variable $x \cap y$ is contained in the first set but not in the second one. However, equality holds if only base components of the constraint are considered:

$$\mathcal{C}(\mathcal{V}(x = y \wedge \phi)) \cap BV \;=\; \mathcal{C}(\mathcal{V}(x = y \wedge \phi[y/x])) \cap BV .$$

We can now specify our algorithm for constraint simplification: It is given by the rules in Figures 2 and 3. The rules in Figure 2 are known from usual rational tree unification. Only the application condition of (*Elim*) and (*Clash*) are original to our setting. The (*Clash*) rule contains as special cases:

$$\frac{x = f(\overline{y}) \wedge x = g(\overline{z}) \wedge \phi}{\perp} \; f \neq g \qquad \text{and} \qquad \frac{x \subseteq f(\overline{y}) \wedge x \subseteq g(\overline{z}) \wedge \phi}{\perp} \; f \neq g .$$

$(Decom)$	$\dfrac{x{=}f(\overline{y}) \wedge \phi}{\overline{y}{=}\overline{z} \wedge \phi}$	$x{=}f(\overline{z})$ in ϕ.
$(Clash)$	$\dfrac{\phi}{\bot}$	$X{\leq_\phi}f(\overline{U})$, $Y \in \mathcal{C}(X)$, $Y{\leq_\phi}g(\overline{V})$, and $f \neq g$.
$(Elim)$	$\dfrac{x{=}y \wedge \phi}{x{=}y \wedge \phi[y/x]}$	$x \in \mathcal{C}(\mathcal{V}(\phi))$, and $x \neq y$.

Fig. 2. Rational Tree Unification

$(Propagate1)$	$\dfrac{x{\subseteq}Y \wedge x{\subseteq}Z \wedge \phi}{x{\subseteq}Y{\cap}Z \wedge \phi}$	
$(Propagate2)$	$\dfrac{X{\subseteq}f(\overline{U}) \wedge \phi}{X{\subseteq}f(\overline{U}{\cap}\overline{V}) \wedge \phi}$	$Y \in \mathcal{C}(X)$, $Y{\leq_\phi}f(\overline{V})$, $\overline{U}{\cap}\overline{V} \not\equiv \overline{U}$.
$(Collapse)$	$\dfrac{x{\subseteq}Y \wedge \phi}{x{\subseteq}Y{\cap}Z \wedge \phi}$	$u \in \mathcal{C}(Y)$, $u{\subseteq}Z$ in ϕ, and $Y{\cap}Z \not\equiv Z$.
$(Descend1)$	$\dfrac{x{=}f(\overline{u}) \wedge \phi}{x{=}f(\overline{u}) \wedge \overline{u}{\subseteq}\overline{V} \wedge \phi}$	$x{\leq_\phi}f(\overline{V})$, not exists $\overline{W}$ such that $\overline{u}{\subseteq}\overline{V}{\cap}\overline{W}$ in ϕ
$(Descend2)$	$\dfrac{\phi}{X{\subseteq}f(\overline{U}) \wedge \phi}$	$X \in \mathcal{V}(\phi), Y \in \mathcal{C}(X)$, $Y{\leq_\phi}f(\overline{U})$, and not exists g and $\overline{V}$ such that $X{\subseteq}g(\overline{V})$ in ϕ or $X{=}g(\overline{V})$ in ϕ

Fig. 3. Simplifying Inclusion Constraints

Its full power comes in interaction with the rules in Figure 3.

The rules *(Propagate1)* and *(Propagate2)* propagate intersection variables into the right hand side of containment contraints. The *(Collapse)* rule collapses chains of variables related via containment constraints. In other words, these rules propagate upper bounds with respect to the containment relation.

The rules *(Descend1)* and *(Descend2)* replace the problematic rule *(Descend)* above. The Descend rules are the only rules introducing new containment constraints. Observe that both preserve well-formedness of constraints. The rule *(Descend2)* introduces a constructor for an intersection variable X by adding a constraint of the form $X{\subseteq}f(\overline{U})$. If the rule is applied, then the intersection of the components of X is forced to be nonempty. A constraint ϕ implies nonemptiness of every variable $X \in \mathcal{V}(\phi)$ (e.g., in $y{\subseteq}f(X)$).

Note that *(Descend1)* and *(Descend2)* are carefully equipped with side conditions for termination. For example, the following derivations are *not* possible:

$$\frac{x{=}f(u)}{x{=}f(u) \wedge x{\subseteq}f(u)} \qquad \frac{x{\subseteq}y \wedge x{=}f(x) \wedge x{\subseteq}f(y)}{x{\subseteq}y \wedge x{\subseteq}y \wedge x{=}f(x) \wedge x{\subseteq}f(y)} \qquad \frac{x{=}f(y)}{y{\subseteq}y \wedge x{=}f(y)} \ .$$

Example 1 (Simplifying Intersections). Assume $a \neq b$ and consider the non-satisfiable constraint $x{\subseteq}y{\cap}z \wedge \phi$ where $\phi \equiv y{=}f(y\ u) \wedge z{=}f(z\ v) \wedge\ u{=}a \wedge v{=}b$:

$$\begin{array}{cl}
x\subseteq y\cap z \wedge \phi & \\ \hline
x\subseteq y\cap z \wedge y\cap z\subseteq f(y\ u)\ \phi & \text{Descend2} \\ \hline
x\subseteq y\cap z \wedge y\cap z\subseteq f(y\cap z\ \ u\cap v) \wedge \phi & \text{Propagate2} \\ \hline
x\subseteq y\cap z \wedge y\cap z\subseteq f(y\cap z\ \ u\cap v) \wedge u\cap v\subseteq a \wedge \phi & \text{Descend2} \\ \hline
\perp & \text{Clash}
\end{array}$$

Example 2 (How (Descend) is circumvented). Consider the non-satisfiable constraint $x\subseteq y \wedge \phi$ where $\phi \equiv y{=}f(u) \wedge u{=}a \wedge z{=}f(x) \wedge x\subseteq z$. The derivation of $\perp$ looks as follows:

$$\begin{array}{cl}
x\subseteq y \wedge \phi & \\ \hline
x\subseteq y\cap z \wedge \phi & \text{Propagate1} \\ \hline
x\subseteq y\cap z \wedge y\cap z\subseteq f(u) \wedge \phi & \text{Descend2} \\ \hline
x\subseteq y\cap z \wedge y\cap z\subseteq f(u\cap x) \wedge \phi & \text{Propagate2} \\ \hline
x\subseteq y\cap z \wedge y\cap z\subseteq f(u\cap x) \wedge u\cap x\subseteq a \wedge \phi & \text{Descend2} \\ \hline
\perp & \text{Clash}
\end{array}$$

Example 3 (Deep Substitution). The substitution operation ensures (e.g.) that rules (Propagate1) and (Elim) are interchangeable. This is a requirement for completeness since we cannot fix the order of rule application in an incremental algorithm. Consider $\phi \equiv z\subseteq a \wedge u\subseteq z \wedge y\subseteq b$ with $a \neq b$ in:

$$\begin{array}{cl}
x{=}y \wedge u\subseteq x \wedge \phi & \\ \hline
x{=}y \wedge u\subseteq x\cap z \wedge \phi & \text{Propagate1} \\ \hline
x{=}y \wedge u\subseteq x\cap z \wedge x\cap z\subseteq a \wedge \phi & \text{Descend2} \\ \hline
x{=}y \wedge u\subseteq y\cap z \wedge y\cap z\subseteq a \wedge \phi & \text{Elim} \\ \hline
\perp & \text{Clash}
\end{array}$$

5 Correctness

The standard model theoretic semantics allows $\{\{a\}/y, \{b\}/z, \{c\}/y\cap z\}$ as a satisfying substitution of $x\subseteq y\cap z \wedge y{=}a \wedge z{=}b$. This conflicts with the intended semantics of intersection variables, as well as with our algorithm which derives a clash. However, the algorithm performs equivalence transformations if we restrict ourselves to so-called *intersection-correct* substitutions:

Definition 3 Intersection Correct. We say that a substitution $\sigma : V \to \mathsf{Sets}$ is *intersection-correct for X and Y*, if $\sigma(X\cap Y) = \sigma(X) \cap \sigma(Y)$. We call a substitution σ *intersection-correct*, if for all intersection variables X and Y:

- If X, Y, $X\cap Y \in \mathsf{dom}(\sigma)$, then σ is intersection-correct for X and Y.
- If X, $X\cap Y \in \mathsf{dom}(\sigma)$, then σ is intersection-correct for $X\cap Y$ and Y.

Note that σ is intersection-correct for X and $X \cap Y$ iff $\sigma(X \cap Y) \subseteq \sigma(X)$. Call a constraint ϕ *intersection-satisfiable*, if ϕ has an intersection-correct solution.

Proposition 4. *Let ϕ be a constraint only containing base variables. Then ϕ is satisfiable, if and only if it is intersection satisfiable.*

The set of all intersection-correct solutions of ϕ is written $\mathrm{Sol}^I(\phi)$. Let $\mathrm{Ext}^I_V(\sigma)$ denote the set of all intersection correct substitutions $\tilde{\sigma}$ such that $\mathsf{dom}(\tilde{\sigma}) = \mathsf{dom}(\sigma) \cup V$ and σ and $\tilde{\sigma}$ coincide on $\mathsf{dom}(\sigma)$. For two constraints ϕ and ψ we say that ϕ *intersection-implies* ψ, written $\phi \models^I \psi$, if

$$\mathrm{Ext}^I_{\mathcal{V}(\psi)}(\mathrm{Sol}^I(\phi)) \;\subseteq\; \mathrm{Sol}^I(\psi) \quad \text{and} \quad \mathrm{Sol}^I(\phi) = \emptyset \text{ if } \mathrm{Ext}^I_{\mathcal{V}(\psi)}(\mathrm{Sol}^I(\phi)) = \emptyset$$

By the first condition, every solution of ϕ must be correctly extensible to a solution of ψ. The second condition excludes for instance: $\phi \models^I \phi \wedge z \subseteq x \cap y$ where $\phi = x{=}a \wedge y{=}b$. The set $\mathrm{Ext}^I_{\mathcal{V}(\phi \wedge z \subseteq x \cap y)}(\mathrm{Sol}^I(\phi))$ is empty, since the variable $x \cap y$ prevents any intersection correct extension of $\{x \mapsto a, y \mapsto b\}$. We call ϕ and ψ *intersection-equivalent* if $\phi \models^I \psi$ and $\psi \models^I \phi$, and write $\phi \mathrel{|\!\!\!=\!\!|}^I \psi$.

Lemma 5. *If ϕ is not intersection satisfiable, then $\phi \models^I \psi$ holds trivially for all ψ. Furthermore, if $\phi \mathrel{|\!\!\!=\!\!|}^I \psi$, then ϕ is intersection satisfiable if and only if ψ is.*

Theorem 6 Termination. *The rule system from Figures 2 and 3 terminates.*

Theorem 7 Correctness and Completeness. *For an arbitrary constraint ϕ the following statements are equivalent:*

1. *ϕ is intersection-satisfiable.*
2. *There exists an irreducible constraint derivable from ϕ.*
3. *There exists an irreducible constraint that is intersection-equivalent to ϕ.*
4. *$\bot$ cannot be derived from ϕ.*

Proof. We define a normal form of constraints and prove that the algorithm (i) performs int.-equivalence transformations, (ii) always yields either $\bot$ or a normal form constraint, and that (iii) normal forms always are int.-satisfiable.

6 Towards a Realistic System

Much of the flexibility of soft type systems is due to the capability to express disjunctive types [WC94, AWL94]. Disjunctive types are necessary to handle program expressions with branching control such as conditionals. Consider the following three-way case statement:

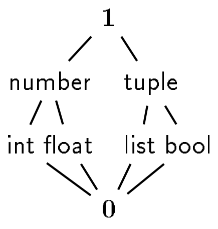
case x of $f(y\ z)$ then $x{=}f(a\ b)$ [] 1 then $true$ [] $f(c\ y)$ then $y{=}d$ end

For this expression to be free of run-time errors, x must be bound to one of $f(a\ b)$, 1, or $f(c\ d)$.[5] This set of values can be approximated with different precision, for instance by one of the following types:

$$\mathsf{int} \sqcup \mathsf{tuple} \qquad \{f, 1\} \qquad f(a{\cup}c\ b{\cup}d){\cup}1 \qquad f(a\ b){\cup}f(c\ d){\cup}1$$

We call the first two approximations *sorts*, and the latter two *union types.*

Sorts are formalised by reference to a complete lattice $(\mathbf{S},\sqsubseteq)$ with minimal and maximal elements $\mathbf{0}$ and $\mathbf{1}$.[6] A sorting function Sort() assigns every constructor f a non-empty sort $S \in \mathbf{S}$, for instance $Sort(nil) = Sort(cons) = \mathsf{list}$, $Sort(1) = \mathsf{int}$, and $Sort(1.2) = \mathsf{float}$. The constraints are extended by a sort constraint $x{\subseteq}S$ where $x{\subseteq}\mathbf{0}$ is equivalent to $\bot$.



Using sorts, we can easily derive the type tuple for x if we assume the equation $x = f(e\ d)$ in the context which constrains x to sort tuple:

$$x{\subseteq}\mathsf{int} \sqcup \mathsf{tuple} \wedge x{\subseteq}\mathsf{tuple} \ \rightarrow\ x{\subseteq}(\mathsf{int} \sqcup \mathsf{tuple}) \sqcap \mathsf{tuple} \ \equiv\ x{\subseteq}\mathsf{tuple}$$

To cover also disjunctions of structured types, we allow union types:

$$\phi\ ::=\ \ldots \mid x{\subseteq}S \mid x{\subseteq}y_1{\cup}\ldots{\cup}y_n \mid x{\subseteq}f(\overline{y_1}){\cup}\ldots{\cup}f(\overline{y_n})$$

With union types, the type of x can be described more precisely by

$$x{\subseteq}f(a\ b){\cup}f(c\ d) \wedge x{\subseteq}f(e\ d) \tag{1}$$

Many soft typing systems allow only "tidy" or "deterministic" unions where the top-level constructors in a union must be different (e.g. [WC94]). In such systems, type information like $f(a\ b){\cup}f(c\ d)$ is immediately approximated by $f(a{\cup}c\ b{\cup}d)$. In contrast, we allow the *representation* of such types, but formalize their *propagation behaviour* operationally in terms of reduction rules. From the constraint (1) above we derive $e{\subseteq}a{\cup}c \wedge d{\subseteq}b{\cup}d$ by an operation similar to antiunification, which in turn reduces to $\bot$ immediately. The inconsistency of the constraint $x{\subseteq}f(a\ a){\cup}f(b\ b) \wedge x{\subseteq}f(a\ b)$, however, cannot be derived.

It is a common technique in constraint programming to implement hard constraints incompletely via "propagators" which continuously watch the already accumulated constraints, and which (*i*) disappear once they are entailed and (*ii*) emit additional constraints if certain conditions are met. This view point allows us to exactly specify the amount of incompleteness with which hard constraints are treated. In addition, one can elegantly express overloaded types like $(\mathsf{int} \times \mathsf{int} \times \mathsf{int}) \cup (\mathsf{real} \times \mathsf{real} \times \mathsf{real})$. Assuming $+$ to have this type, the application $x{+}y{=}z$ would immediately constrain x, y, and z to have at least type $\mathsf{num} = \mathsf{int} \cup \mathsf{real}$: When one of x, y, z gets constrained to int later, it will also constrain y and z to int, and then disappear.

[5] The logic variable used in constraint languages adds the possibility of x not being bound at all.

[6] This tiny fragment here is part of the type hierarchy of Oz data structures [Smo95].

7 Summary

We have presented an incremental algorithm for solving equational and containment constraints. For satisfiability of these constraints we have given three equivalent models drawing intuitions from very different fields. The kernel algorithm has be shown to be terminating, correct and complete.

We have argued the use of these constraints for soft type inference. Based on the kernel algorithm, we have sketched a number of extensions which carry it to type diagnosis for realistic programs. Our approach can carry over propagation techniques from the constraint programming field. These can be used to operationally specify the exact amount of incompleteness with which hard constraints (e.g. for union types) are handled.

Acknowledgements. We would like to thank Ralf Treinen for pointing us to Dörre's paper and Alex Aiken for detailed discussions on set constraints.

References

[AW93] A. Aiken and E. Wimmers. Type Inclusion Constraints and Type Inference. In *6th ACM Conference on Functional Programming and Computer Architecture*, pp. 31–41, Copenhagen, Denmark, June 1993.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft Typing with Conditional Types. In *21st ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

[CF91] R. Cartwright and M. Fagan. Soft Typing. In *ACM Conference on Programming Language Design and Implementation*, pp. 278–292, June 1991.

[Dör94] Jochen Dörre. Feature-Logic with Weak Subsumption Constraints. In *Constraints, Languages, and Computation*, ch. 7, pp. 187–203. Acad. Press, 1994.

[DR90] Jochen Dörre and William C. Rounds. On Subsumption and Semiunification in Feature Algebras. In *IEEE Symposium on Logic in Computer Science*, pp. 300–310, 1990.

[HD91] Pascal Van Hentenryck and Yves Deville. The Cardinality Operator: A New Logical Connective for Constraint Logic Programming. In Koichi Furukawa, editor, *8th International Conference on Logic Programming*, pp. 745–759, Paris, France, 1991. The MIT Press.

[Hei92] Nevin Heintze. Practical Aspects of Set Based Analysis. In *International Conference and Symposium on Logic Programming*, pp. 765–779, 1992.

[Hen88] F. Henglein. Type Inference and Semi-Unification. In *ACM Conference on LISP and Functional Programming*, pp. 184–197, January 1988.

[HSW95] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pp. 27–48. The MIT Press, Cambridge, MA, 1995. To appear.

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *International Logic Programming Symposium*, pp. 167–186, 1991.

[KTU90] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The Undecidability of the Semi-Unification Problem. In *ACM Symposium on Theory of Computation*, pp. 468–476, May 1990.

[KTU93] A. J. Kfoury, J. Tiuryn, and Urzycyn. Type Recursion in the Presence of Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, pp. 290–311, 1993.

[MN95a] Martin Müller and Joachim Niehren. A Type is a Type is a Type. Draft Research Report, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, 1995.

[MN95b] Martin Müller and Joachim Niehren. Weak Subsumption Constraints for Type Diagnosis: An Incremental Algorithm. In *Joint COMPULOGNET/ELSNET/EAGLES Workshop on Computational Logic for Natural Language Processing*, Edinburgh, Scotland, April 3–5 1995.

[MPSW94] Tobias Müller, Konstantin Popow, Christian Schulte, and Jörg Würtz. Constraint programming in Oz. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, September 1992.

[Myc84] Alan Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming, LNCS 167*, 1984.

[Nie94] Joachim Niehren. *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen.* Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, 66041 Saarbrücken, Germany, December 1994.

[NM95] Joachim Niehren and Martin Müller. Constraints for Free in Concurrent Computation. In *First International Workshop on Concurrent Constraint Programming*, Venice, Italy, May29–31 1995. Submitted to CP'95.

[NPT93] Joachim Niehren, Andreas Podelski, and Ralf Treinen. Equational and Membership Constraints for Infinite Trees. In Claude Kirchner, editor, *Proceedings of the RTA '93*, pp. 106–120, 1993.

[PS94] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems.* Wiley, Chichester, England, 1994.

[Smo94] Gert Smolka. A Foundation for Concurrent Constraint Programming. In Jean-Pierre Jouannaud, editor, *Constraints in Computational Logics, LNCS 845*, pp. 50–72, München, Germany, 7–9 September 1994.

[Smo95] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, LNCS 910, pp. 251–292. Springer, 1995.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pp. 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer-Verlag.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[Tha90] S. R. Thatte. Quasi-static Typing. In 7^{th} *ACM Symposium on Principles of Programming Languages*, pp. 367–381. CACM, January 1990.

[Wan87] Mitchell Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[WC94] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. In *ACM Conference on LISP and Functional Programming*, pp. 250–262, June 1994.