Chapter 1

Detecting Common Elements of Types

Manfred Widera[1] and Christoph Beierle[2]

Abstract: We describe an algorithm approximating the following question: Given two types t_1 and t_2, are there instances $\sigma(t_1)$ and $\sigma(t_2)$ denoting a common element? By answering this question we solve a main problem towards a type checking algorithm for non-disjoint types that raises an error just for function calls that cannot be executed successfully. For dynamically typed functional languages such a type checker can extend actual soft typing systems in order to reject provably ill-typed programs.

1.1 INTRODUCTION

When type inference and type checking for functional programs is done according to the approach first presented in [Mil78] different types denote disjoint sets of values. Type checking is done using the binary relation $=$, i.e. for every function call $(f\ a)$ in the analyzed program the type inferred for the argument a and the input type expected by f have to be equal.

By introducing type languages with non-disjoint types equality of types became a too strong restriction. Therefore, subtyping relations $\sqsubseteq$ where introduced modeling the question whether a type t_1 denotes a subset of the values denoted by t_2. For a function call $(f\ a)$ a type checker based on $\sqsubseteq$ performs a test whether the type inferred for the argument a is a subtype of the expected input type of f. The test fails for all the cases where the type inferred for a contains at least one value f is not applicable to. This sound approach to type checking can be used in the following ways:

[1]Fachbereich Informatik, FernUniversität Hagen, D-58084 Hagen, Germany. Email: Manfred.Widera@Fernuni-Hagen.de

[2]Fachbereich Informatik, FernUniversität Hagen, D-58084 Hagen, Germany. Email: Christoph.Beierle@Fernuni-Hagen.de

- In a strictly typed language as e.g. Haskell [HPF97] the type checker prevents ill-typed programs from execution. It is integrated in the language definition excluding ill-typed programs from the set of valid programs.
- When using a sound static type checker for a dynamically typed functional language the set of valid programs in the language is not affected by the type checker. Since dynamically typed languages are not designed with a precise static type checking in mind extensive use of the language's expressiveness can easily confuse the type checker. The output messages are therefore interpreted as *warnings* instead of *errors* yielding the concept of soft-typing (see e.g. [CF91]).

In this work we present another extension of [Mil78] to type languages with non-disjoint types that addresses dynamically typed languages. We introduce a binary relation on types that is defined by an algorithm $CE(t_1,t_2)$ and that approximates the test whether there is a common element denoted by both t_1 and t_2. A type checker based on this relation checks for every function call $(f\ a)$ whether the type inferred for a and the expected input type of f have common elements. The test fails if no common elements are detected. In this case no evaluation of the call can succeed. (We assume the usual situation that a type t of an expression e denotes all values e can be evaluated to, but may denote additional values.)

This is the basis of what we call *complete type checking* [WB00]: The type checker is complete if it accepts every program in which all function calls can be executed without raising a type error.

The best use for a type checker based on the relation on types induced by *CE* is a combination with a soft typing system or a system with an output similar to a soft typing system [Fla97, FF99]:

- The new type checker focuses on a set of proven errors (i.e. program parts that cannot be executed without raising an error). Obviously, in any case such errors must be corrected. If instead these errors were only indicated as warnings within in a large number of type warnings given by the soft typing system they could be overlooked quite easily.
- Errors that cannot be proven and thus are overlooked by the new type checker are caught by the soft typing system and are presented as warnings.

Example 1.1. Consider the function call $c = (\textit{vector-ref}\ v\ i)$ in the functional language Scheme [KCE98] with a vector expression v and an index expression i. Assume that k is an expression with inferred type `posint` (the type of all positive integers). Depending on the expression i the type checkers behave as follows:

- If $i = (+\ k\ 3)$ then the type `posint` can be inferred for i. The function call c is well-typed in every type checker.
- For $i = (-\ k\ 3)$ just the type `int` (the type of all integers) can be inferred. A soft typing system raises a warning because i may be a negative integer causing an error in c. A complete type checker based on *CE* accepts the call c.

- If $i = (* \; k \; -2)$ the most special type inferred for *i* is `negint` (the type of all negative integers). A soft typing system still just raises a warning. A complete type checker based on *CE* raises an error for *c* because this call cannot succeed with a negative number as vector index.

The rest of the paper introduces an algorithm *CE* for checking types for common elements and is organized as follows: Section 1.2 gives a definition of the type language used throughout the paper. In Sec. 1.3 the algorithm *CE* for detecting common elements is presented. Its main component called *S-CE* is given in 1.3.2. Section 1.4 gives some conclusions and points out further work.

1.2 THE TYPE LANGUAGE

The following definition Def. 1.1 introduces the set of all types for a given set of variables V. The definition of the type syntax is standard except for the function types.

Definition 1.1 (standard types). *Let V be a set of variables. The set $T(V)$ of all type terms over V is defined as the smallest set with:*

1. $B \subseteq T(V)$ *where B is a set of all base types containing $\bot$ and $\top$ defined as usual.*
2. $V \subseteq T(V)$.
3. $c \in K, e_1, \ldots, e_{a(c)} \in T(V) \Rightarrow (c \; e_1 \; \ldots \; e_{a(c)}) \in T(V)$ *where K is a set of type constructors and $a(c)$ is the arity of c.*[3]
4. $e_1, \ldots, e_{k \geq 0} \in T(V) \Rightarrow (\cup \; e_1 \; \ldots \; e_k) \in T(V)$.
5. $Tfunc, Tfunc_P, Tfunc_U \in T(V)$ *where Tfunc is called the type of all function definitions (functions for short), $Tfunc_P$ is the type of all predefined functions and $Tfunc_U$ the type of all user defined functions.*
6. *If $X \in V$ is a variable and $t \in T(V)$ contains X then $\mu X.t \in T(V)$.*

The function types introduced in this definition differ from the usual approach: We intend to build a type checker that just raises an error for function calls $(f \; a)$ if the type inferred for *a* and the input type expected by f do not have any common elements. In order to perform this test we need information about *all* valid input values to f. The usual function type constructor that is anti-monotonic in its first argument does not provide this.

The function types defined here do not carry much information. They can just distinguish predefined and user defined function definitions. A more sophisticated definition of function types is given in [WB01].

[3] For instance, the binary "cons" operator "." is in K with $a(.) = 2$.

Definition 1.2 (closed/ground types). *The set $T_C(V) \subset T(V)$ of* closed types *consists of all types not containing any variables not bound by μ. The set $T_G \subset T_C(V)$ of* ground types *consists of all types without any variables.*

Since the only variables occurring in $T_C(V)$ are bound by μ and can be renamed the set V does not matter for closed types. We therefore often write T_C instead of $T_C(V)$.

We define positions in terms as usual as lists of natural numbers (written as $p = p_1.p_2.\ldots.p_k$) with ε denoting the empty list. For every type term t we have $t_\varepsilon := t$. For $k > 0$ and p defined as above the subterm of t at p is defined as usual by $t_p = t'_{p'}$ where t' is the p_1^{th} subterm of t and $p' = p_2.\ldots.p_k$. We write $t_p = \texttt{undefined}$ if there is no subterm at position p in t.

If the second subterm of a recursive type term is selected we introduce an implicit unfolding step in the subterm selection denoted by t_p: For $t = \mu X.t'$ we define that t_2 does not yield t' but $t'[X|t]$, i.e. t' with every free occurrence of X replaced by t.

The notion $t[p|t']$ for type terms t and t' and a position p is introduced for the type t with position p changed to t'.

Definition 1.3 (unfolding recursive bindings). *Let $t \in T$ be type. We define the function* unfold *as unfold$(t) = t'$ where t' differs from t at exactly those positions p with:*

- *$t_p = \mu X.\tilde{t}$ is a recursive type.*
- *There is no proper prefix p' of p such that $t_{p'}$ is a recursive type, i.e. t_p is not a subterm of any recursive type except of itself.*

For each of these positions p the returned type term t' fulfills $t'_p = t_{p.2}$.

The semantics of the type language defined above is as usual. For $t \in T$ the semantics of t is denoted by $\langle\![t]\!\rangle$. A detailed definition of $\langle\![\cdot]\!\rangle$ is given in [WB01].

1.3 CHECKING TYPES FOR COMMON ELEMENTS

The central question occurring in complete type checking is the following: Given two types t_1 and t_2. Is there a substitution σ such that $\langle\![\sigma(t_1)]\!\rangle \cap \langle\![\sigma(t_2)]\!\rangle \neq \{\bot\}$?

In this section an algorithm *CE* is introduced that approximates the answer in the following sense: For every existing substitution with the given property, *CE* returns a more general substitution. On the other hand *CE* is really an approximation and not a decision procedure. Thus, there may be situations where *CE* returns a substitution even if t_1 and t_2 cannot have common elements, e.g. when one of the types does not have any elements at all.

The description of *CE* is done in the following subsections: Section 1.3.1 contains the preliminaries used to define *CE*. In Sec. 1.3.2 an algorithm *S-CE* is introduced that calculates constraints on the variable instantiations. The remaining part of *CE* sketched in Sec. 1.3.3 transforms the constraint sets to idempotent substitutions σ and returns a set of them.

1.3.1 Preliminaries

In this section we will define some structures forming the result or intermediate values of *CE*. The goal of *CE* is to find a set of type substitutions more general than the closed type substitutions transforming two types t_1 and t_2 to closed types with common elements. Type substitutions are defined as follows:

Definition 1.4 (type substitution). *A* type substitution *is a function mapping type variables $X \in V$ to types t_X. A type substitution σ assigning the type t_i to the type variable X_i for all $i \in \{1,\ldots,k\}$ is denoted by $[X_1 \leftarrow t_1,\ldots,X_k \leftarrow t_k]$; its domain is denoted by $dom(\sigma) = \{X_1,\ldots,X_k\}$. A substitution is called* idempotent *if all assigned values t_X do not contain any variables $Y \in dom(\sigma)$ as subterms.*

The return value of *CE* is an s-collection defined as follows:

Definition 1.5 (s-collection). *An* s-collection *is a finite set of type substitutions.*

During the calculation of *CE*, constraints on the possible instantiations of type variables are collected in structures called *constraint sets*:

Definition 1.6 (constraint sets). *A variable constraint is a pair (X,t) (often written as $X \leftarrow t$) where $X \in V$ and $t \in T$. A* constraint set *is a set of variable constraints $\{(X_1,t_1),\ldots,(X_n,t_n)\}$ with pairwise distinct variables X_i. For such a constraint set σ, $dom(\sigma) = \{X_1,\ldots,X_n\}$ and $\sigma(X_i) = t_i$.*

The intermediate values occurring in *CE* are not constraint sets directly, but sets of constraint sets called *c-collections*:

Definition 1.7 (c-collection). *A* c-collection *is a finite set of constraint sets.*

1.3.2 The Algorithm *S-CE*

The main task of calculating substitutions that cover all common elements of two types is done by the algorithm *S-CE*. The way *S-CE* decomposes structures to compare the elements is quite similar to term unification [Rob65]. There are, however, several differences e.g. for the processing of unions and for the repeated unfolding of recursive types.

S-CE is presented as a function *S-CE*(t_1,t_2,σ,r) where

- t_1 and t_2 are the types that are checked.
- σ is a constraint set.
- $r = ((t_{1,1},t_{2,1}),\ldots,(t_{1,m},t_{2,m}))$ is a list of type pairs called recursion history. r is just used when processing recursive types. It contains all pairs of types with at least one recursive type that have already been processed in a previous recursive call.

An initial call of *S-CE* of the form *S-CE*$(t_1, t_2, \{\}, ())$ is initiated by *CE*. Its arguments are two types t_1 and t_2 to be checked for common elements, the empty constraint set $\sigma = \{\}$ and the empty recursion history $r = ()$.

As result *S-CE* returns a c-collection. For every common element v this c-collection contains a constraint set that describes a restriction of t_1 and t_2 to types both containing v. The c-collection is empty if no common elements were detected.

The behaviour of *S-CE* is determined by a case distinction on the the two first parameters t_1 and t_2. We will now describe *S-CE* for each of the possible cases. Since the cases are not disjoint we will present them in the fixed order in which they are checked.

$\top$ *in one of the Arguments*

$\top$ has common elements with every type (except of $\bot$). So if one of the types is $\top$ and the other one is not $\bot$ then *S-CE* directly returns with success. This is formalized in the rules (*Top1*) (and (*Top2*) defined analogously):

$$\frac{\textit{S-CE}(\top, t_2, \sigma, r)}{\{\sigma\}} \quad t_2 \neq \bot \qquad (\textit{Top1})$$

Recursive Types

When one of the types t_1 and t_2 is a recursive one *S-CE* essentially performs a further test with the recursive type unfolded. But since types constructed by the recursive type constructor correspond to infinite syntax trees we need a special termination condition when working on recursive types. During descending an infinite branch of the syntax tree there is just a finite number of different type pairs t_1 and t_2 that are checked by *S-CE*. The number of different type pairs containing at least one recursive type is also finite, but there must be an infinite number of recursive calls to *S-CE* to get an infinite execution. Thus, the only possibility to get an infinite execution is that after a finite number of calls to *S-CE* there is a call with an argument pair already given in one of the calls before.

Consider a recursive call to *S-CE* with types t_1 and t_2 with:

- One of the types is a recursive one.
- There is a call to *S-CE* in the recursive history with the same types t_1 and t_2.

The new call to *S-CE* will not yield any evidence against common elements not already detected in processing the former call with the same arguments. Hence, the actual call can return without introducing any restrictions. This if formalized by rule (*RecT*).

$$\frac{\textit{S-CE}(t_1, t_2, \sigma, r)}{\{\sigma\}} \quad (t_1, t_2) \in r \qquad (\textit{RecT})$$

Now we can explain the unfolding of recursive types in the first or second argument done by the rules (*Rec1*) and (*Rec2*), respectively:

$$\frac{\textit{S-CE}(t_1 = \mu X.t_1', t_2, \sigma, r)}{\textit{S-CE}(\textit{unfold}(t_1), t_2, \sigma, ((t_1, t_2)\ .\ r))} \qquad (\textit{Rec1})$$

(*Rec2*) is defined analogously to (*Rec1*) for unfolding the second argument.

Type Variables

The following auxiliary functions are used by *S-CE* for type variables:

extend-constraint(X, t, σ) adds a term t to the constraint of a variable X in σ. If X was not constrained in σ this is straightforward. Otherwise, X is constrained to the union of the previous constraint of X and t:

Definition 1.8 (*extend-constraint*). *Let σ be a constraint set, $X \in V$ a variable and t a term. $\sigma' = \textit{extend-constraint}(X, t, \sigma)$ is the constraint set differing from σ just in the binding of X as follows:*

- *If X is unconstrained in σ then $X \leftarrow t \in \sigma'$.*
- *If $X \leftarrow t' \in \sigma$ then $X \leftarrow (\cup\, t'\ t) \in \sigma'$.*

add-var-constraint(t, σ) updates the constraints of all variables occurring in a term t by adding the constraints to newly introduced variables:

Definition 1.9 (*add-var-constraint*). *Let t be a type term and σ a constraint set. The call add-var-constraint(t, σ) constrains every variable occurring in t to a fresh variable. The new constraints are added to σ without destroying previous constraints:*

forall** variables Y occurring in t **do
  ***let** Y' be a new variable.*
  $\sigma :=$ *extend-constraint*(Y, Y', σ)
***return** σ.*

We can now discuss the cases with at least one variable as argument. When both types are type variables then their bindings both are extended by a new variable:

$$\frac{\textit{S-CE}(t_1, t_2, \sigma, r)}{\textit{extend-constraint}(t_1, X', \textit{extend-constraint}(t_2, X'), \sigma))} \quad t_1, t_2 \in V, X' \,\text{new}\, (\textit{BothVar})$$

When just t_1 is a type variable then t_2 is united to the old binding of t_1 in σ. Furthermore, the variables occurring in t_2 are constrained to new variables to allow arbitrary instantiations afterwards:

$$\frac{\textit{S-CE}(t_1, t_2, \sigma, r)}{\{\textit{add-var-constraint}(t_2, \textit{extend-constraint}(X, t_2, \sigma))\}} \qquad t_1 \in V, t_2 \notin V \quad (\textit{Var1})$$

The case for just $t_2 \in V$ is given by rule (*Var2*) defined analogously to (*Var1*).

Union Types

When one of the types is a union type then a check has to be performed with the individual union elements and the results must be united. This is formalized in the rules (*U1*) and (*U2*) for a union type t_1 and t_2, respectively:

$$\frac{\textit{S-CE}((\cup\ t_{1,1}\ \ldots\ t_{1,k}), t_2, \sigma, r)}{\bigcup_{i=1}^{k} \textit{S-CE}(t_{1,i}, t_2, \sigma, r)} \qquad (U1)$$

(*U2*) is defined analogously to (*U1*) for a union type in the second argument.

Free Type Constructors

When both types are constructed by the same free type constructor c then the argument pairs of each position have to be checked sequentially collecting the restrictions. This is formalized in the following rule:

$$\frac{\textit{S-CE}((c\ t_{1,1}\ \ldots\ t_{1,k})\ (c\ t_{2,1}\ \ldots\ t_{2,k}), \sigma, r)}{\textit{SCE-list-reduce}(((t_{1,1}, t_{2,1}), \ldots, (t_{1,k}, t_{2,k})), \{\sigma\}, r))} \qquad (\textit{Constr})$$

The function *SCE-list-reduce* used by rule (*Constr*) expects a list of type pairs and an initial c-collection. It initiates sequential calls to *S-CE* with the types given in the type pairs as arguments. This is done for all constraint sets in the actual c-collection and the results are united. The actual c-collection is the initial one when processing the first type pair and the result of processing the previous type pair else. The recursion history r is just passed through to the calls to *S-CE*.

Definition 1.10 (*SCE-list-reduce*). *SCE-list-reduce expects a list L of tuples (t_1, t_2) where t_1 and t_2 are types, a c-collection Σ and a recursion history r.*

The result of SCE-list-reduce is a c-collection. SCE-list-reduce is defined by the following rules where (SCE-list-reduce1) applies for empty L and (SCE-list-reduce2) for non-empty L:

$$\frac{\textit{SCE-list-reduce}((), \Sigma, r)}{\Sigma} \qquad (\textit{SCE-list-reduce1})$$

$$\frac{\textit{SCE-list-reduce}(((t_i, t_i'), l_{i+1}, \ldots, l_k), \Sigma, r)}{\textit{SCE-list-reduce}((l_{i+1}, \ldots, l_k), \bigcup_{\sigma \in \Sigma} \textit{S-CE}(t_i, t_i', \sigma, r), r)} \qquad (\textit{SCE-list-reduce2})$$

Example 1.2 (SCE-list-reduce). Consider the following call to *SCE-list-reduce*:

$$\textit{SCE-list-reduce}(((A, \texttt{string}), (\texttt{num}, \texttt{int})), \{\{\}\}, r)$$

This call is processed by rule (*SCE-list-reduce2*). For every constraint set in the given c-collection (just $\{\}$ in this example) *S-CE* is called with the first type pair A and `string` as arguments, i.e.

$$\Sigma' := \{\sigma'\} = \textit{S-CE}(A, \texttt{string}, \{\}, r) \text{ with } \sigma' := \{A \leftarrow \texttt{string}\}$$

is calculated. The remaining list of type pairs is processed recursively by

$$\textit{SCE-list-reduce}(((\texttt{num},\texttt{int})),\Sigma',r)\,.$$

Again *SCE-list-reduce* initiates one calls to *S-CE* with the first type pair as arguments because Σ' just contains one constraint set:

$$\Sigma'' := \textit{S-CE}(\texttt{num},\texttt{int},\sigma',r)$$

yielding $\Sigma'' = \Sigma'$. This is returned by the last recursive call to *SCE-list-reduced* which is processed by (*SCE-list-reduce1*).

Function Types

Definition 1.11. *The function CEfunc* : $T \times T \to \{\texttt{true},\texttt{false}\}$ *expects two types* t_1 *and* t_2 *and returns true if* t_1 *and* t_2 *are both function types that are either equal or at least one of them is Tfunc. In all other cases CEfunc returns* `false`*.*

If $\textit{CEfunc}(t_1,t_2) = \texttt{true}$, no further restrictions must be introduced.

$$\frac{\textit{S-CE}(t_1,t_2,\sigma,r)}{\{\sigma\}} \qquad \textit{CEfunc}(t_1,t_2) = \texttt{true} \qquad (\textit{Func})$$

Base Types

When t_1 and t_2 are both base types then the question whether they have common elements is answered by a function *CEbase* fulfilling Assumption 1.1:

$$\frac{\textit{S-CE}(t_1,t_2,\sigma,r)}{\{\sigma\}} \quad \textit{CEbase}(t_1,t_2) = \texttt{true} \qquad (\textit{Base})$$

Assumption 1.1 (*CE* on base types) *CEbase* : $B \times B \to \{\texttt{true},\texttt{false}\}$ *approximates the question whether two base types have common elements as follows:*

$$\langle\![b_1]\!\rangle \cap \langle\![b_2]\!\rangle \neq \{\bot\} \Rightarrow \textit{CEbase}(b_1,b_2) = \texttt{true}\,.$$

There is an algorithm also called *CEbase* that returns the value of the function *CEbase* for every input by just checking a finite table of possible inputs.

Integrating the Cases

The following definition presents the algorithm *S-CE* by integrating the previously introduced rules in a fixed order:

Definition 1.12 (algorithm *S-CE*). *The algorithm S-CE tests the rules given above for applicability in the following order and returns the result given by the first applicable rule:*

$$(\textit{Top1}),(\textit{Top2}),(\textit{RecT}),(\textit{Rec1}),(\textit{Rec2}),(\textit{BothVar}),(\textit{Var1}),$$
$$(\textit{Var2}),(\textit{U1}),(\textit{U2}),(\textit{Constr}),(\textit{FunQ}),(\textit{Base})\,.$$

If none of the rules is applicable S-CE returns the empty c-collection $\emptyset$*.*

Example 1.3 (S-CE). Consider the call $S\text{-}CE(t_1, t_2, \sigma, r)$ with:

$$t_1 := \mu X.(\cup\ \mathtt{nil}\ (A\ .\ X)),\ t_2 := (\mathtt{bool}\ .\ (\mathtt{int}\ .\ \mathtt{nil}))),\ \sigma := \emptyset,\ r := ()$$

1. $S\text{-}CE(t_1, t_2, \sigma, r)$ is processed by rule (*Rec1*) resulting in the call

   $$S\text{-}CE((\cup\ \mathtt{nil}\ (A\ .\ t_1)), (\mathtt{bool}\ .\ (\mathtt{int}\ .\ \mathtt{nil}), \sigma, r') \text{ where } r' = ((t_1, t_2))\,.$$

2. By rule (*U1*) this call is splitted into two subcalls:

   (a) $S\text{-}CE(\mathtt{nil}, (\mathtt{bool}\ .\ (\mathtt{int}\ .\ \mathtt{nil})), \sigma, r') = \emptyset$
   (b) $S\text{-}CE((A\ .\ t_1), (\mathtt{bool}\ .\ (\mathtt{int}\ .\ \mathtt{nil})), \sigma, r')$

3. The second call is processed by rule (*Constr*) calling *SCE-list-reduce*. The result of call number i to *S-CE* is denoted by Σ_i:

   (a) $\Sigma_1 = S\text{-}CE(A, \mathtt{bool}, \sigma, r') = \{\sigma'\}$ where $\sigma' := \{A \leftarrow \mathtt{bool}\}$
   (b) $\Sigma_2 = S\text{-}CE(t_1, (\mathtt{int}\ .\ \mathtt{nil}), \sigma', r')$

4. The second call is processed by rule (*Rec1*) and the resulting call by (*U1*) producing the calls

   (a) $\Sigma_1 = S\text{-}CE(A, \mathtt{int}, \sigma, r') = \{\sigma''\}$ where $\sigma'' := \{A \leftarrow (\cup\ \mathtt{bool}\ \mathtt{int})\}$
   (b) $\Sigma_2 = S\text{-}CE(t_1, \mathtt{nil}, \sigma'', r'')$

   where $r'' := ((t_1, (\mathtt{int}\ .\ \mathtt{nil})), (t_1, t_2))$.

5. Σ_2 is calculated by applying (*Rec1*) and (*U1*) as before yielding the calls

   (a) $S\text{-}CE(\mathtt{nil}, \mathtt{nil}, \sigma'', r''') = \{\sigma''\}$ by rule (*Base*).
   (b) $S\text{-}CE((A\ .\ t_1), \mathtt{nil}, \sigma'', r''') = \emptyset$

   where $r''' := ((t_1, \mathtt{nil}), (t_1, (\mathtt{int}\ .\ \mathtt{nil})), (t_1, t_2))$.

The result $\{\sigma''\}$ is returned through all the stages of recursive calls to *S-CE*. When finishing the processing of the union types it is united with $\emptyset$ from the other branch, respectively, but remains unchanged by calculating these unions. Thus, $\{\sigma''\}$ is the result of the initial call.

1.3.3 The Algorithm *CE*

The work of the algorithm *CE* consists of calling *S-CE* with the empty constraint set and empty recursion history and transforming the resulting c-collection into an s-collection. Due to lack of space we do not present the transformation of c-collections to s-collections performed by *CE* in detail, but give some examples for the main tasks.

A c-collection is a set of constraint sets. Every constraint set in the c-collection returned by *S-CE* is transformed into an idempotent substitution independently. The resulting substitutions are collected into an s-collection.

The first task performed by *CE* is the replacement of variables by their constrains: When the right hand side of a variable constraint contains a variable constrained in the constraint set then the occurrences of this variable are essentially replaced by their constraining type terms:

Example 1.4 (replacement of variables). Consider a call to *CE* with the types $t_1 := (X\ .\ (\texttt{string}\ .\ Z))$ and $t_2 := ((Y\ .\ \texttt{num})\ .\ (Y\ .\ (Y\ .\ \texttt{nil})))$. The c-collection returned by *S-CE* is

$$\{\{X \leftarrow (Y\ .\ \texttt{num}),\ Y \leftarrow \texttt{string},\ Z \leftarrow (Y\ .\ \texttt{nil})\}\}.$$

The only constraint set is transformed by inserting the value for Y into the right hand sides of X and Z:

$$\{X \leftarrow (\texttt{string}\ .\ \texttt{num}),\ Y \leftarrow \texttt{string},\ Z \leftarrow (\texttt{string}\ .\ \texttt{nil})\}.$$

An iterated insertion of constraining terms for variables is just possible for constraint sets not containing cyclic dependencies. If cyclic dependencies occur iterated insertion of constraining terms will lead to variables constrained to terms containing themselves. For such variables recursive bindings are introduced.

Example 1.5 (introducing recursive bindings). Consider a call to *CE* with $t_1 = (X\ .\ (Y\ .\ (Z\ .\ \texttt{nil})))$ and $t_2 = ((Y\ .\ X)\ .\ (Z\ .\ ((\texttt{num}\ .\ X)\ .\texttt{nil})))$ as arguments. The c-collection returned by *S-CE* is

$$\{\{X \leftarrow (Y\ .\ X), Y \leftarrow Z, Z \leftarrow (\texttt{num}\ .\ X)\}\}$$

It is transformed by *CE* into the following s-collection:

$$\begin{aligned}\{\{&X \leftarrow \mu V_X.(\mu V_Z.(\texttt{num}\ .\ (V_Z\ .\ V_X))\ .\ V_X),\\ &Y \leftarrow \mu V_Z.(\texttt{num}\ .\ \mu V_X.(V_Z\ .\ V_X)),\ Z \leftarrow \mu V_Z.(\texttt{num}\ .\ \mu V_X.(V_Z\ .\ V_X))\}\}\end{aligned}$$

Theorem 1.2 (correctness of *CE*). *Let $t_1, t_2 \in T$. If there exists a value $v \neq \bot$ and a substitution ρ such that $v \in \langle\!|\rho(t_1)|\!\rangle \cap \langle\!|\rho(t_2)|\!\rangle$ then there exist substitutions $\sigma \in CE(t_1,t_2)$ and τ with $v \in \langle\!|\tau \circ \sigma(t_1)|\!\rangle \cap \langle\!|\tau \circ \sigma(t_2)|\!\rangle$.*

A proof of this theorem in an extended framework and a termination proof for *CE* can be found in [WB01].

1.4 CONCLUSIONS AND FURTHER WORK

For a type language with non-disjoint types this paper introduced an algorithm *CE* that approximates the test for common elements in two types. It gives a sound answer to the question whether the value sets denoted by two types are disjoint. It therefore gives a sound approximation for the question whether a function call *must* fail because of a type error.

The algorithm consists of two components: The first component (called *S-CE*) decomposes its argument types step by step and collects constraints on the

bindings of variables in order to provide certain common elements. The second component transforms the resulting constraint sets into idempotent substitutions for type variables.

Though this work shows how the question for common elements of two types can be answered we do not have an efficient tool for solving sets of common element constraints yet. A problem is the fact that the common element relation is not transitive as e.g. subtype relations usually are. A type checker based on *CE* therefore needs a different mechanism to solve sets of such constraints. In a restricted framework prototypes of abstract interpreters solving every constraint on the fly after generating it gave promising results.

REFERENCES

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[FF99] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Syatems*, 21(2):370–416, 1999.

[Fla97] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, Houston, Texas, May 1997.

[HPF97] Paul Hudak, John Peterson, and Joseph H. Fasel. *A Gentle Introduction to Haskell – Version 1.4 –*, March 1997.

[KCE98] Richard Kelsey, William Clinger, and Jonatan Rees (Editors). Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[WB00] Manfred Widera and Christoph Beierle. How to combine the benefits of strict and soft typing. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*. Intellect, 2000.

[WB01] Manfred Widera and Christoph Beierle. An approach to checking the non-disjointness of types in functional programming. Informatik Berichte, FernUniversität Hagen, 2001.