An Algorithm for Checking the Disjointness of Types

Manfred Widera
Fachbereich Informatik
FernUniversität Hagen
58084 Hagen
Germany
E-Mail:Manfred.Widera@Fernuni-Hagen.de

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 any input arguments. For dynamically typed functional languages as e.g. Scheme such a type checker can extend current soft typing systems in order to reject provably ill-typed programs.

1. INTRODUCTION

When type inference and type checking for functional programs is done according to Milner's approach [8] 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 $\sqsubseteq$ were introduced modeling the question whether a type t_1 denotes a subset of the values denoted by t_2. In a type system where `posint` denotes positive integers, *negint* denotes negative integers and `int` denotes the set of all integers we usually have:

- `posint` $\sqsubseteq$ `int`.
- `list posint` $\sqsubseteq$ `list int`.
- `posint` $\not\sqsubseteq$ `negint`.

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 according to its type, e.g. if the type inferred for a is `int`, but f just expects `posint`. This sound approach to type checking can be used in the following ways:

- In a strictly typed language as e.g. Haskell [6] 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 usually 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* denoting those program parts which need a runtime type check and should be checked by the programmer for type correctness. This yields the concept of soft-typing (see e.g. [1]).

In this work we present another extension of the equality relation used by Milner [8] to type languages with non-disjoint types (that especially addresses dynamically typed languages). We introduce a commutative 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. E.g. we have:

- $CE(\texttt{int}, \texttt{posint}) = \texttt{true}$.
- $CE(\texttt{list int}, \texttt{list posint}) = \texttt{true}$.
- $CE(\texttt{negint}, \texttt{posint}) = \texttt{false}$.

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. If the test fails (i.e. if no common elements are detected) *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* [15]: A type checker is called complete if it accepts every program in which all program parts can be executed without raising a type error for at least one tuple of input arguments.

The intended use for a complete 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 [3, 4] with the following benefits:

- The new type checker focuses on a set of provable type errors (i.e. program parts that cannot be executed without raising an error). Obviously, 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, then they could be overlooked quite easily.
- Function calls that cannot be proven erroneous and are hence overlooked by the new type checker are caught by the soft typing system and are presented as warnings.

Using the new approach the programmer can start correcting provable type errors. Correcting an existing error can eliminate several related warnings which are just generated by a confused type checker. Thus, the new approach provides a better understanding of the errors in a program. After correcting all provable errors the number of warnings should have been reduced significantly and the remaining warnings can be checked as usual.

Example 1.1 *Consider the function call* $c = (vector\text{-}ref\ v\ i)$ *(which expects a positive integer in the second argument) in the functional language Scheme [7]. Let* v *be a vector expression and* i *an index expression. Assume that* k *is an expression with inferred type* `posint` *(the type of all positive integers). Depending on the expression* i *the different 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*: since we cannot be sure that* i *is indeed a negative number, there is no reason for the complete type checker to reject the call.*
- *If* $i = (*\ k\ \ -2)$ *the most special type that can be 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.*

In practical programs a situation comparable to Ex. 1.1 is often caused by partial types [11, 12]: Consider some list l with several valid element types providing the input argument for a call to a function f. The three cases of the example arise if f is applicable to all, some or none of the valid element types of l.

The rest of the paper is organized as follows: Section 2 sketches the type language used throughout the work. In Sec. 3 the algorithm *CE* for checking types for common elements is presented. The first stage traversing the structure of the given type terms was already presented in [14] and is therefore just sketched in Sec. 3. (A more detailed representation is given in App. A.) The second stage transforming the resulting constraints into idempotent substitutions is discussed in detail. In Sec. 4 an abstract semantics for functional languages like Scheme is sketched. This abstract semantics implements a complete type inference system. It extensively uses *CE* and therefore gives a motivation for the definition of *CE*. Section 5 finally gives some conclusion.

2. THE TYPE LANGUAGE

The following definition Def. 2.1 introduces the set of all types for a given set V of variables. The definition of the type syntax is essentially standard except of the missing function types.

Definition 2.1 (standard types) *Let* V *be a set of variables. The set* $\mathcal{T}(V)$ *of all type terms over* V *is defined as the smallest set with:*

1. $\mathcal{B} \subseteq \mathcal{T}(V)$ *where* $\mathcal{B}$ *is a set of all base types containing* $\perp$ *defined as usual.*[1]
2. $V \subseteq \mathcal{T}(V)$.
3. $c \in \mathcal{K}, e_1, \ldots, e_{a(c)} \in \mathcal{T}(V) \Rightarrow (c\ e_1\ \ldots\ e_{a(c)}) \in \mathcal{T}(V)$ *where* $\mathcal{K}$ *is a set of type constructors and* $a(c)$ *is the arity of* c*.*[2]
4. $e_1, \ldots, e_{k\geq 0} \in \mathcal{T}(V) \Rightarrow (\cup\ e_1\ \ldots\ e_k) \in \mathcal{T}(V)$.
5. $e_1, \ldots, e_{k\geq 0} \in \mathcal{T}(V) \Rightarrow (\cap\ e_1\ \ldots\ e_k) \in \mathcal{T}(V)$.
6. $e \in \mathcal{T}(V) \Rightarrow \mathcal{C}e \in \mathcal{T}(V)$
7. *If* $X \in V$ *is a variable and* $t \in \mathcal{T}(V)$ *contains* X *then* $\mu X.t \in \mathcal{T}(V)$.

The type of all values is denoted by $\top$*. It just abbreviates a recursive binding of a variable* $X_\top$ *in a union type of the base types and all constructed types with a type constructor from* $\mathcal{K}$ *and* $X_\top$ *as argument types.*

If the set V *of variables is not of interest the set* $\mathcal{T}(V)$ *is sometimes written as* $\mathcal{T}$.

Besides the type terms generated by this definition we use difference types $t_1 \setminus t_2$ denoting all values of t_1 that are not denoted by t_2 in some of the examples. Internally, difference

[1] In the examples throughout this work we use the base types `int` for integer numbers, `num` for arbitrary numbers, `bool` for boolean values and `nil` for the empty list.

[2] For instance, the binary "cons" operator "." is in $\mathcal{K}$ with $a(.) = 2$. In the examples it is denoted by $(\cdot\ .\ \cdot)$.

types are expressed using intersection types and complement types $\mathcal{C}t$ with respect to $\top$, i.e.

$$t_1 \setminus t_2 = (\cap\ t_1\ \mathcal{C}t_2)\,.$$

Intersection types and complement types are not discussed any further in order to simplify the presentation of the following algorithms. A detailed discussion can be found in [16] and [13].

Note that by defining union types our type language can express partial types according to [11, 12].

Example 2.2 (partial types) *The (partial) type of all lists with integers and strings (denoted by the type* `string`*) as elements is expressible as*

$$\mu X.(\cup\ \textit{nil}\ ((\cup\ \textit{int}\ \textit{string})\ .\ X))\,.$$

Function types are not introduced in this definition because of the following reason: We intend to build a type checker that just raises an error for function calls $(f\ a)$ if the type inferred for a does not contain any elements the function f is applicable to.

In order to approximate this test by a test based on an input type t_{in} of f we need information about *all* valid input values of f in t_{in}. The usual function type constructor that is antimonotonic in its first argument does not provide this.

For the complete type inference process we extend the type language by a modified function type constructor whose input type covers the whole domain of the denoted functions f:

Definition 2.3 (I/O-representations) *An I/O-representation of a function f is given by a set of I/O-representation pairs $IN_i \dashrightarrow OUT_i$ with $IN_i, OUT_i \in \mathcal{T}$ such that:*

- $dom(f) \subseteq \bigcup_i \langle\!|IN_i|\!\rangle$[3]
- $\forall_i f(\langle\!|IN_i|\!\rangle) \subseteq \langle\!|OUT_i|\!\rangle \cup \{\texttt{error}\}$ *(where* `error` *denotes a type error caused by applying a function to an inappropriate argument).*

Note that I/O-representations look similar to refinement types [5] and intersection types [9]. Indeed comparable to these type definitions, every I/O-representation pair expresses a subset of the properties a denoted function must have. However, they differ in the following important aspect: While the function types in [5] and [9] have input types that denote a *subset* of the domain of every denoted function the input types of an I/O-representation pair denotes a *superset* of the denoted function's domains.

I/O-representations are used for expressing the types of predefined functions and for providing an external representation of user defined functions. A detailed discussion of I/O-representations is given in [16] and [13].

[3] $\langle\!|t|\!\rangle$ denotes the semantics of t for every type t.

Definition 2.4 (closed/ground types) *The set $\mathcal{T}_C(V) \subset \mathcal{T}(V)$ of* closed types *consists of all types not containing any variables not bound by μ. The set $\mathcal{T}_G \subset \mathcal{T}_C(V)$ of* ground types *consists of all types without any variables.*

Since the only variables occurring in $\mathcal{T}_C(V)$ are bound by μ and can be renamed without changing the meaning of the containing type the set V does not matter for closed types. We therefore often write $\mathcal{T}_C$ instead of $\mathcal{T}_C(V)$.

For a terms t and t' and a variable X the notion $t[X|t']$ denotes the type term that results from t by changing every occurrence of X that is not bound by μ to t'.

The semantics of the type language defined above is as usual. For $t \in \mathcal{T}$ the semantics of t is denoted by $\langle\!|t|\!\rangle$.[4] For every type t the value $\perp$ (for non-termination of a computation) is denoted by t, i.e.

$$\forall t \in \mathcal{T}\,.\perp \in \langle\!|t|\!\rangle\,.$$

A detailed definition of $\langle\!|\cdot|\!\rangle$ is given in [16].

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 $\langle\!|\sigma(t_1)|\!\rangle \cap \langle\!|\sigma(t_2)|\!\rangle \neq \{\perp\}$?

In this section an algorithm *CE* is introduced that approximates the answer to the above question in the following sense: For every existing substitution with the required property *CE* returns a more general substitution.

The description of *CE* is done in the following subsections: Section 3.1 contains the preliminaries used to define *CE*. In Sec. 3.2 an algorithm *S-CE* is sketched that calculates constraints on the variable instantiations. The remaining part *SMR* of *CE* introduced in Sec. 3.3 transforms the constraint sets to idempotent substitutions σ and returns a set of them.

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 such that for every common element v of the argument types t_1 and t_2 there is a substitution σ preserving the common element v. Type substitutions are defined as follows:

Definition 3.1 (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 type 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:

[4] Using $\langle\!|\cdot|\!\rangle$ for the type semantics leaves $[\![\cdot]\!]$ free to denote the semantics of program expressions.

Definition 3.2 (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 3.3 (constraint sets) *A variable constraint is a pair (X, t) (often written as $X \leftarrow t$) where $X \in V$ and $t \in \mathcal{T}$. 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 $\sigma(X_i) = t_i$.*

Constraint sets and type substitutions are defined in a similar manner and are even denoted in the same way. We will sometimes consider a constraint set as the type substitution denoted in the same way without making the transformation explicit.

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

Definition 3.4 (c-collection) *A* c-collection *is a finite set of constraint sets.*

3.2 The Algorithm S-CE

An important task of calculating substitutions that cover all common elements of two types consists of decomposing the structures of the types to compare the elements. The decomposition is quite similar to term unification [10]. There are, however, several extensions and differences e.g. for the processing of unions, for constraining variables and for the repeated unfolding of recursive types. The algorithm performing this decomposition task is called *S-CE*.

S-CE can be defined as a recursive 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 to *S-CE* of the form *S-CE*$(t_1, t_2, \{\}, ())$ is performed 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 of t_1 and t_2 this c-collection contains a constraint set that describes a restriction of variables occurring in t_1 and t_2 as follows: When every variable X constrained to a type t_X is substituted by a type denoting at least the elements denoted by t_X then the resulting types both contain v. The c-collection is empty if no common elements were detected.

Since constraints on variables can be given in terms of other variables the replacement often has to be iterated and a further substitution has to be used to eliminate unrestricted variables. The need for iterated instantiations is removed in Sec. 3.3.

The behaviour of *S-CE* is determined by a case distinction on the the two first parameters t_1 and t_2. A detailed description of the individual cases is given in [14] and [16]. For easy access they are summarized in App. A. The following cases are of special interest and differ from e.g. unification approaches:

- If both types are base types then the question whether these base types have common elements can be answered using an appropriate table of all pairs of base types.
- If one of the types is a type variable X then X can be restricted to the values denoted by the other type t in principle. One just has to make sure that different restrictions of the same variable X have to be composed to a *union* type in order not to loose common elements.
  If both types are variables then both of them are restricted to the same new variable denoting the (unknown) common elements. Again several restrictions of the same variable have to be united.
- Recursive types are unfolded and the decomposition is continued. The recursion history given in the fourth argument of *S-CE* stores all type pairs with one type getting unfolded. When the same type pair occurs again no further unfolding is performed: a further unfolding can just yield restrictions that have already been found by processing the previous call stored in the recursion history. Thus, breaking the processing of these cases essentially does not change the result provided by *S-CE* but enforces the termination of the algorithm.

The following examples illustrate the behaviour of *S-CE*:

The first example is important as it shows how *S-CE* can correctly instantiate a variable element type of lists with the union of several element types occurring in another list:

Example 3.5 (*S-CE*(1)) *Consider the function call*

$$(length\ (list\ \#f\ 42))\,.$$

In order to check whether the expected type of length and the inferred argument type have common elements one can use the call

$$S\text{-}CE(t_1, t_2, \sigma, r)$$

with:

- $t_1 := \mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X))$

- $t_2 := (\texttt{bool} \ . \ (\texttt{int} \ . \ \texttt{nil}))$
- $\sigma := \emptyset$
- $r := ()$

where A is a type variable. The result of this call is a c-collection $\{\sigma'\}$ containing a single constraint set

$$\sigma' := \{A \leftarrow (\cup \ \texttt{bool} \ \texttt{int})\}\,.$$

The following example is a more artificial one. Its purpose is to show that even complicated situations with the same variables occurring in both types can be handled by *S-CE* in a meaningful manner.

Example 3.6 (*S-CE*(2)) *Consider the call*

$$S\text{-}CE(t_1, t_2, \emptyset, ())$$

with:

- $t_1 := (X \ . \ (Z \ . \ \texttt{nil}))$
- $t_2 := ((Z \ . \ X) \ . \ ((\texttt{num} \ . \ X) \ .\texttt{nil}))$

where X and Z are type variables. The result of this call is a c-collection $\{\sigma\}$ containing a single constraint set

$$\sigma := \{X \leftarrow (Z \ . \ X), Z \leftarrow (\texttt{num} \ . \ X)\}\,.$$

3.3 The Algorithm SMR

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.

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 transformation of an constraint set to an idempotent substitution is done by the algorithm *SMR* as follows: Let $X_1, \ldots, X_k$ be the variables constrained by the given constraint set and for every $X_i \in dom(\sigma)$ let t_i be the type X_i is constrained to. In a certain order every X_i is eliminated from all t_j by inserting t_i instead. If a t_i contains X_i itself then a recursive binding has to be introduced. This is formally described in Def. 3.7.

SMR needs an arbitrary but fixed ordering $<_V$ on the set of variables. Along this ordering it replaces the variable X by the term assigned to it in the terms of all variables Y with $X < Y$. Afterwards the same procedure is done upside down. In every step occurrences of a variable X_i in its own assigned term t_i are expressed by a recursive type:

Definition 3.7 (algorithm *SMR*) *The algorithm SMR (simplify mutual recursion) expects a constraint set or a substitution as input and returns an idempotent substitution. The algorithm is presented in Fig. 1.*

The following examples illustrate the work of *SMR* and *CE*:

Example 3.8 (*CE*(1)) *Consider the call to S-CE discussed in Ex. 3.5 and the only constraint set*

$$\sigma' := \{A \leftarrow (\cup \ \texttt{bool} \ \texttt{int})\}$$

of its result. SMR transforms a given constraint set just by replacing variables in the right hand sides of the bindings. In σ' the binding of the only variable A does not contain any variables. Thus, $SMR(\sigma') = \sigma'$.

Example 3.9 (*CE*(2)) *Consider the call to S-CE discussed in Ex. 3.6 and the only constraint set*

$$\sigma := \{X \leftarrow (Z \ . \ X), Z \leftarrow (\texttt{num} \ . \ X)\}$$

of its result. Assume that $X <_V Z$ holds. The call $SMR(\sigma)$ is processed as follows:

When processing X in the first loop on i SMR removes X from its own right hand side t_X by introducing a recursive binding for a new variable $\tilde{X}$. Afterwards, the updated right hand side is inserted into the right hand side of Z. The result is:

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

Processing Z in the first i loop introduces a recursive binding for the new variable $\tilde{Z}$ in the right hand side of Z. The result is

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

During processing Z in the second i loop SMR replaces Z by its right hand side in the right hand side of X:

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

The second i loop does not perform any further changes for X. In the last step of SMR the nested recursive binding of $\tilde{X}$ is eliminated:

$$\begin{aligned}\sigma' := \{X &\leftarrow \mu\tilde{X} \,.\, (\mu\tilde{Z} \,.\, (\texttt{num} \ . \ (\tilde{Z} \ . \ \tilde{X})) \ . \ \tilde{X}),\\ Z &\leftarrow \mu\tilde{Z} \,.\, (\texttt{num} \ . \ \mu\tilde{X} \,.\, (\tilde{Z} \ . \ \tilde{X}))\}\,.\end{aligned}$$

This substitution σ' is the result of $SMR(\sigma)$.

The main property of *CE* is stated in the following theorem:

Theorem 3.10 (correctness of *CE*) *Let $t_1, t_2 \in \mathcal{T}$, both in set normalized form. Let there exist a value $v \neq \perp$ such that*

$$\exists \rho \,.\, v \in \langle\!\langle \rho(t_1) \rangle\!\rangle \cap \langle\!\langle \rho(t_2) \rangle\!\rangle\,.$$

Then there exists a substitution $\sigma \in CE(t_1, t_2)$ such that

$$\exists \tau \,.\, v \in \langle\!\langle \tau \circ \sigma(t_1) \rangle\!\rangle \cap \langle\!\langle \tau \circ \sigma(t_2) \rangle\!\rangle\,.$$

Sketch of proof The proof consists of two steps: Step 1 shows that for every common element v of t_1 and t_2 there is a constraint set σ' in the result of *S-CE* and a number $k_0 \in \mathbb{N}$ such that for every $k \geq k_0$:

$$\exists \tau \,.\, v \in \langle\!\langle \tau \circ {\sigma'}^k(t_1) \rangle\!\rangle \cap \langle\!\langle \tau \circ {\sigma'}^k(t_2) \rangle\!\rangle$$

where ${\sigma'}^k$ denotes applying σ' k times. This proof is done by structural induction on the types t_1 and t_2 and considering

Algorithm: SMR (simplify mutual recursion)
Input: A constraint set or substitution $\sigma = \{X_1 \leftarrow t_{X_1}, \ldots, X_k \leftarrow t_{X_k}\}$ (where $X_i <_V X_{i+1}$ for all i).
Output: A substitution $\tilde{\sigma}$.

for $i = 1$ **to** k **do**
  $t := \sigma(X_i)$
  if X_i occurs in t (* remove X_i from its own right hand side *)
    then $t' := \mu\tilde{X}_i.t[X_i|\tilde{X}_i]$ with a new variable $\tilde{X}_i \in V$
    else $t' := t$
  $\sigma := \sigma \setminus \{X_i \leftarrow t\} \cup \{X_i \leftarrow t'\}$ (* $\sigma := \sigma|_{dom(\sigma)\setminus\{X_i\}} \cup \{X_i \leftarrow t'_{X_i}\}$ *)
  for $j = i + 1$ **to** k **do** (* remove X_i from all t_{X_j} with $j > i$ *)
    (* $\sigma := (\sigma|_{\{X_i\}} \circ \sigma|_{\{X_{i+1},\ldots,X_k\}}) \cup \sigma|_{\{X_1,\ldots,X_i\}}$ *)
    $t_{X_j} := \sigma(X_j)$
    $t'_{X_j} := t_{X_j}[X_i|\sigma(X_i)]$
    $\sigma := \sigma \setminus \{X_j \leftarrow t_{X_j}\} \cup \{X_j \leftarrow t'_{X_j}\}$
  end (* for j *)
end (* for i *)
for $i = k$ **downto** 1 **do**
  for $j = i - 1$ **downto** 1 **do** (* remove X_i from all t_{X_j} with $j < i$ *)
    $t_{X_j} := \sigma(X_j)$
    $t'_{X_j} := t_{X_j}[X_i|\sigma(X_i)]$
    $\sigma := \sigma \setminus \{X_j \leftarrow t_{X_j}\} \cup \{X_j \leftarrow t'_{X_j}\}$
  end (* for j *)
end (* for i *)
In every t_{X_j} change every $\mu\tilde{X}_i.t$ in a position where $\tilde{X}_i$ is already bound by some μ constructor to $\tilde{X}_i$.

Figure 1: Algorithm SMR

the different decomposition and constraining rules of *S-CE* independently.

Step 2 of the proof shows that for every substitution σ' the algorithm *SMR* returns a substitution σ with

$$\sigma \circ \sigma'(t) = \sigma(t)$$

for every term t. With step 1 of the proof and induction on k_0 as chosen there this implies the theorem. $\square$

A detailed proof of Theorem 3.10 can be found in [13].

4. TYPE INFERENCE BY ABSTRACT INTERPRETATION

The definition of the algorithm *CE* is motivated by the fact that it is needed for complete type inference. In order to illustrate this we give a short summary of the type inference process using *CE* in the rest of this section.

The complete type inference process is presented in terms of an abstract interpretation. This follows [2] where abstract interpretation was shown to be appropriate to express different well known type inference systems. By using abstract interpretation we are able to solve constraints of non-empty intersections of types by *S-CE* on the fly directly when generating them. This avoids the need for an algorithm solving *sets* of such constraints which seems more difficult to find because these constraints are not transitive as e.g. equality constraints and subtype constraints are.

To perform complete type inference, an abstract semantics for the purely functional subset of e.g. Scheme is straightforward for most program expressions (e.g. for generation and application of lambda-closures, for if-expressions, constants,...). Environments are also abstracted in a natural manner.

The main difference to usual approaches used for sound type checkers is the processing of calls to predefined functions. Using the algorithm *CE* presented in Sec. 3.3 the type inference system checks for every function call $(f\ a)$ with input type t_{in} of f[5] and type t_a inferred for a whether

$$\langle\!\langle t_a \rangle\!\rangle \cap \langle\!\langle t_{\text{in}} \rangle\!\rangle \neq \emptyset$$

holds. Whenever *CE* returns an empty intersection for t_a and t_{in} we can conclude that

$$\langle\!\langle t_a \rangle\!\rangle \cap \langle\!\langle t_{\text{in}} \rangle\!\rangle = \emptyset$$

holds and therefore a type error has been detected. Otherwise, the output substitutions of *CE* are used to restrict type variables. Special care has to be taken not to destroy the information which variables were restricted to certain types. This is necessary to enable further restrictions:

Example 4.1 *Consider the following piece of code:*

```
(if (null? l)
    1
    (+ (car l) 42))
```

[5]The input type is taken with respect to an I/O-representation of f, i.e. $dom(f) \subseteq t_{\text{in}}$ holds.

The function null? used in the conditional expression has the following I/O-representation:

$$\{\ \mathtt{nil} \dashrightarrow \mathit{Ttrue},\ \top \setminus \mathtt{nil} \dashrightarrow \mathit{Tfalse}\ \}.$$

If l is unrestricted, i.e. is bound to a variable X then the calls

$$CE(X, \mathtt{nil}) = \{\{X \leftarrow \mathtt{nil}\}\}$$

and

$$CE(X, \top \setminus \mathtt{nil}) = \{\{X \leftarrow \top \setminus \mathtt{nil}\}\}$$

are performed. The results of these calls introduce the following restrictions for l:

- $X \leftarrow \mathtt{nil}$ *for the* then*-case.*
- $X \leftarrow \top\setminus\mathtt{nil}$ *for the* else*-case. The expression* (car l) *with I/O-representation*

  $$\{(A\ .\ \top) \dashrightarrow A\}$$

  further restricts l to a pair $(A\ .\ \top)$ with a fresh type variable A. This is just possible if l was not restricted to $\top \setminus \mathtt{nil}$ during the first restriction, but l is still restricted to X. Then the constraint of X can be refined from $\top \setminus \mathtt{nil}$ to $(A\ .\ \top)$.

The choice of the appropriate case of the conditional expression is done according to the output type of the currently considered I/O-representation pair.

With the abstract interpreter working as sketched before we can perform type inference for a program P as follows: For every user defined function f with arity k in P a call $(f\ A_1\ \ldots\ A_k)$ with new type variables A_i is generated and interpreted abstractly. The I/O-representation denoting f is given by the restrictions of the A_i as input type and result of the abstract interpretation as output type.

The sketched type inference system is indeed complete in the following sense: For every expression e it infers a set of types whose union covers all possible values of e. If during evaluating e the type `error` is inferred for a subexpression e' then a type error is risen for e'. Additionally, we can calculate conditions on e that cause an execution of e' and therefore a runtime type error during evaluating e. This is stated and proven formally in [13].

We finish the sketch of our complete type inference system by a well-typed and an ill-typed example of a program for multiplying two vectors given as lists of their elements.

Example 4.2 (well-typed example) *Consider the following Scheme program for the multiplication of two vectors:*

```
(define (v-v-mult row column)
  (cond ((and (null? row) (null? column)) 0)
        ;; non-recursive case
        (else
         (+ (* (car row) (car column))
                ;; multiply first elements
                ;; and process the vector
                ;; rests recursively
            (v-v-mult (cdr row) (cdr column))))))
```

The result of type inference for this program is an I/O-presentation with a single input type

$$IN = \mu V_R.(\cup\ \mathtt{nil}\ (\mathsf{num}\ .\ V_R)) \quad \times \mu V_C.(\cup\ \mathtt{nil}\ (\mathsf{num}\ .\ V_C))$$

and the corresponding output type $OUT = \mathsf{num}$.

Example 4.3 (ill-typed example) *Now consider the following ill-typed modification of the program of Ex. 4.2:*

```
(define (v-v-mult row column)
  (cond ((and (null? row) (null? column)) 0)
        ;; non-recursive case
        (else
         (+ (* (car row) (car column))
                ;; multiply first elements
                ;; and process the vector
                ;; rests recursively
            (* row column))))) ; Ill-typed call
                               ; to *
```

The result of type checking this program is a type error for the call to $*$ in the last line because row and column are lists which cannot be multiplied.

In contrast to soft typing one can rely on the error messages of the complete type checker as in the example above. No runtime type checks can fix the problems of such a erroneous call that must fail whenever reached.

A detailed definition of the complete type inference system together with a step to step presentation of the examples above and a theorem stating the completeness of the resulting type checker can be found in [13].

5. 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 of 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. It therefore provides a tool for adding some sort of strength to soft typing in dynamically typed functional languages by rejecting provably ill-typed programs.

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 (called *SMR*) 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. We sketched an abstract interpreter with the possibility of several iterated variable restrictions as such a mechanism. In a restricted framework prototypes of abstract interpreters solving every constraint on the fly after generating it gave promising results.

6. REFERENCES

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

[2] P. Cousot. Types as abstract interpretations. In *Conference Record of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 316–331, Jan. 1997.

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

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

[5] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, Toronto, Ontario Canada, 26–28 June 1991. *SIGPLAN Notices* 26(6), June 1991.

[6] P. Hudak, J. Peterson, and J. H. Fasel. *A Gentle Introduction to Haskell – Version 1.4 –*, Mar. 1997.

[7] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998.

[8] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, Dec. 1978.

[9] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 11, 1996.

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

[11] S. R. Thatte. Type inference with partial types. *Springer Verlag (Heidelberg, FRG and NewYork NY, USA) LNCS 317, Automata, Languages and Programming, 15th International Colloquium*, pages 615–629, 1988.

[12] S. R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124(1):127–148, 14 Feb. 1994.

[13] M. Widera. *Complete Type Inference in Functional Programming.* PhD thesis, University of Hagen, Germany, Apr. 2001. (submitted).

[14] M. Widera and C. Beierle. Detecting common elements of types. In S. Gilmore, editor, *Trends in Functional Programming*, volume 2. Intellect, 2000.

[15] M. Widera and C. Beierle. How to combine the benefits of strict and soft typing. In G. Michaelson, P. Trinder, and H.-W. Loidl, editors, *Trends in Functional Programming.* Intellect, 2000.

[16] M. Widera and C. Beierle. An approach to checking the non-disjointness of types in functional programming. Informatik Berichte 281, FernUniversität Hagen, Jan. 2001.

APPENDIX

A. THE CASES OF S-CE

As sketched in Sec. 3.2 the algorithm *S-CE* decomposes a pair of given types t_1 and t_2 and generates constraints on the instantiations of variables preserving the common elements of t_1 and t_2. In this appendix *S-CE* is presented in more detail.

The individual cases of the algorithm *S-CE* are presented in Fig. 2. *S-CE* behaves completely symmetric in its first two arguments t_1 and t_2. I.e. for the rules (*Top1*), (*Rec1*), (*Var1*) and (*U1*) there are symmetric rules (*Top2*), (*Rec2*), (*Var2*) and (*U2*).

The helper functions used in Fig. 2 are defined as follows:

- *unfold*(t) unfolds a recursive argument type t one time. This is defined as usual.
- *combine-cs* takes a c-collection σ as input and returns the c-collection σ' containing all combinations of subsets of σ. If several different constraint sets are generated after unfolding a recursive type with breaking an infinite unfolding chain then the unrestricted chain would generate combinations of all sets of these constraint sets. After breaking the infinite unfolding this combinations must be calculated by *combine-cs*. This is motivated and defined in detail in [13]. We omit the definition here.
- *extend-constraint*(V, t, σ) extends the constraint of V by t in σ:
  - If V is unconstrained in σ then the constraint $V \leftarrow t$ is added.
  - If σ contains a constraint $V \leftarrow t'$ for V then this constraint is replaced by $V \leftarrow (\cup\ t'\ t)$.
- *constrain-all-free*(t, σ) expects a type t and a constraint set σ as input. For every variable X occurring in t without a recursive binding it introduces a new variable X' and extends the constraint of X by X'.
  After the termination of *CE* the new variable X' is still unrestricted and expresses that X possibly has to cover additional values that are not known yet. During type inference a restriction of X' might become necessary.
- *SCE-list-reduce*$(((t^{(1)}, \tilde{t}^{(1)}), \ldots, (t^{(k)}, \tilde{t}^{(k)})), \{\sigma\}, r)$ expects a list of type pairs, an initial c-collection, and an initial recursion history. It successively calls *S-CE* on all the type pairs $(t^{(1)}, \tilde{t}^{(1)})$. The definition is given by the rules of Fig. 3.

$$\frac{S\text{-}CE(\top, t_2, \sigma, r)}{\{constrain\text{-}all\text{-}free(\sigma)\}} \quad \text{for } t_2 \neq \bot \qquad (Top1)$$

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

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

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

$$\frac{S\text{-}CE(t_1, t_2, \sigma, r)}{\{constrain\text{-}all\text{-}free(t_2, extend\text{-}constraint(t_1, t_2, \sigma))\}} \quad \text{for } t_1 \in V, t_2 \notin V \qquad (Var1)$$

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

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

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

Figure 2: The rules of *S-CE*

$$\frac{SCE\text{-}list\text{-}reduce((), \Sigma, r)}{\Sigma} \qquad (SCE\text{-}list\text{-}reduce\text{-}term)$$

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

Figure 3: The rules of *SCE-list-reduce*

- *CEbase* solves the question of common elements for base types. It returns `true` whenever both arguments are base types and have common elements. The definition of *CEbase* depends on the set $\mathcal{B}$ of base types and can be given as a finite enumeration.

The individual rules of *S-CE* cover the following cases:

- (*Top1*) or (*Top2*) apply if one of the types is the type $\top$ of all types and the other type t is not $\bot$. In this case a common element can be found for every restriction of the variables. The unrestricted occurrence of variables in t is expressed by *constrain-all-free.*
- The rules (*RecT*), (*Rec1*) and (*Rec2*) cover recursive types: (*Rec1*) and (*Rec2*) perform an unfolding step of a recursive type in the first or second argument position, respectively. (*RecT*) enforces termination by breaking infinite unfolding chains.
- Rule (*BothVar*) applies if both types are variables. A new variable X denoting the unrestricted common elements is introduced to extend the constraints of t_1 and t_2.
- The rules (*Var1*) and (*Var2*) apply if exactly one of the types is a variable X. Its constraint is extended by the other type t and the constraints of the variables occurring in t are extended by fresh variables to denote unrestricted occurrences.
- (*U1*) and (*U2*) process union types: *S-CE* is called for every union element independently. The results are combined in a c-collection.
- The rule (*Constr*) covers the case of two constructed types with the same type constructor. They have common elements exactly when every pair of argument types have common elements. This is checked by *SCE-list-reduce.* The constraints generated for all these pairs are collected in a common c-collection.
- Rule (*Base*) applies whenever *CEbase* returns `true`, i.e. if both types are base types and have common elements.

In all cases not covered by any of the above rules there are no common elements. The correctness of *S-CE* is proven in [13].