An Approach to Checking the Non-Disjointness of Types in Functional Programming

Manfred Widera, Christoph Beierle
Fachbereich Informatik
FernUniversität Hagen
58084 Hagen
e-mail: {Manfred.Widera | Christoph.Beierle}@FernUni-Hagen.de

January 2001

Abstract

This work motivates a new approach to type checking functional programs building a bridge between static typing and soft typing. A type language for this approach can be very precise; e.g. it may contain large and detailed subtyping hierarchies. Furthermore, certain requirements on the type language must be met that differ from usual type definitions. A type language appropriate for the new typing approach is defined in the first part of this work.
It turns out that it is a central question for our new type checker to check whether two types denote sets of values that are non-disjoint. Since our type language allows for the occurrence of type variables we are furthermore interested in restricting these variables as much as possible without reducing the set of common elements of two types. The second part of this work specifies an algorithm *CE* that approximates this question and proves the main properties of *CE*.

Contents

1 Introduction **1**

2 Preliminaries **4**

2.1 Term Rewriting Systems 4

2.1.1 Terms 4

2.1.2 Usual Term Rewriting Systems 5

2.1.3 Extended Term Rewriting Syntax 7

2.2 Graph Theory 9

3 The Type Language **11**

3.1 The Standard Types 11

3.1.1 Syntax of the Standard Types 12

3.1.2 Semantics of the Standard Types 20

3.1.3 Set Normalized Standard Types 27

3.1.4 Equivalence Classes of Types 29

3.2 Value Assignments 30

3.3 I/O-Representations of Function Types 30

3.4 The Subtype Hierarchy on Semi-Closed Types 33

3.4.1 The Algorithm ST 34

3.4.2 Examples of ST 36

3.4.3 Properties of ST 37

4 Checking Types for Common Elements **39**

4.1 Preliminaries 40

4.1.1 Structures used by CE 40

4.1.2 CE on Base Types and Value Assignments 42

4.2 The Algorithm S-CE 43

4.2.1 Globally Used Auxiliary Functions 43

4.2.2 $\top$ in one of the Arguments 46

4.2.3 Recursive Types 46

4.2.4 Free Type Variables 48

4.2.5 Union Types . . . 50
4.2.6 Intersection Types . . . 50
4.2.7 Complement Types . . . 51
4.2.8 Free Type Constructors . . . 51
4.2.9 Frame Types . . . 51
4.2.10 Environment Types . . . 52
4.2.11 Function Types and Quantified Variables . . . 52
4.2.12 Base Types and Value Assignments . . . 53
4.2.13 Integrating the Cases . . . 53
4.2.14 An *S-CE* Example in Detail . . . 53
4.2.15 Properties of *S-CE* . . . 55
4.2.16 The Order of *S-CE* Rules . . . 59
4.2.17 Further Optimizations . . . 61
4.3 The Algorithm *CE* . . . 63
4.3.1 The Core Component of *CE* . . . 65
4.3.2 Auxiliary Functions used by *CE* . . . 65
4.3.3 Examples of Calls to *CE* . . . 68
4.3.4 Properties of *CE* . . . 70

5 Checking Types for Common Elements **74**
5.1 Preliminaries . . . 75
5.1.1 Structures used by *CE* . . . 75
5.1.2 *CE* on Base Types and Value Assignments . . . 77
5.2 The Algorithm *S-CE* . . . 78
5.2.1 Globally Used Auxiliary Functions . . . 78
5.2.2 $\top$ in one of the Arguments . . . 81
5.2.3 Recursive Types . . . 81
5.2.4 Free Type Variables . . . 83
5.2.5 Union Types . . . 85
5.2.6 Intersection Types . . . 85
5.2.7 Complement Types . . . 86
5.2.8 Free Type Constructors . . . 86
5.2.9 Frame Types . . . 86
5.2.10 Environment Types . . . 87
5.2.11 Function Types and Quantified Variables . . . 87
5.2.12 Base Types and Value Assignments . . . 88
5.2.13 Integrating the Cases . . . 88
5.2.14 An *S-CE* Example in Detail . . . 88
5.2.15 Properties of *S-CE* . . . 90
5.2.16 The Order of *S-CE* Rules . . . 94
5.2.17 Further Optimizations . . . 96

5.3 The Algorithm *CE* 98
5.3.1 The Core Component of *CE* 100
5.3.2 Auxiliary Functions used by *CE* 100
5.3.3 Examples of Calls to *CE* 103
5.3.4 Properties of *CE* 105

6 Conclusions and Future Work **109**

Bibliography **111**

A Proofs of Chapter 3 **113**
A.1 Proofs of Section 3.1 113
A.2 Proofs of Section 3.4 116

B Proofs of Chapter 4 **122**
B.1 Proofs of Section 4.2 122
B.2 Proofs of Section 4.3 136

C The Type Language for Scheme **144**
C.1 Base Types 144
C.1.1 The Boolean Types 145
C.1.2 The Symbol Types 145
C.1.3 The Number Types 145
C.1.4 The Character Types 148
C.1.5 Input/Output and Ports 148
C.1.6 *STbase* and *CEbase* on value assignments 149
C.1.7 Correctness of *STbase* and *CEbase* 149
C.2 Type Constructors 150
C.2.1 Pairs and lists 150
C.2.2 Strings 151
C.2.3 Vectors 151

Chapter 1

Introduction

When type inference and type checking for functional programs were considered in [Mil78] for the first time different types denoted disjoint sets of values. Type checking was 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 had 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. The type checkers based on $\sqsubseteq$ test for a function call $(f\ a)$ whether the type inferred for an argument a is a subtype of the expected input type of f. The test fails for all the cases where a contains at least one value f is not applicable to.
This sound approach to type checking can be used in the following ways:

- 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]).

For dynamically typed languages like e.g. Scheme [KCE98] a static type checker has the disadvantage that there is the possibility of rejecting programs that can be executed without

a type error. Soft typing on the other hand does not reject *any* programs even if they contain fatal type errors.

A new approach to type checking is a type checker that is *complete* in the sense that it accepts every well-typed program and that every rejected program must indeed contain a type error [WB00]. A promising scenario for using a complete type checker is a combination with a soft typing system or a system with an output similar to a soft typing system [Fla97, FF99]:

- Errors that are overlooked by the complete type checker are caught by the soft typing system and are presented as warnings.
- The complete type checker emphasizes a small set of errors that otherwise could be overlooked in a large number of type warnings given by the soft typing system.

Example 1.0.1 *Consider the function call c = (vector-ref v i) in the functional language Scheme 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 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 raises a warning. A complete type checker raises an error for c because this call cannot succeed with a negative number as vector index.*

As the example shows complete type checking allows a very powerful type language that e.g. divides intergers into positive intergers, negative integers and zero (not used in the example). For a sound type checker refining the expected input types of function definitions causes more erroneous warnings to be risen because the refined input types are not precisely met by the types of the arguments in every case.

In this work we define a type language that covers most typing constructs found in the literature and therefore allows very precise types. We give this definition in order to provide a powerful type language for Scheme, but we believe that the core of the type language can be carried over to different functional languages easily.

We furthermore 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 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.) Thus, CE turns out to be a main component of a complete type checker.

The rest of the work is organized as follows: In Chapter 2 some tools and techniques used throughout the work are summarized. The topics mentioned there are:

- Term rewriting systems: Besides summarizing results known from literature an extended syntax is presented that fits into the context of term constructors with variable arity.
- Graph theory: Especially strongly connected components in directed graphs are needed to express dependencies in type assignments.

Chapter 3 introduces the type language and some algorithms working on types: In Sec. 3.1 syntax and semantics of types are defined as usual. Furthermore a normalized form of set operators in types is introduced. Section 3.2 introduces so called value assignments, i.e. types denoting exactly one value. The goal of value assignments is to increase the precision of types available throughout type inference. In Sec. 3.3 we discuss problems caused by the usual function type constructor in our framework and give an alternative definition of function types. Section 3.4 finishes the introduction of types by defining a subtype relation on types. This relation is defined constructively by presenting an algorithm.

One of the main questions in a complete type checker is whether two given types (more precisely certain instances of them) have common elements or not. A formal definition of common elements and an algorithm approximating a solution to this question are presented in Chapter 5. After presenting some preliminaries in Sec. 5.1 the problem is divided into two subproblems: In Sec. 5.2 we describe how constraints on the instances of the types can be collected by recursively traversing the types. Section 5.3 explains how the collected sets of constraints can be transformed into idempotent substitutions.

The proofs of the theorems stated throughout the work are given in the appendix. Appendix A contains the proofs of Chapter 3 and App. B for Chapter 5.

Though the system presented here was designed generally enough to work on several purely functional languages it was primarily designed to work on programs written in Scheme [KCE98]. The general types used in the Chapters 3 and 5 can be specialized in order to fit the properties of Scheme. These specializations of the type language are done in App. C.

Chapter 2

Preliminaries

2.1 Term Rewriting Systems

Term rewriting systems as presented in this section provide a formalism for transforming terms. First we give the usual definition of terms in Subsec. 2.1.1 and summarize term rewriting systems and their main properties in Subsec. 2.1.2. Since we are going to apply term rewriting systems to terms containing constructors of variable arity, we introduce an appropriate term rewriting framework in Subsec. 2.1.3.

For introductory literature on term rewriting see e.g. [DJ90], [Jou95].

2.1.1 Terms

The set of terms over a given signature is defined as follows:

Definition 2.1.1 (terms) *Let $\mathcal{F}_n$ be a set of function symbols of arity n for all $n \in \mathbb{N}_0$ with $\mathcal{F}_n \cap \mathcal{F}_m = \emptyset$ for $n \neq m$, let $\mathcal{F} = \cup_{n \in \mathbb{N}_0} \mathcal{F}_n$ and let $\mathcal{X}$ be a set of variable symbols. The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ over $\mathcal{F}$ and $\mathcal{X}$ is the smallest set with:*

- *$x \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for all $x \in \mathcal{X}$.*
- *If $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for $i = 1, \ldots, n$ and $f \in \mathcal{F}_n$ then $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.*[1]

[1] For $n = 0$ we often write f instead of $f()$.

Terms can be transformed into other terms by instantiation and by selecting or updating subterms:

Definition 2.1.2 (positions and subterms) *Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be a term. A position is a list of integers: $p = i_1.i_2. \ldots .i_k.\epsilon$ with the empty list denoted by ϵ. The subterm of t at position p (denoted by $t_{|p}$) is defined as follows:*

- *$t_{|p} = t$ for $p = \epsilon$.*
- *If $t = f(t_1, \ldots, t_n)$ and $p = i_1.i_2. \ldots .i_k$ with $1 \leq i_1 \leq n$ then $t_{|p} = t'_{|p'}$ with $t' = t_{|i_1}$ and $p' = i_2. \ldots .i_k$.*
- *In all other cases $t_{|p} = \texttt{undefined}$.*

If t is a term and p a position with $t_{|p} \neq \texttt{undefined}$ then p is called a position in t.

When t is a term and p is a position in t then the update of t at p is defined as follows:

Definition 2.1.3 (term update at a position) *Let $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and let p be a position in t. The term update of t at p to t' (denoted by $t[p|t']$) is defined as follows:*

- *$t[\epsilon|t'] = t'$.*
- *$f(t_1, \ldots, t_n)[i_1.i_2. \ldots .i_k|t'] = f(t_1, \ldots, t_{i_1-1}, t'_{i_1}, t_{i_1+1}, \ldots, t_n)$ with the new subterm at position i_1 defined as $t'_{i_1} = t_{i_1}[i_2. \ldots .i_k|t']$.*

Note that $t[p|t']$ is undefined if $t_{|p}$ is undefined.

We can now define substitutions and instances of terms:

Definition 2.1.4 (substitutions, instances of terms) *A substitution is a function $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ that differs from the identity for a finite number of inputs only. σ can be extended to a function from terms to terms as follows: $\sigma(t) = t'$ where t' is generated from t by updating all these positions p of t with $t_{|p} \in \mathcal{X}$ and $\sigma(t_{|p}) \neq t_{|p}$ to $\sigma(t_{|p})$.*

2.1.2 Usual Term Rewriting Systems

A term rewriting system is given by a set of rewriting rules $l \to r$:

Definition 2.1.5 (term rewriting system) *Let $\mathcal{X}$ be a set of variables and $\mathcal{F}$ a set of function symbols. A term rewriting rule is a pair (l, r) usually written as $l \to r$ with $l, r \in \mathcal{T}(\mathcal{X}, \mathcal{F})$ fulfilling:*

- *$l \notin \mathcal{X}$.*
- *If $x \in \mathcal{X}$ occurs in r, i.e. if there is a position p in r with $r_{|p} = x$ then x also occurs in l.*

A term rewriting system is a set of term rewriting rules.

A term rewriting system R induces a rewrite relation $\to_R$ on $\mathcal{T}(\mathcal{X}, \mathcal{F})$:

Definition 2.1.6 (rewrite relation, normal form) *Let $\mathcal{X}$ be a set of variables, $\mathcal{F}$ a set of function symbols and R a term rewriting system. The rewrite relation $\to_R$ induced by R is the binary relation containing all pairs (t, t') of terms (written $t \to_R t'$) with the following property: There is a variable-renamed instance of a rewrite rule $l \to r \in R$ (not containing any variables occurring in t)[2], a substitution σ just changing variables in l and r and a position p in t such that $\sigma(l) = t_{|p}$ and $t' = t[p|\sigma(r)]$.*

If $\to_R$ is a rewrite relation then its reflexive, transitive closure is denoted by $\to_R^*$.

A term t with $t \not\to_R t'$ for every term t' is called normal form (with respect to R).

For a term rewriting system termination and confluence are the main properties. Termination guarantees that applying a term rewriting system R to a term t yields a term t' in normal form after a finite number of steps:

Definition 2.1.7 (termination of term rewriting systems) *Let R be a term rewriting system. R is* terminating *if every chain $t_0 \to_R t_1 \to_R t_2 \to_R \ldots$ leads to a term t_k with $t_k \not\to_R t$ for every term t after a finite number of steps.*

For a given term t there might be different positions rules from R are applicable to or different rules might by applicable. The confluence of R states that whenever t can be transformed into different terms t_1 and t_2 by several steps of R there must be a term t' both t_1 and t_2 can be transformed to:

Definition 2.1.8 (confluence of term rewriting systems) *Let R be a term rewriting system. R is* confluent *if for all terms t with $t \to_R^* t_1$ and $t \to_R^* t_2$ there is a term t' with $t_1 \to_R^* t'$ and $t_2 \to_R^* t'$.*

[2] This situation can always be achieved by renaming the variables in $l \to r$.

Unfortunately termination and confluence are undecidable properties. But for a terminating term rewriting system R confluence is equivalent to local confluence and therefore decidable [KB67]. Local confluence is defined as follows:

Definition 2.1.9 (local confluence of term rewriting systems) *Let R be a term rewriting system. R is* locally confluent *if for all terms t with $t \to_R t_1$ and $t \to_R t_2$ there is a term t' with $t_1 \to_R^* t'$ and $t_2 \to_R^* t'$.*

An algorithm for checking local confluence provided in [KB67] just has to check critical pairs:

Definition 2.1.10 (critical pairs) *Let $r = l_1 \leftarrow r_1$ and $r' = l_2 \leftarrow r_2$ be rewriting rules in R not containing a common variable. Let p be a position in l_1 such that:*

- $(l_1)_{|p} \notin \mathcal{X}$.
- *There is a substitution σ with $\sigma((l_1)_{|p}) = \sigma(l_2)$. ($\sigma$ should be the most general unifier, i.e. σ is not more restrictive than necessary in order to fulfill this property.)*

Then $(\sigma(r_2), \sigma(l_1[p|r_1]))$ is a critical pair.

A term rewriting system R is locally confluent if all critical pairs have a common normal form.

2.1.3 Extended Term Rewriting Syntax

The terms we want to transform using term rewriting systems differ from the definition in Subsec. 2.1.1 in the fact that our function symbols need not have a fixed arity. For example, we will use unions, intersections and Cartesian products of arbitrary arities. We want to transform terms using these constructors independently of the arity a function symbol actually occurs with in a given term.

In principle such function symbols can be understood as function symbols of arity 2 with associativity as additional property. Methods developed in [PS81], [JK86] can be used to express associativity. But since parts of this work spots on functional programming languages with operators of variable arity (like e.g. Scheme [KCE98]) we prefer a notion that corresponds to the understanding of function symbols with variable arity.

The notions presented in the following are a subset of syntactic constructs presented in [WB99]. They are realized in the interpreter for term rewriting systems described in [Fro98].

For processing function symbols of variable arity we introduce the following notions:

- For $i \in \{1, 2, \dots\}$ the pattern $<i\dots>$ represents an argument list of variable arity. A pattern $<i\dots>$ is always instantiated with the sequence of its elements without parentheses. When showing substitutions in examples we will use square brackets [] to make the start and the end of a list visible. The concatenation of lists is just written as $<i\dots><j\dots>$. Note that the whole expression $<i\dots>$ is handled as a single syntactic keyword. Several of these expressions can occur in one rule and have to be distinguished from each other by i. i is called *list counter*. Two occurrences $<i\dots>$ and $<j\dots>$ with $i = j$ denote the same list.

  Example:

  $$(+\ \ <1\dots>\ \ (+\ \ <2\dots>)\ \ <3\dots>) \rightarrow (+ <1\dots> <2\dots> <3\dots>)$$

  transforms a nested sum (e.g. $(+\ a\ b\ c\ (+\ d\ e)\ f))$ to a single sum containing all elements of the original sum in the same order (e.g. $(+\ a\ b\ c\ d\ e\ f))$. The one step reduction of $(+\ a\ b\ c\ (+\ d\ e)\ f)$ in the given example is achieved by the pattern matching $\{<1\dots>\leftarrow [a, b, c], <2\dots>\leftarrow [d, e], <3\dots>\leftarrow [f]\}$. The nested sum $(+\ a\ b\ (+\ c\ (+\ d\ e))\ f)$ of depth 2 can be flattened either by the sequence

  $$\begin{aligned} &(+\ a\ b\ (+\ c\ (+\ d\ e))\ f) \\ \rightarrow &(+\ a\ b\ c\ (+\ d\ e)\ f) \\ \rightarrow &(+\ a\ b\ c\ d\ e\ f) \end{aligned}$$

  or by the sequence

  $$\begin{aligned} &(+\ a\ b\ (+\ c\ (+\ d\ e))\ f) \\ \rightarrow &(+\ a\ b\ (+\ c\ d\ e)\ f) \\ \rightarrow &(+\ a\ b\ c\ d\ e\ f)\,. \end{aligned}$$

- The pattern $<a_i\dots e_i>$ is used instead of $<i\dots>$ if the individual list elements are changed in a uniform manner. $<a_i\dots e_i>$ is a syntactical keyword, too, where the reader can think of a_i as the first list element and of e_i as the last one. There is only one common list counter i for $<i\dots>$ and $<a_i\dots e_i>$, and constructs of different kinds must not share the same index.
  There is an alternative form $<t(a_i)\dots t(e_i)>$ where $t(x)$ stands for any term containing a variable x. This form expresses a list where all elements are instances of the pattern $t(x)$. The pattern $<t(a_i)\dots t(e_i)>$ matches every list $(v_1\ v_2\dots v_k)$ where each v_j is of the form $t(u_j)$, i.e.

  $$\forall j \in \{1, \dots, k\}\, \exists u_j\,.\, v_j = \sigma_j(t(x)) \text{ with } \sigma_j = \{x \leftarrow u_j\}$$

  Please note the following correspondence between the patterns $<t(a_i)\dots t(e_i)>$ and $<a_i\dots e_i>$: If some substitution τ assigns the list $(v_1\ v_2\dots v_k)$ to $<t(a_i)\dots t(e_i)>$

it implicitly contains the assignment of $(u_1\ u_2 \ldots u_k)$ to $< a_i \ldots e_i >$ as well as the assignment of $(v'_1\ v'_2 \ldots v'_k)$ $(v'_j := \sigma_j(t'(x)))$ to $<t'(a_i) \ldots t'(e_i)>$ and vice versa. This ensures that all patterns $< a_i \ldots e_i >$ and $< t'(a_i) \ldots t'(e_i) >$ occurring on the right hand side of a rule are well-defined whenever any of the patterns $<t(a_i) \ldots t(e_i)>$ or $<a_i \ldots e_i>$ occurs on the left hand side of the rule.

Example: The rule

$$(*\ (+\ \ <a_1 \ldots e_1>)\ t) \rightarrow (+\ \ <(*\ a_1\ t) \ldots (*\ e_1\ t)>)$$

eliminates sums inside of products by distributivity: E.g. it transforms the input term $(*\ (+\ t_1\ t_2\ t_3)\ t')$ in one step to $(+\ (*\ t_1\ t')\ (*\ t_2\ t')\ (*\ t_3\ t'))$ with the substitution $\{<a_1 \ldots e_1> \leftarrow [t_1, t_2, t_3], t \leftarrow t'\}$.
As in the example, a rule usually contains some $< a_i \ldots e_i >$ together with the corresponding $<t(a_i) \ldots t(e_i)>$, one on each side of the rule.

Termination and confluence of a term rewriting system using the extended syntax can be checked by the following observations:

- Every rewriting rule using the extended syntax represents a rule scheme of (usually an infinite number of) standard rewriting rules. For an extended term rewriting system R we denote the set of represented standard rewriting rules by $\varphi(R)$.
- An reduction step in the extended TRS R is possible iff it is possible in the represented standard TRS $\varphi(R)$. It is therefore sufficient to check $\varphi(R)$ for termination and local confluence.
- Since the represented standard rules steaming from one extended rule process the given terms uniformly it is sufficient to consider a finite subset of the rules in $\varphi(R)$ in order to find a termination ordering or critical pairs. (The maximal number of arguments to a single function can be calculated from R.)

Thus, while providing a convenient way of dealing with function symbols of arbitrary arity, the extended syntax for TRS represented in this section can be reduced completely to standard TRS techniques [WB01].

2.2 Graph Theory

This section gives an overview over directed graphs and operations to be performed on them. The main notions used in graph theory can be found in [GLS93].

Definition 2.2.1 (directed graph) *A* directed graph *is a pair $G = (V, R)$ with a set V of nodes and a set $R \subseteq V \times V$ of arcs $r = (v, v')$ where v is called the* initial node *of v and v' the* terminal node*.*
For an arc r its initial node is denoted by $\alpha(r)$ and its terminal node by $\omega(r)$.
An arc r is called a loop if $\alpha(r) = \omega(r)$.

For a given graph we are often not only interested in the relation R directly but in elements of its transitive closure. This motivates the following definition of paths:

Definition 2.2.2 (paths and circuits in a directed graph) *Let $G = (V, R)$ be a directed graph. A path in G is a finite sequence $w = r_1, r_2, \ldots, r_k$ with $r_i \in R$ for all $i = 1, \ldots, k$ and $\omega(r_i) = \alpha(r_{i+1})$ for $i = 1, \ldots, k-1$.*
We extended α and ω to paths $w = r_1, \ldots, r_k$ as follows: $\alpha(w) = \alpha(r_1)$ and $\omega(w) = \omega(r_k)$.
A path w is called a circuit if $\alpha(w) = \omega(w)$.
When $w = r_1, \ldots, r_m$ and $w' = r'_1, \ldots, r'_n$ are paths in G with $\omega(w) = \alpha(w')$ then the concatenation of w and w' is $w \circ w' = r_1, \ldots, r_m, r'_1, \ldots, r'_n$.

Definition 2.2.3 (connected nodes, strongly connected components) *Let $G = (V, R)$ be a directed graph and let $v, v' \in V$. We say v is* connected *to v' (written $v \equiv v'$) if either $v = v'$ or there exist paths w and w' with $\alpha(w) = \omega(w') = v$ and $\alpha(w') = \omega(w) = v'$.*
$\equiv$ is an equivalence relation. The equivalence classes are called the strongly connected components *of G.*

Definition 2.2.4 (component graph) *Let $G = (V, R)$ and let $ZK_1, \ldots, ZK_k$ be the strongly connected components of G. The* component graph *of G is*

$$
\begin{aligned}
&G' = (V', R') \text{ with} \\
&V' = \{ZK_1, \ldots, ZK_k\} \text{ and} \\
&R' = \{(ZK, ZK') \in V' \times V' \mid ZK \neq ZK' \land \exists v \in ZK, v' \in ZK' \,.\, (v, v') \in R\}\,.
\end{aligned}
$$

The component graph is especially useful in the following sense: Every graph that does not contain a circuit visiting more than one node induces a partial ordering on the set of nodes. Two nodes $m, n \in V$ with $m \neq n$ fulfill $m < n$ if there exists a path from m to n in G.
When a graph G contains a circuit visiting more than one node (this is equivalent to the fact that G has at least one strongly connected component consisting of more than one node) a partial ordering can no longer be defined in the same manner. In this case there is still some ordering information given in G because there is a partial ordering on the strongly connected components of G that is induced by the component graph of G (see [CLR90, Ex. 23.5-4]) in the manner stated before.

Chapter 3

The Type Language

In this chapter we define the type language used throughout this work and certain operations on types. Types in this work can essentially be understood as sets of values. The definition of the type language is given in two steps: Section 3.1 introduces the standard types. The concept of standard types is similar to the usual approach of types. In Section 3.2 the concept of types is extended by value assignments. They introduce types containing exactly one constant value. The goal of introducing value assignments is a uniform framework of types which also allows the use of values directly instead of generalizing them to types. Section 3.3 introduces an external representation for function types that is special to this approach due to the non standard requirements of complete type inference. An algorithm approximating a subtype hierarchy is presented in Sec. 3.4.

The type language presented in this chapter is a general one that is appropriate to different functional programming languages. In order to adapt it to a certain programming language some definitions have to be refined and instantiated. For Scheme [KCE98] this is done in App. C. The instantiations given in this appendix are already used for examples throughout this chapter.

3.1 The Standard Types

Standard types are terms built from base types, type constructors and type variables. In the following we will formally define these components of standard types.

3.1.1 Syntax of the Standard Types

This section gives a definition of type terms (types for short). Types are built from a set of base types, a set of type constructors, type variables, and syntactic constructs for binding type variables.

The following definition Def. 3.1.1 introduces the set of all standard types for a given set of variables V. The set $V = V_f \cup V_q$ is divided into sets V_f of free variables and V_q of quantified variables where free variables can be instantiated but the quantified ones must not. To distinguish free and quantified variables easily quantified variables are denoted by a subscript $\forall$, e.g. $A_\forall, B_\forall, \ldots$.
The different type constructors are explained in detail later.

Definition 3.1.1 (standard types) *Let $V = V_f \cup V_q$ be a set of type variables. The set $\mathcal{T}_S(V)$ of all standard type terms (or standard types for short) over V is defined as the smallest set fulfilling the following properties:*

1. *$\mathcal{B} \subseteq \mathcal{T}_S(V)$ where $\mathcal{B}$ is the set of all type constants or base types introduced in Def. 3.1.12. Especially $\texttt{sym} \in \mathcal{B}$.*
2. *$V \subseteq \mathcal{T}_S(V)$.*
3. *$c \in \mathcal{K}, e_1, \ldots, e_{a(c)} \in \mathcal{T}_S(V) \Rightarrow (c\ e_1\ \ldots\ e_{a(c)}) \in \mathcal{T}_S(V)$ where $\mathcal{K}$ is the set of type constructors as introduced in Def. 3.1.13.*
4. *$e_1, \ldots, e_{k \geq 0} \in \mathcal{T}_S(V) \Rightarrow (\cup\ e_1\ \ldots\ e_k), (\cap\ e_1\ \ldots\ e_k) \in \mathcal{T}_S(V)$.*
5. *$e \in \mathcal{T}_S(V)$, e does not contain a subterm $e' \in V \Rightarrow \mathcal{C}e \in \mathcal{T}_S(V)$.*
6. *$e_1, e_2 \in \mathcal{T}_S(V), e_2$ does not contain a subterm $e' \in V \Rightarrow e_1 \setminus e_2 \in \mathcal{T}_S(V)$.*
7. *$Tfunc, Tfunc_P, Tfunc_U \in \mathcal{T}_S(V)$ with the function types $Tfunc, Tfunc_P, Tfunc_U$ given in Def. 3.1.16.*
8. *$s \in \texttt{sym}, t \in \mathcal{T}_S(V) \Rightarrow (bind\ s\ t) \in \mathcal{T}_S(V)$ with the binding type constructor bind described in Def. 3.1.17.*
9. *$u_i = (bind\ s_i\ t_i) \in \mathcal{T}_S(V), s_i \neq s_j$ for $i \neq j \Rightarrow (frame\ u_1\ \ldots\ u_n) \in \mathcal{T}_S(V)$ with the frame type constructor frame given in Def. 3.1.17.*
10. *e is an environment type and f a frame type $\Rightarrow (env\ e\ f) \in \mathcal{T}_S(V)$ with the environment type constructor env introduced in Def. 3.1.19.*

11. *If $X \in V_f$ is a free variable and $t \in \mathcal{T}_S(V)$ contains X then $\mu X.t \in \mathcal{T}_S(V)$. The recursive type constructor is explained in detail in Def. 3.1.21.*

An important restriction is made by (5) and (6): The argument of the complement type constructor must not contain any type variables. This is also true for the second argument of the difference type constructor that is internally represented using complements.

Definition 3.1.2 ((semi-)closed/ground standard types) *The set $\mathcal{T}_{SCS}(V) \subset \mathcal{T}_S(V)$ if semi-closed standard types consists of all type not containing any free type variables $X \in V_f$ not bound by μ.*
The set $\mathcal{T}_{CS}(V) \subset \mathcal{T}_{SCS}(V)$ of closed standard types *consists of all semi-closed standard types not containing any quantified type variables $X_\forall \in V_q$. The set $\mathcal{T}_{GS} \subset \mathcal{T}_{CS}(V)$ of* ground standard types *consists of all types not containing any variables.*

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

With the set of standard type terms defined we can now define the notion of positions in standard types and subterms at a given position. Most parts of this definition are standard except of recursive types and environment types: Environment types are handled as lists of frames. Instead of choosing the 1st subterm $j-1$ times and the 2nd subterm once in order to get the jth frame one can specify the jth subterm of the environment directly. When the second subterm of a recursive type is selected this implicitly performs an unfolding step.

Definition 3.1.3 (positions and subterms at positions) *A position in a term is a list p of the form $p = p_1.p_2. \ldots .p_k$ with $p_i \in \mathbb{N}$. The empty list is denoted by ϵ. For a type $t \in \mathcal{T}_S$ and a position p the subterm of t at p, written $t_{|p}$, is either $t_{|p} =$ **`undefined`** or $t_{|p} \in \mathcal{T}_S$ defined as follows:*

- $t_{|\epsilon} = t$.
- *For $k \geq 1$, $t_{|p} = t'_{|p'}$ with $p' := p_2. \ldots .p_k$. and t' calculated as follows:*
  - *If $t = (\cup\ t_1\ \ldots\ t_j)$, $t = (\cap\ t_1\ \ldots\ t_j)$ or $t = (c\ t_1\ \ldots\ t_j)$ with $c \in \mathcal{K}$ and if $p_1 \in \{1, \ldots, j\}$ then $t' = t_{|p_1}$.*
  - *If $t = t_1 \setminus t_2$ and $p_1 \in \{1, 2\}$ then $t' = t_{|p_1}$.*
  - *If $t = \mathcal{C}\tilde{t}$ and $p_1 = 1$ then $t' = \tilde{t}$.*
  - *If $t = (bind\ s\ \tilde{t})$ then:*
    - *If $p_1 = 1$ then $t' = s$.*
    - *If $p_1 = 2$ then $t' = \tilde{t}$.*

- *If* $t = (frame\ u_1\ u_2\ \dots\ u_j)$ *and* $p_1 \in \{1, \dots, j\}$ *then* $t' = u_{|p_1}$.
- *If* $t = (env\ e\ f)$ *then:*
  - *If* $p_1 = 1$ *then* $t' = f$.
  - *If* $p_1 > 1$ *and* $e_{|(p_1-1)} \neq$ ***undefined*** *then* $t' = e_{|(p_1-1)}$.
- *If* $t = \mu X.\tilde{t}$ *then:*
  - *If* $p_1 = 1$ *then* $t' = X$.
  - *If* $p_1 = 2$ *then* $t' = \tilde{t}[X/t]$, *i.e. the type generated from* $\tilde{t}$ *by unfolding it (replacing every free occurrence of* X *by* t*).*
- *If* p_1 *is greater than the number of direct subterms of* t *or if* $t =$ ***undefined*** *then* $t' :=$ ***undefined****.*

For expressing a position p in a term $t = t'_{|p'}$ in terms of t' we need the concatenation of positions:

Definition 3.1.4 (concatenation of positions) *Let* $p = p_1.p_2.\dots.p_m.$ *be a position in* t *and* $p' = p'_1.p'_2.\dots.p'_n$ *a position in* $t_{|p}$. *The concatenation of* p *and* p' *defined by* $p.p' = p_1.p_2.\dots.p_m.p'_1.p'_2.\dots.p'_n$ *is a position in* t.

The set of all subterms of a given term t can now be formalized as follows:

Definition 3.1.5 (set of all subterms) *For a given term* t *the set of all subterms of* t *is defined as*

$$subterms(t) = \{t' \mid \exists p\,.\,t' = t_{|p} \neq \textbf{\textit{undefined}}\}\,.$$

Note that in the presence of recursive type constructors the set of all subterms has some unusual properties:

Remark 3.1.6 (properties of the set of all subterms) *Let* $t = \mu X.\tilde{t}$ *with* X *occurring freely in* $\tilde{t}$. *Then*

$$subterms(t) = subterms(t_{|2})$$

because especially $t \in subterms(t_{|2})$ *and therefore* $subterms(t_{|2})$ *also contains all subterms of* t.

$subterms(t)$ *has a finite cardinality because after the first unfolding of every recursive type constructor in* t *there is just a finite number of subterms to generate without further unfoldings and every further unfolding step yields a term already seen before.*

Example 3.1.7 (set of all subterms) *Consider the type of all lists of argument type A:*

$$t = \mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X))$$

This term has the following subterms:

$$\begin{aligned} t_{|\epsilon} &= \mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X)) \\ t_{|1} &= X \\ t_{|2} &= (\cup\ \boldsymbol{nil}\ (A\ .\ t)) \\ t_{|2.1} &= \boldsymbol{nil} \\ t_{|2.2} &= (A\ .\ t) \\ t_{|2.2.1} &= A \\ t_{|2.2.2} &= t \end{aligned}$$

Calculating subterms of $t_{|2.2.2}$ does not yield any new terms because $t_{|2.2.2} = t$. Therefore

$$subterms(t) = \{\mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X)), X, (\cup\ \boldsymbol{nil}\ (A\ .\ t)), \boldsymbol{nil}, (A\ .\ t), A\}\,.$$

As the following examples states defining the proper subterms of a term t as usual is not sufficient:

Example 3.1.8 (proper subterms (1)) *Consider the term t given in Ex. 3.1.7.*

- *Usually one can define the set of all proper subterms of t as the set of all subterms of t at a position $p \neq \epsilon$. As Ex. 3.1.7 shows for $t = \mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X))$ we get $t_{|2.2.2} = t$ as proper subterm of t.*
- *In a different definition the set of all proper subterms of t is the set of all subterms of t except t itself. The relation of proper subterms given by this definition has the following properties:*
  - *It is not antisymmetric because $t = t_{|2.2.2}$ is a proper subterm of $t_{|2.2} = (A\ .\ t)$ and $(A\ .\ t) = t_{|2.2}$ is a proper subterm of t.*
  - *It is not transitive because in the case of transitivity from t, $t_{|2.2}$ and $t_{|2.2.2} = t$ we get that $t = \mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X))$ must be a proper subterm of itself.*

The following definition yields a relation of proper subterms without the disadvantages stated above:

Definition 3.1.9 (proper subterm) *Let t be a term. A term t' is a proper subterm of t if one of the following conditions holds:*

1. *t' is a direct proper subterm of t, i.e. $t' = t_{|i}$ with $i \in \mathbb{N}$ and with $i \neq 2$ or t is not a recursive type.*

2. *t' is a proper subterm of a direct proper subterm of t type.*

By not allowing unfolding steps in calculating proper subterms we circumvent the problems stated before as the following example shows:

Example 3.1.10 (proper subterms (2)) *Reconsidering the terms of Ex. 3.1.7 we get:*

- *The term $t = \mu X.(\cup\ \mathtt{nil}\ (A\ .\ X))$ just has X as proper subterm.*
- *For the term $t' = (\cup\ \mathtt{nil}\ (A\ .\ t))$ with t as above we get $\{\mathtt{nil}, (A\ .\ t), t, X\}$ as the set of proper subterms.*

We can now formalize the update of a type term at a position.

Definition 3.1.11 (term update at a position) *Let t be a type term, p a position and $t_{|p} \neq$ **undefined**. Then $t[p|t']$ is the* term update *of t* at position *p to t', i.e. the term generated from t by replacing $t_{|p}$ by t'.*

The following definitions will introduce the different syntactic components used to build types in Def. 3.1.1.

Definition 3.1.12 (base types) *The set $\mathcal{B} = \{\perp, \mathcal{O}, \mathtt{sym}, b_1, \ldots, b_v\}$ for some $v \in \mathbb{N}$ is called the set of* base types *or* type constants. *$\perp$ is called the* empty type*, $\mathcal{O}$ is called the* zero type*, and* **sym** *is called the type of all symbols.*

Definition 3.1.13 (free type constructors) *The set $\mathcal{K} = \{c_1, \ldots, c_w\}$ for some $w \in \mathbb{N}$ is called the set of free type constructors or tuple like type constructors. Every free type constructor $c_i \in \mathcal{K}$ has a fixed arity $a(c_i)$.*

Some base types and free type constructors used in the following are given in Ex. 3.1.14. A full set of base types and type constructors can be found in App. C.

Example 3.1.14 (base types and free type constructors) *The set $\mathcal{B}$ of base types contains the types* **bool** *of boolean values,* **int** *of integer values,* **posint** *of positive integer values and* **string** *of string values.*
The set $\mathcal{K}$ of type constructors contains the pair type constructor $(\cdot\ .\ \cdot)$.

Definition 3.1.15 (set oriented type constructors) *The following type constructors are called set oriented type constructors:*

- $\cup$ *with arbitrary arity* $a(\cup) \in \mathbb{N}_0$ *is called the* union type constructor*.*
- $\cap$ *with arbitrary arity* $a(\cap) \in \mathbb{N}_0$ *is called the* intersection type constructor*.*
- $\setminus$ *with* $a(\setminus) = 2$ *is called the* difference type constructor*.*
- $\mathcal{C}$ *with* $a(\mathcal{C}) = 1$ *is called the* complement type constructor*.*

The typing of functions in our approach is done differently from the usual approach. Usually one defines a function type constructor $A \rightarrow B$ that expresses the type of functions mapping the values in A to values in B. This function type constructor is anti-monotonic in its first element. Thus, the domain type A usually denotes a subtype of the exact domain of a given function. (An exception is given by certain partial functions like e.g. the division operator /. It usually contains the value 0 in the second argument position of its domain type, but is not defined for the value 0 in the second argument.)

In our approach it is important to express *all* values a function is applicable to, i.e. we need a supertype of the exact domain. Since this would cause problematic properties of the function type constructor (neither monotonic nor anti-monotonic in the first argument) we just introduce a type of all functions with two subtypes that distinguish user defined from predefined functions.
More precisely when talking about functions we mean *function definitions*, i.e. internal definitions of predefined functions inside an interpreter and user defined function definitions given in a program's source code. In the following we will use *functions* instead of *function definitions* again.

Definition 3.1.16 (function types) *The type Tfunc is called the function type. The two restricted function types* $Tfunc_P$ *and* $Tfunc_U$ *are called the type of predefined functions and the type of user defined functions, respectively.*

The function types described here do not carry much information compared to the usual function types in functional type systems. In fact we expect a complete type checker or type inference system using the type system described here to use value assignments (i.e. types whose semantics contains exactly one value) as described in Sec. 3.2 for functions very often. An external representation providing information similar to usual function types is presented in Sec. 3.3.

In the presented type language environments are handled as first class objects. Thus, we need a type constructor to define types of environments. This constructor is given by the following definitions:

Definition 3.1.17 (symbol bindings and frame types) bind *is called the binding type constructor. The generated symbol bindings have the form* (*bind* s t) *where* $s \in$ `sym` *and* t *is*

a type.
frame *is called the frame type constructor. The frame types generated by this constructor have the form* $(\mathit{frame}\ u_1\ \ldots\ u_k)$ *for every* $k \in \mathbb{N}$ *where the* $u_i = (\mathit{bind}\ s_i\ t_i)$ *are symbol bindings of pairwise disjoint symbols* s_i*. For*

$$(\mathit{frame}\ (\mathit{bind}\ s_1\ t_1)\ \ldots\ (\mathit{bind}\ s_k\ t_k))$$

we often use the notation

$$(\mathit{frame}\ s_1 : t_1\ \ldots\ s_k : t_k)\,.$$

frame *is an abbreviation for the type of all frames.*

A frame F can be understood as a partial function mapping symbols to types. The function given by $F = (\mathit{frame}\ s_1 : t_1\ \ldots\ s_k : t_k)$ is defined exactly for the symbols $s_1, \ldots, s_k$ and maps every s_i to t_i. Especially the order of the s_i occurring in the frame type definition does not matter.

Instead of $(\mathit{frame}\ s_1 : t_1\ \ldots\ s_k : t_k)$ we often write frame types as $[s_1 \mapsto t_1, \ldots, s_k \mapsto t_k]$.

Example 3.1.18 (frame types) *Examples for frame types are*

$$\begin{aligned} F_1 &= (\mathit{frame}\ x : \mathtt{int}\ s : \mathtt{string}\ f : \mathit{Tfunc}) \\ F_2 &= (\mathit{frame}\ x : \mathtt{posint}\ y : \mathtt{posint}) \end{aligned}$$

or in the alternative notation

$$\begin{aligned} F_1 &= [x \mapsto \mathtt{int}, s \mapsto \mathtt{string}, f \mapsto \mathit{Tfunc}] \\ F_2 &= [x \mapsto \mathtt{posint}, y \mapsto \mathtt{posint}] \end{aligned}$$

Definition 3.1.19 (environment types) *Environment types are either the empty environment type* $\mathtt{env}_0$ *or are generated by the environment type constructor* env *and have the form* $(\mathit{env}\ f\ e)$ *where* e *is an environment type and* f *is a frame type.*

The type of all environments is denoted by ***env****.*

By Def. 3.1.19 all environment types have the form

$$(\mathit{env}\ F_1\ (\mathit{env}\ F_2\ \ldots (\mathit{env}\ F_n\ \mathtt{env}_0) \ldots))$$

with frame types $F_1, \ldots, F_n$. They are generated from frame types as lists of frames with *env* behaving as the cons-operator and $\mathtt{env}_0$ as list terminator. We introduce the additional notation

$$(F_1\ F_2\ \ldots\ F_n)$$

for the environment

$$(env\ F_1\ (env\ F_2\ \ldots(env\ F_n\ \texttt{env}_0)\ldots))$$

In the context of abstract interpretation we will refer to environment types as *simple abstract environments* and use the list notation introduced above.

Example 3.1.20 (environment types) *An example for an environment type is*

$$E = (env\ F_1\ (env\ F_2\ \boldsymbol{env}_0))$$

where F_1 and F_2 are as in Ex. 3.1.18. In the notation of simple abstract environments we just write the list

$$E = (F_1\ F_2)$$

When a type t has to be defined in a recursive manner i.e. t is defined by a term containing itself as subtype the recursive type constructor μ has to be used.

Definition 3.1.21 (recursive types) *Recursive types are defined by the recursive type constructor μ and are of the form $\mu X.t$ where $X \in V_f$ is a free type variable and t is a type containing X.*

Example 3.1.22 (recursive types) *The classic example for recursive types are lists generated recursively from pairs: For every element type A the type of all lists with elements of type A is defined as*

$$\mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X))$$

where ***nil*** *stands for the type just containing the empty list* ().[1]

As stated in Def. 3.1.3 calculating the second subterm of a recursive type term $\mu X.t$ yields a term t' generated from a subterm of t by certain syntactical changes. This kind of change is called unfolding. The following definition provides an operation *unfold* that explicitly performs such changes on a given term:

Definition 3.1.23 (unfolding recursive bindings) *Let $t \in \mathcal{T}_S$ 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.*

[1] A different way of representing the type of the empty list is given by the value assignments presented in Sec. 3.2.

- *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 $t'_{|p} = t_{|2.p}$.

The sequence of unfoldings of t is defined by:

$$\begin{aligned}
\mathit{unfold}^0(t) &= t \\
\mathit{unfold}^1(t) &= \mathit{unfold}(t) = \mathit{unfold}(\mathit{unfold}^0(t)) \\
&\vdots \\
\mathit{unfold}^k(t) &= \mathit{unfold}(\mathit{unfold}^{k-1}(t))
\end{aligned}$$

Example 3.1.24 (unfolding a recursive type) *The first unfolding results of the type of lists with element type A are given as follows:*

$$\begin{aligned}
\mathit{unfold}^0(\mu X.(\cup\ \texttt{nil}\ (A\ .\ X))) &= \mu X.(\cup\ \texttt{nil}\ (A\ .\ X)) \\
\mathit{unfold}^1(\mu X.(\cup\ \texttt{nil}\ (A\ .\ X))) &= (\cup\ \texttt{nil}\ (A\ .\ \mu X.(\cup\ \texttt{nil}\ (A\ .\ X)))) \\
\mathit{unfold}^2(\mu X.(\cup\ \texttt{nil}\ (A\ .\ X))) &= (\cup\ \texttt{nil}\ (A\ .\ (\cup\ \texttt{nil}\ (A\ .\ \mu X.(\cup\ \texttt{nil}\ (A\ .\ X))))))
\end{aligned}$$

Note that every unfolding step affects exactly those recursive subterms in a type that are not a proper subterm of another recursive type: When several recursive bindings occur in a nested manner every unfolding step unfolds just the outermost binding. If on the other hand a type contains several recursive bindings that are not nested these bindings are unfolded in parallel.

Definition 3.1.25 (type of all values) *The type of all values is denoted by $\top$ and is defined as*

$$\top := \mu X.(\cup\ \mathcal{B} \bigcup_{c \in \mathcal{K}} (c\ \underbrace{X\ X\ \ldots\ X}_{a(c)})\ \texttt{frame}\ \texttt{env}\ \mathit{Tfunc})$$

3.1.2 Semantics of the Standard Types

Defining the semantics for most types is straight forward. But since our type language contains the recursive type constructor a special kind of values called cyclic values occurs. Example 3.1.26 shows how these values can be used before we formally define the semantic domain of types in Def. 3.1.27.

Example 3.1.26 (days of week) *Consider a calendar tool that cycles the sequence of week days. A first way to provide the week days could be the following list:*[2]

$$l := (\text{“mon”} \; . \; (\text{“tue”} \; . \; (\text{“wed”} \; . \; (\text{“thu”} \; . \; (\text{“fri”} \; . \; (\text{“sat”} \; . \; (\text{“sun”} \; . \; ())))))))$$

For switching from Sunday to Monday we need a test whether the list of week days is empty and a Further symbol holding the whole list to start over again.

The situation becomes much easier when a list l' can contain itself as a sublist. Analogously to recursive types the following notion is used in the example:

$$l' := \mu v.(\text{“mon”} \; . \; (\text{“tue”} \; . \; (\text{“wed”} \; . \; (\text{“thu”} \; . \; (\text{“fri”} \; . \; (\text{“sat”} \; . \; (\text{“sun”} \; . \; v)))))))$$

Now traversing the list l' yields an infinite sequence of week days.

The semantics of closed types is defined by a semantic function $\langle\!|\cdot|\!\rangle_c : \mathcal{T}_c \rightarrow \mathcal{P}(\mathcal{V})$ where $\mathcal{V}$ denotes the semantic domain, i.e. the set of all values expressible in a functional language. $\mathcal{V}$ is defined as follows:

Definition 3.1.27 (semantic domain of types) *The semantic domain $\mathcal{V}$ of types consists of the following (pairwise disjoint) subsets:*

- *A set $\mathcal{V}_S$ of* simple values.
- *Sets $\mathcal{V}_c = \{(c \; v_1 \; \ldots \; v_k) \mid v_i \in \mathcal{V}\}$ for every type constructor c with arity k. Thus, every tuple like type constructor corresponds to a free data constructor of the same arity.*
- *PFunc is the set of predefined functions.*
- *LC (lambda closures) is the set of user defined functions.*
- *Frame is the set of frames, i.e. the set of all functions mapping a finite number of symbols to values.*
- *Env is the set of environments, i.e. the set containing exactly the element* **`emptyenv`** *and all pairs $(e' \; f)$ with $e' \in Env$ and $f \in Frame$.*
- *The set $\mathcal{V}_{rec}$ contains values with a cyclic definition. Cyclic values are written in the form $v = \mu x.v'$ where v' is a value that contains v (denoted as x) as a component of a data constructor or as the value bound to a symbol in a frame.* [3]

[2] We use the same notation for data constructors as for type constructors.

[3] We use the same constructor μ for both recursive types and cyclic values. This should not cause confusion since usually the context implies whether a term is a type or a value.

Example 3.1.28 (cyclic value) *The standard example for cyclic values is given by structures as e.g. nested pairs containing themselves as arguments:*

$$v := \mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))$$

We can define positions in value terms and subterms at a position analogously to type terms (where the elements of *PFunc* and *LC* have no subtypes at positions other than ϵ). The cyclic data constructor μ behaves identically to the recursive type constructor when selecting the second subterm.

We can now define the unfolding of recursive values analogously to recursive types:

Definition 3.1.29 (unfolding of values) *Let $v = \mathcal{V}$ be a value. The operation $unfold_V$ transforms v to a value v' as follows: $unfold_V(v) = v'$ where v and v' differ exactly on the positions p fulfilling:*

- *$v_{|p} = \mu x.\tilde{v}$ is a cyclic value.*
- *There is no proper prefix p' of p such that $v_{|p'}$ is a cyclic value, i.e. $v_{|p}$ is not a subterm of any cyclic value except of itself.*

The sequence of unfoldings of v is defined by:

$$\begin{aligned} unfold_V^0(v) &= v \\ unfold_V^1(v) &= unfold_V(v) = unfold_V(unfold_V^0(v)) \\ &\vdots \\ unfold_V^k(v) &= unfold_V(unfold_V^{k-1}(v)) \end{aligned}$$

The function $unfold_V$ implies a equivalence relation induced by

$$v_1 = unfold_V(v_2) \Rightarrow v_1 \equiv v_2 \,.$$

In the following we consider the equivalence classes on cyclic values instead of the values themselves. The individual representants of an equivalence class are considered as representations of the same value.

Example 3.1.30 (unfolding values) *The results of the first unfolding steps of the cyclic value v from Ex. 3.1.28 are:*

$$\begin{aligned} &unfold_V^0(\mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))) = \mu x.(1 \;.\; (2 \;.\; (3 \;.\; x))) \\ &unfold_V^1(\mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))) = (1 \;.\; (2 \;.\; (3 \;.\; \mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))))) \\ &unfold_V^2(\mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))) = (1 \;.\; (2 \;.\; (3 \;.\; (1 \;.\; (2 \;.\; (3 \;.\; \mu x.(1 \;.\; (2 \;.\; (3 \;.\; x))))))))) \end{aligned}$$

In the following we define the semantic function $\langle\!\langle\cdot\rangle\!\rangle_c$ for the different standard types. We often use the notation $e : t$ for $e \in \langle\!\langle t\rangle\!\rangle_c$.[4]

Definition 3.1.31 (base types) *The object $\bot \in \mathcal{V}$ denotes non-termination of a computation. The value set $\langle\!\langle t\rangle\!\rangle_c$ of every closed type t contains $\bot$. The value set of the empty type $\bot$ just contains the value for non-termination, i.e. $\langle\!\langle\bot\rangle\!\rangle_c = \{\bot\}$.*[5]

The only value of the zero type $\mathcal{O}$ (except of $\bot$ that is value of every type but is often not mentioned explicitly) is the zero value $\mathcal{O}$ denoting an empty input or output of a function. (One can interpret the zero type $\mathcal{O}$ as a Cartesian product of 0 elements.)

For all base types $b_1, \dots, b_v$ there are sets $\langle\!\langle b_i\rangle\!\rangle_c$. For our example type language the sets $\langle\!\langle b_i\rangle\!\rangle_c$ are explained in App. C.

Definition 3.1.32 (intersection existence property) *Let $\mathcal{B}$ a set of base types and $\langle\!\langle\cdot\rangle\!\rangle_c$ a semantic function. $\mathcal{B}$ has the* intersection existence property *if the following holds:*
If $b, b' \in \mathcal{B}$ with $\langle\!\langle b\rangle\!\rangle_c \cap \langle\!\langle b'\rangle\!\rangle_c \neq \emptyset$ then there exists some $\tilde{b} \in \mathcal{B}$ with $\langle\!\langle\tilde{b}\rangle\!\rangle_c = \langle\!\langle b\rangle\!\rangle_c \cap \langle\!\langle b'\rangle\!\rangle_c$.

In the following we assume the intersection existence property to hold for $\mathcal{B}$.

Definition 3.1.33 (type constructors) *The semantic function $\langle\!\langle\cdot\rangle\!\rangle_c$ is extended to constructed types as follows:*

- $\langle\!\langle(\cup\ t_1\ \dots\ t_k)\rangle\!\rangle_c = (\cup\ \langle\!\langle t_1\rangle\!\rangle_c\ \dots\ \langle\!\langle t_k\rangle\!\rangle_c)$. *For $k = 1$, $\langle\!\langle(\cup\ t_1)\rangle\!\rangle_c = \langle\!\langle t_1\rangle\!\rangle_c$ and for $k = 0$* $\langle\!\langle(\cup)\rangle\!\rangle_c = \langle\!\langle\bot\rangle\!\rangle_c$.
- $\langle\!\langle(\cap\ t_1\ \dots\ t_k)\rangle\!\rangle_c = (\cap\ \langle\!\langle t_1\rangle\!\rangle_c\ \dots\ \langle\!\langle t_k\rangle\!\rangle_c)$. *For $k = 1$, $\langle\!\langle(\cap\ t_1)\rangle\!\rangle_c = \langle\!\langle t_1\rangle\!\rangle_c$ and for $k = 0$* $\langle\!\langle(\cap)\rangle\!\rangle_c = \langle\!\langle\top\rangle\!\rangle_c$.
- $\langle\!\langle t_1 \setminus t_2\rangle\!\rangle_c = \langle\!\langle t_1\rangle\!\rangle_c \setminus \langle\!\langle t_2\rangle\!\rangle_c$.
- $\langle\!\langle\mathcal{C}t\rangle\!\rangle_c = \mathcal{V} \setminus \langle\!\langle t\rangle\!\rangle_c =: \mathcal{C}_{\mathcal{V}}\langle\!\langle t\rangle\!\rangle_c$.
- *For every tuple like type constructor c_i (with $i \in \{1, \dots, w\}$) there is a corresponding data constructor d_i such that* $\langle\!\langle(c_i\ t_1\ \dots\ c_{a(c_i)})\rangle\!\rangle_c = \{(d_i\ e_1\ \dots\ e_{a(c_i)}) \mid e_1 : t_1, \dots, e_{a(c_i)} : t_{a(c_i)}\}$.

[4]The notation $\langle\!\langle\cdot\rangle\!\rangle$ (sometimes with subscript) for the semantics of types is a non-standard one here because we will use the usual notation $[\![\cdot]\!]$ for the semantic function of the functional language.

[5]We use the same notation $\bot$ for the empty type and its only value, the non-terminating computation. This should not cause confusion because usually it is clear from the context whether we speak of the type or the value $\bot$.

Example 3.1.34 (semantics of constructed types) *For* $t := (\texttt{posint}\ .\ \texttt{lower-char})$ *we have*

$$\langle\!\langle t\rangle\!\rangle_c = \{(v_1\ .\ v_2) \mid v_1 \in \langle\!\langle \texttt{posint}\rangle\!\rangle_c, v_2 \in \langle\!\langle \texttt{lower-char}\rangle\!\rangle_c\} = \{(1\ .\ a), (1\ .\ b), \ldots, (2\ .\ f), \ldots\}\,.$$

The function types express the set of all functions, of all predefined functions and of all user defined functions. This is formalized in the following definition:

Definition 3.1.35 (function types) *The set of function definitions in our framework is divided into a set PFunc of predefined function definitions and a set LC of user defined function definitions. The semantics of the function types is given by:*

- $\langle\!\langle Tfunc_P\rangle\!\rangle_c = PFunc.$
- $\langle\!\langle Tfunc_U\rangle\!\rangle_c = LC.$
- $\langle\!\langle Tfunc\rangle\!\rangle_c = \langle\!\langle Tfunc_P\rangle\!\rangle_c \cup \langle\!\langle Tfunc_U\rangle\!\rangle_c.$

In the following we will use *function* as an abbreviation of *function definition*.

Knowledge of the internal structure of the semantic domains of function types is not essential at the moment. This can be deferred to the definition of a type inference system.

Frame types and environment types essentially emulate the structure of frames and environments. The semantics of these types is described in the following definition:

Definition 3.1.36 (frame types and environment types) *For a frame type*

$$f = (\textit{frame}\ s_1 : t_1\ \ldots\ s_n : t_n)$$

the set $\langle\!\langle f\rangle\!\rangle_c$ *is the set of those frames* $f_v \in$ *Frame that are defined exactly on the symbols* $s_1, \ldots, s_n$ *and assign a value* $v_i \in \langle\!\langle t_i\rangle\!\rangle_c$ *to* s_i.

For an environment type $e = (\textit{env}\ e'\ f)$ *with another environment type* e' *and a frame type* f $\langle\!\langle e\rangle\!\rangle_c$ *contains exactly those environments* $(e'_v\ f_v)$ *that consist of a parent environment* $e'_v \in \langle\!\langle e'\rangle\!\rangle_c$ *and a frame* $f_v \in \langle\!\langle f\rangle\!\rangle_c$.
$\langle\!\langle \texttt{env}_0\rangle\!\rangle_c$ *contains just the empty environment* `emptyenv`.

Example 3.1.37 (semantics of frame and environment types) *Consider the frame types given in Ex. 3.1.18. They have the following semantics:*

$$\langle\!\langle F_1\rangle\!\rangle_c = \{[x \mapsto v_1, s \mapsto v_2, f \mapsto v_3] \mid v_1 \in \langle\!\langle \texttt{int}\rangle\!\rangle_c, v_2 \in \langle\!\langle \texttt{string}\rangle\!\rangle_c, v_1 \in \langle\!\langle Tfunc\rangle\!\rangle_c\} =$$
$$= \{[x \mapsto 42, s \mapsto \text{“Hello world!”}, f \mapsto f_+], [x \mapsto -3, s \mapsto \text{“abc”}, f \mapsto f_{map}], \ldots\}^6$$

$$\langle\!\langle F_2 \rangle\!\rangle_c = \{[x \mapsto v_1, y \mapsto v_2] \mid v_1 \in \langle\!\langle \textbf{\textit{posint}} \rangle\!\rangle_c, v_2 \in \langle\!\langle \textbf{\textit{posint}} \rangle\!\rangle_c\} = \\ = \{[x \mapsto 1, y \mapsto 1], [x \mapsto 256, y \mapsto 42], \ldots\}$$

The semantics of the environment E in Ex. 3.1.20 is given by a list of frames taken from the semantics of the corresponding frame types:

$$\langle\!\langle E \rangle\!\rangle_c = \{(f_1\ f_2) \mid f_i \in \langle\!\langle F_i \rangle\!\rangle_c\} = \{([x \mapsto -3, s \mapsto \text{“abc”}, f \mapsto f_{map}]\ [x \mapsto 256, y \mapsto 42]), \ldots\}$$

The semantics of a frame does not depend on the order in which the symbol bindings are given. We therefore define the function *fs* that expects a frame type as argument and returns the set of all frame types with the same semantics:

Definition 3.1.38 (shuffles of frame types) *Let $F = [x_1 \mapsto t_1, \ldots, x_k \mapsto t_k]$ be a frame type. The function fs (frame shuffle) on frame types is defined by:*

$$fs(F) = \{[x_{i_1} \mapsto t_{i_1}, \ldots, x_{i_k} \mapsto t_{i_k}] \mid \{i_1, \ldots, i_k\} = \{1, \ldots, k\}\}$$

Every $F' \in fs(F)$ is called a shuffle of F.

The types that do not have a defined semantics up to now are the types containing variables. Types with free variables are defined under a given assignment of types to the variables:

Definition 3.1.39 ((closed) 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 the assigned values t_X do not contain any variables $Y \in dom(\sigma)$ as subterms. It is called* closed *if it assigns closed types to all variables.*

The set of all type substitutions is denoted by TS. TS_C is the set of all closed type substitutions.

The effect of type substitutions to types is formalized by the following definition of substitution applications:

Definition 3.1.40 (application of type substitutions) *If t is a type and $\sigma = [x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k]$ is a type substitution then $\sigma(t)$ denotes the* application *of σ to t, i.e. the type generated from t by replacing every free occurrence of $x_i \in dom(\sigma)$ by t_i.*

[6]For a symbol *xyz* f_{xyz} denotes the function usually bound to *xyz*.

Sometimes we need a special kind of type substitution transforming a given type to a closed type. A set of such substitutions is given by the following definition:

Definition 3.1.41 (appropriate closed type substitutions) *A closed type substitution σ is* appropriate *for a type t if σ assigns a type to all type variables occurring freely in t.*

Definition 3.1.42 (semantics of types with variables) *Let t be a type containing the type variables $X_1, \ldots, X_k$ and let σ be an appropriate closed type substitution for t.*

The semantic function for types with variables is $\langle\!\langle \cdot \rangle\!\rangle(\cdot) : \mathcal{T} \times TS \to \mathcal{V}$. It is based on the semantic function for ground types as follows:

$$\langle\!\langle t \rangle\!\rangle(\sigma) = \langle\!\langle \sigma(t) \rangle\!\rangle_c$$

In the following we write $\langle\!\langle t \rangle\!\rangle$ instead of $\langle\!\langle t \rangle\!\rangle(\sigma)$ if t is a closed type and the semantics is independent from a certain type substitution.

The semantics of a recursive type t is defined by finite approximations of t, i.e. finite unfoldings of t:

Definition 3.1.43 (finite approximation of recursive types) *Let $t \in \mathcal{T}_S(V)$ a type and let $\diamond$ denote an additional type that does not contain* any *values (even not the non-termination $\perp$). Then the kth approximation of t is*

$$approx^k(t) = cut(unfold^k(t))$$

where cut replaces every occurrence of a recursive type in its argument by $\diamond$.

Example 3.1.44 *As in Ex. 3.1.24 the first approximations of lists with element type A are presented:*

$$\begin{aligned}
&approx^0(\mu X.(\cup\ \textbf{\textit{nil}}\ (A\ .\ X))) = \diamond \\
&approx^1(\mu X.(\cup\ \textbf{\textit{nil}}\ (A\ .\ X))) = (\cup\ \textbf{\textit{nil}}\ (A\ .\ \diamond)) \\
&approx^2(\mu X.(\cup\ \textbf{\textit{nil}}\ (A\ .\ X))) = (\cup\ \textbf{\textit{nil}}\ (A\ .\ (\cup\ \textbf{\textit{nil}}\ (A\ .\ \diamond))))
\end{aligned}$$

Definition 3.1.45 (semantics of recursive types (1)) *The semantics of recursive types (with respect to finite values) is defined by*

$$\langle\!\langle \mu X.t \rangle\!\rangle = \{v \in \mathcal{V} \mid \exists k \in \mathbb{N}.\, v \in \langle\!\langle approx^k(\mu X.t) \rangle\!\rangle\} = \bigcup_{k \geq 1} \langle\!\langle approx^k(\mu X.t) \rangle\!\rangle$$

Example 3.1.46 *Let $t := \mu X.(\cup\ \textbf{\textit{nil}}\ (\textbf{\textit{int}}\ .\ X))$ the definition above yields for $\langle\!\langle t \rangle\!\rangle$ the set of all lists of finite length with arguments of type* ***int****. E.g. the list $v :=$ (1 2 3) fulfills $v \in \langle\!\langle approx^4(t) \rangle\!\rangle$ and therefore $v \in \langle\!\langle t \rangle\!\rangle$.*

Unfortunately this definition does not cover infinite values as they are given by cyclic value definitions. To cover these values too we must define finite approximations of values:

Definition 3.1.47 (finite approximation of values) *Let v be a value and let $\star$ denote a variable for a value whose type does not matter, i.e. $\forall t \in \mathcal{T} . \star : t$ (even $\star : \diamond$). The finite approximations of v are defined as follows:*

$$approx_V^k(v) = cut_V(unfold_V^k(t))$$

where cut_V replaces all occurrences of a cyclic value in the given value (more exactly value representation) by $\star$.

Example 3.1.48 *The first approximations of the cyclic value $v := \mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))$ introduced in Ex. 3.1.28 are:*

$$\begin{aligned}
&approx_V^0(\mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))) = \star \\
&approx_V^1(\mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))) = (1 \;.\; (2 \;.\; (3 \;.\; \star))) \\
&approx_V^2(\mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))) = (1 \;.\; (2 \;.\; (3 \;.\; (1 \;.\; (2 \;.\; (3 \;.\; \star))))))
\end{aligned}$$

Now we can define an improved semantics of recursive types.

Definition 3.1.49 (semantics of recursive types (2)) *The semantics of recursive types is defined by*

$$\begin{aligned}
\langle\!|\mu X.t|\!\rangle = \{v \in \mathcal{V} \mid \exists k \in \mathbb{N} . v \in \langle\!|approx^k(\mu X.t)|\!\rangle\} \cup \\
\cup \{v \in \mathcal{V}_{rec} \mid \forall i \in \mathbb{N}\; \exists j \in \mathbb{N} . approx_V^i(v) \in \langle\!|approx^j(\mu X.t)|\!\rangle\}
\end{aligned}$$

In Def. 3.1.49 the first union element is identical to Def. 3.1.45. The second element contains all recursive values denoted by the recursive type.

Example 3.1.50 *For $v = \mu x.(1 \;.\; (2 \;.\; (3 \;.\; x)))$ and $t = \mu X.(\cup\; \textbf{nil}\; (int \;.\; X))$ we have*

$$\forall i \in \mathbb{N} . approx_V^i(v) \in \langle\!|approx^{3i+1}(t)|\!\rangle$$

and therefore $v \in \langle\!|t|\!\rangle$. Analogously the list l' of all week days as defined in Ex. 3.1.26 fulfills $l' \in \langle\!|\mu X.(\cup\; \textbf{nil}\; (string \;.\; X))|\!\rangle$.

3.1.3 Set Normalized Standard Types

The type constructors $\cup$, $\cap$, $\setminus$ and $\mathcal{C}$ model the behaviour of the corresponding set operations for types. These set operators allow the representation of sets in different ways. Since some of the algorithms presented in Sec. 3.4 and 5 rely on a certain representation we define types in set normalized form:

Definition 3.1.51 (set normalized form) *A type t is in set normalized form if the following properties are fulfilled:*

1. *t does not contain a difference type constructor $\backslash$.*
2. *There are no directly nested occurrences of the complement type constructor of the form $\mathcal{C}\mathcal{C}t'$.*
3. *For every occurrence $\mathcal{C}t'$ of the complement type constructor the argument t' has no union or intersection as top level constructor.*

An algorithm to transform every type t to set normalized form is given in form of a term rewriting system. Since type constructors of variable arity occur we use the extended notion of term rewriting rules as presented in Sec. 2.1:

Definition 3.1.52 *The term rewriting system R_{SN} is defined by the following term rewriting rules:*

$$\begin{aligned} (\backslash\ t_1\ t_2) &\rightarrow (\cap\ t_1\ \mathcal{C}t_2) \\ \mathcal{C}(\cup\ <a_1 \ldots e_1>) &\rightarrow (\cap\ <\mathcal{C}a_1 \ldots \mathcal{C}e_1>) \\ \mathcal{C}(\cap\ <a_1 \ldots e_1>) &\rightarrow (\cup\ <\mathcal{C}a_1 \ldots \mathcal{C}e_1>) \\ \mathcal{C}\mathcal{C}t &\rightarrow t \end{aligned}$$

We now prove several properties of this term rewriting system:

Lemma 3.1.53 (termination of R_{SN}) *The term rewriting system R_{SN} terminates for every input type.*

Proof: See App. A.1, Page 113. □

Lemma 3.1.54 (confluence of R_{SN}) *The term rewriting system R_{SN} is confluent.*

Proof: See App. A.1, Page 114. □

Lemma 3.1.55 (syntactic correctness of *set-normalize*) *The result type t' returned when applying R_{SN} to an arbitrary type t is in set normalized form.*

Proof: See App. A.1, Page 114. □

Lemma 3.1.56 (semantic correctness of *set-normalize*) *Let t be an arbitrary type and t' be the result of applying R_{SN} to a type t. Then $\langle\!\langle t \rangle\!\rangle(\sigma) = \langle\!\langle t' \rangle\!\rangle(\sigma)$ for every appropriate closed type substitution σ.*

Proof: See App. A.1, Page 115. □

Example 3.1.57 (set normalized form) *Consider the type*

$$t = \textbf{\textit{num}} \setminus (\cup\ \mathcal{C}\texttt{posreal}\ \texttt{int-e})\,.$$

Applying R_{SN} to t yields the following transformation sequence (writing $\rightarrow_i$ for rule number i applied:

$$t \rightarrow_1 (\cap\ \textbf{\textit{num}}\ \mathcal{C}(\cup\ \mathcal{C}\texttt{posreal}\ \texttt{int-e})) \rightarrow_2 (\cap\ \textbf{\textit{num}}\ (\cap\ \mathcal{C}\mathcal{C}\texttt{posreal}\ \mathcal{C}\texttt{int-e})) \rightarrow_4$$
$$\rightarrow_4 (\cap\ \textbf{\textit{num}}\ (\cap\ \texttt{posreal}\ \mathcal{C}\texttt{int-e}))$$

Since the term rewriting system R_{SN} is confluent it shows a functional behaviour. In the following we often write *set-normalize*(t) for the result of applying R_{SN} to t.

3.1.4 Equivalence Classes of Types

The idea behind the set normalized form of types is the fact that different types can be semantically equivalent, i.e. denote the same set of values. This is the case not only for types not in set normalized form as the following example shows:

Example 3.1.58 (semantically equivalent types) *The following set of types are semantically equivalent:*

- $\{\top, (\cup\ \top\ t)\}$ *for every type t.*
- $\{t, (\cup\ t\ \bot), (\cup\ t\ t), (\cap\ \top\ t)\}$ *for every type t.*

For several of these equivalences one can define normal forms and present algorithms normalizing the types. We do not want to discuss here whether it is possible to define a unique normal form for every type such that semantically equivalent types have the same normal form. We rather define equivalence classes on the types as usual. In the following (unless stated otherwise) we will consider equivalent types as equal, i.e. we will not consider syntactical differences of semantically equivalent types.

3.2 Value Assignments

During type inference there are sometimes exact values known for certain argument positions of a function call. This is usually due to constant values occurring in the program source code. For a type inference system just working on standard types we have to transform every constant value to its most special type. This usually causes loss of information and may result in fewer type errors to be caught.

Example 3.2.1 *Suppose the following piece of code:*

```
(let ((x 5)
      (y 3)
      (z 2))
  (/ (- x (+ y z))))
```

When using the type **`posint-e`** *(type of exact positive integers) for x, y and z we can only infer the type* **`int-e`** *(type of exact integers) for the argument to /. But taking into account the given values for x, y and z yields the value* 0 *for the argument to / which causes a type error.*

The following definition of a *value assignment* allows to work directly on constant values as long as this is possible:

Definition 3.2.2 (value assignment) *Let $v \in \mathcal{V}$ be a value with $v : t_b$ for some base type t_b.*
Then the value assignment *$\mathcal{A}(v)$ is a type with $\langle\!\langle \mathcal{A}(v) \rangle\!\rangle = \{v\}$. We identify $\langle\!\langle \mathcal{A}(v) \rangle\!\rangle$ with its only element v.*

Definition 3.2.3 ((semi-)closed/ground types) *The set $\mathcal{T}$ of all types can now be defined analogously to $\mathcal{T}_S$ in Def. 3.1.1 except of adding value assignments and types containing value assignments as subterms to $\mathcal{T}$. The sets $\mathcal{T}_{SC}$ of all semi-closed types $\mathcal{T}_C$ of all closed types and $\mathcal{T}_G$ of all ground types are defined analogously to $\mathcal{T}_{SCS}$, $\mathcal{T}_{CS}$ and $\mathcal{T}_{GS}$ in Def. 3.1.2.*

3.3 I/O-Representations of Function Types

The usual definition of function types (cf. e.g. [Mil78]) is done by a function type constructor $\rightarrow$ with the following semantics:
A type $t_1 \rightarrow t_2$ represents all functions f with

- $\langle\!\langle t_1 \rangle\!\rangle \subseteq dom(f)$
- $f(\langle\!\langle t_1 \rangle\!\rangle) \subseteq \langle\!\langle t_2 \rangle\!\rangle$

This definition of function types is appropriate for sound type systems. For an argument type t and an input type t_1 those systems usually perform the test $\langle\!\langle t \rangle\!\rangle \subseteq \langle\!\langle t_1 \rangle\!\rangle$ approximated by a subtype relation $\sqsubseteq$ with

$$t \sqsubseteq t_1 \Rightarrow \langle\!\langle t \rangle\!\rangle \subseteq \langle\!\langle t_1 \rangle\!\rangle \subseteq dom(f)$$

This implies that the test $t \sqsubseteq t_1$ is a sound approximation of the test for the applicability of f to t.

In a complete type checker we rather want to approximate the test $\langle\!\langle t \rangle\!\rangle \cap dom(f) \neq \emptyset$, i.e. instead of raising an error when *some* elements of t cause an error we raise an error when *all* elements of t are no valid arguments to f. Unfortunately a function type defined in the way above is not of use for a complete type checker because there might be cases in which $\langle\!\langle t \rangle\!\rangle \cap dom(f) \neq \emptyset$, but $\langle\!\langle t \rangle\!\rangle \cap \langle\!\langle t_1 \rangle\!\rangle = \emptyset$ raises a type error.

Example 3.3.1 *According to the usual definition of function types the unary division operator should have the type* `num` $\setminus$ `zero` $\rightarrow$ `num`. *But because of the problem of deciding whether the argument is 0 one often states the division-by-zero problem to be outside the scope of the type checker.*

Putting the division-by-zero problem into the scope of the type checker is easily possible for a complete type checker that can detect some of the division-by-zero errors, but not all of them. To do this a complete type checker needs a function type definition different from that given above.

A type inference system based on the type language described here could e.g. use the directed data flow properties of abstract interpretations to infer abstract predefined functions or abstract lambda closures that can be understood as value assignments for predefined and user defined functions, respectively, instead of function types. Since it seems quite difficult to find a short and understandable representation for the output of such function value assignments we will define I/O-representations for functions. The I/O-representations expressing the main properties of a function can be used in the output of types.

Definition 3.3.2 (I/O-representations) *An I/O-representation of a function is given by a set of I/O-representation pairs* $IN_i \dashrightarrow OUT_i$ *with:*

- $dom(f) \subseteq \bigcup_i \langle\!\langle IN_i \rangle\!\rangle$

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

By the first part of the definition every value that can be processed by a function must be member of at least one IN_i. A type checker cannot raise an error because of calling a function f with an input argument $v \in dom(f)$ that is not member of f's input type.
By the second part of the definition applying the function to an argument $v : IN_i$ cannot yield a result that is not of type OUT_i (except of error). Thus, uniting all those OUT_i such that IN_i has common elements with the argument type yields a type that covers all values possible for a function application.

Example 3.3.3 (I/O-representations of functions) *Consider the following I/O-represention of the function* add *where every line represents one I/O-represention pair, e.g. one element of the I/O-representation set:*

$$\begin{aligned} \textit{posint} \times \textit{posint} &\dashrightarrow \textit{posint} \\ \textit{int} \times \textit{int} &\dashrightarrow \textit{int} \\ \textit{num} \times \textit{num} &\dashrightarrow \textit{num} \\ \textit{string} \times \textit{string} &\dashrightarrow \textit{string} \end{aligned}$$

One could imagine that add is the usual addition on the three mentioned number types and the concatenation function on strings.

I/O-representations can occur in every place a function type could occur.

When a function f expects a function f' as input the inferred output type of f' is known just from checking uses of f' for non-empty intersection with other types. Thus, the input type of f' is not completely known, but we know a types PIN_i Furthermore, when an output type $POUT_i$ is inferred for a given input PIN_i we can just expect the property $f(\langle\!\langle PIN_i \rangle\!\rangle) \cup \langle\!\langle POUT_i \rangle\!\rangle \neq \emptyset$ to hold. Therefore we define the output-partial I/O-representation (PI/PO-representation) of a function as follows:

Definition 3.3.4 (PI/PO-representations) *A PI/PO-representation of a function f is given by a set of PI/PO-representation pairs $PIN_i \overset{\approx}{\dashrightarrow} POUT_i$ with:*

- $\forall i \,.\, dom(f) \cap \langle\!\langle PIN_i \rangle\!\rangle \neq \emptyset$
- $\forall i \,.\, f(\langle\!\langle PIN_i \rangle\!\rangle) \cap \langle\!\langle POUT_i \rangle\!\rangle \neq \emptyset$

The set of all Pi/PO-representations is denoted by **pipo***.*

Example 3.3.5 (PI/PO-representation) *Consider the following function map1 implementing the usual map operation for just unary functions and one list and its use:*

```
(define (map1 f l)
  (if (null? l) ()
      (cons (f (car l)) (map1 f (cdr l)))))

(define (usemap1 f l)
  (let ((l-new (map1 f l)))
    (+ (car l-new) (car (cdr l-new)))))
```

An I/O-representation for usemap1 is

$$\{A \overset{\approx}{\dashrightarrow} \mathbf{num}\} \times (\mathit{list}\ A) \dashrightarrow (\mathit{list}\ B)$$

From the use of l-new in usemap1 we know that the output type of the first argument of usemap1 must contain numbers. But as long as the input function is not known further output values might be possible. Analogously from applying f to car l) we know that f must be applicable to some values of the element type of l. But again, f may be applicable to other values, too.

3.4 The Subtype Hierarchy on Semi-Closed Types

An important property of the type language presented in this chapter is a partial ordering on the set of ground types we will describe here. We call this partial ordering the subtype hierarchy on ground types. In usual sound type inference the existence of a type hierarchy allows to restrict types to common subtypes or generalize them to common supertypes. Work on sound type inference with subtyping can be found e.g. in [Smi94].

It is important for a type inference system with subtyping to decide whether a type t_1 is subtype of another type t_2 or not. Such an algorithm is described in [AC93]. The algorithm described in this section shares many ideas with that one in [AC93] but employs certain modifications to work on the function types specific to this work as well as on intersection types and complement types. We do not consider the work presented in [Dam94] on subtyping in the presents of union types, intersection types and recursive types because of its restriction to infinite base types.

Since the main question for a pair of types in this work is the non-empty intersection property discovered in Sec. 5 rather then the subtype property a sound subtyping algorithm is sufficient.

3.4.1 The Algorithm *ST*

In this section the algorithm *ST* defining the subtype relation on semi-closed types is presented. The definition of *ST* is based on an algorithm *STbase* that checks the subtype property for base types. The definition of base types in App. C contains the corresponding definition of *STbase*.

Assumption 3.4.1 (termination and correctness of *STbase*) *Let b_1 and b_2 be base types or value assignments. There is an algorithm STbase such that the call $STbase(b_1, b_2)$ terminates and approximates the subset property of the denoted sets of values as follows:*

$$STbase(b_1, b_2) = \textbf{\textit{true}} \Rightarrow \langle\!\langle b_1 \rangle\!\rangle \subseteq \langle\!\langle b_2 \rangle\!\rangle$$

The algorithm *ST* approximates the question whether a type t_1 denotes a subset of the values denoted by t_2 for arbitrary closed types. In one case it calls an algorithm *CE* that tests types for common elements and is presented in Chap. 5. Since *ST* and *CE* are mutually recursive we start the presentation with *ST* and use certain assumptions on the behaviour of *CE* to state the needed properties of *ST*. The algorithm *ST* is defined as follows:

Algorithm: ST
Input: Two semi-closed types t_1 and t_2.
Output: A boolean value.

The algorithm is given by the following case distinction where $\mathcal{NB}$ represents the condition that none of the cases given before is applicable.

1. If $t_1 = t_2$ then $ST(t_1, t_2) := \texttt{true}$.

2. If $t_1 = \bot$ or $t_2 = \top$ then $ST(t_1, t_2) := \texttt{true}$.

3. If t_1 and t_2 are base types or value assignments then $ST(t_1, t_2) := STbase(t_1, t_2)$.

4. If $t_1 \in \{Tfunc_P, Tfunc_U\}$ and $t_2 = Tfunc$ then $ST(t_1, t_2) := \texttt{true}$.

5. If $t_1 = (c\ t_{1,1}\ \ldots\ t_{1,k})$ and $t_2 = (c\ t_{2,1}\ \ldots\ t_{2,k})$ for $c \in \mathcal{K}$ then

$$ST(t_1, t_2) := \bigwedge_{i=1}^{k} ST(t_{1,i}, t_{2,i})\,.$$

6. If $f_1 = (\textit{frame}\ s_1 : t_1\ \dots\ s_k : t_k)$ and $f_2 = (\textit{frame}\ s'_1 : t'_1\ \dots\ s'_k : t'_{k'})$ with $\{s_1, \dots, s_k\} = \{s'_1, \dots, s'_{k'}\}$ then

$$ST(f_1, f_2) := \forall_{i \in \{1,\dots,k\}} \exists_{j \in \{1,\dots,k'\}} \,.\, s_i = s'_j \wedge ST(t_i, t'_j)\,.$$

7. If $e_1 = (\textit{env}\ e'_1\ f_1)$ and $e_2 = (\textit{env}\ e'_2\ f_2)$ then

$$ST(e_1, e_2) := ST(e'_1, e'_2) \wedge ST(f_1, f_2)\,.$$

8. Recursive types are unfolded before continuing the check until the same parameters have already occurred in one of the recursive subcalls before:
   (a) If the stack of function calls to ST already contains a call to ST with the same pair (t_1, t_2) of arguments then $ST(t_1, t_2) := \texttt{true}$.
   (b) If $\mathcal{NB}$ and $t_1 = \mu X.t'_1$ then $ST(t_1, t_2) := ST(\textit{unfold}(t_1), t_2)$.
   (c) If $\mathcal{NB}$ and $t_2 = \mu X.t'_2$ then $ST(t_1, t_2) := ST(t_1, \textit{unfold}(t_2))$.

9. For the set operations the following correspondences hold:
   (a) If $\mathcal{NB}$ and $t_1 = (\cup\ t_{1,1}\ \dots\ t_{1,k})$ then

$$ST(t_1, t_2) := \bigwedge_{i=1}^{k} ST(t_{1,i}, t_2)\,.$$

   (b) If $\mathcal{NB}$ and $t_2 = (\cup\ t_{2,1}\ \dots\ t_{2,k})$ then

$$ST(t_1, t_2) := \bigvee_{i=1}^{k} ST(t_1, t_{2,i})\,.$$

   (c) If $\mathcal{NB}$ and $t_1 = (\cap\ t_{1,1}\ \dots\ t_{1,k})$ then

$$ST(t_1, t_2) := \bigvee_{i=1}^{k} ST(t_{1,i}, t_2)\,.$$

(d) If $\mathcal{NB}$ and $t_2 = (\cap\ t_{2,1}\ \ldots\ t_{2,k})$ then

$$ST(t_1, t_2) := \bigwedge_{i=1}^{k} ST(t_1, t_{2,i})\,.$$

(e) If $\mathcal{NB}$ and $t_1 = \mathcal{C}t'_1$ and $t_2 = \mathcal{C}t'_2$ then $ST(t_1, t_2) := ST(t'_2, t'_1)$.

(f) If $\mathcal{NB}$ and $t_2 = \mathcal{C}t'_2$ and $CE(\tau(t_1), t'_2) = \emptyset$[7] then $ST(t_1, t_2) :=$ `true` where τ instantiates every variable $X_\forall \in V_q$ occurring in t_1 with $\top$.

10. If $\mathcal{NB}$ then $ST(t_1, t_2) =$ `false`.

We can now defined a subtype relation $\sqsubseteq$ on closed types that approximates the question whether a type t_1 denotes a subset of the values denoted by a type t_2. This subtype relation is defined by the syntactical test performed by ST:

Definition 3.4.2 (subtype relation on closed types) *The subtype relation $\sqsubseteq$ on closed types is defined using the algorithm ST by:*

$$t_1 \sqsubseteq t_2 \Leftrightarrow ST(t_1, t_2)\,.$$

3.4.2 Examples of *ST*

The following example presents some type pairs fulfilling the subtype property:

Example 3.4.3 (subtypes) *The following pairs of types fulfill the subtype relation:*

$$\mathit{int} \sqsubseteq \mathit{num}$$
$$(\mathit{int}\ .\ \mathit{string}) \sqsubseteq (\mathit{num}\ .\ \mathit{string})$$
$$(\cup\ \mathit{env}\ (\mathit{int}\ .\ \mathit{string})) \sqsubseteq (\cup\ \mathit{env}\ (\mathit{num}\ .\ \mathit{string}))$$

The a calculation of ST is given in the following example:

Example 3.4.4 *Consider the types*

$$t_1 = ((\cup\ \mathit{posint}\ \mathit{negint})\ .\ \mathit{string}) \text{ and } t_2 = (\mathit{num}\ .\ \mathit{string})\,.$$

[7] I.e. there are no common elements between $\tau(t_1)$ and t'_2 detected by CE.

The call $ST(t_1, t_2)$ is processed by case 5. It is transformed into the conjunction of the subcalls

$$c_1 \doteq ST((\cup\ \texttt{posint negint}), \texttt{num}) \text{ and } c_2 \doteq ST(\texttt{string}, \texttt{string})\,.$$[8]

c_1 is processed by case 9a and yields the conjunction of the subcalls

$$c_{1,1} \doteq ST(\texttt{posint}, \texttt{num}) \text{ and } c_{1,2} \doteq ST(\texttt{negint}, \texttt{num})\,.$$

Both of these cases are processed by case 3 and yield the result **true**. *This is also the result of c_1 and since c_2 yields* **true** *by case 1 we get*

$$ST(((\cup\ \texttt{posint negint})\ .\ \texttt{string}), (\texttt{num}\ .\ \texttt{string})) = \texttt{true}\,.$$

3.4.3 Properties of *ST*

In Case (9f) the algorithm *ST* is mutually recursive with the algorithm *CE* that will be defined in Chap. 5. More exactly there are chains of recursive calls of the form

$$ST \to CE \to S\text{-}CE \to ST\,.$$

For the moment we formulate two assumptions on the properties of *CE*. A complete proof of the algorithms occurring in this chain will be given when all the algorithms are defined.

Assumption 3.4.5 (termination assumption for *CE*) *The algorithm CE terminates for every input.*

Assumption 3.4.6 (correctness assumption for *CE*) *Let $t_1, t_2 \in \mathcal{T}$ be closed types, both in set normalized form. If there exists a $v \in \langle\!\langle t_1 \rangle\!\rangle \cap \langle\!\langle t_2 \rangle\!\rangle$ then $CE(t_1, t_2) =$* **true**.

The following Lemma 3.4.7 states the main property of ST: The test syntactically performed on two types is a sound approximation of the subset relation on the type semantics:

Lemma 3.4.7 *Let CE be an algorithm fulfilling Assumption 3.4.5 and let STbase fulfill Assumption 3.4.1. Then every call to ST for an arbitrary pair of input arguments in set normalized form terminates.*

Proof: See App. A.2, Page 116. □

A prove for the termination of *CE* used in Lemma 3.4.7 is given in Theorem 5.3.14.

[8]We use $\doteq$ to denote syntactic equality of function calls (e.g. ST) in contrast to $=$ denoting the equality of the results.

Lemma 3.4.8 *Let $t_1, t_2 \in \mathcal{T}_{SC}$ be semi-closed types in set normalized form. Let STbase fulfill Assumption 3.4.1 and let CE fulfill Assumption 3.4.6. Then*

$$ST(t_1, t_2) = \textbf{\textit{true}} \Rightarrow \langle\!|\sigma(t_1)|\!\rangle \subseteq \langle\!|\sigma(t_2)|\!\rangle$$

for every closed type substitution σ appropriate for t_1 and t_2.

Proof: See App. A.2, Page 116. □

In the following we will always assume Assumption 3.4.1 to hold.

Chapter 4

Checking Types for Common Elements

One main question occurring in complete type checking is the following: Given two types t_1 and t_2. Is there a substitution σ such that $\langle\!\langle\sigma(t_1)\rangle\!\rangle \cap \langle\!\langle\sigma(t_2)\rangle\!\rangle \neq \{\bot\}$? More precisely this property should be independent from the instantiation of quantified variables i.e. members of V_q. For every such substitution the property of common elements must not be violated by changing the assignment to a quantified variable.

Example 4.0.9 *Consider the types $t_1 = (\textbf{\texttt{int}} \;.\; X_\forall)$ and $t_2 = (\textbf{\texttt{posint}} \;.\; Y)$ where $X_\forall \in V_q$ and $Y \in V_f$. Then t_1 and t_2 have common elements because we can e.g. choose substitutions $\rho_f = \{Y \leftarrow X_\forall\}$ and $\rho_q = \{X_\forall \leftarrow \textbf{\texttt{int}}\}$ and a value $v = 5$ and get*

$$v \in \langle\!\langle\rho_q \circ \rho_f(t_1)\rangle\!\rangle \cap \langle\!\langle\rho_q \circ \rho_f(t_2)\rangle\!\rangle \,.$$

Furthermore, when choosing an arbitrary ρ'_q with $\rho'_q(X_\forall) \neq \bot$ there always exists a v' with

$$v' \in \langle\!\langle\rho'_q \circ \rho_f(t_1)\rangle\!\rangle \cap \langle\!\langle\rho'_q \circ \rho_f(t_2)\rangle\!\rangle \,.$$

On the other hand t_1 and $t_3 = (\textbf{\texttt{posint}} \;.\; \textbf{\texttt{negint}})$ do not have common elements because we can instantiate $X_\forall \in V_q$ in a way such that no common elements exist.

The notion of common elements is formalized as follows:

Definition 4.0.10 *Let t_1 and t_2 be two types. These types have common elements if*

$$\begin{aligned} &\exists \rho_f \forall \rho_q (dom(\rho_f) \subseteq V_f \wedge dom(\rho_q) \subseteq V_q \wedge \forall X_\forall \in dom(\rho_q) \,.\, \rho_q(X_\forall) \neq \bot \wedge \\ &\qquad \wedge \rho_q \circ \rho_f \text{ is appropriate for } t_1, t_2) \Rightarrow \\ &\qquad\qquad \Rightarrow \exists v \in \langle\!\langle\rho_q \circ \rho_f(t_1)\rangle\!\rangle \cap \langle\!\langle\rho_q \circ \rho_f(t_2)\rangle\!\rangle \wedge v \neq \bot \end{aligned} \tag{4.1}$$

We call every v with the properties given in (5.1) a common element *of t_1 and t_2 and denote the set of all common elements of t_1 and t_2 by $t_1 \boxdot t_2$.*

Note that quantified variables are treated in a special way in Def. 5.0.19: They must not be instantiated in a special way in order to get common elements. Common elements of types containing quantified variables are rather considered just for those types with common elements for *every* instantiation of the quantified variables.

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* may return a substitution even if t_1 and t_2 cannot have common elements especially when one of the types already does not have any elements.

The description of *CE* is done in the following subsections: Section 5.1 contains the preliminaries used to define *CE*. In Sec. 5.2 an algorithm *S-CE* is introduced that calculates constraints on the variable instantiations. The algorithm *CE* presented in Sec. 5.3 transforms these constraint sets to idempotent substitutions σ.

4.1 Preliminaries

In this section we will give some definitions needed to define *CE*. First, we will define some structures forming the result or intermediate values of *CE*. In a second step we will formulate the needed properties of a function *CEbase* realizing *CE* on base types and value assignments.

4.1.1 Structures used by *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. The substitutions considered by *CE* are restricted in the following way: If σ is a substitution returned by *CE* then σ just assigns types to *free* type variables. This motivates the following definition:

Definition 4.1.1 (free type substitutions) *A type substitution σ is called a* free type substitution *if σ is just defined for free type variables, i.e. if*

$$dom(\sigma) \subseteq V_f \, .$$

The set of all free type substitutions is denoted by FTS, the set of all closed free type substitutions by FTS_C.

CE will return sets of free type substitutions called s-collections:

Definition 4.1.2 (s-collection) *An* s-collection *is a finite set of free type substitutions.*

During the processing of CE constraints on the possible instantiations of type variables are collected in structures called *constraint sets* that are defined as follows:

Definition 4.1.3 (free constraint sets) *A free variable constraint is a pair* (X, t) *(often written as* $X \leftarrow t$ *where* $X \in V_f$ *and* $t \in \mathcal{T}$*. A* free constraint set *is a set of free variable constraints* $\{(X_1, t_1), \ldots, (X_n, t_n)\}$ *with pairwise disjoint variables* X_i*. For such a free constraint set* $dom(\sigma) = \{X_1, \ldots, X_n\}$ *and* $\sigma(X_i) = t_i$*.*

The intermediate values occurring in CE are not free constraint sets directly, but sets of free constraint sets. These sets are called *c-collections* and defined as follows:

Definition 4.1.4 (c-collection) *A* c-collection *is a finite set of free constraint sets.*

Constraint sets and c-collections make restrictions to the values assigned to certain variables. The free type substitutions or s-collections fulfilling all these constraints are called compatible with a free constraint set of c-collection, respectively:

Definition 4.1.5 *Let* σ *be a free constraint set. A free type substitution* σ' *is compatible with* σ *if for every* X *with* $X \leftarrow t \in \sigma$ *the following holds:*

- $X \in dom(\sigma')$
- $\langle\!\langle t \rangle\!\rangle \subseteq \langle\!\langle \sigma'(X) \rangle\!\rangle$

An s-collection Σ' *is compatible with a c-collection* Σ *if there is a one to one assignment of* $\sigma' \in \Sigma'$ *to* $\sigma \in \Sigma$ *with* σ' *compatible to* σ*.*

For every free constraint set σ' there is the natural free type substitution σ of σ' defined as follows:

Definition 4.1.6 (natural free type substitution) *Let* σ' *be a free constraint set. The natural free type substitution of* σ' *(denoted by* $subst(\sigma')$*) is*

$$subst(\sigma') = \{X \leftarrow \sigma'(X) \mid X \in dom(\sigma')\}\,.$$

For a c-collection Σ' *the natural s-collection of* Σ' *is*

$$subst(\Sigma') = \{subst(\sigma') \mid \sigma' \in \Sigma'\}\,.$$

The union of c-collections is based on the union of free constraint sets defined as follows:

Definition 4.1.7 (union of free constraint sets) *Let σ_1 and σ_2 be two free constraint sets. Their union is defined as*

$$\begin{aligned}\sigma_1 \otimes \sigma_2 =&\{A \leftarrow (\cup\ t_1\ t_2) \mid A \leftarrow t_1 \in \sigma_1 \wedge A \leftarrow t_2 \in \sigma_2\}\\ &\cup\{A \leftarrow t_1 \mid A \leftarrow t_1 \in \sigma_1 \wedge A \notin dom(\sigma_2)\}\\ &\cup\{A \leftarrow t_2 \mid A \leftarrow t_2 \in \sigma_2 \wedge A \notin dom(\sigma_1)\}\,.\end{aligned}$$

We can now define the union of c-collections:

Definition 4.1.8 (union of c-collections) *Let Σ_1 and Σ_2 be two c-collections the union of Σ_1 and Σ_2 is defined as*

$$\Sigma_1 \otimes \Sigma_2 = \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in \Sigma_1 \wedge \sigma_2 \in \Sigma_2\}\,.$$

We use the same operator $\otimes$ for both free constraint sets and c-collections. When identifying free constraint sets σ with the one element c-collections $\{\sigma\}$ the correspondence becomes obvious.

4.1.2 *CE* on Base Types and Value Assignments

The algorithm *CE* works by decomposing its argument types. It therefore needs an algorithm *CEbase* to decide the common element question for base types and value assumptions.

Assumption 4.1.9 (common elements of base types) *The function CEbase* $: \mathcal{B} \times \mathcal{B} \rightarrow \{\texttt{true}, \texttt{false}\}$ *approximates the question whether two base types or value assignments have common elements as follows:*

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

There is an algorithm also called CEbase that terminates for every input and returns the value of the function CEbase.

Especially *CEbase* fulfills

$$CEbase(b_1, b_2) = \texttt{true} \text{ if } (b_1 = \top \wedge b_2 \neq \bot) \vee (b_2 = \top \wedge b_1 \neq \bot)$$

but

$$CEbase(b_1, b_2) = \texttt{false} \text{ if } b_1 = \bot \vee b_2 = \bot\,.$$

App. C contains a definition of *CEbase* for the example base types presented there.

4.2 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 free 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

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

is initiated by *CE*. Its arguments are two types t_1 and t_2 to be checked for common elements, the empty free 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 free 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 where 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 a fixed order in which they are checked.

4.2.1 Globally Used Auxiliary Functions

To describe the actions of *S-CE* in the individual cases we start with defining some auxiliary functions that are needed in several cases:

In several cases *S-CE* decomposes a constructor and performs recursive calls sequentially to the arguments. This is formalized by the function *SCE-list-reduce*.

SCE-list-reduce 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 free

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. A recursion history r is just passed through to the calls to *S-CE*.

Definition 4.2.1 (*SCE-list-reduce*) *The function SCE-list-reduce expects the following arguments:*

- *A list L of tuples (t_1, t_2) where t_1 and t_2 are types.*
- *A c-collection Σ.*
- *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 4.2.2 (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 free 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 the call

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

Again SCE-list-reduce initiates a number of calls to S-CE with the first type pair as arguments. Since Σ' just contains one free constraint set in the example just one call to S-CE is necessary:

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

The resulting c-collection Σ'' is used in the recursive call

$$\textit{SCE-list-reduce}((), \Sigma'', r)$$

to SCE-list-reduce. Since the list of type pairs is empty rule (SCE-list-reduce1) applies and Σ'' is returned.

For changing a constraint in a given constraint set the functions *extend-constraint* and *replace-constraint* can be used:

Definition 4.2.3 (*extend-constraint*) *Let σ be a free constraint set, $X \in V_f$ a free variable and t a term. $\sigma' = extend\text{-}constraint(X, t, \sigma)$ is the free constraint set differing from σ just in the constraint of X as follows:*

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

Definition 4.2.4 (*replace-constraint*) *Let σ be a free constraint set, $X \in V_f$ a free variable and t a term. $\sigma' = replace\text{-}constraint(X, t, \sigma)$ is defined by*

$$\sigma'(Y) = \begin{cases} t & \text{if } Y = X \\ \sigma(Y) & \text{else} \end{cases}$$

Note that *extend-constraint* and *replace-constraint* behave equivalently if $X \notin dom(\sigma)$.

In some cases a given free constraint set has to be updated by constraining all free variables occurring in a term in a certain manner. This is done by the functions *constrain-all-free* and *free-to-top*:

Definition 4.2.5 (*constrain-all-free*) *Let t be a type term and σ a free constraint set. The call constrain-all-free(t, σ) constrains every free variable occurring in t to a fresh quantified variable. The new constraints are added to σ without destroying previous constraints:*

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

Definition 4.2.6 (*free-to-top*) *Let t be a type term and σ a free constraint set. The call free-to-top(t, σ) constrains every free variable occurring in t to $\top$ in σ:*

forall *free variables Y in t* ***do***
  $\sigma :=$ replace-constraint$(Y, \top, \sigma)$
return *σ.*

The following subsections describe the actions of *S-CE* for certain cases of t_1 and t_2:

4.2.2 $\top$ 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 if formalized in the rules (*Top1*) and (*Top2*):

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

$$\frac{\textit{S-CE}(t_1, \top, \sigma, r)}{\{\sigma\}} \quad t_1 \neq \bot \qquad (\textit{Top2})$$

4.2.3 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.

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 that are not already detected in processing the former call with the same arguments. Hence, the actual call can return without introducing any further constraints. 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:

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

$$\frac{\textit{S-CE}(t_1, t_2 = \mu X.t_2', \sigma, r)}{\textit{combine-cs}(\textit{S-CE}(t_1, \textit{unfold}(t_2), \sigma, ((t_1, t_2)\ .\ r)))} \qquad (\textit{Rec2})$$

The function *combine-cs* used in (*Rec1*) and (*Rec2*) expects and returns a c-collection and is defined as follows:

$$\begin{aligned} \textit{combine-cs}(\emptyset) &= \emptyset \\ \textit{combine-cs}(\Sigma) &= \{\bigotimes_{\sigma \in \Sigma'} \sigma \mid \emptyset \neq \Sigma' \subseteq \Sigma\} \text{ for } \Sigma \neq \emptyset \end{aligned}$$

Example 4.2.7 *Consider the c-collection $\Sigma = \{\sigma_1, \sigma_2\}$ with $\sigma_1 = \{X \leftarrow \textbf{int}, Y \leftarrow \textbf{int}\}$ and $\sigma_2 = \{X \leftarrow \textbf{bool}, Z \leftarrow \textbf{bool}\}$.*

The non-empty subsets of Σ are

$$\begin{aligned} \Sigma_1 &= \{\sigma_1\} \\ \Sigma_2 &= \{\sigma_2\} \\ \Sigma_3 &= \Sigma\,. \end{aligned}$$

Each of these subsets Σ_i yields a free constraint set that is generated by combining all elements of Σ_i by $\otimes$ with the following results:

$$\begin{aligned} &\bigotimes_{\sigma \in \Sigma_1} = \sigma_1 \\ &\bigotimes_{\sigma \in \Sigma_2} = \sigma_2 \\ &\bigotimes_{\sigma \in \Sigma_3} = \sigma_3 \textit{ with } \sigma_3 = \{X \leftarrow (\cup\ \textbf{int}\ \textbf{bool}), Y \leftarrow \textbf{int}, Z \leftarrow \textbf{bool}\}\,. \end{aligned}$$

Altogether combine-cs$(\Sigma) = \{\sigma_1, \sigma_2, \sigma_3\}$.

The reason for applying *combine-cs* on the result of the recursive subcall becomes obvious in the following example:[1]

Example 4.2.8 (recursive types) *Assume that (Rec1) and (Rec2) just return the results of their subcalls without applying combine-cs. Consider the two types*

$$\begin{aligned} t_1 &= \mu X.((\cup\ f(\textbf{bool})\ f(\textbf{int}))\ .\ (\cup\ \textbf{nil}X)) \\ t_2 &= \mu Y.(f(A)\ .\ (\cup\ \textbf{nil}\ Y)) \end{aligned}$$

[1] Some of the rules used for the example are defined afterwards. We just sketch the results here.

where f is a unary free type constructor (or a type constructor with arity n and $n-1$ argument positions fixed).

Processing the call $S\text{-}CE(t_1, t_2, \emptyset, ())$ starts with applying the rule (Rec1) and afterwards the rule (Rec2). The result is a recursive call

$$S\text{-}CE(t_1', t_2', \emptyset, r) \text{ with } t_1' = unfold(t_1),\ t_2' = unfold(t_2) \text{ and } r = ((t_1', t_2)\ (t_1, t_2))$$

This call is processed by rule (Constr) and is divided into two subcalls. The first one is processed by rule (U1) with each of its subcalls decomposed by (Constr) and processed by (Var2). It yields

$$S\text{-}CE((\cup\ f(\texttt{bool})\ f(\texttt{int})), f(A), \emptyset, r) = \{\{A \leftarrow \texttt{bool}\}, \{A \leftarrow \texttt{int}\}\} =: \Sigma_1\,.$$

The second subcall (more precisely we have two subcalls for every element $\sigma \in \Sigma_1$) is decomposed by (U1) and (U2) and some of the subcalls are processed by (RecT), some by (Base) and some fail. These subcalls do not introduce any new constraints and Σ_1 is returned.

Unfortunately there is a value $v = (\#t\ .\ (5\ .\ \texttt{nil}))$ with $v \in t_1 \boxdot t_2$ (e.g. with $\rho_f = \{A \leftarrow (\cup\ \texttt{bool}\ \texttt{int})\}$ and $\rho_q = \emptyset$), but $v \notin \sigma(t_1) \boxdot \sigma(t_2)$ for all $\sigma \in \Sigma_1$.

When we assume the rule (*RecT*) not to exist the processing of the call in Example 5.2.8 does not terminate, but the constraints generated during the infinite loop give insight in the problem of the example. After the second unfolding of t_1 and t_2 the first call performed by *SCE-list-reduce* generates the free constraint sets $\sigma_{1,1} = \{A \leftarrow (\cup\ \texttt{bool}\ \texttt{bool})\}$ and $\sigma_{1,2} = \{A \leftarrow (\cup\ \texttt{bool}\ \texttt{int})\}$ from $\{A \leftarrow \texttt{bool}\}$ and $\sigma_{2,1} = \{A \leftarrow (\cup\ \texttt{int}\ \texttt{bool})\}$ and $\sigma_{2,2} = \{A \leftarrow (\cup\ \texttt{int}\ \texttt{int})\}$ from $\{A \leftarrow \texttt{int}\}$. After the third unfolding eight constraint sets containing three element unions are generated and so on.

More generally, when after unfolding types by (*Rec1*) and (*Rec2*) a union type constructor causes several free constraint sets to be generated, an iterated unfolding can generate constraints containing unions as above. Since the generation of these unions is blocked by (*RecT*) we have to calculate these unions after processing the first unfolding. This is done by *combine-cs*.

4.2.4 Free Type Variables

When at least one of the types is a free type variable several cases have to be distinguished: There is either exactly one or two free type variables that can already be member of the domain of σ or not.

When both types are free type variables then a new variable is introduced in order to name the common elements possibly occurring at this point. This variable must not be restricted by *S-CE* any further and is therefore quantified:

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

Example 4.2.9 *Let $t_1 = \#(X\ \textbf{int}\ X)$ and $t_2 = \#(\textbf{bool}\ Y\ Y)$. The call S-CE$(t_1, t_2, \emptyset, ())$ is processed by rule (Constr) given below. The processing of this rule initiates the call SCE-list-reduce$(((X, \textbf{bool}), (\textbf{int}, Y), (X, Y)), \emptyset, ())$. The first two subcalls to S-CE are processed by the rules (Var1) and (Var2) given below. They yield a intermediate c-collection consisting of exactly one free constraint set $\sigma' = \{X \leftarrow \textbf{bool}, Y \leftarrow \textbf{int}\}$.*

The result of the initial call to S-CE is the result of the third subcall S-CE$(X, Y, \sigma', ())$ which is processed by rule (BothVar). Its result is $\{\{X \leftarrow (\cup\ \textbf{bool}\ Z), Y \leftarrow (\cup\ \textbf{int}\ Z)\}\}$. with a new free variable Z.

Note that the introduction of Z is necessary in order to preserve e.g. the common element $v = \#(\#t\ 42\ 40.5)$ with a non-integer number in the third vector position.

When just t_1 is a free type variable then the constraint of t_1 in σ is updated to contain t_2:

$$\frac{\textit{S-CE}(t_1, t_2, \sigma, r)}{\{\textit{constrain-all-free}(t_2, \textit{extend-constraint}(X, t_2, \sigma))\}} \quad t_1 \in V_f, t_2 \notin V_f \quad (\textit{Var1})$$

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

Example 4.2.10 *Let $t_1 = (X\ .\ \textbf{int})$ and $t_2 = ((Y\ .\ \textbf{int})\ .\ Y)$. The call S-CE$(t_1, t_2, \emptyset, ())$ yields the call SCE-list-reduce$(((X, (Y\ .\ \textbf{int})), (\textbf{int}, Y)), \emptyset, r)$ by rule (Constr) given below.*

The first subcall to S-CE is S-CE$(X, (Y\ .\ \textbf{int}), \emptyset, ())$. It is processed by rule (Var1) and yields the result $\Sigma_1 = \{\sigma_1\}$ with $\sigma_1 = \{X \leftarrow (Y\ .\ \textbf{int}), Y \leftarrow Y'\}$. The constraint of X is introduced by extend-constraint while the constraint of Y results from applying constrain-all-free to $(Y\ .\ \textbf{int})$.

In second subcall initiated by SCE-list-reduce is S-CE$(\textbf{int}, Y, \sigma_1, ())$. Its result is $\Sigma_2 = \{\sigma_2\}$ with $\sigma_2 = \{X \leftarrow (Y\ .\ \textbf{int}), Y \leftarrow (\cup\ Y'\ \textbf{int})\}$ by rule (Var2).

Note that without applying constrain-all-free the result in this example is $\Sigma_2' = \{\sigma_2'\}$ with $\sigma_2' = \{X \leftarrow (Y\ .\ \textbf{int}), Y \leftarrow \textbf{int}\}$. While the value $v = ((\#t\ .\ 168)\ .\ 42)$ is a common element of t_1 and t_2 under σ_2 it is not a common element when applying σ_2' instead.

4.2.5 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:

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

$$\frac{\textit{S-CE}(t_1, (\cup\ t_{2,1}\ \ldots\ t_{2,k}), \sigma, r)}{\bigcup_{i=1}^{k} \textit{S-CE}(t_1, t_{2,i}, \sigma, r)} \qquad (U2)$$

4.2.6 Intersection Types

When one of the types is an intersection type the other argument type has to be checked against all intersection elements cumulating the detected restrictions. This is done by the rules (*I1*) and (*I2*) for an intersection type in the first or second argument, respectively using the function *SCE-list-reduce* given in Def. 5.2.1 on page 79:

$$\frac{\textit{S-CE}((\cap\ t_{1,1}\ \ldots\ t_{1,k}), t_2, \sigma, r)}{\textit{SCE-list-reduce}(((t_{1,1}, t_2), \ldots, (t_{1,k}, t_2)), \{\sigma\}, r)} \qquad (I1)$$

$$\frac{\textit{S-CE}(t_1, (\cap\ t_{2,1}\ \ldots\ t_{2,k}), \sigma, r)}{\textit{SCE-list-reduce}(((t_1, t_{2,1}), \ldots, (t_1, t_{2,k})), \{\sigma\}, r))} \qquad (I2)$$

Unfortunately this definition of (*I1*) and (*I2*) yields the following unintuitive results:

Example 4.2.11 *Let* $t_1 = (A\ .\ \texttt{nil})$ *and* $t_2 = (\cap\ (\texttt{num}\ .\ nil)\ (\texttt{bool}\ .\ nil))$*. For this pair of types S-CE returns the c-collection* $\{A \leftarrow (\cup\ \texttt{num}\ \texttt{bool})$ *even though* t_2 *does not contain any values except* $\perp$*.*

The behaviour above does not violate any of the statements proven for *S-CE* below. For refining *S-CE* one could think of a different mode in variable instantiation in order to distinguish constraints introduced by different intersection elements. A different way of refinement is an equivalent transformation of intersection types, e.g. transforming t_2 to $((\cap\ \texttt{num}\ \texttt{bool})\ .\ \texttt{nil})$ with an additional check for emptiness of the resulting intersection in Ex. 5.2.11 above.

4.2.7 Complement Types

If one of the the types is a complement type then there are common elements if the other type is not a subtype of the complement's argument.

When the type t (either t_1 or t_2) that is checked against the complement type is not a semi-closed term then we enforce this property be replacing every free variable by $\top$:

$$\frac{\textit{S-CE}(\mathcal{C}t_1', t_2, \sigma, r)}{\{\sigma'\}} \quad \sigma'(t_2) \not\sqsubseteq t_1', \sigma' := \textit{free-to-top}(t_2, \sigma) \qquad (\textit{Comp1})$$

$$\frac{\textit{S-CE}(t_1, \mathcal{C}t_2', \sigma, r)}{\{\sigma'\}} \quad \sigma'(t_1) \not\sqsubseteq t_2', \sigma' := \textit{free-to-top}(t_1, \sigma) \qquad (\textit{Comp2})$$

Note that $\sigma'(t_2) \not\sqsubseteq t_1'$ abbreviates the test $\neg ST(\sigma'(t_2), t_1')$. σ' is generated from σ in order to get a semi-closed term $\sigma'(t_2)$ in (*Comp1*) or $\sigma'(t_1)$ in (*Comp2*).

The calls to ST performed when checking the $\not\sqsubseteq$-conditions (cf. Def. 3.4.2) can cause a recursive call to *S-CE*, again, because of case (9f) of ST. Because of this the recursion info r must be passed over to ST and back to *S-CE* via *CE*. We omit this parameter here in order to simplify the representation of the algorithms. r is not changed at all in ST or *CE*, but is just passed through to *S-CE* again.

Example 4.2.12 *Let $t_1 = \mathcal{C}\,\texttt{int}$ and $t_2 = (X\ .\ Y)$. The call S-CE$(t_1, t_2, \emptyset, ())$ is processed by rule (Comp1). This rule generates the free constraint set $\sigma' = \textit{free-to-top}(t_2, \emptyset) = \{X \leftarrow \top, Y \leftarrow \top\}$ and performs the test $\neg ST((\top\ .\ \top), \texttt{int})$. Since this test succeeds the c-collection $\{\sigma'\}$ is returned.*

4.2.8 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 using *SCE-list-reduce*:

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

4.2.9 Frame Types

For two frame types F_1 and F_2 the existence of common elements depends on two facts:

- The sets of bound symbols must be equal.
- For every symbol the assigned types must have common elements.

After determining an order of the symbols the types assigned to the same symbol in both frames must be checked against each other. The result restrictions are collected:

$$\frac{\textit{S-CE}(F_1, F_2, \sigma, r), [s_1 \mapsto t_{1,i}, \ldots, s_k \mapsto t_{k,i}] \in \textit{fs}(F_i) \text{ for } i \in \{1,2\}}{\textit{SCE-list-reduce}(((t_{1,1}, t_{2,1}), \ldots, (t_{1,k}, t_{2,k})), \{\sigma\}, r))} \qquad (\textit{Frame})$$

4.2.10 Environment Types

When both types are environment types then they are decomposed and the pairs of frames at the same position are checked collecting the results. This is done similar to the processing of types constructed by the free type constructors:

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

4.2.11 Function Types and Quantified Variables

If both types t_1 and t_2 are function types then they have common elements when they are equal or one of them is the type *TFunc* of all functions. A quantified variable just has common elements with itself. These cases are formalized in the following function:

Definition 4.2.13 *The function $fq : \mathcal{T} \times \mathcal{T} \to \{\texttt{true}, \texttt{false}\}$ expects two types t_1 and t_2 and returns true in the following cases:*

- *t_1 and t_2 are the same function type.*
- *t_1 and t_2 are both function types and one of them is the type TFunc of* all *functions.*
- *$t_1 = t_2$ and $t_1, t_2 \in V_q$.*

In all cases not mentioned above fq returns ***`false`***.

In all the cases with $\texttt{fq}(t_1, t_2) = \texttt{true}$ no further restrictions must be included into the result of *S-CE*. This yields the following rule:

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

4.2.12 Base Types and Value Assignments

When t_1 and t_2 are both base types or value assignments then the question whether they have common elements is answered by *CEbase* fulfilling Assumption 5.1.9.

The behaviour of *S-CE* is formalized in the following rule:

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

4.2.13 Integrating the Cases

In this subsection we will present the algorithm *S-CE* by integrating the previously introduced rules in a fixed order:

Definition 4.2.14 (algorithm *S-CE*) *The algorithm S-CE checks the previously defined rules in the following order and returns the result given by the first applicable rule:*

$$(\mathit{Top1}), (\mathit{Top2}), (\mathit{RecT}), (\mathit{Rec1}), (\mathit{Rec2}), (\mathit{BothVar}), (\mathit{Var1}), (\mathit{Var2}),$$
$$(\mathit{U1}), (\mathit{U2}), (\mathit{I1}), (\mathit{I2}), (\mathit{Comp1}), (\mathit{Comp2}), (\mathit{Constr}), (\mathit{Frame}), (\mathit{Env}), (\mathit{FunQ}), (\mathit{Base})\,.$$

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

4.2.14 An *S-CE* Example in Detail

The following example shows in detail how calls to *S-CE* are processed and how the use of unions by *extend-constraint* enables *S-CE* to process hetreogenous lists correctly.

Example 4.2.15 (*S-CE*) *Consider the call S-CE(t_1, t_2, σ, r) with:*

$$t_1 = \mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X))$$
$$t_2 = (\boldsymbol{nat}\ .\ (\boldsymbol{string}\ .\ \boldsymbol{nil}))$$
$$\sigma = \emptyset$$
$$r = ()$$

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

$$\mathit{S\text{-}CE}((\cup\ \boldsymbol{nil}\ (A\ .\ t_1)), t_2, \emptyset, r' := ((t_1, t_2)))\,.$$

2. *By rule (U1) the S-CE-call is recursively splitted into two subcalls:*

   (a) S-CE($\textbf{nil}, t_2, \emptyset, r'$) $= \emptyset$

   (b) S-CE($(A\ .\ t_1), t_2, \emptyset, r'$)

3. *S-CE($(A\ .\ t_1), t_2, \emptyset, r'$) is processed by rule (Constr). SCE-list-reduce performs the following subcalls (the result of subcall number i is denoted by Σ_i):*

   (a) $\Sigma_1 =$ S-CE($A, \textbf{nat}, \emptyset, r'$) $= \{\{A \leftarrow \textbf{nat}\}\}$ *by rule (Var1).*

   (b) $\Sigma_2 =$ S-CE($t_1, (\textbf{string}\ .\ \textbf{nil}), \sigma' := \{A \leftarrow \textbf{nat}\}, r'$)

4. *S-CE($t_1, (\textbf{string}\ .\ \textbf{nil}), \sigma', r'$) is processed by rule (Rec1) resulting in the call*

$$\text{S-CE}((\cup\ \textbf{nil}\ (A\ .\ t_1)), (\textbf{string}\ .\ \textbf{nil}), \sigma', r'' = ((t_1, (\textbf{string}\ .\ \textbf{nil})), (t_1, t_2)))\,.$$

5. *By rule (U1) we have two subcalls for the contained call to S-CE:*

   (a) S-CE($\textbf{nil}, (\textbf{string}\ .\ \textbf{nil}), \sigma', r''$) $= \emptyset$

   (b) S-CE($(A\ .\ t_1), (\textbf{string}\ .\ \textbf{nil}), \sigma', r''$)

6. *S-CE($(A\ .\ t_1), (\textbf{string}\ .\ \textbf{nil}), \sigma', r''$) is processed by rule (Constr):*

   (a) $\Sigma'_1 =$ S-CE($A, \textbf{string}, \sigma', r''$) $= \{\sigma'' := \{A \leftarrow (\cup\ \textbf{nat}\ \textbf{string})\}\}$

   (b) $\Sigma'_2 =$ S-CE($t_1, \textbf{nil}, \sigma'', r''$)

7. *The second call is processed by rule (Rec1) producing the call*

$$\text{S-CE}((\cup\ \ \textbf{nil}\ \ (A\ \ .\ \ t_1)), \textbf{nil}, \sigma'', r'''\ \ :=\ \ ((t_1, \textbf{nil}), (t_1, (\textbf{string}\ \ .\ \ \textbf{nil})), (t_1, t_2)))\,.$$

8. *By rule (U1) the call to S-CE causes the following subcalls:*

   (a) S-CE($\textbf{nil}, \textbf{nil}, \sigma'', r'''$) $= \{\sigma''\}$ *by rule (Base).*

   (b) S-CE($(A\ .\ t_1), \textbf{nil}, \sigma'', r'''$) $= \emptyset$

   The call containing the union $(\cup\ \textbf{nil}\ (A\ .\ t_1))$ in step (7) yields the result $\{\sigma''\}$. This is also the result for Σ'_2.

9. *The result of step (5) is given by the union of the two subcalls. Since the first subcall yielded the result $\emptyset$ this is equal to the result of the second subcall which is still $\{\sigma''\}$. Thus, $\{\sigma''\}$ is the result of step (4), of Σ_2 and therefore of step (3).*

10. *The result of step (2) is given by the union of $\emptyset$ from the first subcall and $\{\sigma''\}$ from the second one. $\{\sigma''\}$ is also the union result and the result of step (1).*

4.2.15 Properties of *S-CE*

Lemma 4.2.16 (termination of *S-CE*) *If the algorithm ST terminates for every pair of closed terms in set normalized form then the algorithm S-CE terminates for every input with arguments 1 and 2 in set normalized form.*

Proof: See App. B.1, Page 122. □

The remaining condition on *ST* for termination of *S-CE* is removed later. The unrestricted termination result is given in Cor. 5.3.16.

In the following we will prove certain properties of the result of *S-CE* for every intermediate recursive call occurring during the processing of an initial call to *S-CE*. These intermediate calls are sometimes incomplete because there might be computations that were started before initiating the actual intermediate call but were not finished. Hence, we need the notion of *implicit constraint sets*. Implicit constraint sets are defined in the context of executions of *SCE-list-reduce*.

Informally an implicit constraint set is the union of the results of all *open S-CE calculations* i.e. calculations that are given in the argument list of a call to *SCE-list-reduce* but where not yet processed. Open *S-CE* calculations are defined as follows:

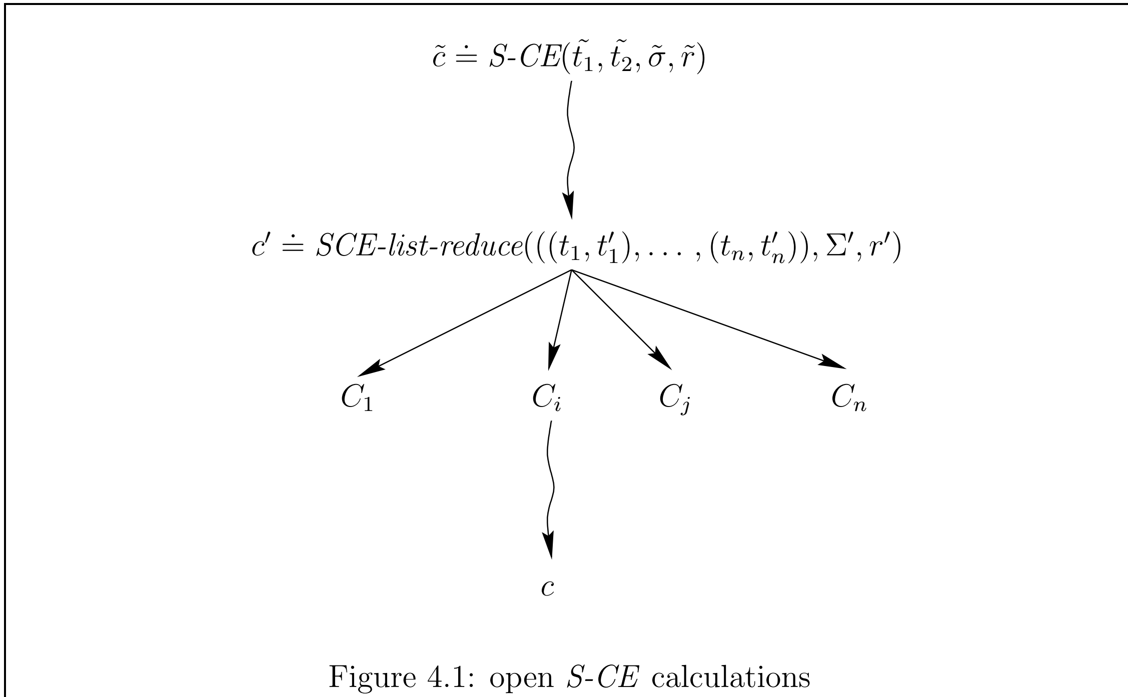

Figure 4.1: open *S-CE* calculations

Definition 4.2.17 (open *S-CE* calculation) *Let*

$$\tilde{c} \doteq \textit{S-CE}(\tilde{t_1}, \tilde{t_2}, \tilde{\sigma}, \tilde{r})$$

be a call to S-CE and

$$c' \doteq \textit{SCE-list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r')$$

a call to SCE-list-reduce occurring as (maybe indirect) subcall of $\tilde{c}$. Let the first argument of SCE-list-reduce be a list with the n elements (t_i, t'_i) for $i \in \{1, \ldots, n\}$. For every $j \in \{1, \ldots, n\}$ let

$$c_{j,l} \doteq \textit{S-CE}(t_j, t'_j, \sigma_{j,l}, r) \text{ with } l \in \{1, \ldots, m_j\}$$

be the calls to S-CE initiated by SCE-list-reduce when processing the jth list element and let

$$C_j = \{c_{j,1}, \ldots, c_{j,m_j}\}\,.$$

Let c be a call to S-CE that is either $c_{i,l} \in C_i$ or a subcall of $c_{i,l}$ for some i and some l (cf. Fig. 5.1). Then every C_j with $i \leq j \leq n$ is an open *S-CE* calculation *of c with respect to $\tilde{c}$.*

The fact that C_i is an open *S-CE* calculation of c might be confusing. But this definition is necessary in order to provide the constraints given by C_i for the $c_{i,l}$ being processed after c in the following definition of implicit constraint sets.

When applying *SCE-list-reduce* the constraints for the individual list entries of the first argument could essentially be calculated independently from each other (with the empty free constraint set as third argument) and combined afterwards. For the presentation of *SCE-list-reduce* we calculated them step by step and combined an intermediate result directly on the fly. The idea behind implicit constraint sets given in Def. 5.2.18 is to perform all open *S-CE* calculations independently from an already known intermediate result and to combine the results.

Note that in the following definition there can be several calls c' for given c and $\tilde{c}$. The elements set of *all* corresponding open *S-CE* calculations are enumerated from 1 to n.

Definition 4.2.18 (implicit constraint sets) *Let $c := \textit{S-CE}(t, t', \sigma, r)$ be a call to S-CE steaming from an initial call $\tilde{c} := \textit{S-CE}(\tilde{t}, \tilde{t'}, \emptyset, ())$. Let $C_1, \ldots, C_n$ be the open S-CE calculations of c with respect to $\tilde{c}$. Let $C_j = \{c_{j,1}, \ldots, c_{j,m_j}\}$ with $c_{j,l} \doteq \textit{S-CE}(t_j, t'_j, \sigma_{j,l}, r)$ and let $\Sigma_j = \textit{S-CE}(t_j, t'_j, \emptyset, r)$ for every $j \in \{1, \ldots, n\}$.*

Then the implicit constraint set of c with respect to $\tilde{c}$ is

$$\textit{implicit-cs}_{\tilde{c}}(c) := \bigotimes_{j=1,\ldots,n} \Sigma_j\,.$$

If the set of open S-CE calculations of c with respect to $\tilde{c}$ *is empty we define*

$$implicit\text{-}cs_{\tilde{c}}(c) := \{\emptyset\}\,.$$

The following Lemma 5.2.19 states the main property of implicit constraint sets: When processing a call to *SCE-list-reduce* we can stop the calculation at every list element of the first argument and combine the intermediate result with the corresponding implicit constraint set without changing the result.

The following lemma states that calculating *SCE-list-reduce* as given in Def. 5.2.1 is equivalent to calculating the implicit constraint set of the initial call. I.e. the constraints generated by the subcalls to *S-CE* initiated by *SCE-list-reduce* can also be generated independently and be combined afterwards.

Lemma 4.2.19 (implicit constraint sets) *Let* $\tilde{c}$ *be a call to S-CE and* c' *the first call to SCE-list-reduce performed when processing* $\tilde{c}$*. Let* Σ *be the result of a call*

$$c' \doteq SCE\text{-}list\text{-}reduce((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)$$

and let C_j *denote the set of recursive calls to S-CE when processing the* j^{th} *list element. Let* Σ_j *be the c-collection used in the recursive call*

$$SCE\text{-}list\text{-}reduce(((t_{j+1}, t'_{j+1}), \ldots, (t_n, t'_n)), \Sigma_j, r)$$

for all j *and* $\Sigma'_j = S\text{-}CE(t_j, t'_j, \emptyset, r)$ *then:*

$$\forall_{l=0}^{n} combine\text{-}cs(\Sigma) = combine\text{-}cs(\Sigma_l \otimes \bigcup_{j=l}^{k} \Sigma'_j) = combine\text{-}cs(\Sigma_l \otimes implicit\text{-}cs_{\tilde{c}}(C_l))\,.$$

where $\Sigma_0 = \Sigma' = \Sigma'_0$.

Proof: See App. B.1, Page 124. □

The following lemma states the main property of *S-CE*'s output: Consider two types t_1 and t_2 that have a common element v under a free constraint set $\tilde{\sigma}$ (precisely its natural free type substitution). Consider a call to *S-CE* with t_1, t_2 and $\tilde{\sigma}$ as arguments and a recursion history r that is not arbitrary chosen but generated by *S-CE* starting with an initial call with empty recursion history. Then the result contains a free constraint set σ that essentially restrict t_1 and t_2 to types containing v.

But the free constraint sets σ (more precisely the corresponding natural free type substitutions) in general are not idempotent. Thus, the lemma precisely states common elements of t_1 and t_2 after σ has been applied to t_1 and t_2 an arbitrary number k of times.

The correctness of *S-CE* is stated by the following Lemma 5.2.20: If during evaluating the call $S\text{-}CE(\tilde{t_1}, \tilde{t_2}, \emptyset, ())$ there is a subcall $S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$ then the set of common elements of the type pair (t_1, t_2) is preserved under building all instances $\sigma(t_1)$ and $\sigma(t_2)$ with $\sigma \in \Sigma = S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$. Furthermore, changing $\tilde{\sigma}$ to one of the $\sigma \in \Sigma$ just enlarges the set of values each variable is constrained to.

Lemma 4.2.20 (correctness of *S-CE*) *Let $t_1, t_2 \in \mathcal{T}$, both in set normalized form and let $c := S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$ be a call to S-CE that occurs as a recursive call after an initial call $\tilde{c}$ to S-CE with empty recursion information $()$. Let CE fulfill Assumption 3.4.6. Let there exist a value $v \neq \perp$ such that*

$$\forall k \in \mathbb{N} \,.\, v \in subst(\tilde{\sigma})^k(t_1) \boxdot subst(\tilde{\sigma})^k(t_2)\,. \tag{4.2}$$

Then there exist a free constraint set

$$\sigma \in combine\text{-}cs\text{-}cond_{(c,\tilde{c})}(S\text{-}CE(t_1, t_2, \tilde{\sigma}, r) \otimes implicit\text{-}cs_{\tilde{c}}(c))\ ^2$$

such that the free type substitution $\sigma' = subst(\sigma)$ compatible with σ fulfills

$$\forall k \in \mathbb{N} \,.\, v \in \sigma'^k(t_1) \boxdot \sigma'^k(t_2)\,. \tag{4.3}$$

Furthermore, if X is a free variable with $X \in dom(\tilde{\sigma})$ and $\tilde{\sigma}(X) = \tilde{t}_X$ then $X \in dom(\sigma)$ for every $\sigma \in S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$ and $t_X = \sigma(X)$ fulfills:

$$\langle\!\lbrack \tilde{t}_X \rbrack\!\rangle(\tau) \subseteq \langle\!\lbrack t_X \rbrack\!\rangle(\tau) \tag{4.4}$$

for every closed type substitution τ appropriate for $\tilde{t}_X$ and t_X.

Proof: See App. B.1, Page 128. □

Note that the second part of (5.2) in Lemma 5.2.20 is necessary because of the special understanding of quantified variables. Two types are just considered as types with common elements if there are common elements under *every* instantiation of the quantified variables with types $\neq \top$. E.g. the type (`list` $X_\forall$) has common elements with (`list` $\top$) but not with (`list` $Y_\forall$).

[2]The function *combine-cs-cond* conditionally calls *combine-cs* on its argument depending on its subscripts: It behaves like *combine-cs* if c is a (maybe indirect) recursive subcall of an execution of Rule (*Rec1*) or (*Rec2*) in the context of $\tilde{c}$. Otherwise, *combine-cs-cond* is the identity.

4.2.16 The Order of *S-CE* Rules

Though Lemma 5.2.20 states the correctness of *S-CE* independently from the order in which the rules are checked the precision of *S-CE* depends on the given order. This problem becomes obvious when considering intersection types not containing any elements as the following example shows:

Example 4.2.21 (order of *S-CE* rules) *Consider the call*

$$\textit{S-CE}(t_1, t_2, \emptyset, ()) \text{ with } t_1 = (\cup\ \textbf{int}\ \textbf{bool}), t_2 = (\cap\ \textbf{int}\ \textbf{bool})\,.$$

When decomposing the union first the result is given by the union of the results of the two subcalls S-CE($\textbf{int}, t_2, \emptyset, ()$ *) and S-CE(* $\textbf{bool}, t_2, \emptyset, ()$ *). Both of them fail and return the empty c-collection. As a result the c-collection returned by rule (U1) is also empty denoting no common elements.*

If on the other hand the intersection is decomposed first the result of

$$\textit{SCE-list-reduce}(((t_1, \textbf{int}), (t_1, \textbf{bool})), \emptyset, ())$$

is returned. SCE-list-reduce performs the subcalls

$$\textit{S-CE}(t_1, \textbf{int}, \emptyset, ()) \text{ and } \textit{S-CE}(t_1, \textbf{bool}, \emptyset, ())\,.$$

Both of these calls return $\Sigma_0 = \{\emptyset\}$*. The result of SCE-list-reduce is* Σ_0*. This is less precise than the failure in the case above.*

A discussion on all order dependencies between two rules is given in the following Remark 5.2.22:

Remark 4.2.22 (ordering of the *S-CE* cases) *On the one hand S-CE contains rules with conditions based on the structure of both terms* t_1 *and* t_2*. On the other hand the applicability of the rules (Rec1), (Rec2), (Var1), (Var2), (U1), (U2), (I1), (I2), (Comp1) and (Comp2) just relies on one of the terms. We discuss here that the chosen ordering of the last kind of rules is reasonable in order to get as precise results as possible.*

- *As we will see below there are indeed cases whose order is crucial for the desired result. The unfolding of recursive types itself does not interfere with any of the other cases directly. But when performed to late constructors that have to be decomposed early can be hidden by the* μ*-constructor. If e.g.* $t_1 = \mu X.(\cup\ \ldots\ C[X]\ \ldots)$ *and* $t_2 = (\cap\ \ldots)$ *then the unfolding of* t_1 *has to be done before decomposing* t_2*. The correct solution is to perform the unfolding of recursive types in the rules (Rec1) and (Rec2) before all other cases discussed here.*

- *To treat free variables in the next step is correct and useful in situations where one type to be tested is a free type variable X and the other one is constructed by $\cup$, $\cap$ or $\mathcal{C}$. In this case the most precise result we can get is to constrain X to the constructed type. This is just possible when this constructed type has not been decomposed, already.*

- *Let $t_1 = (\cup\ t_{1,1}\ \dots\ t_{1,k})$ and $t_2 = (\cap\ t_{2,1}\ \dots\ t_{2,k'})$. A value $v \in \langle\!|\tau(t_1)|\!\rangle \cap \langle\!|\tau(t_2)|\!\rangle$ for an arbitrary substitution τ must occur in* one *of the $\langle\!|\tau(t_{1,i})|\!\rangle$ and in* all *of the $\langle\!|\tau(t_{2,j})|\!\rangle$. Therefore to get common elements of t_1 and t_2* all *of the $\tau(t_{2,j})$ must have common elements with* the same *$\tau(t_{1,i})$. When first decomposing the intersection of t_2 S-CE checks whether all $\tau(t_{2,j})$ have common elements with* any *of the $\tau(t_{1,i})$. When on the other hand the union of t_1 is decomposed first then the desired stricter property is checked. It is therefore reasonable to decompose unions by the rules (U1) and (U2) before decomposing intersections by the rules (I1) and (I2).*
  Now let $t_1 = (\cup\ t_{1,1}\ \dots\ t_{1,k})$ and $t_2 = \mathcal{C}t'_2$. We know that t'_2 is not a union or intersection type because otherwise t_2 not in set normalized form. When first processing the complement type then we check the property

  $$\neg ST(\sigma(t_1), t'_2) \Leftrightarrow \neg \bigwedge_{i=1}^{k} ST(\sigma(t_{1,i}), t'_2) \Leftrightarrow \bigvee_{i=1}^{k} \neg ST(\sigma(t_{1,i}), t'_2)$$

  Decomposing the union first unites the results of checking $\neg ST(\sigma(t_{1,i}), t'_2)$ where the union behaves like disjunction when t_1 is already a semi-closed type. Thus, for a semi-closed t_1 both orders yield the same result, but for t_1 containing free variables processing the complement first introduces constraints that might be unnecessary. On the other hand decomposing the union first will introduce constraints containing $\top$ (done by free-to-top) just for certain free constraint sets in the returned c-collection and hence it is correct and reasonable to decompose unions before processing complements.
  Altogether it is correct to check the rules (U1) and (U2) before (I1), (I2), (Comp1) and (Comp2).

- *Let $t_1 = (\cap\ t_{1,1}\ \dots\ t_{1,k})$ and $t_2 = \mathcal{C}t'_2$. Again, t'_2 is not a union or intersection type because t_2 is in set normalized form and therefore the complement constructor cannot hide a constructor that must be processed first. If t_1 is a semi-closed type then processing the complement first yields the test*

  $$\neg ST(\sigma(t_1), t'_2) \Leftrightarrow \neg \bigvee_{i=1}^{k} ST(\sigma(t_{1,i}), t'_2) \Leftrightarrow \bigwedge_{i=1}^{k} \neg ST(\sigma(t_{1,i}), t'_2)$$

  On the other hand decomposing the intersection first causes a sequence of tests of the form $\neg ST(\sigma(t_{1,i}), t'_2)$ without changing the substitution because both types are already semi-closed. This sequence of tests is semantically equivalent to the conjunction above. Because of this it is correct to check the rules (I1) and (I2) before (Comp1) and (Comp2).

In the steps above we have proven the correctness and reasonability of the order (Rec1), (Rec2) → (Var1), (Var2) → (U1), (U2) → (I1), (I2) → (Comp1), (Comp2) in which S-CE applies the cases that just depend one one of the argument types. The order of the other cases can be chosen arbitrarily (except of (RecT) that must be checked before (Rec1) and (Rec2) in order to guarantee termination).

4.2.17 Further Optimizations

Recall the use of the function *combine-cs* in the rules (*Rec1*) and (*Rec2*) of *S-CE*. They were used to combine several free constraint sets generated after unfolding one of the argument types. Unfortunately, the application of *combine-cs* destroys some of the precision we got from the use of several different free constraint sets as results of the rules (*U1*) and (*U2*).

In the proof of Lemma 5.2.20 the fact that (*Rec1*) and (*Rec2*) apply *combine-cs* was just used for proving the correctness of (*RecT*). Indeed, when a recursive subcall generated by (*Rec1*) or (*Rec2*) returns without applying (*RecT*) we did not cut an execution that would lead to combinations of the different free constraint sets given in the intermediate result of the subcall of (*Rec1*) or (*Rec2*) and therefore the application of *combine-cs* is unnecessary.

Example 4.2.23 *Consider the types*

$$t_1 = \mu X.((\cup\ f(\textbf{\textit{bool}})\ f(\textbf{\textit{int}}))\ .\ (\cup\ \textbf{\textit{nil}}\ X))$$
$$t_2 = (f(A)\ .\ (f(A)\ .\ \textbf{\textit{nil}}))\,.$$

The call S-CE$(t_1, t_2, \emptyset, ())$ *is processed by rule (Rec1) initiating the subcall S-CE*$(t_1', t_2, \emptyset, r)$ *with* $t_1' = \mathit{unfold}(t_1)$ *and* $r = ((t_1, t_2))$. *This call is decomposed by rule (Constr) and the first subcall S-CE*$((\cup\ f(\textbf{\textit{bool}})\ f(\textbf{\textit{int}})), f(A), \emptyset, r)$ *processed by (U1) and (Constr) and (Var2) for every subcall yields* $\Sigma_1 = \{\{A \leftarrow \textbf{\textit{bool}}\}, \{A \leftarrow \textbf{\textit{int}}\}\}$.

For every free constraint set $\sigma \in \Sigma_1$ *the function SCE-list-reduce initiates a subcall*

$$\textit{S-CE}((\cup\ \textbf{\textit{nil}}\ t_1), (f(A)\ .\ \textbf{\textit{nil}}), \sigma, r)\,.$$

(We will just discuss the call for $\sigma_1 = \{A \leftarrow \textbf{\textit{bool}}\}$ *in detail.) The call is processed by (U1) with the first subcall S-CE*$(\textbf{\textit{nil}}, (f(A)\ .\ \textbf{\textit{nil}}), \sigma_1, r)$ *returning with the empty c-collection as result. The second subcall S-CE*$(t_1, (f(A)\ .\ \textbf{\textit{nil}}), \sigma_1, r)$ *is processed by rule (Rec1) yielding the subcall*

$$\textit{S-CE}(t_1', (f(A)\ .\ \textbf{\textit{nil}}), \sigma_1, r') \text{ with } r' = ((t_1, (f(A)\ .\ \textbf{\textit{nil}})), (t_1, t_2))\,.$$

Again, (Constr) initiates subcalls for two type pairs. The first of them returns

$$\Sigma_{2,1} = \{\{A \leftarrow (\cup\ \textbf{\textit{bool}}\ \textbf{\textit{bool}})\}, \{A \leftarrow (\cup\ \textbf{\textit{bool}}\ \textbf{\textit{int}})\}\}\,.$$

The two subcalls S-CE$((\cup\ \boldsymbol{nil}\ t_1), \boldsymbol{nil}, \sigma, r')$ *for* $\sigma \in \Sigma_{2,1}$ *just return* σ. *Thus,* $\Sigma_{2,1}$ *is returned.*

Analogously, for $\sigma_2 = \{A \leftarrow \boldsymbol{int}\} \in \Sigma_1$ *we get*

$$\Sigma_{2,2} = \{\{A \leftarrow (\cup\ \boldsymbol{int}\ \boldsymbol{bool})\}, \{A \leftarrow (\cup\ \boldsymbol{int}\ \boldsymbol{int})\}\}\,.$$

Altogether, without applying combine-cs the initial call to S-CE returns $\Sigma = \Sigma_{2,1} \cup \Sigma_{2,2}$. *This result is correct because there was no application of rule (RecT) cutting a necessary computation.*

Furthermore, the application of *combine-cs* is just necessary if (*RecT*) was applied to the argument pair (t_1, t_2) the actual application of (*Rec1*) or (*Rec2*) inserted into the recursion information. For all other applications of (*RecT*) there exists a corresponding application of (*Rec1*) or (*Rec2*) and it suffices to apply *combine-cs* there.

Example 4.2.24 *Recall Example 5.2.8. The initial call S-CE*$(t_1, t_2, \emptyset, ())$ *is processed by (Rec1) yielding the call S-CE*$(t_1', t_2, \emptyset, ((t_1, t_2)))$. *This call is processed by (Rec2) which yields S-CE*$(t_1', t_2', \emptyset, ((t_1', t_2), (t_1, t_2)))$.[3] *The only type pair processed by the rule (RecT) is* (t_1, t_2) *and it suffices to apply combine-cs to the result of the subcall of (Rec1).*

In order to perform this optimization we have to extend the return value of *S-CE*: Instead of just returning a c-collection Σ, *S-CE* has to return a pair $(\Sigma, \mathcal{R})$ where Σ is a c-collection as before and $\mathcal{R}$ is a set of type pairs of types (*RecT*) was applied on. This *recursion termination history* $\mathcal{R}$ is maintained as follows:

- For a call *S-CE*(t_1, t_2, σ, r) the rules (*RecT*) returns $(\{\sigma\}, \{(t_1, t_2)\})$.
- The behaviour of (*Rec1*) and (*Rec2*) applied to a type pair (t_1, t_2) depends on the result $(\Sigma, \mathcal{R})$ of its recursive subcall as follows:
  - If $(t_1, t_2) \in \mathcal{R}$ then the return value of (*Rec1*) or (*Rec2*), respectively, is

    $$(\textit{combine-cs}(\Sigma), \mathcal{R} \setminus \{(t_1, t_2\})\,.$$

  - Otherwise, $(\Sigma, \mathcal{R})$ is returned.
- The rules (*U1*), (*U2*), (*I1*), (*I2*), (*Constr*), (*Frame*), (*Env*) return the union of the recursion termination histories of their subcalls.
- The rules (*Top1*), (*Top2*), (*BothVar*), (*Var1*), (*Var2*), (*Comp1*), (*Comp2*), (*FunQ*), (*Base*) return an empty recursion termination history.

[3] $t_1 = \mu X.((\cup\ f(\texttt{bool})\ f(\texttt{int}))\ .\ (\cup\ \texttt{nil}\ X))$, $t_2 = \mu Y.(f(A)\ .\ (\cup\ \texttt{nil}\ Y))$, $t_i' = \textit{unfold}(t_i)$ as in Example 5.2.8.

Example 4.2.25 *Consider the following two types*

$$t_1 = \mu X.(\cup\ \textbf{\textit{nil}}\ ((\textbf{\textit{int}}\ .\ A)\ .\ X))$$
$$t_2 = ((\cup\ (\textbf{\textit{int}}\ .\ \textbf{\textit{posint}})\ (\textbf{\textit{int}}\ .\ \textbf{\textit{bool}}))\ .\ \textbf{\textit{nil}})$$

After the first unfolding step by rule (Rec1) S-CE checks $(\cup\ \textbf{\textit{nil}}\ ((\textbf{\textit{int}}\ .\ A)\ .\ t_1))$ *and* t_2*. Rule (U1) initiates two subcalls checking*

1. ***nil*** *and* t_2
2. $((\textbf{\textit{int}}\ .\ A)\ .\ t_1)$ *and* t_2

The first of these subcalls yields the empty c-collection while the second subcall is processed by rule (Constr) which initiates a call to SCE-list-reduce with the following argument pairs:

- $(\textbf{\textit{int}}\ .\ A)$ *and* $(\cup\ (\textbf{\textit{int}}\ .\ \textbf{\textit{posint}})\ (\textbf{\textit{int}}\ .\ \textbf{\textit{bool}}))$
- t_1 *and* ***nil***

The first of these subcalls yields the c-collection $\Sigma = \{\sigma_1, \sigma_2\}$ *with* $\sigma_1 = \{A \leftarrow \textbf{\textit{posint}}\}$ *and* $\sigma_2 = \{A \leftarrow \textbf{\textit{bool}}\}$*. The second subcall succeeds for both* σ_1 *and* σ_2 *without changing any of these free constraint sets.*

In the optimized algorithm S-CE the rule (Rec1) just passes through the result Σ *of unfolding* t_1 *the first time. The original algorithm applies combine-cs to* Σ *yielding*

$$\Sigma' = \{\sigma_1, \sigma_2, \{A \leftarrow (\cup \textbf{\textit{posint}}\ \textbf{\textit{bool}}\}\}\,.$$

By introducing a further free constraint set the unnecessary call to combine-cs causes the loss of information.

As the example shows calling *combine-cs* just when necessary causes the rules (*Rec1*) and (*Rec2*) to provide a more precise output. Furthermore, *combine-cs* is a quite expensive operation and calling it just when necessary makes *S-CE* more efficient.

4.3 The Algorithm *CE*

The algorithm *CE* is the main algorithm approximating the question of common elements of two given types t_1 and t_2. Its first step is to call *S-CE* with the types t_1 and t_2, an

empty constraint set and an empty recursion information as input. The c-collection resulting from this call (in the case of success) consists of free constraint sets whose natural free type substitutions are not idempotent. *CE* transforms these free constraint sets into idempotent free type substitutions independently from each other. This is done by repeatedly inserting the corresponding values for the variables into the right hand sides occurring in the substitution. Recursive dependencies between variables are eliminated by introducing recursive bindings by the μ constructor.

The following example shows the intended result of *CE* for a variable depending on itself:

Example 4.3.1 *Consider a call to S-CE which result just contains the free constraint set* $\{X \leftarrow f(X)\}$ *with a free type constructor* $f(\cdot)$ *of arity* 1*. We want CE to transform this free constraint set into the idempotent free type substitution* $\{X \leftarrow \mu Y.f(Y)\}$*.*

For several variables that mutually depend on each other the intended result is presented in the following example:

Example 4.3.2 *Consider a call to S-CE returning the c-collection* $\Sigma = \{\sigma\}$

$$\{\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(X_1)\}\}\,.$$

The dependencies between the variables are given in Fig. 5.2 where an arrow from a variable V *to a type* t *expresses that* V *is constrained to* t*.*

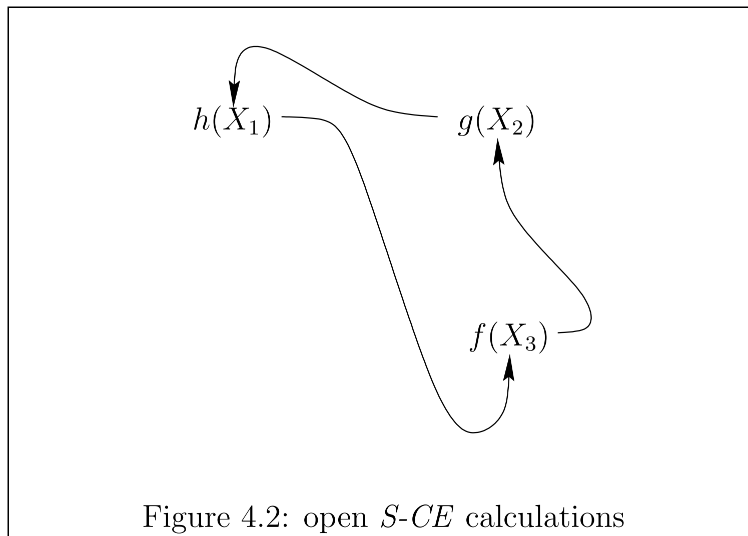

Figure 4.2: open *S-CE* calculations

The intended result of CE is an s-collection consisting of the idempotent free type substitution

$$\begin{aligned}\{X_1 &\leftarrow f(g(\mu V_3\,.\,h(f(g(V_3))))),\\ X_2 &\leftarrow g(\mu V_3\,.\,h(f(g(V_3)))),\\ X_3 &\leftarrow \mu V_3\,.\,h(f(g(V_3)))\}\,.\end{aligned}$$

4.3.1 The Core Component of *CE*

The following definition presents the core component of *CE*. It consists of a call to *S-CE* and a loop transforming every free constraint set σ' in the result Σ' of *S-CE* into a free type substitution. All these free type substitutions are collected in an s-collection Σ.

Definition 4.3.3 (algorithm *CE*) *The algorithm CE takes two terms and returns an s-collection* Σ*.*

Algorithm: *CE*
Input: *Two terms* t_1 *and* t_2*.*
Output: *An s-collection* Σ*.*

$\Sigma' := \textit{S-CE}(t_1, t_2, \emptyset, ())$
$\Sigma := \emptyset$
forall $\sigma' \in \Sigma'$ ***do*** *(* Transform free constraint sets into free type substitutions *)*
  $\sigma'' := \textit{GIS}(\sigma')$ *(* generate idempotent substitution from* σ' **)*
  $\Sigma := \Sigma \cup \{\sigma''\}$ *(* collect result substitutions in s-collection* Σ **)*
end *(* forall *)*
return Σ

4.3.2 Auxiliary Functions used by *CE*

The main task of *CE*, i.e. the elimination of the variables from the right hand sides of a single free type substitution is done by *GIS*. It is done by calculating an order in which the assignments to the variables depend on each other. Following this order, *GIS* inserts the right hand sides of already processed variables into the right hand sides of different variables. If a variable depends on itself a recursive type is generated. *GIS* is defined as follows:

Definition 4.3.4 (algorithm *GIS*) *The algorithm GIS expects a free type substitution and returns a free type substitution that is idempotent.*

Algorithm: *GIS (generate idempotent substitution)*
Input: *A substitution* σ
Output: *An idempotent substitution* $\tilde{\sigma}$*.*

$G := (V, R)$ *with* $V = dom(\sigma')$ *and* $R = \{(y, x) \mid x \neq y, x \leftarrow t \in \sigma'$ *and* y *is subterm of* $t\}$
$G' := (V', R')$ *is the component graph of* G*. (* cf. Def. 2.2.4 *)*
Mark all nodes $v \in V'$ *with* 0*.*
$\sigma'' := \emptyset$
while *there are nodes* $v \in V'$ *marked with* 0 ***do***
  select $v' \in V'$ *with all predecessors of* v' *marked with* 1
  mark v' *with* 1
  if v' *represents a single node in* G ***then***
    $t := \sigma'(v')$ *(* lookup* v' *in* σ' **)*
    $t' := \sigma''(t)$ *(* apply already known substitution* σ'' **)*
    if t' *contains* v' *as subterm*
      then $t' := \mu X.t'[v'/X]$ *with a new variable* $X \in V_f$
    $\sigma'' := \sigma'' \cup \{v' \leftarrow t'\}$
  else *(** v' *represents more than one node in* V **)*
    Let $\tilde{V} \subseteq V$ *be the set of nodes* $v \in V$ *represented by* v'*.*
    $\sigma_r := \{X \leftarrow t \mid X \in \tilde{V}, X \leftarrow \tilde{t} \in \sigma', t = \sigma''(\tilde{t})\}$ *(** $\sigma_r = \sigma'' \circ \sigma'|_{\tilde{V}}$ **)*
    $\tilde{\sigma} := SMR(\sigma_r)$
    $\sigma'' := \sigma'' \cup \tilde{\sigma}$
  end *(* if-then-else *)*
end *(* while *)*
return σ''

For nodes $v' \in V'$ denoting more than one node in V (i.e. for several variables mutually depending on each other) *GIS* extracts a free type substitution restricted to the nodes denoted by v' and passes the processing to *SMR*.
The algorithm *SMR* performs insertions of variables and the introduction of recursive types in a certain order to eliminate cyclic dependencies. This order $<_V$ on the variables must be fixed but can be chosen arbitrarily. *SMR* is defined as follows:

Definition 4.3.5 (algorithm *SMR*) *SMR needs an 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 *in its own assigned term are expressed by a recursive type:*

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

for $i = 1$ ***to*** k ***do***
  $t := \sigma(X_i)$
  if X_i *occurs in* t_{X_i} *(* 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* $\tilde{X}_i \in V_f$
    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* $j > i$ **)*
    (* $\sigma := (\sigma|_{\{X_i\}} \circ \sigma_{\{X_{i+1},...,X_k\}}) \cup \sigma_{\{X_1,...,X_i\}}$ ***)**
    $t_{X_j} := \sigma(X_j)$
    $t'_{X_j} := t_{X_j}[X_i/t'_{X_i}]$
    $\sigma := \sigma \setminus \{X_j \leftarrow t_{X_j}\} \cup \{X_j \leftarrow t'_{X_j}\}$
  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* $j < i$ **)*
    $t_{X_j} := \sigma(X_j)$
    $t'_{X_j} := t_{X_j}[X_i/t'_{X_i}]$
    $\sigma := \sigma \setminus \{X_j \leftarrow t_{X_j}\} \cup \{X_j \leftarrow t'_{X_j}\}$
  end *(* for j *)*
end *(* for i *)*
In every t_{X_j} *replace every* $\mu\tilde{X}_i.t$ *in a position where* $\tilde{X}_i$ *is already bound by some* μ *constructor.*

Example 4.3.6 (*SMR*) *Consider a call to SMR with the input substitution* σ *defined by*

$$\sigma = \{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(X_1)\}\,.$$

Assume that $X_1 <_V X_2 <_V X_3$*. Processing the first i loop for* $i = 1$ *SMR inserts* X_1 *into the right hand side of the variables that are greater with respect to* $<_V$*. This insertion yields*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(f(X_2))\}\,.$$

For $i = 2$*,* X_2 *is inserted into the right hand side of* X_3*. The result is:*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(f(g(X_3)))\}\,.$$

For $i = 3$*, the occurrence of* X_3 *in its own right hand side is eliminated by introducing a recursive type:*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow \mu V_3\,.\,h(f(g(V_3)))\}$$

Now the second i-loop is processed. For $i = 3$*, SMR inserts the value of* X_3 *into the right hand sides of all variables that are smaller with respect to* $<_V$*. The result is*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(\mu V_3\,.\,h(f(g(V_3)))), X_3 \leftarrow \mu V_3\,.\,h(f(g(V_3)))\}\,.$$

With $i = 2$ the same is done for X_2:

$$\{X_1 \leftarrow f(g(\mu V_3 \,.\, h(f(g(V_3))))), X_2 \leftarrow g(\mu V_3 \,.\, h(f(g(V_3)))), X_3 \leftarrow \mu V_3 \,.\, h(f(g(V_3)))\}$$

For $i = 1$, there is nothing to be done and the substitution above is returned as result of SMR.

Note that the result substitutions provided by *GIS* and *SMR* are *idempotent*.

4.3.3 Examples of Calls to *CE*

Recalling the input of Example 5.2.15 we get the following example for *CE*:

Example 4.3.7 (*CE*) *Consider t_1 and t_2 as defined in Ex. 5.2.15 and a call CE(t_1, t_2). This call causes the subcall to S-CE discussed in Ex. 5.2.15 and yields the result calculated there:*

$$\Sigma' := \{\{A \leftarrow (\cup\ \mathtt{string}\ \mathtt{nat})\}\}$$

For $\sigma' = \{A \leftarrow (\cup\ \mathtt{string}\ \mathtt{nat})\}$, the algorithm CE generates the graph $G = (V := \{A\}, R := \emptyset)$ and the component graph $G' = G$. The only node $A \in V'$ represents the single node $A \in V$ and $t' = \sigma(A)$ does not contain A as a subterm. Thus, $\sigma'' := \{A \leftarrow (\cup\ \mathtt{string}\ \mathtt{nat})\}$ and $\Sigma := \{\sigma''\}$ is returned.

The following example resents some extended work of *CE* including real work for the subroutines *GIS* and *SMR*:

Example 4.3.8 (*CE*) *Consider the types*

$$t_1 = (X\ .\ (Y\ .\ (Z\ .\ \mathtt{nil})))$$

and

$$t_2 = ((Y\ .\ X)\ .\ (Z\ .\ ((\mathtt{num}\ .\ X)\ .\mathtt{nil})))\,.$$

When fixing $<_V$ to $X <_V Y <_V Z$ then S-CE returns the following c-collection:

$$\Sigma' := \{\{X \leftarrow (Y\ .\ X), Y \leftarrow Z, Z \leftarrow (\mathtt{num}\ .\ X)\}\}$$

The graph G calculated in GIS for the only element σ' of Σ' is given in Fig. 5.3. It has exactly one strongly connected component and thus the component graph G' consists of exactly one node.

Processing the only node of G' yields a call to SMR with the σ' as argument. The individual iterations of the first i-loop performs the following changes:

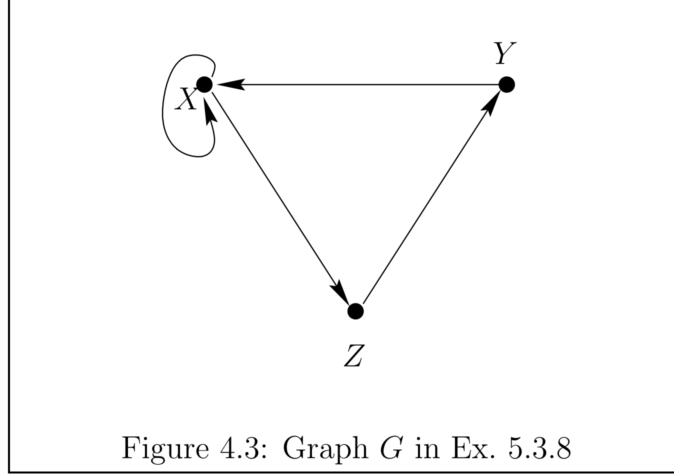

Figure 4.3: Graph G in Ex. 5.3.8

1. *X is recursively bound by a μ constructor in $t_X = (Y\ .\ X)$ yielding $t'_X = \mu X.(Y\ .\ X)$. In t_Z X is replaced by t'_X yielding $(\mathit{num}\ .\ \mu X.(Y\ .\ X))$.*
2. *For $i = 2$, t_Y is not changed because it does not contain Y as subterm. In t_Z as generated in the step before Y is replaced by t_Y yielding $(\mathit{num}\ .\ \mu X.(Z\ .\ X))$.*
3. *The occurrence of Z in t_Z from the step before is recursively bound by μ. The resulting term is $\mu Z.(\mathit{num}\ .\ \mu X.(Z\ .\ X))$*

After processing the first i-loop the intermediate free variable constraint is:

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

The second i-loop has a descending argument. For the individual values of i the following tasks are performed:

3. *$t_Y = Z$ is replaced by $t_Z = \mu Z.(\mathit{num}\ .\ \mu X.(Z\ .\ X))$.*
2. *Y is replaced by t_Y from the step before in t_X yielding $\mu X.(\mu Z.(\mathit{num}\ .\ \mu X.(Z\ .\ X))\ .\ X)$.*
1. *Nothing to do.*

The resulting free constraint set after executing the second i-loop is:

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

The nested μ binding of X in t_X can now be removed, yielding $\mu X.(\mu Z.(\textit{num} . (Z . X)) . X)$.

The resulting substitution is the only element of the s-collection Σ returned by CE, i.e.:

$$\begin{aligned}\Sigma = \{\{X &\leftarrow \mu X.(\mu Z.(\textit{num} . (Z . X)) . X),\\ Y &\leftarrow \mu Z.(\textit{num} . \mu X.(Z . X)),\\ Z &\leftarrow \mu Z.(\textit{num} . \mu X.(Z . X))\}\}\end{aligned}$$

We can show that this is an s-collection by renaming of the μ-bounded variables resulting in:

$$\begin{aligned}\Sigma = \{\{X &\leftarrow \mu V.(\mu W.(\textit{num} . (W . V)) . V),\\ Y &\leftarrow \mu W.(\textit{num} . \mu V.(W . V)),\\ Z &\leftarrow \mu W.(\textit{num} . \mu V.(W . V))\}\}\end{aligned}$$

In the example above the only free type substitution $\sigma \in \Sigma$ has the property that every $v \in t_1 \boxdot t_2$ fulfills $v \in \sigma(t_1) \boxdot \sigma(t_2)$.

4.3.4 Properties of *CE*

In this section we prove several properties of *CE* that are necessary in order to make *CE* practically usable.

In order to prove termination and correctness of *CE* for every input we first prove these properties for the auxiliary functions *SMR* and *GIS*:

Lemma 4.3.9 (termination of *SMR*) *The algorithm SMR terminates for every input substitution with a finite domain $dom(\sigma)$.*

Proof: See App. B.2, Page 136. $\square$

Lemma 4.3.10 (correctness of *SMR*) *Let σ' be a substitution such that the graph $G = (V, R)$ defined as in GIS with $V = dom(\sigma')$ and $R = \{(y, x) \mid x \neq y, x \leftarrow t \in \sigma'$ and y is subterm of $t\}$ contains more than one node and consists of a single strongly connected component. Let $\sigma = SMR(\sigma')$. Then*

$$\langle\!|\sigma \circ \sigma'(t)|\!\rangle(\phi) = \langle\!|\sigma(t)|\!\rangle(\phi)$$

for every type term t and every closed type substitution ϕ appropriate for $\sigma \circ \sigma'(t)$ and $\sigma(t)$.

Proof: See App. B.2, Page 136. $\square$

Lemma 4.3.11 (termination of *GIS*) *The algorithm GIS terminates for every input substitution with a finite domain dom(σ).*

Proof: See App. B.2, Page 138. $\Box$

Lemma 4.3.12 (correctness of *GIS*) *Let σ' be a substitution fulfilling with the following properties:*

1. *σ' does not contain a variable binding $A \leftarrow B$ with $B \in dom(\sigma')$.*
2. *If σ' contains a variable binding $A \leftarrow C[B]$ with a context C and a variable $B \in dom(\sigma')$ then there exists a variable $B' \notin dom(\sigma')$ such that B is bound to B' or a union containing B' in σ'.*

Let v be a value fulfilling

$$\forall k \in \mathbb{N} \,.\, v \in {\sigma'}^k(t_1) \boxdot {\sigma'}^k(t_2)$$

and let $\sigma = GIS(\sigma')$. Then

$$v \in \sigma(t_1) \boxdot \sigma(t_2) \,.$$

Proof: See App. B.2, Page 138. $\Box$

The following lemma essentially states the termination of *CE*. The termination proof in this lemma relies on the termination of *ST*. The termination of *CE* without this restriction is proven afterwards.

Lemma 4.3.13 *If the algorithm S-CE terminates for every pair of terms in set normalized form (and empty free constraint set and empty recursion information) then CE terminates for every pair of input types in set normalized form.*

Proof: See App. B.2, Page 141. $\Box$

We can now prove the unrestricted termination of *CE*. With Lemma 5.3.13 given, the proof consists of proving the termination of the loop

$$CE \rightarrow S\text{-}CE \rightarrow ST \rightarrow CE \tag{4.5}$$

between the mutually dependent algorithms *CE*, *S-CE* and *ST*. Figure 5.4 shows the termination dependencies. An arrow from an Algorithm 1 to an Algorithm 2 expresses that the

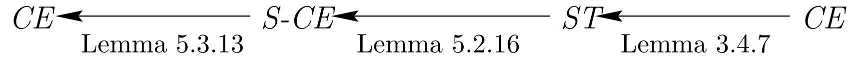

Figure 4.4: termination dependencies between the algorithms

termination of Alg. 2 depends on termination of Alg. 1. The corresponding lemma is given under every arrow.

Informally, the termination proof for *CE* (Theorem 5.3.14) uses the fact that every execution of the loop given in (5.5) reduces the number of complement type constructors occurring in the two types that are checked. None of the algorithms processed during this loop can introduce new complement type constructors. Thus, there is a finite bound for the number of executions of this loop. This is formally stated in the following theorem:

Theorem 4.3.14 (termination of *CE*) *The algorithm CE terminates for every pair of input types in set normalized form.*

Proof: See App. B.2, Page 141. □

Given Theorem 5.3.14 we can now easily conclude the termination of *ST* for every input. This extends the termination proof given in Lemma 3.4.7.

Corollary 4.3.15 *The algorithm ST terminates for every pair of ground input types in set normalized form.*

Proof: See App. B.2, Page 142. □

Analogously we can extend Lemma 5.2.16 as follows:

Corollary 4.3.16 *The algorithm S-CE terminates for every pair of input types in set normalized form.*

Proof: See App. B.2, Page 142. □

Theorem 4.3.17 (correctness of *CE*) *Let* $t_1, t_2 \in \mathcal{T}$*, both in set normalized form. Let there exist a value* $v \neq \bot$ *such that*

$$v \in t_1 \boxdot t_2 \,.$$

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

$$v \in \sigma(t_1) \boxdot \sigma(t_2) \,.$$

Proof: See App. B.2, Page 142. □

Chapter 5

Checking Types for Common Elements

One main question occurring in complete type checking is the following: Given two types t_1 and t_2. Is there a substitution σ such that $\langle\!\langle \sigma(t_1) \rangle\!\rangle \cap \langle\!\langle \sigma(t_2) \rangle\!\rangle \neq \{\bot\}$? More precisely this property should be independent from the instantiation of quantified variables i.e. members of V_q. For every such substitution the property of common elements must not be violated by changing the assignment to a quantified variable.

Example 5.0.18 *Consider the types* $t_1 = (\textbf{\textit{int}} \;.\; X_\forall)$ *and* $t_2 = (\textbf{\textit{posint}} \;.\; Y)$ *where* $X_\forall \in V_q$ *and* $Y \in V_f$*. Then* t_1 *and* t_2 *have common elements because we can e.g. choose substitutions* $\rho_f = \{Y \leftarrow X_\forall\}$ *and* $\rho_q = \{X_\forall \leftarrow \textbf{\textit{int}}\}$ *and a value* $v = 5$ *and get*

$$v \in \langle\!\langle \rho_q \circ \rho_f(t_1) \rangle\!\rangle \cap \langle\!\langle \rho_q \circ \rho_f(t_2) \rangle\!\rangle \,.$$

Furthermore, when choosing an arbitrary ρ'_q *with* $\rho'_q(X_\forall) \neq \bot$ *there always exists a* v' *with*

$$v' \in \langle\!\langle \rho'_q \circ \rho_f(t_1) \rangle\!\rangle \cap \langle\!\langle \rho'_q \circ \rho_f(t_2) \rangle\!\rangle \,.$$

On the other hand t_1 *and* $t_3 = (\textbf{\textit{posint}} \;.\; \textbf{\textit{negint}})$ *do not have common elements because we can instantiate* $X_\forall \in V_q$ *in a way such that no common elements exist.*

The notion of common elements is formalized as follows:

Definition 5.0.19 *Let* t_1 *and* t_2 *be two types. These types have common elements if*

$$\begin{aligned} \exists \rho_f \forall \rho_q (dom(\rho_f) \subseteq V_f \wedge dom(\rho_q) \subseteq V_q \wedge \forall X_\forall \in dom(\rho_q) \,.\, \rho_q(X_\forall) \neq \bot \wedge \\ \wedge\, \rho_q \circ \rho_f \textit{ is appropriate for } t_1, t_2) \Rightarrow \\ \Rightarrow \exists v \in \langle\!\langle \rho_q \circ \rho_f(t_1) \rangle\!\rangle \cap \langle\!\langle \rho_q \circ \rho_f(t_2) \rangle\!\rangle \wedge v \neq \bot \end{aligned} \quad (5.1)$$

We call every v with the properties given in (5.1) a common element *of t_1 and t_2 and denote the set of all common elements of t_1 and t_2 by $t_1 \boxdot t_2$.*

Note that quantified variables are treated in a special way in Def. 5.0.19: They must not be instantiated in a special way in order to get common elements. Common elements of types containing quantified variables are rather considered just for those types with common elements for *every* instantiation of the quantified variables.

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* may return a substitution even if t_1 and t_2 cannot have common elements especially when one of the types already does not have any elements.

The description of *CE* is done in the following subsections: Section 5.1 contains the preliminaries used to define *CE*. In Sec. 5.2 an algorithm *S-CE* is introduced that calculates constraints on the variable instantiations. The algorithm *CE* presented in Sec. 5.3 transforms these constraint sets to idempotent substitutions σ.

5.1 Preliminaries

In this section we will give some definitions needed to define *CE*. First, we will define some structures forming the result or intermediate values of *CE*. In a second step we will formulate the needed properties of a function *CEbase* realizing *CE* on base types and value assignments.

5.1.1 Structures used by *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. The substitutions considered by *CE* are restricted in the following way: If σ is a substitution returned by *CE* then σ just assigns types to *free* type variables. This motivates the following definition:

Definition 5.1.1 (free type substitutions) *A type substitution σ is called a* free type substitution *if σ is just defined for free type variables, i.e. if*

$$dom(\sigma) \subseteq V_f\ .$$

The set of all free type substitutions is denoted by FTS, the set of all closed free type substitutions by FTS_C.

CE will return sets of free type substitutions called s-collections:

Definition 5.1.2 (s-collection) *An* s-collection *is a finite set of free type substitutions.*

During the processing of *CE* constraints on the possible instantiations of type variables are collected in structures called *constraint sets* that are defined as follows:

Definition 5.1.3 (free constraint sets) *A free variable constraint is a pair (X, t) (often written as $X \leftarrow t$ where $X \in V_f$ and $t \in \mathcal{T}$. A* free constraint set *is a set of free variable constraints $\{(X_1, t_1), \ldots, (X_n, t_n)\}$ with pairwise disjoint variables X_i. For such a free constraint set $dom(\sigma) = \{X_1, \ldots, X_n\}$ and $\sigma(X_i) = t_i$.*

The intermediate values occurring in *CE* are not free constraint sets directly, but sets of free constraint sets. These sets are called *c-collections* and defined as follows:

Definition 5.1.4 (c-collection) *A* c-collection *is a finite set of free constraint sets.*

Constraint sets and c-collections make restrictions to the values assigned to certain variables. The free type substitutions or s-collections fulfilling all these constraints are called compatible with a free constraint set of c-collection, respectively:

Definition 5.1.5 *Let σ be a free constraint set. A free type substitution σ' is compatible with σ if for every X with $X \leftarrow t \in \sigma$ the following holds:*

- $X \in dom(\sigma')$
- $\langle\!|t|\!\rangle \subseteq \langle\!|\sigma'(X)|\!\rangle$

An s-collection Σ' is compatible with a c-collection Σ if there is a one to one assignment of $\sigma' \in \Sigma'$ to $\sigma \in \Sigma$ with σ' compatible to σ.

For every free constraint set σ' there is the natural free type substitution σ of σ' defined as follows:

Definition 5.1.6 (natural free type substitution) *Let σ' be a free constraint set. The natural free type substitution of σ' (denoted by $subst(\sigma')$) is*

$$subst(\sigma') = \{X \leftarrow \sigma'(X) \mid X \in dom(\sigma')\}\,.$$

For a c-collection Σ' the natural s-collection of Σ' is

$$subst(\Sigma') = \{subst(\sigma') \mid \sigma' \in \Sigma'\}\,.$$

The union of c-collections is based on the union of free constraint sets defined as follows:

Definition 5.1.7 (union of free constraint sets) *Let σ_1 and σ_2 be two free constraint sets. Their union is defined as*

$$\begin{aligned}\sigma_1 \otimes \sigma_2 =& \{A \leftarrow (\cup\ t_1\ t_2) \mid A \leftarrow t_1 \in \sigma_1 \wedge A \leftarrow t_2 \in \sigma_2\} \\ &\cup \{A \leftarrow t_1 \mid A \leftarrow t_1 \in \sigma_1 \wedge A \notin dom(\sigma_2)\} \\ &\cup \{A \leftarrow t_2 \mid A \leftarrow t_2 \in \sigma_2 \wedge A \notin dom(\sigma_1)\}\,.\end{aligned}$$

We can now define the union of c-collections:

Definition 5.1.8 (union of c-collections) *Let Σ_1 and Σ_2 be two c-collections the union of Σ_1 and Σ_2 is defined as*

$$\Sigma_1 \otimes \Sigma_2 = \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in \Sigma_1 \wedge \sigma_2 \in \Sigma_2\}\,.$$

We use the same operator $\otimes$ for both free constraint sets and c-collections. When identifying free constraint sets σ with the one element c-collections $\{\sigma\}$ the correspondence becomes obvious.

5.1.2 *CE* on Base Types and Value Assignments

The algorithm *CE* works by decomposing its argument types. It therefore needs an algorithm *CEbase* to decide the common element question for base types and value assumptions.

Assumption 5.1.9 (common elements of base types) *The function CEbase* $: \mathcal{B} \times \mathcal{B} \rightarrow \{\texttt{true}, \texttt{false}\}$ *approximates the question whether two base types or value assignments have common elements as follows:*

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

There is an algorithm also called CEbase that terminates for every input and returns the value of the function CEbase.

Especially *CEbase* fulfills

$$CEbase(b_1, b_2) = \texttt{true} \text{ if } (b_1 = \top \wedge b_2 \neq \bot) \vee (b_2 = \top \wedge b_1 \neq \bot)$$

but

$$CEbase(b_1, b_2) = \texttt{false} \text{ if } b_1 = \bot \vee b_2 = \bot\,.$$

App. C contains a definition of *CEbase* for the example base types presented there.

5.2 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 free 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

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

is initiated by *CE*. Its arguments are two types t_1 and t_2 to be checked for common elements, the empty free 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 free 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 where 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 a fixed order in which they are checked.

5.2.1 Globally Used Auxiliary Functions

To describe the actions of *S-CE* in the individual cases we start with defining some auxiliary functions that are needed in several cases:

In several cases *S-CE* decomposes a constructor and performs recursive calls sequentially to the arguments. This is formalized by the function *SCE-list-reduce*.

SCE-list-reduce 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 free

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. A recursion history r is just passed through to the calls to *S-CE*.

Definition 5.2.1 (*SCE-list-reduce*) *The function SCE-list-reduce expects the following arguments:*

- *A list L of tuples (t_1, t_2) where t_1 and t_2 are types.*
- *A c-collection Σ.*
- *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 5.2.2 (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 free 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 the call

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

Again SCE-list-reduce initiates a number of calls to S-CE with the first type pair as arguments. Since Σ' just contains one free constraint set in the example just one call to S-CE is necessary:

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

The resulting c-collection Σ'' is used in the recursive call

$$\textit{SCE-list-reduce}((), \Sigma'', r)$$

to SCE-list-reduce. Since the list of type pairs is empty rule (SCE-list-reduce1) applies and Σ'' is returned.

For changing a constraint in a given constraint set the functions *extend-constraint* and *replace-constraint* can be used:

Definition 5.2.3 (*extend-constraint*) *Let σ be a free constraint set, $X \in V_f$ a free variable and t a term. $\sigma' = extend\text{-}constraint(X, t, \sigma)$ is the free constraint set differing from σ just in the constraint of X as follows:*

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

Definition 5.2.4 (*replace-constraint*) *Let σ be a free constraint set, $X \in V_f$ a free variable and t a term. $\sigma' = replace\text{-}constraint(X, t, \sigma)$ is defined by*

$$\sigma'(Y) = \begin{cases} t & \text{if } Y = X \\ \sigma(Y) & \text{else} \end{cases}$$

Note that *extend-constraint* and *replace-constraint* behave equivalently if $X \notin dom(\sigma)$.

In some cases a given free constraint set has to be updated by constraining all free variables occurring in a term in a certain manner. This is done by the functions *constrain-all-free* and *free-to-top*:

Definition 5.2.5 (*constrain-all-free*) *Let t be a type term and σ a free constraint set. The call constrain-all-free(t, σ) constrains every free variable occurring in t to a fresh quantified variable. The new constraints are added to σ without destroying previous constraints:*

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

Definition 5.2.6 (*free-to-top*) *Let t be a type term and σ a free constraint set. The call free-to-top(t, σ) constrains every free variable occurring in t to $\top$ in σ:*

forall *free variables Y in t* ***do***
  $\sigma := replace\text{-}constraint(Y, \top, \sigma)$
return *σ.*

The following subsections describe the actions of *S-CE* for certain cases of t_1 and t_2:

5.2.2 $\top$ 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 if formalized in the rules (*Top1*) and (*Top2*):

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

$$\frac{\textit{S-CE}(t_1, \top, \sigma, r)}{\{\sigma\}} \quad t_1 \neq \bot \qquad (\textit{Top2})$$

5.2.3 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.

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 that are not already detected in processing the former call with the same arguments. Hence, the actual call can return without introducing any further constraints. 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:

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

$$\frac{\textit{S-CE}(t_1, t_2 = \mu X.t_2', \sigma, r)}{\textit{combine-cs}(\textit{S-CE}(t_1, \textit{unfold}(t_2), \sigma, ((t_1, t_2)\ .\ r)))} \qquad (\textit{Rec2})$$

The function *combine-cs* used in (*Rec1*) and (*Rec2*) expects and returns a c-collection and is defined as follows:

$$\textit{combine-cs}(\emptyset) = \emptyset$$
$$\textit{combine-cs}(\Sigma) = \{\bigotimes_{\sigma \in \Sigma'} \sigma \mid \emptyset \neq \Sigma' \subseteq \Sigma\} \text{ for } \Sigma \neq \emptyset$$

Example 5.2.7 *Consider the c-collection* $\Sigma = \{\sigma_1, \sigma_2\}$ *with* $\sigma_1 = \{X \leftarrow \textbf{int}, Y \leftarrow \textbf{int}\}$ *and* $\sigma_2 = \{X \leftarrow \textbf{bool}, Z \leftarrow \textbf{bool}\}$.

The non-empty subsets of Σ *are*

$$\Sigma_1 = \{\sigma_1\}$$
$$\Sigma_2 = \{\sigma_2\}$$
$$\Sigma_3 = \Sigma\,.$$

Each of these subsets Σ_i *yields a free constraint set that is generated by combining all elements of* Σ_i *by* $\otimes$ *with the following results:*

$$\bigotimes_{\sigma \in \Sigma_1} = \sigma_1$$
$$\bigotimes_{\sigma \in \Sigma_2} = \sigma_2$$
$$\bigotimes_{\sigma \in \Sigma_3} = \sigma_3 \textit{ with } \sigma_3 = \{X \leftarrow (\cup\ \textbf{int}\ \textbf{bool}), Y \leftarrow \textbf{int}, Z \leftarrow \textbf{bool}\}\,.$$

Altogether combine-cs$(\Sigma) = \{\sigma_1, \sigma_2, \sigma_3\}$.

The reason for applying *combine-cs* on the result of the recursive subcall becomes obvious in the following example:[1]

Example 5.2.8 (recursive types) *Assume that (Rec1) and (Rec2) just return the results of their subcalls without applying combine-cs. Consider the two types*

$$t_1 = \mu X.((\cup\ f(\textbf{bool})\ f(\textbf{int}))\ .\ (\cup\ \textbf{nil} X))$$
$$t_2 = \mu Y.(f(A)\ .\ (\cup\ \textbf{nil}\ Y))$$

[1]Some of the rules used for the example are defined afterwards. We just sketch the results here.

where f is a unary free type constructor (or a type constructor with arity n and $n-1$ argument positions fixed).

Processing the call S-CE$(t_1, t_2, \emptyset, ())$ starts with applying the rule (Rec1) and afterwards the rule (Rec2). The result is a recursive call

$$\textit{S-CE}(t'_1, t'_2, \emptyset, r) \text{ with } t'_1 = \textit{unfold}(t_1),\ t'_2 = \textit{unfold}(t_2) \text{ and } r = ((t'_1, t_2)\ (t_1, t_2))$$

This call is processed by rule (Constr) and is divided into two subcalls. The first one is processed by rule (U1) with each of its subcalls decomposed by (Constr) and processed by (Var2). It yields

$$\textit{S-CE}((\cup\ f(\texttt{bool})\ f(\texttt{int})), f(A), \emptyset, r) = \{\{A \leftarrow \texttt{bool}\}, \{A \leftarrow \texttt{int}\}\} =: \Sigma_1\,.$$

The second subcall (more precisely we have two subcalls for every element $\sigma \in \Sigma_1$) is decomposed by (U1) and (U2) and some of the subcalls are processed by (RecT), some by (Base) and some fail. These subcalls do not introduce any new constraints and Σ_1 is returned.

Unfortunately there is a value $v = (\#t\ .\ (5\ .\ \texttt{nil}))$ with $v \in t_1 \boxdot t_2$ (e.g. with $\rho_f = \{A \leftarrow (\cup\ \texttt{bool int})\}$ and $\rho_q = \emptyset$), but $v \notin \sigma(t_1) \boxdot \sigma(t_2)$ for all $\sigma \in \Sigma_1$.

When we assume the rule (*RecT*) not to exist the processing of the call in Example 5.2.8 does not terminate, but the constraints generated during the infinite loop give insight in the problem of the example. After the second unfolding of t_1 and t_2 the first call performed by *SCE-list-reduce* generates the free constraint sets $\sigma_{1,1} = \{A \leftarrow (\cup\ \texttt{bool bool})\}$ and $\sigma_{1,2} = \{A \leftarrow (\cup\ \texttt{bool int})\}$ from $\{A \leftarrow \texttt{bool}\}$ and $\sigma_{2,1} = \{A \leftarrow (\cup\ \texttt{int bool})\}$ and $\sigma_{2,2} = \{A \leftarrow (\cup\ \texttt{int int})\}$ from $\{A \leftarrow \texttt{int}\}$. After the third unfolding eight constraint sets containing three element unions are generated and so on.

More generally, when after unfolding types by (*Rec1*) and (*Rec2*) a union type constructor causes several free constraint sets to be generated, an iterated unfolding can generate constraints containing unions as above. Since the generation of these unions is blocked by (*RecT*) we have to calculate these unions after processing the first unfolding. This is done by *combine-cs*.

5.2.4 Free Type Variables

When at least one of the types is a free type variable several cases have to be distinguished: There is either exactly one or two free type variables that can already be member of the domain of σ or not.

When both types are free type variables then a new variable is introduced in order to name the common elements possibly occurring at this point. This variable must not be restricted by *S-CE* any further and is therefore quantified:

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

Example 5.2.9 *Let* $t_1 = \#(X\ \boldsymbol{int}\ X)$ *and* $t_2 = \#(\boldsymbol{bool}\ Y\ Y)$*. The call* $\textit{S-CE}(t_1, t_2, \emptyset, ())$ *is processed by rule (Constr) given below. The processing of this rule initiates the call* $\textit{SCE-list-reduce}(((X, \boldsymbol{bool}), (\boldsymbol{int}, Y), (X, Y)), \emptyset, ())$*. The first two subcalls to S-CE are processed by the rules (Var1) and (Var2) given below. They yield a intermediate c-collection consisting of exactly one free constraint set* $\sigma' = \{X \leftarrow \boldsymbol{bool}, Y \leftarrow \boldsymbol{int}\}$*.*

The result of the initial call to S-CE is the result of the third subcall $\textit{S-CE}(X, Y, \sigma', ())$ *which is processed by rule (BothVar). Its result is* $\{\{X \leftarrow (\cup\ \boldsymbol{bool}\ Z), Y \leftarrow (\cup\ \boldsymbol{int}\ Z)\}\}$*. with a new free variable* Z*.*

Note that the introduction of Z *is necessary in order to preserve e.g. the common element* $v = \#(\#t\ 42\ 40.5)$ *with a non-integer number in the third vector position.*

When just t_1 is a free type variable then the constraint of t_1 in σ is updated to contain t_2:

$$\frac{\textit{S-CE}(t_1, t_2, \sigma, r)}{\{\textit{constrain-all-free}(t_2, \textit{extend-constraint}(X, t_2, \sigma))\}} \quad t_1 \in V_f, t_2 \notin V_f \quad (\textit{Var1})$$

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

Example 5.2.10 *Let* $t_1 = (X\ .\ \boldsymbol{int})$ *and* $t_2 = ((Y\ .\ \boldsymbol{int})\ .\ Y)$*. The call* $\textit{S-CE}(t_1, t_2, \emptyset, ())$ *yields the call* $\textit{SCE-list-reduce}(((X, (Y\ .\ \boldsymbol{int})), (\boldsymbol{int}, Y)), \emptyset, r)$ *by rule (Constr) given below.*

The first subcall to S-CE is $\textit{S-CE}(X, (Y\ .\ \boldsymbol{int}), \emptyset, ())$*. It is processed by rule (Var1) and yields the result* $\Sigma_1 = \{\sigma_1\}$ *with* $\sigma_1 = \{X \leftarrow (Y\ .\ \boldsymbol{int}), Y \leftarrow Y'\}$*. The constraint of* X *is introduced by extend-constraint while the constraint of* Y *results from applying constrain-all-free to* $(Y\ .\ \boldsymbol{int})$*.*

In second subcall initiated by SCE-list-reduce is $\textit{S-CE}(\boldsymbol{int}, Y, \sigma_1, ())$*. Its result is* $\Sigma_2 = \{\sigma_2\}$ *with* $\sigma_2 = \{X \leftarrow (Y\ .\ \boldsymbol{int}), Y \leftarrow (\cup\ Y'\ \boldsymbol{int})\}$ *by rule (Var2).*

Note that without applying constrain-all-free the result in this example is $\Sigma_2' = \{\sigma_2'\}$ *with* $\sigma_2' = \{X \leftarrow (Y\ .\ \boldsymbol{int}), Y \leftarrow \boldsymbol{int}\}$*. While the value* $v = ((\#t\ .\ 168)\ .\ 42)$ *is a common element of* t_1 *and* t_2 *under* σ_2 *it is not a common element when applying* σ_2' *instead.*

5.2.5 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:

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

$$\frac{\textit{S-CE}(t_1, (\cup\ t_{2,1}\ \ldots\ t_{2,k}), \sigma, r)}{\bigcup_{i=1}^{k} \textit{S-CE}(t_1, t_{2,i}, \sigma, r)} \qquad (U2)$$

5.2.6 Intersection Types

When one of the types is an intersection type the other argument type has to be checked against all intersection elements cumulating the detected restrictions. This is done by the rules (*I1*) and (*I2*) for an intersection type in the first or second argument, respectively using the function *SCE-list-reduce* given in Def. 5.2.1 on page 79:

$$\frac{\textit{S-CE}((\cap\ t_{1,1}\ \ldots\ t_{1,k}), t_2, \sigma, r)}{\textit{SCE-list-reduce}(((t_{1,1}, t_2), \ldots, (t_{1,k}, t_2)), \{\sigma\}, r)} \qquad (I1)$$

$$\frac{\textit{S-CE}(t_1, (\cap\ t_{2,1}\ \ldots\ t_{2,k}), \sigma, r)}{\textit{SCE-list-reduce}(((t_1, t_{2,1}), \ldots, (t_1, t_{2,k})), \{\sigma\}, r))} \qquad (I2)$$

Unfortunately this definition of (*I1*) and (*I2*) yields the following unintuitive results:

Example 5.2.11 *Let* $t_1 = (A\ .\ \texttt{nil})$ *and* $t_2 = (\cap\ (\texttt{num}\ .\ nil)\ (\texttt{bool}\ .\ nil))$*. For this pair of types S-CE returns the c-collection* $\{A \leftarrow (\cup\ \texttt{num}\ \texttt{bool})$ *even though* t_2 *does not contain any values except* $\bot$*.*

The behaviour above does not violate any of the statements proven for *S-CE* below. For refining *S-CE* one could think of a different mode in variable instantiation in order to distinguish constraints introduced by different intersection elements. A different way of refinement is an equivalent transformation of intersection types, e.g. transforming t_2 to $((\cap\ \texttt{num}\ \texttt{bool})\ .\ \texttt{nil})$ with an additional check for emptiness of the resulting intersection in Ex. 5.2.11 above.

5.2.7 Complement Types

If one of the the types is a complement type then there are common elements if the other type is not a subtype of the complement's argument.

When the type t (either t_1 or t_2) that is checked against the complement type is not a semi-closed term then we enforce this property be replacing every free variable by $\top$:

$$\frac{\textit{S-CE}(\mathcal{C}t'_1, t_2, \sigma, r)}{\{\sigma'\}} \quad \sigma'(t_2) \not\sqsubseteq t'_1, \sigma' := \textit{free-to-top}(t_2, \sigma) \qquad (\textit{Comp1})$$

$$\frac{\textit{S-CE}(t_1, \mathcal{C}t'_2, \sigma, r)}{\{\sigma'\}} \quad \sigma'(t_1) \not\sqsubseteq t'_2, \sigma' := \textit{free-to-top}(t_1, \sigma) \qquad (\textit{Comp2})$$

Note that $\sigma'(t_2) \not\sqsubseteq t'_1$ abbreviates the test $\neg ST(\sigma'(t_2), t'_1)$. σ' is generated from σ in order to get a semi-closed term $\sigma'(t_2)$ in (*Comp1*) or $\sigma'(t_1)$ in (*Comp2*).

The calls to ST performed when checking the $\not\sqsubseteq$-conditions (cf. Def. 3.4.2) can cause a recursive call to *S-CE*, again, because of case (9f) of ST. Because of this the recursion info r must be passed over to ST and back to *S-CE* via *CE*. We omit this parameter here in order to simplify the representation of the algorithms. r is not changed at all in ST or *CE*, but is just passed through to *S-CE* again.

Example 5.2.12 *Let $t_1 = \mathcal{C}\,\texttt{int}$ and $t_2 = (X\ .\ Y)$. The call S-CE$(t_1, t_2, \emptyset, ())$ is processed by rule (Comp1). This rule generates the free constraint set $\sigma' = \textit{free-to-top}(t_2, \emptyset) = \{X \leftarrow \top, Y \leftarrow \top\}$ and performs the test $\neg ST((\top\ .\ \top), \texttt{int})$. Since this test succeeds the c-collection $\{\sigma'\}$ is returned.*

5.2.8 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 using *SCE-list-reduce*:

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

5.2.9 Frame Types

For two frame types F_1 and F_2 the existence of common elements depends on two facts:

- The sets of bound symbols must be equal.
- For every symbol the assigned types must have common elements.

After determining an order of the symbols the types assigned to the same symbol in both frames must be checked against each other. The result restrictions are collected:

$$\frac{\textit{S-CE}(F_1, F_2, \sigma, r), [s_1 \mapsto t_{1,i}, \ldots, s_k \mapsto t_{k,i}] \in \textit{fs}(F_i) \text{ for } i \in \{1, 2\}}{\textit{SCE-list-reduce}(((t_{1,1}, t_{2,1}), \ldots, (t_{1,k}, t_{2,k})), \{\sigma\}, r))} \qquad (\textit{Frame})$$

5.2.10 Environment Types

When both types are environment types then they are decomposed and the pairs of frames at the same position are checked collecting the results. This is done similar to the processing of types constructed by the free type constructors:

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

5.2.11 Function Types and Quantified Variables

If both types t_1 and t_2 are function types then they have common elements when they are equal or one of them is the type *TFunc* of all functions. A quantified variable just has common elements with itself. These cases are formalized in the following function:

Definition 5.2.13 *The function* $\textit{fq} : \mathcal{T} \times \mathcal{T} \rightarrow \{\texttt{true}, \texttt{false}\}$ *expects two types* t_1 *and* t_2 *and returns true in the following cases:*

- t_1 *and* t_2 *are the same function type.*
- t_1 *and* t_2 *are both function types and one of them is the type TFunc of* all *functions.*
- $t_1 = t_2$ *and* $t_1, t_2 \in V_q$.

In all cases not mentioned above fq returns $\texttt{false}$.

In all the cases with $\texttt{fq}(t_1, t_2) = \texttt{true}$ no further restrictions must be included into the result of *S-CE*. This yields the following rule:

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

5.2.12 Base Types and Value Assignments

When t_1 and t_2 are both base types or value assignments then the question whether they have common elements is answered by *CEbase* fulfilling Assumption 5.1.9.

The behaviour of *S-CE* is formalized in the following rule:

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

5.2.13 Integrating the Cases

In this subsection we will present the algorithm *S-CE* by integrating the previously introduced rules in a fixed order:

Definition 5.2.14 (algorithm *S-CE*) *The algorithm S-CE checks the previously defined rules in the following order and returns the result given by the first applicable rule:*

$$(\textit{Top1}), (\textit{Top2}), (\textit{RecT}), (\textit{Rec1}), (\textit{Rec2}), (\textit{BothVar}), (\textit{Var1}), (\textit{Var2}),$$
$$(\textit{U1}), (\textit{U2}), (\textit{I1}), (\textit{I2}), (\textit{Comp1}), (\textit{Comp2}), (\textit{Constr}), (\textit{Frame}), (\textit{Env}), (\textit{FunQ}), (\textit{Base})\,.$$

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

5.2.14 An *S-CE* Example in Detail

The following example shows in detail how calls to *S-CE* are processed and how the use of unions by *extend-constraint* enables *S-CE* to process hetreogenous lists correctly.

Example 5.2.15 (*S-CE*) *Consider the call S-CE*(t_1, t_2, σ, r) *with:*

$$\begin{aligned} t_1 &= \mu X.(\cup\ \boldsymbol{nil}\ (A\ .\ X)) \\ t_2 &= (\boldsymbol{nat}\ .\ (\boldsymbol{string}\ .\ \boldsymbol{nil})) \\ \sigma &= \emptyset \\ r &= () \end{aligned}$$

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

$$\textit{S-CE}((\cup\ \boldsymbol{nil}\ (A\ .\ t_1)), t_2, \emptyset, r' := ((t_1, t_2)))\,.$$

2. *By rule (U1) the S-CE-call is recursively splitted into two subcalls:*

   (a) S-CE($\textbf{nil}, t_2, \emptyset, r'$) $= \emptyset$

   (b) S-CE($(A\ .\ t_1), t_2, \emptyset, r'$)

3. *S-CE($(A\ .\ t_1), t_2, \emptyset, r'$) is processed by rule (Constr). SCE-list-reduce performs the following subcalls (the result of subcall number i is denoted by Σ_i):*

   (a) $\Sigma_1 =$ S-CE($A, \textbf{nat}, \emptyset, r'$) $= \{\{A \leftarrow \textbf{nat}\}\}$ *by rule (Var1).*

   (b) $\Sigma_2 =$ S-CE($t_1, (\textbf{string}\ .\ \textbf{nil}), \sigma' := \{A \leftarrow \textbf{nat}\}, r'$)

4. *S-CE($t_1, (\textbf{string}\ .\ \textbf{nil}), \sigma', r'$) is processed by rule (Rec1) resulting in the call*

$$\text{S-CE}((\cup\ \textbf{nil}\ (A\ .\ t_1)), (\textbf{string}\ .\ \textbf{nil}), \sigma', r'' = ((t_1, (\textbf{string}\ .\ \textbf{nil})), (t_1, t_2)))\,.$$

5. *By rule (U1) we have two subcalls for the contained call to S-CE:*

   (a) S-CE($\textbf{nil}, (\textbf{string}\ .\ \textbf{nil}), \sigma', r''$) $= \emptyset$

   (b) S-CE($(A\ .\ t_1), (\textbf{string}\ .\ \textbf{nil}), \sigma', r''$)

6. *S-CE($(A\ .\ t_1), (\textbf{string}\ .\ \textbf{nil}), \sigma', r''$) is processed by rule (Constr):*

   (a) $\Sigma'_1 =$ S-CE($A, \textbf{string}, \sigma', r''$) $= \{\sigma'' := \{A \leftarrow (\cup\ \textbf{nat}\ \textbf{string})\}\}$

   (b) $\Sigma'_2 =$ S-CE($t_1, \textbf{nil}, \sigma'', r''$)

7. *The second call is processed by rule (Rec1) producing the call*

$$\text{S-CE}((\cup\ \textbf{nil}\ (A\ .\ t_1)), \textbf{nil}, \sigma'', r''' := ((t_1, \textbf{nil}), (t_1, (\textbf{string}\ .\ \textbf{nil})), (t_1, t_2)))\,.$$

8. *By rule (U1) the call to S-CE causes the following subcalls:*

   (a) S-CE($\textbf{nil}, \textbf{nil}, \sigma'', r'''$) $= \{\sigma''\}$ *by rule (Base).*

   (b) S-CE($(A\ .\ t_1), \textbf{nil}, \sigma'', r'''$) $= \emptyset$

   The call containing the union $(\cup\ \textbf{nil}\ (A\ .\ t_1))$ in step (7) yields the result $\{\sigma''\}$. This is also the result for Σ'_2.

9. *The result of step (5) is given by the union of the two subcalls. Since the first subcall yielded the result $\emptyset$ this is equal to the result of the second subcall which is still $\{\sigma''\}$. Thus, $\{\sigma''\}$ is the result of step (4), of Σ_2 and therefore of step (3).*

10. *The result of step (2) is given by the union of $\emptyset$ from the first subcall and $\{\sigma''\}$ from the second one. $\{\sigma''\}$ is also the union result and the result of step (1).*

5.2.15 Properties of *S-CE*

Lemma 5.2.16 (termination of *S-CE*) *If the algorithm ST terminates for every pair of closed terms in set normalized form then the algorithm S-CE terminates for every input with arguments 1 and 2 in set normalized form.*

Proof: See App. B.1, Page 122. □

The remaining condition on *ST* for termination of *S-CE* is removed later. The unrestricted termination result is given in Cor. 5.3.16.

In the following we will prove certain properties of the result of *S-CE* for every intermediate recursive call occurring during the processing of an initial call to *S-CE*. These intermediate calls are sometimes incomplete because there might be computations that were started before initiating the actual intermediate call but were not finished. Hence, we need the notion of *implicit constraint sets*. Implicit constraint sets are defined in the context of executions of *SCE-list-reduce*.

Informally an implicit constraint set is the union of the results of all *open S-CE calculations* i.e. calculations that are given in the argument list of a call to *SCE-list-reduce* but where not yet processed. Open *S-CE* calculations are defined as follows:

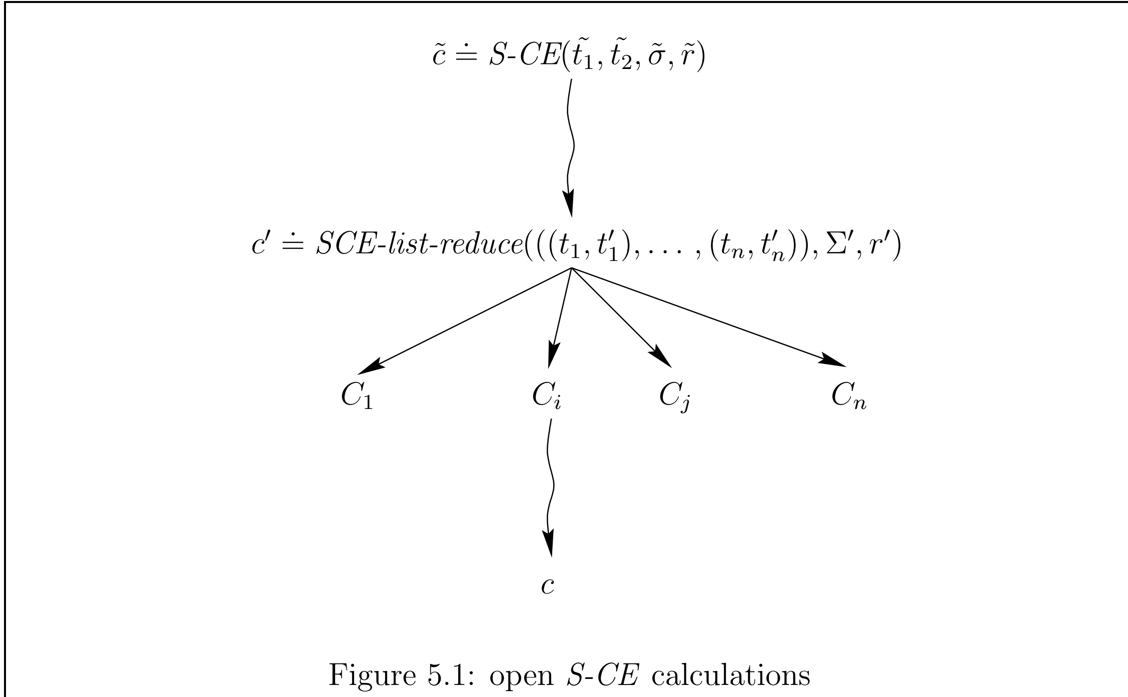

Figure 5.1: open *S-CE* calculations

Definition 5.2.17 (open *S-CE* calculation) *Let*

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

be a call to S-CE and

$$c' \doteq SCE\text{-}list\text{-}reduce((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r')$$

a call to SCE-list-reduce occurring as (maybe indirect) subcall of $\tilde{c}$. *Let the first argument of SCE-list-reduce be a list with the n elements* (t_i, t'_i) *for* $i \in \{1, \ldots, n\}$. *For every* $j \in \{1, \ldots, n\}$ *let*

$$c_{j,l} \doteq S\text{-}CE(t_j, t'_j, \sigma_{j,l}, r) \text{ with } l \in \{1, \ldots, m_j\}$$

be the calls to S-CE initiated by SCE-list-reduce when processing the jth *list element and let*

$$C_j = \{c_{j,1}, \ldots, c_{j,m_j}\} .$$

Let c be a call to S-CE that is either $c_{i,l} \in C_i$ *or a subcall of* $c_{i,l}$ *for some i and some l (cf. Fig. 5.1). Then every* C_j *with* $i \leq j \leq n$ *is an* open *S-CE* calculation *of c with respect to* $\tilde{c}$.

The fact that C_i is an open *S-CE* calculation of c might be confusing. But this definition is necessary in order to provide the constraints given by C_i for the $c_{i,l}$ being processed after c in the following definition of implicit constraint sets.

When applying *SCE-list-reduce* the constraints for the individual list entries of the first argument could essentially be calculated independently from each other (with the empty free constraint set as third argument) and combined afterwards. For the presentation of *SCE-list-reduce* we calculated them step by step and combined an intermediate result directly on the fly. The idea behind implicit constraint sets given in Def. 5.2.18 is to perform all open *S-CE* calculations independently from an already known intermediate result and to combine the results.

Note that in the following definition there can be several calls c' for given c and $\tilde{c}$. The elements set of *all* corresponding open *S-CE* calculations are enumerated from 1 to n.

Definition 5.2.18 (implicit constraint sets) *Let* $c := S\text{-}CE(t, t', \sigma, r)$ *be a call to S-CE steaming from an initial call* $\tilde{c} := S\text{-}CE(\tilde{t}, \tilde{t'}, \emptyset, ())$. *Let* $C_1, \ldots, C_n$ *be the open S-CE calculations of c with respect to* $\tilde{c}$. *Let* $C_j = \{c_{j,1}, \ldots, c_{j,m_j}\}$ *with* $c_{j,l} \doteq S\text{-}CE(t_j, t'_j, \sigma_{j,l}, r)$ *and let* $\Sigma_j = S\text{-}CE(t_j, t'_j, \emptyset, r)$ *for every* $j \in \{1, \ldots, n\}$.

Then the implicit constraint set of c with respect to $\tilde{c}$ *is*

$$implicit\text{-}cs_{\tilde{c}}(c) := \bigotimes_{j=1,\ldots,n} \Sigma_j .$$

If the set of open S-CE calculations of c with respect to $\tilde{c}$ is empty we define

$$\textit{implicit-cs}_{\tilde{c}}(c) := \{\emptyset\}\,.$$

The following Lemma 5.2.19 states the main property of implicit constraint sets: When processing a call to *SCE-list-reduce* we can stop the calculation at every list element of the first argument and combine the intermediate result with the corresponding implicit constraint set without changing the result.

The following lemma states that calculating *SCE-list-reduce* as given in Def. 5.2.1 is equivalent to calculating the implicit constraint set of the initial call. I.e. the constraints generated by the subcalls to *S-CE* initiated by *SCE-list-reduce* can also be generated independently and be combined afterwards.

Lemma 5.2.19 (implicit constraint sets) *Let $\tilde{c}$ be a call to S-CE and c' the first call to SCE-list-reduce performed when processing $\tilde{c}$. Let Σ be the result of a call*

$$c' \doteq \textit{SCE-list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)$$

and let C_j denote the set of recursive calls to S-CE when processing the jth list element. Let Σ_j be the c-collection used in the recursive call

$$\textit{SCE-list-reduce}(((t_{j+1}, t'_{j+1}), \ldots, (t_n, t'_n)), \Sigma_j, r)$$

for all j and $\Sigma'_j = \textit{S-CE}(t_j, t'_j, \emptyset, r)$ then:

$$\forall_{l=0}^{n} \textit{combine-cs}(\Sigma) = \textit{combine-cs}(\Sigma_l \otimes \bigcup_{j=l}^{k} \Sigma'_j) = \textit{combine-cs}(\Sigma_l \otimes \textit{implicit-cs}_{\tilde{c}}(C_l))\,.$$

where $\Sigma_0 = \Sigma' = \Sigma'_0$.

Proof: See App. B.1, Page 124. □

The following lemma states the main property of *S-CE*'s output: Consider two types t_1 and t_2 that have a common element v under a free constraint set $\tilde{\sigma}$ (precisely its natural free type substitution). Consider a call to *S-CE* with t_1, t_2 and $\tilde{\sigma}$ as arguments and a recursion history r that is not arbitrary chosen but generated by *S-CE* starting with an initial call with empty recursion history. Then the result contains a free constraint set σ that essentially restrict t_1 and t_2 to types containing v.

But the free constraint sets σ (more precisely the corresponding natural free type substitutions) in general are not idempotent. Thus, the lemma precisely states common elements of t_1 and t_2 after σ has been applied to t_1 and t_2 an arbitrary number k of times.

The correctness of $S\text{-}CE$ is stated by the following Lemma 5.2.20: If during evaluating the call $S\text{-}CE(\tilde{t_1}, \tilde{t_2}, \emptyset, ())$ there is a subcall $S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$ then the set of common elements of the type pair (t_1, t_2) is preserved under building all instances $\sigma(t_1)$ and $\sigma(t_2)$ with $\sigma \in \Sigma = S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$. Furthermore, changing $\tilde{\sigma}$ to one of the $\sigma \in \Sigma$ just enlarges the set of values each variable is constrained to.

Lemma 5.2.20 (correctness of *S-CE*) *Let $t_1, t_2 \in \mathcal{T}$, both in set normalized form and let $c := S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$ be a call to S-CE that occurs as a recursive call after an initial call $\tilde{c}$ to S-CE with empty recursion information $()$. Let CE fulfill Assumption 3.4.6. Let there exist a value $v \neq \bot$ such that*

$$\forall k \in \mathbb{N} \,.\, v \in subst(\tilde{\sigma})^k(t_1) \boxdot subst(\tilde{\sigma})^k(t_2)\,. \tag{5.2}$$

Then there exist a free constraint set

$$\sigma \in combine\text{-}cs\text{-}cond_{(c,\tilde{c})}(S\text{-}CE(t_1, t_2, \tilde{\sigma}, r) \otimes implicit\text{-}cs_{\tilde{c}}(c))\ ^2$$

such that the free type substitution $\sigma' = subst(\sigma)$ compatible with σ fulfills

$$\forall k \in \mathbb{N} \,.\, v \in \sigma'^k(t_1) \boxdot \sigma'^k(t_2)\,. \tag{5.3}$$

Furthermore, if X is a free variable with $X \in dom(\tilde{\sigma})$ and $\tilde{\sigma}(X) = \tilde{t}_X$ then $X \in dom(\sigma)$ for every $\sigma \in S\text{-}CE(t_1, t_2, \tilde{\sigma}, r)$ and $t_X = \sigma(X)$ fulfills:

$$\langle\!\langle \tilde{t}_X \rangle\!\rangle(\tau) \subseteq \langle\!\langle t_X \rangle\!\rangle(\tau) \tag{5.4}$$

for every closed type substitution τ appropriate for $\tilde{t}_X$ and t_X.

Proof: See App. B.1, Page 128. □

Note that the second part of (5.2) in Lemma 5.2.20 is necessary because of the special understanding of quantified variables. Two types are just considered as types with common elements if there are common elements under *every* instantiation of the quantified variables with types $\neq \top$. E.g. the type (`list` $X_\forall$) has common elements with (`list` $\top$) but not with (`list` $Y_\forall$).

[2]The function *combine-cs-cond* conditionally calls *combine-cs* on its argument depending on its subscripts: It behaves like *combine-cs* if c is a (maybe indirect) recursive subcall of an execution of Rule (*Rec1*) or (*Rec2*) in the context of $\tilde{c}$. Otherwise, *combine-cs-cond* is the identity.

5.2.16 The Order of *S-CE* Rules

Though Lemma 5.2.20 states the correctness of *S-CE* independently from the order in which the rules are checked the precision of *S-CE* depends on the given order. This problem becomes obvious when considering intersection types not containing any elements as the following example shows:

Example 5.2.21 (order of *S-CE* rules) *Consider the call*

$$\textit{S-CE}(t_1, t_2, \emptyset, ()) \text{ with } t_1 = (\cup\ \textbf{\textit{int}}\ \textbf{\textit{bool}}), t_2 = (\cap\ \textbf{\textit{int}}\ \textbf{\textit{bool}})\,.$$

When decomposing the union first the result is given by the union of the results of the two subcalls S-CE($\textbf{\textit{int}}, t_2, \emptyset, ()$*) and S-CE(*$\textbf{\textit{bool}}, t_2, \emptyset, ()$*). Both of them fail and return the empty c-collection. As a result the c-collection returned by rule (U1) is also empty denoting no common elements.*

If on the other hand the intersection is decomposed first the result of

$$\textit{SCE-list-reduce}(((t_1, \textbf{\textit{int}}), (t_1, \textbf{\textit{bool}})), \emptyset, ())$$

is returned. SCE-list-reduce performs the subcalls

$$\textit{S-CE}(t_1, \textbf{\textit{int}}, \emptyset, ()) \text{ and } \textit{S-CE}(t_1, \textbf{\textit{bool}}, \emptyset, ())\,.$$

Both of these calls return $\Sigma_0 = \{\emptyset\}$*. The result of SCE-list-reduce is* Σ_0*. This is less precise than the failure in the case above.*

A discussion on all order dependencies between two rules is given in the following Remark 5.2.22:

Remark 5.2.22 (ordering of the *S-CE* cases) *On the one hand S-CE contains rules with conditions based on the structure of both terms* t_1 *and* t_2*. On the other hand the applicability of the rules (Rec1), (Rec2), (Var1), (Var2), (U1), (U2), (I1), (I2), (Comp1) and (Comp2) just relies on one of the terms. We discuss here that the chosen ordering of the last kind of rules is reasonable in order to get as precise results as possible.*

- *As we will see below there are indeed cases whose order is crucial for the desired result. The unfolding of recursive types itself does not interfere with any of the other cases directly. But when performed to late constructors that have to be decomposed early can be hidden by the* μ*-constructor. If e.g.* $t_1 = \mu X.(\cup\ \ldots\ C[X]\ \ldots)$ *and* $t_2 = (\cap\ \ldots)$ *then the unfolding of* t_1 *has to be done before decomposing* t_2*. The correct solution is to perform the unfolding of recursive types in the rules (Rec1) and (Rec2) before all other cases discussed here.*

- *To treat free variables in the next step is correct and useful in situations where one type to be tested is a free type variable X and the other one is constructed by $\cup$, $\cap$ or $\mathcal{C}$. In this case the most precise result we can get is to constrain X to the constructed type. This is just possible when this constructed type has not been decomposed, already.*

- *Let $t_1 = (\cup\ t_{1,1}\ \ldots\ t_{1,k})$ and $t_2 = (\cap\ t_{2,1}\ \ldots\ t_{2,k'})$. A value $v \in \langle\!\langle \tau(t_1) \rangle\!\rangle \cap \langle\!\langle \tau(t_2) \rangle\!\rangle$ for an arbitrary substitution τ must occur in* one *of the $\langle\!\langle \tau(t_{1,i}) \rangle\!\rangle$ and in* all *of the $\langle\!\langle \tau(t_{2,j}) \rangle\!\rangle$. Therefore to get common elements of t_1 and t_2* all *of the $\tau(t_{2,j})$ must have common elements with* the same *$\tau(t_{1,i})$. When first decomposing the intersection of t_2 S-CE checks whether all $\tau(t_{2,j})$ have common elements with* any *of the $\tau(t_{1,i})$. When on the other hand the union of t_1 is decomposed first then the desired stricter property is checked. It is therefore reasonable to decompose unions by the rules (U1) and (U2) before decomposing intersections by the rules (I1) and (I2).*
  Now let $t_1 = (\cup\ t_{1,1}\ \ldots\ t_{1,k})$ and $t_2 = \mathcal{C}t'_2$. We know that t'_2 is not a union or intersection type because otherwise t_2 not in set normalized form. When first processing the complement type then we check the property

$$\neg ST(\sigma(t_1), t'_2) \Leftrightarrow \neg \bigwedge_{i=1}^{k} ST(\sigma(t_{1,i}), t'_2) \Leftrightarrow \bigvee_{i=1}^{k} \neg ST(\sigma(t_{1,i}), t'_2)$$

  Decomposing the union first unites the results of checking $\neg ST(\sigma(t_{1,i}), t'_2)$ where the union behaves like disjunction when t_1 is already a semi-closed type. Thus, for a semi-closed t_1 both orders yield the same result, but for t_1 containing free variables processing the complement first introduces constraints that might be unnecessary. On the other hand decomposing the union first will introduce constraints containing $\top$ (done by free-to-top) just for certain free constraint sets in the returned c-collection and hence it is correct and reasonable to decompose unions before processing complements.
  Altogether it is correct to check the rules (U1) and (U2) before (I1), (I2), (Comp1) and (Comp2).

- *Let $t_1 = (\cap\ t_{1,1}\ \ldots\ t_{1,k})$ and $t_2 = \mathcal{C}t'_2$. Again, t'_2 is not a union or intersection type because t_2 is in set normalized form and therefore the complement constructor cannot hide a constructor that must be processed first. If t_1 is a semi-closed type then processing the complement first yields the test*

$$\neg ST(\sigma(t_1), t'_2) \Leftrightarrow \neg \bigvee_{i=1}^{k} ST(\sigma(t_{1,i}), t'_2) \Leftrightarrow \bigwedge_{i=1}^{k} \neg ST(\sigma(t_{1,i}), t'_2)$$

  On the other hand decomposing the intersection first causes a sequence of tests of the form $\neg ST(\sigma(t_{1,i}), t'_2)$ without changing the substitution because both types are already semi-closed. This sequence of tests is semantically equivalent to the conjunction above. Because of this it is correct to check the rules (I1) and (I2) before (Comp1) and (Comp2).

In the steps above we have proven the correctness and reasonability of the order (Rec1), (Rec2) → (Var1), (Var2) → (U1), (U2) → (I1), (I2) → (Comp1), (Comp2) in which S-CE applies the cases that just depend one one of the argument types. The order of the other cases can be chosen arbitrarily (except of (RecT) that must be checked before (Rec1) and (Rec2) in order to guarantee termination).

5.2.17 Further Optimizations

Recall the use of the function *combine-cs* in the rules (*Rec1*) and (*Rec2*) of *S-CE*. They were used to combine several free constraint sets generated after unfolding one of the argument types. Unfortunately, the application of *combine-cs* destroys some of the precision we got from the use of several different free constraint sets as results of the rules (*U1*) and (*U2*).

In the proof of Lemma 5.2.20 the fact that (*Rec1*) and (*Rec2*) apply *combine-cs* was just used for proving the correctness of (*RecT*). Indeed, when a recursive subcall generated by (*Rec1*) or (*Rec2*) returns without applying (*RecT*) we did not cut an execution that would lead to combinations of the different free constraint sets given in the intermediate result of the subcall of (*Rec1*) or (*Rec2*) and therefore the application of *combine-cs* is unnecessary.

Example 5.2.23 *Consider the types*

$$t_1 = \mu X.((\cup\ f(\boldsymbol{bool})\ f(\boldsymbol{int}))\ .\ (\cup\ \boldsymbol{nil}\ X))$$
$$t_2 = (f(A)\ .\ (f(A)\ .\ \boldsymbol{nil}))\,.$$

The call S-CE$(t_1, t_2, \emptyset, ())$ *is processed by rule (Rec1) initiating the subcall S-CE*$(t_1', t_2, \emptyset, r)$ *with* $t_1' = unfold(t_1)$ *and* $r = ((t_1, t_2))$*. This call is decomposed by rule (Constr) and the first subcall S-CE*$((\cup\ f(\boldsymbol{bool})\ f(\boldsymbol{int})), f(A), \emptyset, r)$ *processed by (U1) and (Constr) and (Var2) for every subcall yields* $\Sigma_1 = \{\{A \leftarrow \boldsymbol{bool}\}, \{A \leftarrow \boldsymbol{int}\}\}$*.*

For every free constraint set $\sigma \in \Sigma_1$ *the function SCE-list-reduce initiates a subcall*

$$S\text{-}CE((\cup\ \boldsymbol{nil}\ t_1), (f(A)\ .\ \boldsymbol{nil}), \sigma, r)\,.$$

(We will just discuss the call for $\sigma_1 = \{A \leftarrow \boldsymbol{bool}\}$ *in detail.) The call is processed by (U1) with the first subcall S-CE*$(\boldsymbol{nil}, (f(A)\ .\ \boldsymbol{nil}), \sigma_1, r)$ *returning with the empty c-collection as result. The second subcall S-CE*$(t_1, (f(A)\ .\ \boldsymbol{nil}), \sigma_1, r)$ *is processed by rule (Rec1) yielding the subcall*

$$S\text{-}CE(t_1', (f(A)\ .\ \boldsymbol{nil}), \sigma_1, r') \text{ with } r' = ((t_1, (f(A)\ .\ \boldsymbol{nil})), (t_1, t_2))\,.$$

Again, (Constr) initiates subcalls for two type pairs. The first of them returns

$$\Sigma_{2,1} = \{\{A \leftarrow (\cup\ \boldsymbol{bool}\ \boldsymbol{bool})\}, \{A \leftarrow (\cup\ \boldsymbol{bool}\ \boldsymbol{int})\}\}\,.$$

The two subcalls S-CE($(\cup\ \textbf{nil}\ t_1), \textbf{nil}, \sigma, r'$) *for* $\sigma \in \Sigma_{2,1}$ *just return* σ. *Thus,* $\Sigma_{2,1}$ *is returned.*

Analogously, for $\sigma_2 = \{A \leftarrow \textbf{int}\} \in \Sigma_1$ *we get*

$$\Sigma_{2,2} = \{\{A \leftarrow (\cup\ \textbf{int}\ \textbf{bool})\}, \{A \leftarrow (\cup\ \textbf{int}\ \textbf{int})\}\}\,.$$

Altogether, without applying combine-cs the initial call to S-CE returns $\Sigma = \Sigma_{2,1} \cup \Sigma_{2,2}$. *This result is correct because there was no application of rule (RecT) cutting a necessary computation.*

Furthermore, the application of *combine-cs* is just necessary if (*RecT*) was applied to the argument pair (t_1, t_2) the actual application of (*Rec1*) or (*Rec2*) inserted into the recursion information. For all other applications of (*RecT*) there exists a corresponding application of (*Rec1*) or (*Rec2*) and it suffices to apply *combine-cs* there.

Example 5.2.24 *Recall Example 5.2.8. The initial call S-CE*($t_1, t_2, \emptyset, ()$) *is processed by (Rec1) yielding the call S-CE*($t_1', t_2, \emptyset, ((t_1, t_2))$). *This call is processed by (Rec2) which yields S-CE*($t_1', t_2', \emptyset, ((t_1', t_2), (t_1, t_2))$).[3] *The only type pair processed by the rule (RecT) is* (t_1, t_2) *and it suffices to apply combine-cs to the result of the subcall of (Rec1).*

In order to perform this optimization we have to extend the return value of *S-CE*: Instead of just returning a c-collection Σ, *S-CE* has to return a pair $(\Sigma, \mathcal{R})$ where Σ is a c-collection as before and $\mathcal{R}$ is a set of type pairs of types (*RecT*) was applied on. This *recursion termination history* $\mathcal{R}$ is maintained as follows:

- For a call *S-CE*(t_1, t_2, σ, r) the rules (*RecT*) returns $(\{\sigma\}, \{(t_1, t_2)\})$.
- The behaviour of (*Rec1*) and (*Rec2*) applied to a type pair (t_1, t_2) depends on the result $(\Sigma, \mathcal{R})$ of its recursive subcall as follows:
  - If $(t_1, t_2) \in \mathcal{R}$ then the return value of (*Rec1*) or (*Rec2*), respectively, is
  $$(\textit{combine-cs}(\Sigma), \mathcal{R} \setminus \{(t_1, t_2\})\,.$$
  - Otherwise, $(\Sigma, \mathcal{R})$ is returned.
- The rules (*U1*), (*U2*), (*I1*), (*I2*), (*Constr*), (*Frame*), (*Env*) return the union of the recursion termination histories of their subcalls.
- The rules (*Top1*), (*Top2*), (*BothVar*), (*Var1*), (*Var2*), (*Comp1*), (*Comp2*), (*FunQ*), (*Base*) return an empty recursion termination history.

[3] $t_1 = \mu X.((\cup\ f(\texttt{bool})\ f(\texttt{int}))\ .\ (\cup\ \texttt{nil}\ X))$, $t_2 = \mu Y.(f(A)\ .\ (\cup\ \texttt{nil}\ Y))$, $t_i' = \textit{unfold}(t_i)$ as in Example 5.2.8.

Example 5.2.25 *Consider the following two types*

$$t_1 = \mu X.(\cup\ \textit{\textbf{nil}}\ ((\textit{\textbf{int}}\ .\ A)\ .\ X))$$
$$t_2 = ((\cup\ (\textit{\textbf{int}}\ .\ \textit{\textbf{posint}})\ (\textit{\textbf{int}}\ .\ \textit{\textbf{bool}}))\ .\ \textit{\textbf{nil}})$$

After the first unfolding step by rule (Rec1) S-CE checks $(\cup\ \textit{\textbf{nil}}\ ((\textit{\textbf{int}}\ .\ A)\ .\ t_1))$ *and* t_2*. Rule (U1) initiates two subcalls checking*

1. $\textit{\textbf{nil}}$ *and* t_2
2. $((\textit{\textbf{int}}\ .\ A)\ .\ t_1)$ *and* t_2

The first of these subcalls yields the empty c-collection while the second subcall is processed by rule (Constr) which initiates a call to SCE-list-reduce with the following argument pairs:

- $(\textit{\textbf{int}}\ .\ A)$ *and* $(\cup\ (\textit{\textbf{int}}\ .\ \textit{\textbf{posint}})\ (\textit{\textbf{int}}\ .\ \textit{\textbf{bool}}))$
- t_1 *and* $\textit{\textbf{nil}}$

The first of these subcalls yields the c-collection $\Sigma = \{\sigma_1, \sigma_2\}$ *with* $\sigma_1 = \{A \leftarrow \textit{\textbf{posint}}\}$ *and* $\sigma_2 = \{A \leftarrow \textit{\textbf{bool}}\}$*. The second subcall succeeds for both* σ_1 *and* σ_2 *without changing any of these free constraint sets.*

In the optimized algorithm S-CE the rule (Rec1) just passes through the result Σ *of unfolding* t_1 *the first time. The original algorithm applies combine-cs to* Σ *yielding*

$$\Sigma' = \{\sigma_1, \sigma_2, \{A \leftarrow (\cup \textit{\textbf{posint}}\ \textit{\textbf{bool}}\}\}\,.$$

By introducing a further free constraint set the unnecessary call to combine-cs causes the loss of information.

As the example shows calling *combine-cs* just when necessary causes the rules (*Rec1*) and (*Rec2*) to provide a more precise output. Furthermore, *combine-cs* is a quite expensive operation and calling it just when necessary makes *S-CE* more efficient.

5.3 The Algorithm *CE*

The algorithm *CE* is the main algorithm approximating the question of common elements of two given types t_1 and t_2. Its first step is to call *S-CE* with the types t_1 and t_2, an

empty constraint set and an empty recursion information as input. The c-collection resulting from this call (in the case of success) consists of free constraint sets whose natural free type substitutions are not idempotent. *CE* transforms these free constraint sets into idempotent free type substitutions independently from each other. This is done by repeatedly inserting the corresponding values for the variables into the right hand sides occurring in the substitution. Recursive dependencies between variables are eliminated by introducing recursive bindings by the μ constructor.

The following example shows the intended result of *CE* for a variable depending on itself:

Example 5.3.1 *Consider a call to S-CE which result just contains the free constraint set* $\{X \leftarrow f(X)\}$ *with a free type constructor* $f(\cdot)$ *of arity* 1*. We want CE to transform this free constraint set into the idempotent free type substitution* $\{X \leftarrow \mu Y.f(Y)\}$*.*

For several variables that mutually depend on each other the intended result is presented in the following example:

Example 5.3.2 *Consider a call to S-CE returning the c-collection* $\Sigma = \{\sigma\}$

$$\{\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(X_1)\}\}\,.$$

The dependencies between the variables are given in Fig. 5.2 where an arrow from a variable V *to a type* t *expresses that* V *is constrained to* t*.*

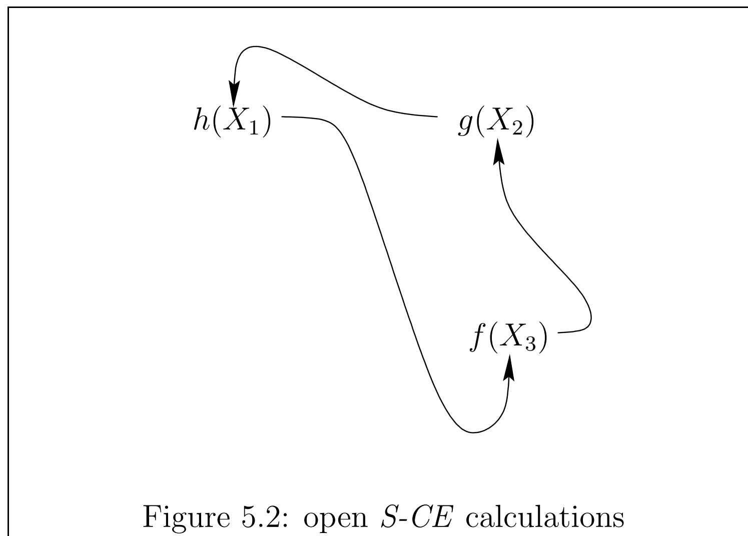

Figure 5.2: open *S-CE* calculations

The intended result of CE is an s-collection consisting of the idempotent free type substitution

$$\{X_1 \leftarrow f(g(\mu V_3\,.\,h(f(g(V_3))))),$$
$$X_2 \leftarrow g(\mu V_3\,.\,h(f(g(V_3)))),$$
$$X_3 \leftarrow \mu V_3\,.\,h(f(g(V_3)))\}\,.$$

5.3.1 The Core Component of *CE*

The following definition presents the core component of *CE*. It consists of a call to *S-CE* and a loop transforming every free constraint set σ' in the result Σ' of *S-CE* into a free type substitution. All these free type substitutions are collected in an s-collection Σ.

Definition 5.3.3 (algorithm *CE*) *The algorithm CE takes two terms and returns an s-collection* Σ*.*

Algorithm: *CE*
Input: *Two terms* t_1 *and* t_2*.*
Output: *An s-collection* Σ*.*

$\Sigma' := \textit{S-CE}(t_1, t_2, \emptyset, ())$
$\Sigma := \emptyset$
forall $\sigma' \in \Sigma'$ ***do*** *(* Transform free constraint sets into free type substitutions *)*
  $\sigma'' := \textit{GIS}(\sigma')$ *(* generate idempotent substitution from* σ' **)*
  $\Sigma := \Sigma \cup \{\sigma''\}$ *(* collect result substitutions in s-collection* Σ **)*
end *(* forall *)*
return Σ

5.3.2 Auxiliary Functions used by *CE*

The main task of *CE*, i.e. the elimination of the variables from the right hand sides of a single free type substitution is done by *GIS*. It is done by calculating an order in which the assignments to the variables depend on each other. Following this order, *GIS* inserts the right hand sides of already processed variables into the right hand sides of different variables. If a variable depends on itself a recursive type is generated. *GIS* is defined as follows:

Definition 5.3.4 (algorithm *GIS*) *The algorithm GIS expects a free type substitution and returns a free type substitution that is idempotent.*

Algorithm: *GIS (generate idempotent substitution)*
Input: *A substitution* σ
Output: *An idempotent substitution* $\tilde{\sigma}$.

$G := (V, R)$ *with* $V = dom(\sigma')$ *and* $R = \{(y, x) \mid x \neq y, x \leftarrow t \in \sigma'$ *and* y *is subterm of* $t\}$
$G' := (V', R')$ *is the component graph of* G. *(* cf. Def. 2.2.4 *)*
Mark all nodes $v \in V'$ *with* 0.
$\sigma'' := \emptyset$
while *there are nodes* $v \in V'$ *marked with* 0 ***do***
  select $v' \in V'$ *with all predecessors of* v' *marked with* 1
  mark v' *with* 1
  if v' *represents a single node in* G ***then***
    $t := \sigma'(v')$ *(* lookup* v' *in* σ' **)*
    $t' := \sigma''(t)$ *(* apply already known substitution* σ'' **)*
    if t' *contains* v' *as subterm*
      then $t' := \mu X.t'[v'/X]$ *with a new variable* $X \in V_f$
    $\sigma'' := \sigma'' \cup \{v' \leftarrow t'\}$
  else *(** v' *represents more than one node in* V **)*
    Let $\tilde{V} \subseteq V$ *be the set of nodes* $v \in V$ *represented by* v'.
    $\sigma_r := \{X \leftarrow t \mid X \in \tilde{V}, X \leftarrow \tilde{t} \in \sigma', t = \sigma''(\tilde{t})\}$ *(** $\sigma_r = \sigma'' \circ \sigma'|_{\tilde{V}}$ **)*
    $\tilde{\sigma} := SMR(\sigma_r)$
    $\sigma'' := \sigma'' \cup \tilde{\sigma}$
  end *(* if-then-else *)*
end *(* while *)*
return σ''

For nodes $v' \in V'$ denoting more than one node in V (i.e. for several variables mutually depending on each other) *GIS* extracts a free type substitution restricted to the nodes denoted by v' and passes the processing to *SMR*.
The algorithm *SMR* performs insertions of variables and the introduction of recursive types in a certain order to eliminate cyclic dependencies. This order $<_V$ on the variables must be fixed but can be chosen arbitrarily. *SMR* is defined as follows:

Definition 5.3.5 (algorithm *SMR*) *SMR needs an 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 *in its own assigned term are expressed by a recursive type:*

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

for $i = 1$ ***to*** k ***do***
  $t := \sigma(X_i)$
  if X_i *occurs in* t_{X_i} *(* 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* $\tilde{X}_i \in V_f$
    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* $j > i$ **)*
    (* $\sigma := (\sigma|_{\{X_i\}} \circ \sigma_{\{X_{i+1},\dots,X_k\}}) \cup \sigma_{\{X_1,\dots,X_i\}}$ ****)***
    $t_{X_j} := \sigma(X_j)$
    $t'_{X_j} := t_{X_j}[X_i/t'_{X_i}]$
    $\sigma := \sigma \setminus \{X_j \leftarrow t_{X_j}\} \cup \{X_j \leftarrow t'_{X_j}\}$
  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* $j < i$ **)*
    $t_{X_j} := \sigma(X_j)$
    $t'_{X_j} := t_{X_j}[X_i/t'_{X_i}]$
    $\sigma := \sigma \setminus \{X_j \leftarrow t_{X_j}\} \cup \{X_j \leftarrow t'_{X_j}\}$
  end *(* for j *)*
end *(* for i *)*
In every t_{X_j} *replace every* $\mu\tilde{X}_i.t$ *in a position where* $\tilde{X}_i$ *is already bound by some* μ *constructor.*

Example 5.3.6 (*SMR*) *Consider a call to SMR with the input substitution* σ *defined by*

$$\sigma = \{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(X_1)\}\,.$$

Assume that $X_1 <_V X_2 <_V X_3$*. Processing the first i loop for* $i = 1$ *SMR inserts* X_1 *into the right hand side of the variables that are greater with respect to* $<_V$*. This insertion yields*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(f(X_2))\}\,.$$

For $i = 2$*,* X_2 *is inserted into the right hand side of* X_3*. The result is:*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow h(f(g(X_3)))\}\,.$$

For $i = 3$*, the occurrence of* X_3 *in its own right hand side is eliminated by introducing a recursive type:*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(X_3), X_3 \leftarrow \mu V_3\,.\,h(f(g(V_3)))\}$$

Now the second i-loop is processed. For $i = 3$*, SMR inserts the value of* X_3 *into the right hand sides of all variables that are smaller with respect to* $<_V$*. The result is*

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(\mu V_3\,.\,h(f(g(V_3)))), X_3 \leftarrow \mu V_3\,.\,h(f(g(V_3)))\}\,.$$

With $i = 2$ the same is done for X_2:

$$\{X_1 \leftarrow f(g(\mu V_3 \,.\, h(f(g(V_3))))), X_2 \leftarrow g(\mu V_3 \,.\, h(f(g(V_3)))), X_3 \leftarrow \mu V_3 \,.\, h(f(g(V_3)))\}$$

For $i = 1$, there is nothing to be done and the substitution above is returned as result of SMR.

Note that the result substitutions provided by *GIS* and *SMR* are *idempotent*.

5.3.3 Examples of Calls to *CE*

Recalling the input of Example 5.2.15 we get the following example for *CE*:

Example 5.3.7 (*CE*) *Consider t_1 and t_2 as defined in Ex. 5.2.15 and a call CE(t_1, t_2). This call causes the subcall to S-CE discussed in Ex. 5.2.15 and yields the result calculated there:*

$$\Sigma' := \{\{A \leftarrow (\cup\ \textit{\textbf{string nat}})\}\}$$

For $\sigma' = \{A \leftarrow (\cup\ \textit{\textbf{string nat}})\}$, the algorithm CE generates the graph $G = (V := \{A\}, R := \emptyset)$ and the component graph $G' = G$. The only node $A \in V'$ represents the single node $A \in V$ and $t' = \sigma(A)$ does not contain A as a subterm. Thus, $\sigma'' := \{A \leftarrow (\cup\ \textit{\textbf{string nat}})\}$ and $\Sigma := \{\sigma''\}$ is returned.

The following example resents some extended work of *CE* including real work for the subroutines *GIS* and *SMR*:

Example 5.3.8 (*CE*) *Consider the types*

$$t_1 = (X\ .\ (Y\ .\ (Z\ .\ \textit{\textbf{nil}})))$$

and

$$t_2 = ((Y\ .\ X)\ .\ (Z\ .\ ((\textit{\textbf{num}}\ .\ X)\ .\textit{\textbf{nil}})))\,.$$

When fixing $<_V$ to $X <_V Y <_V Z$ then S-CE returns the following c-collection:

$$\Sigma' := \{\{X \leftarrow (Y\ .\ X), Y \leftarrow Z, Z \leftarrow (\textit{\textbf{num}}\ .\ X)\}\}$$

The graph G calculated in GIS for the only element σ' of Σ' is given in Fig. 5.3. It has exactly one strongly connected component and thus the component graph G' consists of exactly one node.

Processing the only node of G' yields a call to SMR with the σ' as argument. The individual iterations of the first i-loop performs the following changes:

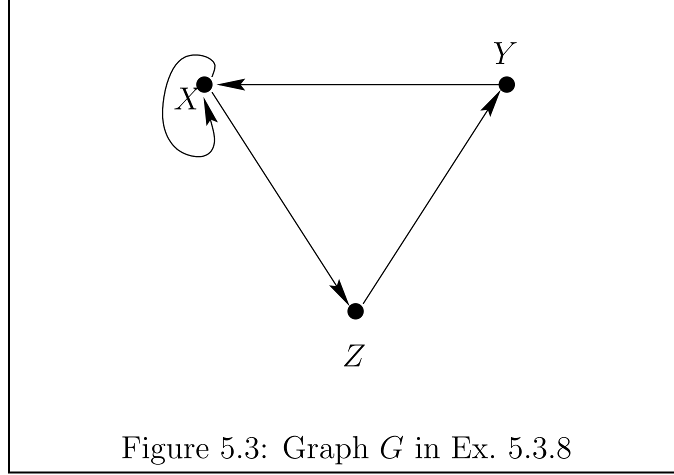

Figure 5.3: Graph G in Ex. 5.3.8

1. *X is recursively bound by a μ constructor in $t_X = (Y\ .\ X)$ yielding $t'_X = \mu X.(Y\ .\ X)$. In t_Z X is replaced by t'_X yielding $(\textbf{num}\ .\ \mu X.(Y\ .\ X))$.*
2. *For $i = 2$, t_Y is not changed because it does not contain Y as subterm. In t_Z as generated in the step before Y is replaced by t_Y yielding $(\textbf{num}\ .\ \mu X.(Z\ .\ X))$.*
3. *The occurrence of Z in t_Z from the step before is recursively bound by μ. The resulting term is $\mu Z.(\textbf{num}\ .\ \mu X.(Z\ .\ X))$*

After processing the first i-loop the intermediate free variable constraint is:

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

The second i-loop has a descending argument. For the individual values of i the following tasks are performed:

3. *$t_Y = Z$ is replaced by $t_Z = \mu Z.(\textbf{num}\ .\ \mu X.(Z\ .\ X))$.*
2. *Y is replaced by t_Y from the step before in t_X yielding $\mu X.(\mu Z.(\textbf{num}\ .\ \mu X.(Z\ .\ X))\ .\ X)$.*
1. *Nothing to do.*

The resulting free constraint set after executing the second i-loop is:

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

The nested μ binding of X in t_X can now be removed, yielding $\mu X.(\mu Z.(\textit{num} \; . \; (Z \; . \; X)) \; . \; X)$.

The resulting substitution is the only element of the s-collection Σ returned by CE, i.e.:

$$\begin{aligned}\Sigma = \{\{X &\leftarrow \mu X.(\mu Z.(\textit{num} \; . \; (Z \; . \; X)) \; . \; X),\\ Y &\leftarrow \mu Z.(\textit{num} \; . \; \mu X.(Z \; . \; X)),\\ Z &\leftarrow \mu Z.(\textit{num} \; . \; \mu X.(Z \; . \; X))\}\}\end{aligned}$$

We can show that this is an s-collection by renaming of the μ-bounded variables resulting in:

$$\begin{aligned}\Sigma = \{\{X &\leftarrow \mu V.(\mu W.(\textit{num} \; . \; (W \; . \; V)) \; . \; V),\\ Y &\leftarrow \mu W.(\textit{num} \; . \; \mu V.(W \; . \; V)),\\ Z &\leftarrow \mu W.(\textit{num} \; . \; \mu V.(W \; . \; V))\}\}\end{aligned}$$

In the example above the only free type substitution $\sigma \in \Sigma$ has the property that every $v \in t_1 \boxdot t_2$ fulfills $v \in \sigma(t_1) \boxdot \sigma(t_2)$.

5.3.4 Properties of *CE*

In this section we prove several properties of *CE* that are necessary in order to make *CE* practically usable.

In order to prove termination and correctness of *CE* for every input we first prove these properties for the auxiliary functions *SMR* and *GIS*:

Lemma 5.3.9 (termination of *SMR*) *The algorithm SMR terminates for every input substitution with a finite domain dom(σ).*

Proof: See App. B.2, Page 136. $\square$

Lemma 5.3.10 (correctness of *SMR*) *Let σ' be a substitution such that the graph $G = (V, R)$ defined as in GIS with $V = dom(\sigma')$ and $R = \{(y, x) \mid x \neq y, x \leftarrow t \in \sigma'$ and y is subterm of $t\}$ contains more than one node and consists of a single strongly connected component. Let $\sigma = SMR(\sigma')$. Then*

$$\llbracket \sigma \circ \sigma'(t) \rrbracket(\phi) = \llbracket \sigma(t) \rrbracket(\phi)$$

for every type term t and every closed type substitution ϕ appropriate for $\sigma \circ \sigma'(t)$ and $\sigma(t)$.

Proof: See App. B.2, Page 136. $\square$

Lemma 5.3.11 (termination of *GIS*) *The algorithm GIS terminates for every input substitution with a finite domain dom(σ).*

Proof: See App. B.2, Page 138. □

Lemma 5.3.12 (correctness of *GIS*) *Let σ' be a substitution fulfilling with the following properties:*

1. *σ' does not contain a variable binding $A \leftarrow B$ with $B \in dom(\sigma')$.*
2. *If σ' contains a variable binding $A \leftarrow C[B]$ with a context C and a variable $B \in dom(\sigma')$ then there exists a variable $B' \notin dom(\sigma')$ such that B is bound to B' or a union containing B' in σ'.*

Let v be a value fulfilling

$$\forall k \in \mathbb{N} \,.\, v \in {\sigma'}^k(t_1) \boxdot {\sigma'}^k(t_2)$$

and let $\sigma = GIS(\sigma')$. Then

$$v \in \sigma(t_1) \boxdot \sigma(t_2)\,.$$

Proof: See App. B.2, Page 138. □

The following lemma essentially states the termination of *CE*. The termination proof in this lemma relies on the termination of *ST*. The termination of *CE* without this restriction is proven afterwards.

Lemma 5.3.13 *If the algorithm S-CE terminates for every pair of terms in set normalized form (and empty free constraint set and empty recursion information) then CE terminates for every pair of input types in set normalized form.*

Proof: See App. B.2, Page 141. □

We can now prove the unrestricted termination of *CE*. With Lemma 5.3.13 given, the proof consists of proving the termination of the loop

$$CE \rightarrow S\text{-}CE \rightarrow ST \rightarrow CE \tag{5.5}$$

between the mutually dependent algorithms *CE*, *S-CE* and *ST*. Figure 5.4 shows the termination dependencies. An arrow from an Algorithm 1 to an Algorithm 2 expresses that the

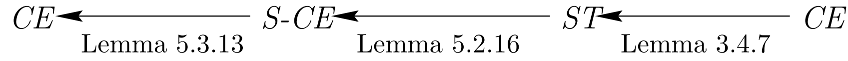

Figure 5.4: termination dependencies between the algorithms

termination of Alg. 2 depends on termination of Alg. 1. The corresponding lemma is given under every arrow.

Informally, the termination proof for *CE* (Theorem 5.3.14) uses the fact that every execution of the loop given in (5.5) reduces the number of complement type constructors occurring in the two types that are checked. None of the algorithms processed during this loop can introduce new complement type constructors. Thus, there is a finite bound for the number of executions of this loop. This is formally stated in the following theorem:

Theorem 5.3.14 (termination of *CE*) *The algorithm CE terminates for every pair of input types in set normalized form.*

Proof: See App. B.2, Page 141. □

Given Theorem 5.3.14 we can now easily conclude the termination of *ST* for every input. This extends the termination proof given in Lemma 3.4.7.

Corollary 5.3.15 *The algorithm ST terminates for every pair of ground input types in set normalized form.*

Proof: See App. B.2, Page 142. □

Analogously we can extend Lemma 5.2.16 as follows:

Corollary 5.3.16 *The algorithm S-CE terminates for every pair of input types in set normalized form.*

Proof: See App. B.2, Page 142. □

Theorem 5.3.17 (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*

$$v \in t_1 \boxdot t_2 \,.$$

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

$$v \in \sigma(t_1) \boxdot \sigma(t_2) \,.$$

Proof: See App. B.2, Page 142. □

Chapter 6

Conclusions and Future 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 is sound with respect to the question whether the value sets denoted by two types are disjoint, i.e. whether no instances denote any common elements. The result can be used to infer a sound approximation for the question whether a function call *must* fail because of a type error.

The underlying type language is an expressive one that contains e.g. value assignments (i.e. types denoting exactly one element besides $\perp$) and can capture the value sets occurring in the functional language Scheme [KCE98] quite precisely. It differs from usual type languages especially in the definition of function types. We explained why the usual function type constructor is inappropriate for complete type checking and presented a new notion of partial function types (pfts). Though pfts might be inappropriate for representing function types during type inference their benefit is to clarify the needed information about functions on a theoretical level and to serve as representation types for functions in the output of a type inference system.

The algorithm *CE* 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 resulting free constraint sets restrict the types t that can be assigned to a certain variable X by stating which values must at least be denoted by t.
The second component transforms the resulting constraint sets into idempotent substitutions for type variables. This is done by iteratedly inserting the assignments of variables into the right hand sides of other variables and by introducing recursive bindings by the μ type constructor. A prototype implementation in Scheme of the algorithm *CE* described in this work will be finished soon.

Though this work shows how a single 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. To implement a type checker based on *CE* we rather concentrate on a different mechanism to solve sets of such constraints. One possibility is the implementation of a type inference system in terms of an abstract interpreter. This allows to solve every non-empty intersection constraint "on the fly" directly after generating it. A prototype of an abstract interpreter that is restricted to *type checking* (i.e. the input types of the main function have to be provided) gave promising results. We are currently working on the specification of a complete *type inference* system based on abstract interpretation.

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[CE91] William Clinger and Jonatan Rees (Editors). *Revised^4 Report on the Algorithmic Language Scheme*, November 1991.

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

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press Chambridge, Massachusetts; London, England, 1990.

[Dam94] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software: International Symposium*, volume 789 of *Lecture Notes in Computer Science*, pages 687–706. Springer-Verlag, April 1994.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 245–320. Elsevier Science Publishers B.V., 1990.

[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.

[Fro98] Dieter Froning. Lokale Normalisierung von Scheme-Programmen. Diplomarbeit, FernUniversität Hagen, April 1998.

[GLS93] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer Verlag Berlin, Heidelberg, New York, second edition, 1993.

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

[JK86] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4):1155–1194, November 1986.

[Jou95] J. P. Jouannaud. Rewrite proofs and computations. International Summer School Marktoberdorf "Logic of computation", Institut für Informatik, Technische Universität München, 1995.

[KB67] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, 1967.

[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.

[PS81] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, April 1981.

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

[Smi94] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

[WB99] Manfred Widera and Christoph Beierle. Local normalization of functional programs. Informatik Berichte 248, FernUniversität Hagen, January 1999.

[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. A term rewriting scheme for function symbols with variable arity. Informatik Berichte 280, FernUniversität Hagen, January 2001.

Appendix A

Proofs of Chapter 3

A.1 Proofs of Section 3.1

Lemma 3.1.53 *The term rewriting system R_{SN} terminates for every input type.*

Proof of Lemma 3.1.53: For a type t and a position p with $t_{|p} \neq$ `undefined` the function *compargs* yields the number of proper prefixes p' of p such that $t_{|p'}$ has $\mathcal{C}$ as top level constructor. For a type t the set *unint-pos* contains all positions $\tilde{p}$ with $t_{|\tilde{p}}$ having the top level constructor $\cup$ or $\cap$. Furthermore, for a type t we define *diffcount*(t) as the number of difference type constructors occurring in t and *compcount*(t) as the number of complement type constructors occurring in t.

Now consider a term t during its normalization by R_{SN}:

- Whenever rule one is applied *diffcount*(t) is decremented by one. Since t has a finite representation *diffcount*(t) is finite. None of the other rules increases *diffcount*(t) and thus the starting value of *diffcount*(t) is a bound for the number of applications of the first rule.

- When applying the second or third rule, $\sum_{p \in \textit{unint-pos}(t)} \textit{compargs}(t,p)$ is decreased by one. The only rule increasing this value is the first one introducing new $\cap$ and $\mathcal{C}$ constructors. We can switch to $\sum_{p \in \textit{unintdiff-pos}(t)} \textit{comp-diff-args}(t,p)$ where *unintdiff-pos* also yields all position with $\setminus$ at top level and *comp-diff-args* differs from *compargs* in also considering those prefixes $p'.2.\epsilon$ with $t_{|p'}$ having $\setminus$ as top level type constructor. This value is still decreased by one with every application of the second or third rule, but no longer increased by any of the other rules. Therefore, the number of applications of the second

and third rules is bounded by this value for the initial t which is finite due to the finite representation of t.

- The fourth rule decreases *compcount*(t) by 2 in every application. Since the starting value of *compcount*(t) is finite and each of the finite number of applications of rules 2 and 3 increases it just by a finite amount the fourth rule can just be applied a finite number of times.

Altogether the application of R_{SN} to an arbitrary type term t terminates. □

Lemma 3.1.54 *The term rewriting system R_{SN} is confluent.*

Proof of Lemma 3.1.54: Since we have termination of R_{SN} from Lemma 3.1.53 we just have to prove local confluence. This can be done by considering the critical pairs of R_{SN}.

There are just two critical pairs:

- Rule 2 and rule 4 have the critical value $\mathcal{CC}(\cup < a_1 \dots e_1 >)$. This can be normalized to $(\cup < a_1 \dots e_1 >)$ by rule 4 directly or we can apply rule 2 followed by rule 3 yielding $(\cup \ < \mathcal{CC}a_1 \dots \mathcal{CC}a_1 >)$. Here we can apply rule 4 to every argument of the union and get the same result $(\cup < a_1 \dots e_1 >)$ as before.

- Rule 3 and rule 4 have the critical value $\mathcal{CC}(\cup < a_1 \dots e_1 >)$. The proof is analogous to the case above with exchanging the application of the rules 2 and 3.

Since all critical pairs have a common normal form and the term rewriting system R_{SN} terminates it is also confluent. □

Lemma 3.1.55 *The result type t' returned when applying R_{SN} to an arbitrary type t is in set normalized form.*

Proof of Lemma 3.1.55: Consider a type $\tilde{t}$ that is not in set-normalized form. Then $\tilde{t}$ must violate at least one of the conditions of Def. 3.1.51:

1. If $\tilde{t}$ violates the first condition then it contains a difference type constructor and therefore rule 1 of R_{SN} applies to $\tilde{t}$.

2. If condition (2) is violated there is a nested occurrence of the complement type constructor and rule 4 applies.

3. If $\tilde{t}$ violates condition (3) then there exists a complement type constructor whose argument is either a union type (rule 2 applies) or an intersection type (rule 3 is applicable).

In all cases of a type $\tilde{t}$ not in set-normalized form there is still a rule in R_{SN} that is applicable to $\tilde{t}$. Thus, all result types of R_{SN} are in set-normalized form. □

Lemma 3.1.56 *Let t be an arbitrary type and t' be the result of applying R_{SN} to a type t. Then $\langle\!|t|\!\rangle(\sigma) = \langle\!|t'|\!\rangle(\sigma)$ for every appropriate closed type substitution σ.*

Proof of Lemma 3.1.56: We just have to prove the lemma for the types t that form the left hand side of one of the rules. Since the semantics of constructed types just depends on the semantics of the argument, but not on the representation the semantics does not change be replacing a subterm by an equivalent one.

By Def. 3.1.33 and Def. 3.1.42 the semantics of the type constructors $\cup$, $\cap$, $\setminus$ and $\mathcal{C}$ directly models the corresponding set operation, respectively. We therefore have to show that the changes made by the rules of R_{SN} are correct transformations for the set operations:

1. First rule: Let A and B be sets and e an element.

$$e \in A \setminus B \Leftrightarrow e \in A \wedge e \notin B \Leftrightarrow e \in A \wedge e \in \mathcal{C}B \Leftrightarrow e \in A \cap \mathcal{C}B\,.$$

2. Second rule: Let $A_1, \dots, A_k$ be sets and e an element.

$$\begin{aligned} & e \in \mathcal{C}(\cup\ A_1\ \dots\ A_k) \\ \Leftrightarrow & e \notin (\cup\ A_1\ \dots\ A_k) \\ \Leftrightarrow & \forall i\,.\, e \notin A_i \\ \Leftrightarrow & \forall i\,.\, e \in \mathcal{C}A_i \\ \Leftrightarrow & e \in (\cap\ \mathcal{C}A_1\ \dots\ \mathcal{C}A_k)\,. \end{aligned}$$

3. Third rule: Let $A_1, \dots, A_k$ be sets and e an element.

$$\begin{aligned} & e \in \mathcal{C}(\cap\ A_1\ \dots\ A_k) \\ \Leftrightarrow & e \notin (\cap\ A_1\ \dots\ A_k) \\ \Leftrightarrow & \exists i\,.\, e \notin A_i \\ \Leftrightarrow & \exists i\,.\, e \in \mathcal{C}A_i \\ \Leftrightarrow & e \in (\cup\ \mathcal{C}A_1\ \dots\ \mathcal{C}A_k)\,. \end{aligned}$$

4. Fourth rule: Let A be a set and e an element.

$$e \in \mathcal{C}\mathcal{C}A \Leftrightarrow e \notin \mathcal{C}A \Leftrightarrow e \in A\,.$$

□

A.2 Proofs of Section 3.4

Lemma 3.4.7 *Let CE be an algorithm fulfilling Assumption 3.4.5 and let STbase fulfill Assumption 3.4.1. Then every call to ST for an arbitrary pair of input arguments in set normalized form terminates.*

Proof of Lemma 3.4.7: The termination is trivial for the cases (1), (2), (4) and (10) because these cases directly return either `true` or `false`. For case (3) the termination just depends on the termination of *STbase* that is given by Assumption 3.4.1 and for case (9f) the termination depends on *CE* which terminates according to Assumption 3.4.5.

The cases (5), (6), (7) and (9) (except of case (9f)) decompose the top level constructor of at least one of the given types and just cause direct recursive calls to *ST*. If neither t_1 nor t_2 contain a recursive type constructor then the number of possible decompositions of t_1 is bounded by the maximal number b_1 of nested type constructors (and analogously by b_2 for t_2). (This holds because all semi-closed types must have a finite representation in terms of base types and type constructors.) The number of recursive calls to *ST* is bounded by $b_1 + b_2$.

If $t = \mu X.t'$ then the maximal number of decompositions of t (including the unfolding step) until reaching t again is bounded by a finite number b' (again because of the finite representation of t). If this number is bounded by b'_1 for all recursively defined subterms in t_1 and by b'_2 for t_2 and we need at most c_1 or c_2 unfolding steps to get the first μ-constructor on a decomposition path then after at most $c_1 + c_2 + b'_1 + b'_2$ recursive calls to *ST* a call is performed with a argument pair that already occurs on the stack of recursive calls. Because of this subcase (8a) guarantees the termination of case (8). □

Lemma 3.4.8 *Let $t_1, t_2 \in \mathcal{T}_g$ be semi-closed types in set normalized form. Let STbase fulfill Assumption 3.4.1 and let CE fulfill Assumption 3.4.6. Then*

$$ST(t_1, t_2) = \textit{\textbf{true}} \Rightarrow \langle\!\langle \sigma(t_1) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t_2) \rangle\!\rangle$$

for every closed type substitution σ appropriate for t_1 and t_2.

Proof of Lemma 3.4.8: First we prove the lemma along the case distinction of the algorithm for those cases directly returning a value without recursion. We therefore assume σ to be an arbitrary closed type substitution appropriate for t_1 and t_2.

- For every semi-closed type $t \in \mathcal{T}_{SCS}$ obviously $\langle\!\langle \sigma(t) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t) \rangle\!\rangle$ holds. Thus, case (1) does not violate the lemma.
- For every closed type t the following properties hold:
  - $\perp \in \langle\!\langle t \rangle\!\rangle$. This implies $\langle\!\langle \perp \rangle\!\rangle \subseteq \langle\!\langle \sigma(t) \rangle\!\rangle$ and therefore proves correctness of the first subcase of case (2).

– $\langle\!\langle \sigma(t) \rangle\!\rangle \subseteq \mathcal{V} = \langle\!\langle \top \rangle\!\rangle$ proves correctness of the second subcase of (2).

- If t_1 and t_2 are base types or value assignments the answer given by *STbase* that is correct due to Assumption 3.4.1. Case (3) is therefore correct.
- Because of $\langle\!\langle \mathit{Tfunc} \rangle\!\rangle = \langle\!\langle \mathit{Tfunc}_P \rangle\!\rangle \cup \langle\!\langle \mathit{Tfunc}_U \rangle\!\rangle$ case (4) is trivially correct.

For the remaining cases we prove the lemma by induction on the number of recursive calls needed to generate an answer for a given pair of arguments. Because of Lemma 3.4.7[1] this number is always finite:

$\mathbf{n = 0}$: This case is always processed by the cases already proven above.

$\mathbf{n \rightarrow n + 1}$: Again the proof is given along the cases of the algorithm. (One has to note in this part of the proof that from the fact that σ is appropriate for t_1 and t_2 this property also holds for every subterm of t_1 and t_2.)

- Types constructed by tuple like constructors can just be subtypes of each other when the top level constructors are equal. In this case:

  $$\langle\!\langle \sigma((c\ t_{1,1}\ \dots\ t_{1,k})) \rangle\!\rangle \subseteq \langle\!\langle \sigma((c\ t_{2,1}\ \dots\ t_{2,k})) \rangle\!\rangle \Leftrightarrow \forall_{i=1}^{k} \langle\!\langle \sigma(t_{1,i}) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t_{2,i}) \rangle\!\rangle$$

  We can apply the induction hypothesis to the pairs $(t_{1,i}, t_{2,i})$ and for ST yielding true because of case (5) we get:

  $$\begin{aligned} ST((c\ t_{1,1}\ \dots\ t_{1,k}), (c\ t_{2,1}\ \dots\ t_{2,k})) = \texttt{true} \Rightarrow \forall_{i=1}^{k} ST(t_{1,i}, t_{2,i}) = \texttt{true} \Rightarrow \\ \forall_{i=1}^{k} \langle\!\langle \sigma(t_{1,i}) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t_{2,i}) \rangle\!\rangle \Leftrightarrow \langle\!\langle \sigma((c\ t_{1,1}\ \dots\ t_{1,k})) \rangle\!\rangle \subseteq \langle\!\langle \sigma((c\ t_{2,1}\ \dots\ t_{2,k})) \rangle\!\rangle\,. \end{aligned}$$

- A frame type f_1 can just be a subtype of another frame type f_2 if the sets of symbols defined in f_1 and f_2 are equal. When $ST(f_1, f_2)$ yields true because of case (6) then for every symbol $s_i = s'_j$ the following holds:

  $$ST(t_i, t'_j) \Rightarrow \langle\!\langle \sigma(t_i) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t'_j) \rangle\!\rangle\,.$$

  By the definition of $\langle\!\langle \cdot \rangle\!\rangle$ on frames this implies $\langle\!\langle \sigma(f_1) \rangle\!\rangle \subseteq \langle\!\langle \sigma(f_2) \rangle\!\rangle$.

- If $ST(e_1, e_2)$ yields true because of case (7) then e_1 and e_2 must be environments $e_1 = (\mathit{env}\ e'_1\ f_1)$ and $e_2 = (\mathit{env}\ e'_2\ f_2)$ with $ST(e'_1, e'_2)$ and $ST(f_1, f_2)$. From the induction hypothesis we can conclude $\langle\!\langle \sigma(e'_1) \rangle\!\rangle \subseteq \langle\!\langle \sigma(e'_2) \rangle\!\rangle$ and $\langle\!\langle \sigma(f_1) \rangle\!\rangle \subseteq \langle\!\langle \sigma(f_2) \rangle\!\rangle$. Together with the definition of $\langle\!\langle \cdot \rangle\!\rangle$ on environments this implies $\langle\!\langle \sigma(e_1) \rangle\!\rangle \subseteq \langle\!\langle \sigma(e_2) \rangle\!\rangle$.

[1]More exactly because of Cor. 5.3.15 not needing Assumption 3.4.5.

- For the set constructors decomposed in case (9) the first part of the proof does not take into account interactions between top level set constructors of t_1 and t_2:
  - If $t_1 = (\cup\ t_{1,1}\ \dots\ t_{1,k})$ and case (9a) yields true then
$$\begin{aligned}\forall_{i\in\{1,\dots,k\}} ST(t_{1,i}, t_2) = \texttt{true} \Rightarrow \forall_{i\in\{1,\dots,k\}} \langle\!\langle \sigma(t_{1,i}) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t_2) \rangle\!\rangle \\ \Rightarrow \langle\!\langle \sigma(t_1) \rangle\!\rangle = (\cup\ \langle\!\langle \sigma(t_{1,1}) \rangle\!\rangle\ \dots\ \langle\!\langle \sigma(t_{1,k}) \rangle\!\rangle) \subseteq \langle\!\langle \sigma(t_2) \rangle\!\rangle\,.\end{aligned}$$
    Therefore, case (9a) is correct.
  - If $t_2 = (\cup\ t_{2,1}\ \dots\ t_{2,k})$ and case (9b) returns true then
$$\begin{aligned}\exists_{i\in\{1,\dots,k\}} ST(t_1, t_{2,i}) = \texttt{true} \Rightarrow \exists_{i\in\{1,\dots,k\}} \langle\!\langle \sigma(t_1) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t_{2,i}) \rangle\!\rangle \\ \Rightarrow \langle\!\langle \sigma(t_1) \rangle\!\rangle \subseteq (\cup\ \langle\!\langle \sigma(t_{2,1}) \rangle\!\rangle\ \dots\ \langle\!\langle \sigma(t_{2,k}) \rangle\!\rangle) = \langle\!\langle \sigma(t_2) \rangle\!\rangle\end{aligned}$$
    implies the correctness of case (9b).
  - If $t_1 = (\cap\ t_{1,1}\ \dots\ t_{1,k})$ and ST returns true because of case (9c) then
$$\begin{aligned}\exists_{i\in\{1,\dots,k\}} ST(t_{1,i}, t_2) = \texttt{true} \Rightarrow \exists_{i\in\{1,\dots,k\}} \langle\!\langle \sigma(t_{1,i}) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t_2) \rangle\!\rangle \\ \Rightarrow \langle\!\langle \sigma(t_1) \rangle\!\rangle = (\cap\ \langle\!\langle \sigma(t_{1,1}) \rangle\!\rangle\ \dots\ \langle\!\langle \sigma(t_{1,k}) \rangle\!\rangle) \subseteq \langle\!\langle \sigma(t_2) \rangle\!\rangle\,.\end{aligned}$$
    This implies the correctness of case (9c).
  - If $t_2 = (\cap\ t_{2,1}\ \dots\ t_{2,k})$ and case (9d) yields the result true then
$$\begin{aligned}\forall_{i\in\{1,\dots,k\}} ST(t_1, t_{2,i}) = \texttt{true} \Rightarrow \forall_{i\in\{1,\dots,k\}} \langle\!\langle \sigma(t_1) \rangle\!\rangle \subseteq \langle\!\langle \sigma(t_{2,i}) \rangle\!\rangle \\ \Rightarrow \langle\!\langle \sigma(t_1) \rangle\!\rangle \subseteq (\cap\ \langle\!\langle \sigma(t_{2,1}) \rangle\!\rangle\ \dots\ \langle\!\langle \sigma(t_{2,k}) \rangle\!\rangle) = \langle\!\langle \sigma(t_2) \rangle\!\rangle\,.\end{aligned}$$
    Thus, case (9d) is correct.
  - If $ST(t_1, t_2) = \texttt{true}$ because of case (9e) then $t_1 = \mathcal{C}t'_1$, $t_2 = \mathcal{C}t'_2$ and
$$ST(t'_2, t'_1) = \texttt{true} \Rightarrow \langle\!\langle t'_2 \rangle\!\rangle \subseteq \langle\!\langle t'_1 \rangle\!\rangle \Rightarrow \langle\!\langle t_1 \rangle\!\rangle = \mathcal{V} \setminus \langle\!\langle t'_1 \rangle\!\rangle \subseteq \mathcal{V} \setminus \langle\!\langle t'_2 \rangle\!\rangle = \langle\!\langle t_2 \rangle\!\rangle\,.$$
    This implies the correctness of case (9e). (Note that the argument of $\mathcal{C}$ must not contain variables and therefore a substitution σ is not necessary in this case.)
  - If $ST(t_1, t_2) = \texttt{true}$ because of case (9f) then $t_2 = \mathcal{C}t'_2$ and $CE(\tilde{t}_1, t'_2) = \texttt{false}$ with $\tilde{t}_1$ generated from t_1 by replacing every quantified variable $X_\forall \in V_q$ by $\top$. By Assumption 3.4.6 and by the fact that all type constructors allowing variables in their arguments are monotonic and therefore $\langle\!\langle \sigma(t_1) \rangle\!\rangle \subseteq \langle\!\langle \tilde{t}_1 \rangle\!\rangle$ this implies:
$$\begin{aligned}&CE(\tilde{t}_1, t'_2) = \texttt{false} \Rightarrow \\ &\qquad \Rightarrow \emptyset = \langle\!\langle \tilde{t}_1 \rangle\!\rangle \cap \langle\!\langle t'_2 \rangle\!\rangle \supseteq \langle\!\langle \sigma(t_1) \rangle\!\rangle \cap \langle\!\langle t_2 \rangle\!\rangle \Rightarrow \\ &\qquad\qquad \Rightarrow \langle\!\langle \sigma(t_1) \rangle\!\rangle \subseteq \mathcal{V} \setminus \langle\!\langle t'_2 \rangle\!\rangle = \langle\!\langle t_2 \rangle\!\rangle\,.\end{aligned}$$
    Therefore, case (9f) is correct.

The only cases where two of the subcases of case (9) yield different results depending on the order in which the types t_1 and t_2 are processed are:

1. $t_1 = (\cup\ t_{1,1}\ \dots\ t_{1,k})$ and $t_2 = (\cup\ t_{2,1}\ \dots\ t_{2,k'})$
2. $t_1 = (\cap\ t_{1,1}\ \dots\ t_{1,k})$ and $t_2 = (\cap\ t_{2,1}\ \dots\ t_{2,k'})$

because $\vee$ and $\wedge$ do not commute with each other. In case 2 the given order yields

$$\bigvee_{i=1}^{k}\bigwedge_{j=1}^{k'} ST(t_{1,i}, t_{2,j})$$

which implies

$$\bigwedge_{j=1}^{k'}\bigvee_{i=1}^{k} ST(t_{1,i}, t_{2,j})\,.$$

The given order already yields the more special check which implies:

$$\bigvee_{i=1}^{k}\bigwedge_{j=1}^{k'} ST(t_{1,i}, t_{2,j}) \Rightarrow \bigvee_{i=1}^{k}\bigwedge_{j=1}^{k'} \langle\!|t_{1,i}|\!\rangle \subseteq \langle\!|t_{2,j}|\!\rangle \Rightarrow \bigvee_{i=1}^{k} \langle\!|t_{1,i}|\!\rangle \subseteq \langle\!|t_2|\!\rangle \Rightarrow \langle\!|t_1|\!\rangle \subseteq \langle\!|t_2|\!\rangle$$

The lemma is not violated by the order in which case 2 is processed.
In case 1 the term representing the performed checks is:

$$\begin{aligned}\bigwedge_{i=1}^{k}\bigvee_{j=1}^{k'} ST(t_{1,i}, t_{2,j}) &\Rightarrow \forall_{i\in\{1,\dots,k\}}\exists_{j\in\{1,\dots,k'\}} \langle\!|t_{1,i}|\!\rangle \subseteq \langle\!|t_{2,j}|\!\rangle \\ &\Rightarrow \forall_{i\in\{1,\dots,k\}} \langle\!|t_{1,i}|\!\rangle \subseteq (\cup\ \langle\!|t_{2,1}|\!\rangle\ \dots\ \langle\!|t_{2,k'}|\!\rangle) \\ &\Rightarrow (\cup\ \langle\!|t_{1,1}|\!\rangle\ \dots\ \langle\!|t_{1,k}|\!\rangle) \subseteq (\cup\ \langle\!|t_{2,1}|\!\rangle\ \dots\ \langle\!|t_{2,k'}|\!\rangle)\end{aligned}$$

- Whenever the top level type constructor of one of the types is the μ constructor case (8) is applied:
  - In the cases (8b) and (8c) the first or second argument is unfolded one step respectively. It is replaced by an equivalent term and the induction hypothesis provides that for the changed term the right answer is generated.
    Since the unfolding is done before unions or intersections are decomposed in case (9) the decomposition order needed there cannot be violated by a set constructor hidden by μ.

– For correctness of case (8a) we must prove that we can assume the subtype property of two types to hold when processing a recursive call with arguments that have been processed before.

Consider the tree given by the calling behaviour of ST as follows:

  * The initial call to ST is represented by the root.
  * Every son of a given node represents a recursive subcall to ST evaluated by the current call.
  * Every leave of the tree represents a boolean value `true` or `false`.

This tree can just be infinite when both initial arguments contain the μ-constructor. Now let t_1 and t_2 be the initial arguments and t_1' and t_2' a pair of arguments of recursive calls that occur an infinite number of times on an infinite path. (We are just interested in infinite paths: Whenever an argument pair (s_1, s_2) occurs a second time on a path the existence of an infinite path is implied by repeating the subpath from the first occurrence of (s_1, s_2) to the second an infinite number of times.) Let recursive calls to t_1' and t_2' occur at depth $d_0, d_1, \ldots$ represented by nodes $n_0, n_1, \ldots$ and let n be a leave representing the value `false` with depth d fulfilling $d_k < d \leq d_{k+1}$ where without loss of generality the node n_k is the last node with a pair of arguments already used before. (If there were nodes with already analyzed argument pairs after n_k then we can switch to the last of these pairs and the corresponding recursive subterm. Since the leave n is fixed such a last node must exist.) Let $c := d - d_k$. Since the recursive calls represented by the nodes n_0 and n_k cannot be distinguished by ST (except by a history of recursive calls) there are equal paths yielding from n_k to n and from n_0 to a leave n' representing `false` with only one call with the argument pair t_1' and t_2' on the path from the root to n' (cf. Fig. A.1).

By an inductive argument on the number of pairs t_1' and t_2' occurring an infinite number of times in the tree we can show that whenever the tree contains a leave representing `false` there is such a leave $\tilde{n}$ with no pair of arguments occurring more than once on the path from the root to $\tilde{n}$. Thus, returning `true` for argument pairs already used before in an recursive call does not violate the lemma and hence case (8a) is correct.

□

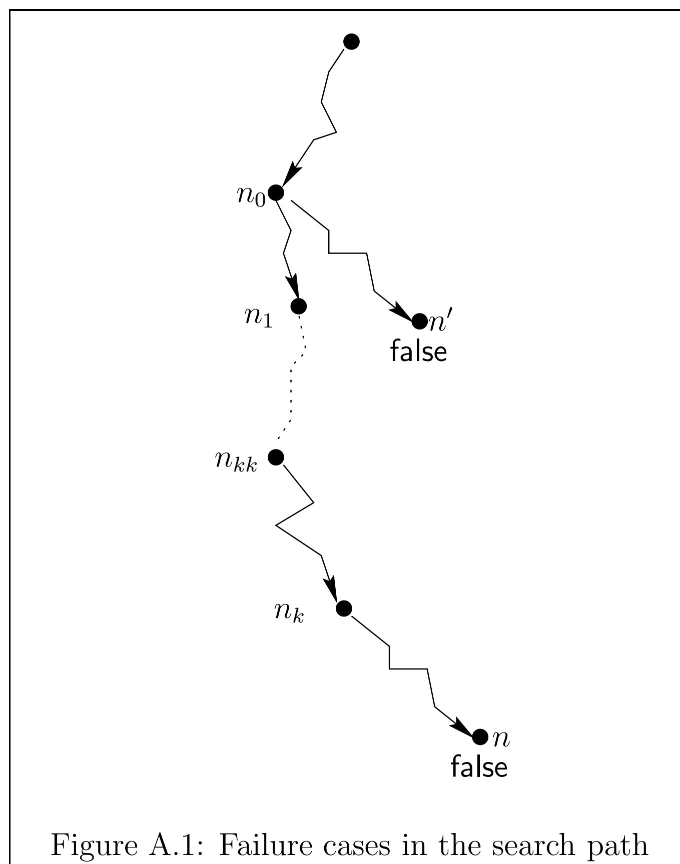

Figure A.1: Failure cases in the search path

Appendix B

Proofs of Chapter 5

B.1 Proofs of Section 5.2

Lemma 5.2.16 *If the algorithm ST terminates for every pair of closed terms in set normalized form then the algorithm S-CE terminates for every input with arguments 1 and 2 in set normalized form.*

Proof of Lemma 5.2.16: First we show that there is no infinite chain of calls to *S-CE*.

Obviously the rules (*Top1*), (*Top2*), (*RecT*), (*BothVar*), (*Var1*), (*Var2*), (*FunQ*) and (*Base*) of *S-CE* terminate directly, i.e. they do not perform a call to any other function. This is also the case when no rule applies and *S-CE* returns the empty c-collection.

The rules (*Comp1*) and (*Comp2*) also just return the given constraint set without any changes. But they perform calls to *ST* that can lead to recursive calls to *S-CE*. But since *ST* is stated terminating in the precondition these recursive calls to *S-CE* terminate, too.

The rules (*U1*), (*U2*), (*I1*), (*I2*), (*Constr*), (*Frame*) and (*Env*) perform recursive calls to *S-CE* with the following property: When t_1 and t_2 are the first two arguments then the first two arguments t'_1 and t'_2 of the direct subcalls to *S-CE* or the calls to *S-CE* performed by *SCE-list-reduce* fulfill:

- t'_1 is a subterm of t_1 and t'_2 is a subterm of t_2.
- At least one of the subterms stated above is a proper subterm.

By defining an ordering on the pairs of types with $(t'_1, t'_2) <_{TP} (t_1, t_2)$ if the two properties

stated above hold we get a termination ordering for *S-CE* for types not containing a recursive type constructor because in the ordering defined above there are no infinite descending chains.

In the presence of recursive type the ordering given above is no longer sufficient to show termination of *S-CE* because the rules (*Rec1*) and (*Rec2*) perform recursive calls to *S-CE* with subtypes as arguments that are no proper subtypes.
Informally termination is given because (*Rec1*) and (*Rec2*) never process the same pair of types twice in a recursive chain. Since the number of different type pairs occurring in such a recursive chain is finite there is just a finite number of invocations of (*Rec1*) and (*Rec2*). Furthermore there is always just a finite number of intermediate calls to *S-CE* between two invocations of (*Rec1*) and (*Rec2*) and thus the whole execution sequence is finite.

Formally we extend the ordering given above by a counter of possible invocations of (*Rec1*) and (*Rec2*). For two types t_1 and t_2 we define

$$\textit{sub-pairs}(t_1, t_2) = \{(t_1', t_2') \mid t_1' \text{ subterm of } t_1 \wedge t_2' \text{ subterm of } t_2\}\,.$$

$\textit{sub-pairs}(t_1, t_2)$ is finite because the set of subterms is finite for every type term t. Furthermore if t_1' is a subterm of t_1 and t_2' is a subterm of t_2 then $\textit{sub-pairs}(t_1', t_2') \subseteq \textit{sub-pairs}(t_1, t_2)$. This is the case because even if t_1 is a subterm of t_1' again, the set of subterms does not increase from t_1 to t_1' as stated in Remark 3.1.6 (analogously for t_2).

We now define a termination ordering for *S-CE* considering the two types t_1 and t_2 and the recursion information r. (We understand r as a *set* here.) $(t_1', t_2', r') < (t_1, t_2, r)$ if one of the following properties holds:

- $\textit{sub-pairs}(t_1', t_2') \setminus r' \subset \textit{sub-pairs}(t_1, t_2) \setminus r$.
- $\textit{sub-pairs}(t_1', t_2') \setminus r' = \textit{sub-pairs}(t_1, t_2) \setminus r \wedge (t_1', t_2') <_{TP} (t_1, t_2)$.

Since $<_{TP}$ does not contain infinite descending chains we can conclude that in every descending chain $\textit{sub-pairs}(t_1, t_2)$ must be smaller or r must be larger after a finite number of steps. Furthermore, $\textit{sub-pairs}(t_1, t_2)$ and therefore $\textit{sub-pairs}(t_1, t_2) \setminus r$ are finite and therefore the first condition can hold just a finite number of times in a descending chain.

For the rules (*U1*), (*U2*), (*I1*), (*I2*), (*Constr*), (*Frame*) and (*Env*) it is obvious that every recursive call performed in these rules has a smaller argument because r remains unchanged and either $(t_1', t_2') <_{TP} (t_1, t_2)$ or $\textit{sub-pairs}(t_1, t_2)$ already decreases because of one of the subterm sets becoming smaller.

If (*Rec1*) or (*Rec2*) is applied to t_1, t_2 and r a recursive call is performed with arguments (without loss of generality for (*Rec1*)) $t_1' = \textit{unfold}(t_1)$, $t_2' = t_2$ and $r' = ((t_1, t_2)\ .\ r)$. This call

fulfills

$$(t_1, t_2) \in \textit{sub-pairs}(t_1, t_2) \setminus r \supset \textit{sub-pairs}(t'_1, t'_2) \setminus r' \not\ni (t_1, t_2)$$

This is the case because *sub-pairs*$(t_1, t_2) \supseteq$ *sub-pairs*(t'_1, t'_2) and $r \subset r'$. Altogether $<$ is a termination ordering for *S-CE*.

The last reason of possible non-termination is the splitting of a call to *S-CE* into several subcalls done by the rules (*U1*), (*U2*) directly and (*I1*), (*I2*), (*Constr*), (*Frame*) and (*Env*) indirectly via *SCE-list-reduce*.

For the rules (*U1*) and (*U2*) processing union types the subcalls are given by splitting the union type into its argument types. Since all union types have a finite number of argument types the number of these subcalls is also finite.

For the rules (*I1*), (*I2*), (*Constr*), (*Frame*) and (*Env*) the number k of calls to *SCE-list-reduce* is given by the number of subterms in the processed intersection type, tuple like type, frame type or environment type, respectively. All these types always have a finite number of arguments and hence $k \in \mathbb{N}$. (Exactly there is always one more call to *SCE-list-reduce* with the empty list of type pairs that is processed by rule (*SCE-list-reduce1*), but the number of calls is still finite.)
We still have to prove that every execution of rule (*SCE-list-reduce2*) just involves a finite number of calls to *S-CE*. For the ith recursive execution of (*SCE-list-reduce2*) calculating the intermediate c-collection Σ_i this number of calls is equal to the number of elements in the previous c-collection Σ_{i-1}. The only rules generating collections of more than one element are (*U1*) and (*U2*) processing union types. The number of elements in the result collection is bounded by the sum of element numbers of the intermediate collections. As explained before the number of intermediate collections is finite and the number of recursive calls to *S-CE* processed by (*U1*) or (*U2*) is finite. Thus, by induction on the maximal number of recursive calls to *S-CE* processed by (*U1*) or (*U2*) necessary to calculate one of the subcollections ensuring the termination of *SCE-list-reduce* we can prove that every result collection of *S-CE* has a finite arity. □

Lemma 5.2.19 *Let $\tilde{c}$ be a call to S-CE and c' the first call to SCE-list-reduce performed when processing $\tilde{c}$. Let Σ be the result of a call*

$$c' \doteq \textit{SCE-list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)$$

and let C_j denote the set of recursive calls to S-CE when processing the jth list element. Let Σ_j be the c-collection used in the recursive call

$$\textit{SCE-list-reduce}(((t_{j+1}, t'_{j+1}), \ldots, (t_n, t'_n)), \Sigma_j, r)$$

for all j and $\Sigma'_j = S\text{-}CE(t_j, t'_j, \emptyset, r)$ *then:*

$$\forall_{l=0}^{n} \mathit{combine\text{-}cs}(\Sigma) = \mathit{combine\text{-}cs}(\Sigma_l \otimes \bigcup_{j=l}^{k} \Sigma'_j) = \mathit{combine\text{-}cs}(\Sigma_l \otimes \mathit{implicit\text{-}cs}_{\tilde{c}}(C_l))\,.$$

where $\Sigma_0 = \Sigma'$.

Proof of Lemma 5.2.19: We first prove the following statement (B.1) used in the rest of the proof:

$$\bigcup_{\sigma \in \Sigma} S\text{-}CE(t_1, t_2, \sigma, r) = \Sigma \otimes S\text{-}CE(t_1, t_2, \emptyset, r) \tag{B.1}$$

In proving (B.1) we first rule out several special cases:

- $\Sigma = \emptyset$. Then both sides trivially yield the empty c-collection $\emptyset$.
- $S\text{-}CE(t_1, t_2, \emptyset, r) = \emptyset$ (i.e. no common elements of t_1 and t_2 detected). Since the applicability of none of the rules in *S-CE* depends on the given free constraint set

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

  for every $\sigma \in \Sigma$. Therefore,

  $$\bigcup_{\sigma \in \Sigma} S\text{-}CE(t_1, t_2, \sigma, r) = \emptyset\,.$$

Now we consider the case that $\Sigma = \{\sigma\}$ and $S\text{-}CE(t_1, t_2, \emptyset, r)$ returns with a c-collection consisting of exactly one element σ', i.e $S\text{-}CE(t_1, t_2, \emptyset, r) = \{\sigma'\}$ holds. We have to show that

$$S\text{-}CE(t_1, t_2, \sigma, r) = \{\sigma \otimes \sigma'\}\,. \tag{B.2}$$

Obviously, both sides of (B.2) can just differ for those variables being constrained by an element of $S\text{-}CE(t_1, t_2, \sigma, r)$ or by σ'.

The behaviour of the individual rules (applicability, arguments 1,2 and 4 of recursive subcalls) does not depend on the provided free constraint set. Therefore, we just have to consider the rules (*BothVar*), (*Var1*), (*Var2*), (*Comp1*) and (*Comp2*). All other rules either just return an unchanged free constraint set or pass through the result of the subcalls. (For those rules calling *SCE-list-reduce* induction on the number of subcalls will give (B.2).)

- (*BothVar*) constrains the two variables t_1 and t_2 to a new variable. If t_i (with $i \in \{1, 2\}$) is not constrained in σ then the new constraint is obviously equal at both sides of (B.2). If $\sigma(t_i) = t$ then t and the new variable are united in the constraint of t_i by *extend-constraint* on the left hand side and by $\otimes$ on the right hand side of (B.2).

- (*Var1*) and (*Var2*) are proven analogously (*BothVar*). For the variable constrained by *extend-constraint* the argumentation is completely analogous. For those variables constrained by *constrain-all-free* we also can use an analogous argument because *constrain-all-free* constrains all variables in terms of *extend-constraint*.
- (*Comp1*) and (*Comp2*) are the only places where constraints on variable are overwritten by *free-to-top*. Since all these constraints are overwritten with $\top$ and the union of every type t with $\top$ is equal to $\top$ both sides of (B.2) are equivalent.

Now let $\Sigma = \{\sigma_1, \dots, \sigma_k\}$ and let $\sigma'_k = \sigma_k \otimes \sigma'$. Analogously to (B.2) we get

$$\forall_{l=1}^{k} \textit{S-CE}(t_1, t_2, \sigma_l, r) = \{\sigma_l \otimes \sigma'\} = \sigma'_l \,.$$

Obviously both sides of (B.1) yield exactly the set $\{\sigma'_1, \dots \sigma'_k\}$ and are therefore equal.

If $\textit{S-CE}(t_1, t_2, \emptyset, r)$ is processed using the rules (*U1*) and (*U2*) the result usually consists of several free constraint sets. We can consider each of these free constraint sets independently as follows:
Let $\sigma_i \in \textit{S-CE}(t_1, t_2, \emptyset, r)$ be an arbitrarily chosen free constraint set. For each union type that yields several free constraint sets when processed we can choose one union element that was involved in calculating σ_i. Replacing the union by this union element and repeating this for all union types yielding several free constraint sets as result we get a type pair t'_1 and t'_2 with

$$\textit{S-CE}(t'_1, t'_2, \emptyset, r) = \{\sigma_i\} \,.$$

For these types t'_1 and t'_2 (B.2) holds. This is the case for all $\sigma_i \in \textit{S-CE}(t_1, t_2, \emptyset, r)$. Furthermore, the rules of *S-CE* show the same behaviour when a free constraint set $\sigma \neq \emptyset$ is used in calling *S-CE*, i.e. there is a direct correspondence between the σ_i and the results of $\sigma_i \in \textit{S-CE}(t_1, t_2, \sigma, r)$ for every σ via the common execution path of *S-CE*. Hence, (B.1) also holds for types t_1 and t_2 yielding an arbitrary number of free constraint sets as result from *S-CE*.

We can now prove the lemma. The situation after processing the first j list elements by *SCE-list-reduce* can be described as follows:

$$\begin{aligned}\textit{SCE-list-reduce}((t_{j+1}, t'_{j+1}), \dots, (t_n, t'_n)), \Sigma_j, r) = \\ = \textit{SCE-list-reduce}((\bar{t}_1, \bar{t}'_1), \dots, (\bar{t}_{n-j}, \bar{t}'_{n-j})), \bar{\Sigma}, r)\end{aligned}$$

Hereby the remaining calculations after processing j list elements in *SCE-list-reduce* can always be described as a not partially processed call to *SCE-list-reduce*. The statement of the

lemma now simplifies to:

$$\textit{combine-cs}(\textit{SCE-list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)) = \\ = \textit{combine-cs}(\Sigma' \otimes \Sigma' \otimes \bigotimes_{j=1}^{n} \textit{S-CE}(t_j, t'_j, \emptyset, r)) \,.$$

Because of

$$\begin{aligned}
&\textit{combine-cs}(\Sigma' \otimes \Sigma' \otimes \tilde{\Sigma}) \\
=&\textit{combine-cs}(\{\sigma_1 \otimes \sigma_2 \otimes \tilde{\sigma} \mid \sigma_1, \sigma_2 \in \Sigma', \tilde{\sigma} \in \tilde{\Sigma}\}) \\
=&\{\bigotimes_{\sigma_1 \in \bar{\Sigma}_1, \sigma_2 \in \bar{\Sigma}_2, \tilde{\sigma} \in \hat{\Sigma}} (\sigma_1 \otimes \sigma_2 \otimes \tilde{\sigma}) \mid \bar{\Sigma}_1, \bar{\Sigma}_2 \subseteq \Sigma' \wedge \hat{\Sigma} \subseteq \tilde{\Sigma} \wedge \bar{\Sigma}_1, \bar{\Sigma}_2, \hat{\Sigma} \neq \emptyset\} \\
=&\{\bigotimes_{\sigma_1 \in \bar{\Sigma}_1, \sigma_2 \in \bar{\Sigma}_2} (\sigma_1 \otimes \sigma_2) \otimes \bigotimes_{\tilde{\sigma} \in \hat{\Sigma}} \tilde{\sigma} \mid \bar{\Sigma}_1, \bar{\Sigma}_2 \subseteq \Sigma' \wedge \hat{\Sigma} \subseteq \tilde{\Sigma} \wedge \bar{\Sigma}_1, \bar{\Sigma}_2, \hat{\Sigma} \neq \emptyset\} \\
=&\{\bigotimes_{\sigma \in \bar{\Sigma}} \sigma \otimes \bigotimes_{\tilde{\sigma} \in \hat{\Sigma}} \tilde{\sigma} \mid \bar{\Sigma} \subseteq \Sigma' \wedge \hat{\Sigma} \subseteq \tilde{\Sigma} \wedge \bar{\Sigma}, \hat{\Sigma} \neq \emptyset\} \\
=&\{\bigotimes_{\sigma \in \bar{\Sigma}, \tilde{\sigma} \in \hat{\Sigma}} (\sigma \otimes \tilde{\sigma}) \mid \bar{\Sigma} \subseteq \Sigma' \wedge \hat{\Sigma} \subseteq \tilde{\Sigma} \wedge \bar{\Sigma}, \hat{\Sigma} \neq \emptyset\} \\
=&\textit{combine-cs}(\{(\sigma \otimes \tilde{\sigma}) \mid \sigma \in \Sigma', \tilde{\sigma} \in \tilde{\Sigma}\}) \\
=&\textit{combine-cs}(\Sigma' \otimes \tilde{\Sigma})
\end{aligned}$$

for an arbitrary c-collection $\tilde{\Sigma}$ the statement above further simplifies to

$$\textit{combine-cs}(\textit{SCE-list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)) = \\ = \textit{combine-cs}(\Sigma' \otimes \bigotimes_{j=1}^{n} \textit{S-CE}(t_j, t'_j, \emptyset, r)) \,.$$

We prove this by showing

$$\textit{SCE-list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r) = \Sigma' \otimes \bigotimes_{j=1}^{n} \textit{S-CE}(t_j, t'_j, \emptyset, r) \,. \tag{B.3}$$

by induction on the number n of list elements:

n = 1:

$$\textit{SCE-list-reduce}((t_1, t'_1)), \Sigma', r) = \textit{SCE-list-reduce}((), \bigcup_{\sigma \in \Sigma'} \textit{S-CE}(t_1, t'_1, \sigma, r), r) = \\ = \bigcup_{\sigma \in \Sigma'} \textit{S-CE}(t_1, t'_1, \sigma, r) = \Sigma' \otimes \textit{S-CE}(t_1, t'_1, \emptyset, r)$$

with the last step because of (B.1) proven above.

$\mathbf{n} \to \mathbf{n+1}$: Let (B.3) hold for all calls to *SCE-list-reduce* with a list of length n as first argument. Consider a call

$$\begin{aligned}&\mathit{SCE\text{-}list\text{-}reduce}((t_1, t'_1), (\bar{t}_1, \bar{t}'_1), \ldots, (\bar{t}_1, \bar{t}'_1)), \Sigma, r) = \\ &\qquad = \mathit{SCE\text{-}list\text{-}reduce}(((\bar{t}_1, \bar{t}'_1), \ldots, (\bar{t}_1, \bar{t}'_1)), \bigcup_{\sigma \in \Sigma} \mathit{S\text{-}CE}(t_1, t'_1, \sigma, r), r)\end{aligned}$$

By applying the induction hypothesis this is equal to

$$\bigcup_{\sigma \in \Sigma} \mathit{S\text{-}CE}(t_1, t'_1, \sigma, r) \otimes \bigotimes_{j=1}^{n} \mathit{S\text{-}CE}(\bar{t}_j, \bar{t}'_j, \emptyset, r)$$

Applying (B.1) we get

$$\Sigma \otimes \mathit{S\text{-}CE}(t_1, t'_1, \emptyset, r) \otimes \bigotimes_{j=1}^{n} \mathit{S\text{-}CE}(\bar{t}_j, \bar{t}'_j, \emptyset, r)$$

After a renaming of the arguments with $(t_{i+1}, t'_{i+1}) = (\bar{t}_i, \bar{t}'_i)$ for all $i \in \{1, \ldots, n\}$ we get (B.3) for argument lists of length $n+1$.

Altogether, the lemma is proven. □

Lemma 5.2.20 *Let $t_1, t_2 \in \mathcal{T}$, both in set normalized form and let $c := \mathit{S\text{-}CE}(t_1, t_2, \tilde{\sigma}, r)$ be a call to S-CE that occurs as a recursive call after an initial call $\tilde{c}$ to S-CE with empty recursion information (). Let CE fulfill Assumption 3.4.6. Let there exist a value $v \neq \bot$ such that*

$$\forall k \in \mathbb{N} \,.\, v \in \mathit{subst}(\tilde{\sigma})^k(t_1) \boxdot \mathit{subst}(\tilde{\sigma})^k(t_2)\,. \tag{5.2}$$

Then there exist a free constraint set

$$\sigma \in \mathit{combine\text{-}cs\text{-}cond}(\mathit{S\text{-}CE}(t_1, t_2, \tilde{\sigma}, r) \otimes \mathit{implicit\text{-}cs}_{\tilde{c}}(c))$$

such that the free type substitution $\sigma' = \mathit{subst}(\sigma)$ compatible with σ fulfills

$$\forall k \in \mathbb{N} \,.\, v \in {\sigma'}^k(t_1) \boxdot {\sigma'}^k(t_2)\,. \tag{5.3}$$

Furthermore, if X is a free variable with $X \in \mathit{dom}(\tilde{\sigma})$ and $\tilde{\sigma}(X) = \tilde{t}_X$ then $X \in \mathit{dom}(\sigma)$ for every $\sigma \in \mathit{S\text{-}CE}(t_1, t_2, \tilde{\sigma}, r)$ and $t_X = \sigma(X)$ fulfills:

$$\langle\!\langle \tilde{t}_X \rangle\!\rangle(\tau) \subseteq \langle\!\langle t_X \rangle\!\rangle(\tau) \tag{5.4}$$

for every closed type substitution τ appropriate for $\tilde{t}_X$ and t_X.

Proof of Lemma 5.2.20: We begin the proof by (mostly) eliminating the use of *combine-cs-cond* in the lemma by the following observation: Since *combine-cs* processes all non-empty subsets of the given c-collection especially the subsets consisting of exactly one element are processed. Because of this, all free constraint sets occurring in the argument of *combine-cs* are also in the result. Therefore, we have

$$\Sigma \subseteq \mathit{combine\text{-}cs\text{-}cond}(\Sigma) \subseteq \mathit{combine\text{-}cs}(\Sigma)$$

for every c-collection Σ. In the following we will use Σ instead of *combine-cs-cond*(Σ) wherever possible. For situations where the application of *combine-cs* is necessary we will show that *combine-cs-cond* indeed behaves like *combine-cs*.

Now consider the cases that can violate the lemma. The lemma is trivially fulfilled for terms t_1 and t_2 not containing any variables. If t_1 or t_2 contains a variable there must be a value $v \in \langle\!\langle \rho(t_1) \rangle\!\rangle \cap \langle\!\langle \rho(t_2) \rangle\!\rangle$ for some ρ such that no $\sigma \in \mathit{S\text{-}CE}(t_1, t_2, \tilde{\sigma}, r)$ fulfills

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

By definition of the type semantics the reason for the failure can be found at those positions p with

- $(t_1)_{|p}$ or $(t_2)_{|p}$ is a variable A (without loss of generality let us assume that $(t_1)_{|p} = A$).
- $(\sigma')^k(A)$ for some k cannot be instantiated in order to contain $v_{|p}$ (i.e. for all τ we have $v_{|p} \notin \langle\!\langle \tau \circ {\sigma'}^k(A) \rangle\!\rangle$).

Informally A is instantiated by σ' in a way that rules out $v_{|p}$ as a possible value.

We can therefore prove the lemma by showing that:

- All constraints generated by *S-CE* do not yield the situation above for the type pair (t_1, t_2) currently considered.
- Constraints already generated for variables occurring in other positions in the types of the initial call to *S-CE* (i.e. constraints already given in $\tilde{\sigma}$) are not changed in a way violating the lemma afterwards (i.e. the types these variables are constrained to are enlarged or remain unchanged).

In the first part of the proof we will show that all the rules generate an output fulfilling the lemma by case distinction on the executed rule. The proof is done by induction on the number of needed subcalls to *S-CE* to process a certain call. (This is possible because according to

Lemma 5.2.16 *S-CE* terminates and therefore the number of needed subcalls to *S-CE* is always finite.)

In the second part we will show that in the case of no applicable rule there are no common elements of $\tilde{\sigma}^k(t_1)$ and $\tilde{\sigma}^k(t_2)$ for some $k \in \mathbb{N}$ and therefore the lemma is satisfied by returning the empty c-collection.

Part 1: Correctness of the individual rules: We show the correctness of the rules by case distinction on the activated rule.

Whenever a $v \in \tilde{\sigma}^k(t_1) \boxdot \tilde{\sigma}^k(t_2)$ exists there are ρ_f and ρ_q such that

$$v \in \langle\!\langle \rho_q \circ \rho_f \circ \tilde{\sigma}^k(t_1) \rangle\!\rangle \cap \langle\!\langle \rho_q \circ \rho_f \circ \tilde{\sigma}^k(t_2) \rangle\!\rangle \,.$$

For every ρ'_q with $dom(\rho'_q) = dom(\rho_q)$ and $\rho'_q(X_\forall) \neq \bot$ for every $X_\forall \in dom(\rho'_q)$ there exists a v' with

$$v' \in \langle\!\langle \rho'_q \circ \rho_f \circ \tilde{\sigma}^k(t_1) \rangle\!\rangle \cap \langle\!\langle \rho'_q \circ \rho_f \circ \tilde{\sigma}^k(t_2) \rangle\!\rangle \,.$$

We can choose a v' that differs from v at exactly those positions where either $\tilde{\sigma}^k(t_1)$ or $\tilde{\sigma}^k(t_2)$ contains a quantified variable. Since ${\sigma'}^k(t_1)$ and ${\sigma'}^k(t_2)$ do not contain additional quantified variables and all quantified variables occurring in $\tilde{\sigma}^k(t_1)$ and $\tilde{\sigma}^k(t_2)$ also occur in ${\sigma'}^k(t_1)$ and ${\sigma'}^k(t_2)$ (maybe as element of a union) or are replaced by $\top$ we can choose τ'_q such that

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

whenever

$$v \in \langle\!\langle \tau_q \circ \tau_f \circ {\sigma'}^k(t_1) \rangle\!\rangle \cap \langle\!\langle \tau_q \circ \tau_f \circ {\sigma'}^k(t_2) \rangle\!\rangle \,. \tag{B.4}$$

Thus, in the following we will just prove (B.4) for the individual rules.

The substitutions used for $\boxdot$ are denoted ρ_q for the quantified and ρ_f for the free variables in (5.2) and analogously τ_q and τ_f in (5.3).

- Let t_1, t_2 and $\tilde{\sigma}$ be processed by (*Top1*).[1] Let v be chosen as in (5.2). Because of $t_1 = \top$ from the condition of (*Top1*) $v \in \langle\!\langle \tau \circ {\sigma'}^k(t_1) \rangle\!\rangle$ for all τ and σ'. Since $\sigma' = \tilde{\sigma}$ we can choose $\tau = \rho$ according to the choice of v and k and trivially get statement (5.3) of the lemma. (5.4) holds trivially because of $\sigma' = \tilde{\sigma}$. For (*Top2*) the proof is done analogously.

[1]In order to simplify notations we identify $subst(\tilde{\sigma})$ and $\tilde{\sigma}$ in the following.

- If t_1, t_2 and $\tilde{\sigma}$ are processed by rule (*FunQ*) (i.e. t_1 and t_2 are function types or quantified variables) or (*Base*) (i.e. t_1 and t_2 are base types or value assignments) then (5.3) is fulfilled because t_1 and t_2 are not affected by σ' and $\tilde{\sigma}$ and we can choose $\tau = \rho$. (Since σ' and $\tilde{\sigma}$ are generated from *free* constraint sets they just affect *free* variables.) (5.4) again holds trivially because of $\sigma' = \tilde{\sigma}$.

- Let the current call be processed by rule (*BothVar*). Then both types t_1 and t_2 are free variables. These variables are both constrained to contain the values of a newly introduced variable. As this variable does not contain as a subtype of any types to be checked it is not constrained by *S-CE* and we can choose τ to instantiate it according to the value v. This yields (5.3). (5.4) holds because of the definition of *extend-constraint*: Whenever one of the variables is already constrained the newly introduced variable and the previous value are united.

- Let (*Var1*) be the rule processing the current call to *S-CE*. Then t_1 is a free variable and t_2 is not a free variable. Because of $v \in \langle\!\langle \rho \circ \tilde{\sigma}^k(t_2) \rangle\!\rangle$ in (5.2) we have $v \in \langle\!\langle \bar{\rho} \circ \rho \circ {\sigma'}^k(t_2) \rangle\!\rangle$ with $\bar{\rho}$ instantiating the variables introduced by *constrain-all-free*. (All free variables occurring in t_2 are instantiated with a newly introduced variable (maybe in a union) by σ'. These variables are not instantiated by further calls of σ' because they do not occur as subtype of one of the checked types. When assuming that ρ was chosen not to instantiate unknown variables we can choose the instantiation of the new variables by $\bar{\rho}$ according to the value v.) Furthermore, we have $v \in \langle\!\langle \bar{\rho} \circ \rho \circ {\sigma'}^{k+1}(t_1) \rangle\!\rangle$ because of the change to the constraint of t_1 performed by *extend-constraint*. We can now choose $\tau_2 = \bar{\rho} \circ \rho$, $\tau_1 = \bar{\rho} \circ \rho \circ \sigma'$ and get $v \in \tau_2 \circ {\sigma'}^k(t_2)$ and $v \in \tau_1 \circ {\sigma'}^k(t_1)$. Since there are valid substitutions τ_1 and τ_2 in both cases and all type constructors that can contain variables in our type language are monotonic we can just choose $\tau = \{A \leftarrow \top \mid A \in dom(\tau_1) \cup dom(\tau_2)\}$. This yields (5.3). The preservation of previous constraints as stated by (5.4) is obvious here because:

  - If there are previous constraints for t_1 they are preserved because of the union calculated by *extend-constraint* in this case.
  - The free variables occurring in t_2 are constrained by *constrain-all-free*. This function preserves previous constraints by calculating unions in *extend-constraints* when necessary.

  The correctness of (*Var2*) is proven analogously.

- If the current call is processed by rule (*Comp1*) or (*Comp2*) then the resulting substitution σ' calculated by *free-to-top* differs from $\tilde{\sigma}$ in instantiating some additional variables. All these variables are instantiated to $\top$. By choosing $\tau = \rho$ we get $\langle\!\langle \rho \circ \tilde{\sigma}^k(A) \rangle\!\rangle \subseteq \langle\!\langle \tau \circ {\sigma'}^k(A) \rangle\!\rangle$ for every variable A and since all type constructors containing variables in their arguments are monotonic (5.3) of the lemma follows. (5.4) holds because for every type t and every substitution τ we have $\langle\!\langle \tau(t) \rangle\!\rangle \subseteq \langle\!\langle \top \rangle\!\rangle$.

- Let (*U1*) be the rule processing the current call. Then $t_1 = (\cup\ t_{1,1}\ \dots\ t_{i,k})$. Let

$$v \in \langle\!\langle \rho \circ \mathit{subst}(\tilde{\sigma})^k(t_1) \rangle\!\rangle \cap \langle\!\langle \rho \circ \mathit{subst}(\tilde{\sigma})^k(t_2) \rangle\!\rangle$$

for some arbitrary k and an appropriate ρ. Because of $v \in \langle\!\langle \rho \circ \mathit{subst}(\tilde{\sigma})^k(t_1) \rangle\!\rangle$ there must be a $t_{1,i}$ with $v \in \langle\!\langle \rho \circ \mathit{subst}(\tilde{\sigma})^k(t_{1,i}) \rangle\!\rangle$ and therefore

$$v \in \langle\!\langle \rho \circ \mathit{subst}(\tilde{\sigma})^k(t_{1,i}) \rangle\!\rangle \cap \langle\!\langle \rho \circ \mathit{subst}(\tilde{\sigma})^k(t_2) \rangle\!\rangle\,.$$

From the induction on the number of needed subcalls to *S-CE* necessary to process a given call we can use the induction hypothesis to show that there is a free constraint set $\sigma \in \mathit{S\text{-}CE}(t_{1,i}, t_2, \tilde{\sigma}, r)$ with $\sigma' = \mathit{subst}(\sigma)$ such that there exists a τ with

$$v \in \langle\!\langle \tau \circ {\sigma'}^k(t_{1,i}) \rangle\!\rangle \cap \langle\!\langle \tau \circ {\sigma'}^k(t_2) \rangle\!\rangle\,.$$

By rule (*U1*) this σ is also a member of the result of $\mathit{S\text{-}CE}(t_1, t_2, \tilde{\sigma}, r)$. (5.3) holds e.g. with τ chosen as above. (5.4) follows trivially from the induction stating (5.4) for the subcalls. The correctness of (*U2*) is proven analogously.

- Let the current call be processed by rule (*I1*), i.e. $t_1 = (\cap\ t_{1,1}\ \dots\ t_{1,n})$ and consider a value v chosen as in (5.2). *SCE-list-reduce* called by rule (*I1*) performs a sequence of subcalls to *S-CE* with the types $(t_{1,1}, t_2)$, $(t_{1,2}, t_2)$, $\dots$, $(t_{1,n}, t_2)$ as first two arguments and free constraint sets that steam from the output of the call before. We will show that the c-collection Σ_i that is used in the recursive subcall $\mathit{SCE\text{-}list\text{-}reduce}((l_{i+1}, \dots , l_n), \Sigma_i, r)$ with $l_j = (t_{1,j}, t_2)$ fulfills:

$$\begin{aligned} \exists \sigma \in \Sigma_i \forall k \in \mathbb{N}\, \exists \tau\,.\, v \in \langle\!\langle \tau \circ \sigma^k(t_{1,1}) \rangle\!\rangle \cap \langle\!\langle \tau \circ \sigma^k(t_2) \rangle\!\rangle \wedge \\ \wedge v \in \langle\!\langle \tau \circ \sigma^k(t_{1,2}) \rangle\!\rangle \cap \langle\!\langle \tau \circ \sigma^k(t_2) \rangle\!\rangle \wedge \\ \vdots \qquad\qquad\qquad\qquad \\ \wedge v \in \langle\!\langle \tau \circ \sigma^k(t_{1,i}) \rangle\!\rangle \cap \langle\!\langle \tau \circ \sigma^k(t_2) \rangle\!\rangle \end{aligned} \tag{B.5}$$

For the last conjunction element this follows directly from part (5.3) of the induction hypothesis applied to the direct *S-CE* calls performed by the call to *SCE-list-reduce* processing l_i.

For the other conjunction elements (making statements about l_j with $j < i$) we can apply (5.3) of the induction hypothesis to the calls calculating Σ_j (defined analogously to Σ_i). All the transformations from Σ_j to Σ_i preserve the needed properties according to (5.4) of the induction hypothesis.

Since the statement (B.5) especially holds for $i = n$ rule (*I1*) fulfills (5.3) because from $v \in \langle\!\langle \tau \circ \sigma^k(t_{1,i}) \rangle\!\rangle$ for all i we can conclude $v \in \langle\!\langle \tau \circ \sigma^k(t_1) \rangle\!\rangle$. (5.4) directly follows from the fact that all subcalls to *S-CE* performed by *SCE-list-reduce* fulfill (5.4) because of the induction hypothesis. The proof of (*I2*) can be done analogously.

- If (*Rec1*) is the rule processing the current call then $t_1 = \mu X.t'$ and the recursive call *S-CE*(*unfold*(t_1), $t_2, \tilde{\sigma}, ((t_1, t_2) \; . \; r)$) fulfills (5.3) and (5.4) by the induction hypothesis. Because of $⟅t_1⟆(\sigma) = ⟅unfold(t_1)⟆(\sigma)$ (see Def. 3.1.49) the lemma also holds for the result of (*Rec1*). For (*Rec2*) the proof is done analogously.
  If the current call is processed by rule (*RecT*) then $(t_1, t_2) \in r$, i.e. the type pair (t_1, t_2) must have been processed in a previous call to *S-CE* already, because r is generated by *S-CE* starting with (). During processing this previous call there are two possibilities for the necessary constraints:
  - They have already been generated.
  - Between the previous and the current call with the current parameters a call to *SCE-list-reduce* was processed and the necessary constraints will be generated during processing the rest of this call to *SCE-list-reduce*. In this case the necessary constraints are already contained in the implicit constraint set of the current call as stated by Lemma 5.2.19.

  There is just one problem left: When calling (*RecT*) the necessary constraint for a variable might be splitted into subconstraints occurring in different free constraint sets in the same c-collection. Since (*RecT*) can only occur as a (maybe indirect) subcall of a call to (*Rec1*) or (*Rec2*) the complete constraint is generated by *combine-cs-cond* that behaves like *combine-cs* in this context.[2]

- Let the current call be processed by rule (*Constr*). Then $t_1 = (c \; t_{1,1} \; \ldots \; t_{1,n})$ and $t_2 = (c \; t_{2,1} \; \ldots \; t_{2,n})$. Via *SCE-list-reduce* a sequence of calls with type pairs $(t_{1,i}, t_{2,i})$ is initiated. Analogously to the proof of (*I1*) we can show that the c-collection Σ_i used in the call *SCE-list-reduce*($(l_{i+1}, \ldots, l_n), \Sigma_i, r$) contains a σ with:

$$\begin{aligned} \forall k \in \mathbb{N} \exists \tau \; . \; & v_{|1} \in ⟅\tau \circ \sigma^k(t_{1,1})⟆ \cap ⟅\tau \circ \sigma^k(t_{2,1})⟆ \wedge \\ & \wedge v_{|2} \in ⟅\tau \circ \sigma^k(t_{1,2})⟆ \cap ⟅\tau \circ \sigma^k(t_{2,2})⟆ \wedge \\ & \vdots \\ & \wedge v_{|i} \in ⟅\tau \circ \sigma^k(t_{1,i})⟆ \cap ⟅\tau \circ \sigma^k(t_{2,i})⟆ \end{aligned}$$

  Since this is especially the case for $i = n$ rule (*Constr*) fulfills (5.3). (5.4) is proven analogously to rule (*I1*). The rules (*Frame*) and (*Env*) are proven correct analogously.

Part 2: No rule applicable: In this part we consider the case that none of the rules of *S-CE* is applicable. In this case the result is the empty c-collection violating (5.3). We have to show

[2] When finishing a call to Rule (*RecT*) the recursive parent calls must also apply *combine-cs* to their result up to that ancestor processed by Rule (*Rec1*) or (*Rec2*) that is finished first. Since all these calls are subcalls of (*Rec1*) or (*Rec2*) *combine-cs-cond* indeed behaves like *combine-cs*. Integrating this into the induction is straightforward. We omit this here for simplification of the presentation of the proof (by dropping case distinctions).

that in this case the precondition of the lemma given in (5.2) does not hold. Equivalently, whenever the precondition of the lemma holds there is an applicable rule in *S-CE* for the current argument pair. We do this by case distinction on the structure of the argument types that can be:

- A base type (including value assignments and $\bot$).
- $\top$
- A type constructed by a tuple like type constructor.
- A frame type.
- An environment type.
- A function type.
- A recursive type.
- A type variable (either free or quantified)
- A union type.
- An intersection type.
- A complement type.

The case distinction is done in the order of the cases in *S-CE*:

- If one of the types is $\top$ then one of the rules (*Top1*) and (*Top2*) is applicable.
- If one of the types is a recursive type then one of the rules (*RecT*), (*Rec1*) and (*Rec2*) applies.
- If one of the checked types is a *free* type variable X then the following cases are possible:
  - Both types are free variables. In this case rule (*BothVar*) is applicable.
  - The other type is not a free type variable: Either (*Var1*) or (*Var2*) applies.

  Thus, in all cases with at least one free variable one of the rules of *S-CE* is applicable.

  In the following we assume that none of the given types is $\top$ or a free type variable.

  If one of the types (without loss of generality t_1) is a *quantified* type variable $Y_\forall$ then the following cases are possible:

- t_2 is a recursive, union, intersection or complement type. In this case the rule corresponding to t_2 is applicable.
- $t_2 = Y_\forall$. Then rule (*FunQ*) applies.
- In every other case there is no rule applicable. But in this case t_2 is a base type $\neq \top$, a type constructed by one of the free type constructors, a frame type or an environment type and hence does not cover all values. Thus, there exists a ground type $\tilde{t}$ having no common elements with any instance of t_2. We can conclude that e.g. for ρ'_q containing $Y_\forall \leftarrow \tilde{t}$ the precondition of the lemma does not hold.

- If one of the types is a union type one of the rules (*U1*) and (*U2*) applies. Analogously, intersection types are covered by either (*I1*) or (*I2*).

- If one of the types is a complement type $t = \mathcal{C}t'$ then the other type s must not be a subtype of t' in order to have common elements. Because of Lemma 3.4.8 $\langle\!\langle s \rangle\!\rangle \not\subseteq \langle\!\langle t' \rangle\!\rangle$ implies $s \not\sqsubseteq t'$. (Lemma 3.4.8 is applicable because of Assumption 3.4.6 holding due to the precondition of the lemma and because *free-to-top* yields a free constraint set eliminating all free variables from the arguments to *ST*.) Since all the type constructors presented here are monotonic all variable instantiations that can result in common elements are covered by σ'. Thus, one of the rules (*Comp1*) and (*Comp2*) applies whenever t_1 and t_2 can have common elements.

For the rest of the case distinction we can assume that none of the types is $\top$, a recursive, union, intersection or complement type or a free type variable because these cases have already be proven above.

- If one of the types is constructed by a tuple like type constructor c then common elements are just possible if the other type is also constructed by c. Then rule (*Constr*) is applicable. The analogous holds for frame types (where additionally the sets of bound symbols must be equal) and environment types. The applicable rules are (*Frame*) and (*Env*), respectively.

- When one of the types is a function type then the other type must be a function type as well and either the types are equal or one of them is `Tfunc`. In all these cases the function *fq* returns *true* and rule (*FunQ*) is applicable.

- If one of the types (e.g. t_1) is a base type or a value assignment then the other type (i.e. t_2) must be a base type or value assignment, too, with $\mathit{CEbase}(t_1, t_2) = \texttt{true}$. In this case rule (*Base*) applies.

By this case distinction all cases of two types with common elements are covered by one of the rules of *S-CE*.[3] Therefore, returning the empty c-collection in case of no applicable rule is correct.

Part 1 and Part 2 together prove the lemma. □

B.2 Proofs of Section 5.3

Lemma 5.3.9 *The algorithm SMR terminates for every input substitution σ with a finite domain $dom(\sigma)$.*

Proof of Lemma 5.3.9: Let $k := |dom(\sigma)| \in \mathbb{N}$. For both loops with index i the number of iterations is bounded by k. This is also the case for both loops with index j for every i. Altogether the bodies of both inner loops are executed at most $k^2 \in \mathbb{N}$ times. □

Lemma 5.3.10 *Let σ' be a substitution such that the graph $G = (V, R)$ defined as in GIS with $V = dom(\sigma')$ and $R = \{(y, x) \mid x \neq y, x \leftarrow t \in \sigma'$ and y is subterm of $t\}$ contains more than one node and consists of a single strongly connected component. Let $\sigma = SMR(\sigma')$. Then*

$$\langle\!\langle \sigma \circ \sigma'(t) \rangle\!\rangle(\phi) = \langle\!\langle \sigma(t) \rangle\!\rangle(\phi)$$

for every type term t and every closed type substitution appropriate for $\sigma \circ \sigma'(t)$ and $\sigma(t)$.

Proof of Lemma 5.3.10: For every variable $X \in dom(\sigma')$ we show $\langle\!\langle \sigma \circ \sigma'(X) \rangle\!\rangle(\phi) = \langle\!\langle \sigma(X) \rangle\!\rangle(\phi)$ for every appropriate closed type substitution ϕ. Because of $dom(\sigma) = dom(\sigma')$ this implies $\langle\!\langle \sigma \circ \sigma'(t) \rangle\!\rangle(\phi) = \langle\!\langle \sigma(t) \rangle\!\rangle(\phi)$ for every type term t.

Let $s := \sigma'(X)$. $\sigma(X)$ and $\sigma \circ \sigma'(X)$ do not differ in those positions of s consisting of closed terms or variables not in the domain of σ'.

Let $Y \in dom(\sigma')$ be a variable occurring in s. We will denote $\sigma'(X) = s =: C[Y]$ indicating a context C that contains the variable Y. For the moment we assume that every right hand side occurring in σ' contains exactly one variable $V \in dom(\sigma')$.

Because of $\sigma \circ \sigma'(X) = \sigma(\sigma'(X)) = \sigma(C[Y]) = C[\sigma(Y)]$[4] we have to show $\langle\!\langle \sigma(X) \rangle\!\rangle(\phi) = \langle\!\langle C[\sigma(Y)] \rangle\!\rangle(\phi)$. We do this by case distinction on the different possible orderings between X and Y:

[3]The only type that was not inspected in the case distinction is $\perp$ that has no common elements with any type except of the non-termination that is not of interest for the lemma.

[4]The last step holds because we assume that there are no other variables $V \in dom(\sigma)$ occurring in $C[Y]$.

1. $Y = X$: $\sigma(Y) = \sigma(X) = \mu A.C[A]$. Therefore, $C[\sigma(Y)] = C[\mu A.C[A]]$ differs from $\sigma(X)$ just by an unfolding step. Both terms are semantically equivalent.

2. $X <_V Y$: When Y is processed in the second loop on i the value $\sigma(Y)$ is inserted into the constraint of X, i.e. $\sigma(X) = C[\sigma(Y)]$.

3. $X >_V Y$: Let there exist variables $V_1, \ldots, V_{m+1}$ fulfilling

$$\begin{aligned} \sigma'(Y) &= C_0[V_1] \\ \sigma'(V_1) &= C_1[V_2] \\ &\vdots \\ \sigma'(V_m) &= C_m[V_{m+1}] \end{aligned}$$

   and let $V_l <_V X$ for $l \leq m$ and $V_{m+1} \geq X$. We prove $\langle\!\langle \sigma(X) \rangle\!\rangle(\phi) = \langle\!\langle C[\sigma(Y)] \rangle\!\rangle(\phi)$ by induction on m.

   $\mathbf{m = 0}$: When processing Y in the first loop on i the assignment to X in σ' is updated to $C[C_0[V_1]]$. The processing of V_1 either yields $C[\sigma(Y)] = \mathit{unfold}(\sigma(X))$ according to (1) for $V_1 = X$ or $C[\sigma(Y)] = \sigma(X)$ according to (2) for $V_1 >_V X$. In both cases $\langle\!\langle \sigma(X) \rangle\!\rangle(\phi) = \langle\!\langle C[\sigma(Y)] \rangle\!\rangle(\phi)$.

   $\mathbf{m \rightarrow m+1}$: Let the statement hold for m. Assume that

$$\begin{aligned} \sigma'(Y) &= C_0[V_1] \\ \sigma'(V_1) &= C_1[V_2] \\ &\vdots \\ \sigma'(V_m) &= C_m[V_{m+1}] \\ \sigma'(V_{m+1}) &= C_{m+1}[V_{(m+1)+1}] \end{aligned}$$

   and let $V_l <_V X$ for $l \leq m+1$ and $V_{m+2} \geq X$. When processing Y in the first loop on i the assignment to X in σ' is updated to $C[C_0[V_1]]$ which can be understood as a context $C'[V_1]$ of V_1. By setting $Y' := V_1$, $V_l' := V_{l+1}$ for $l \in \{1, \ldots, m+1\}$ we can apply the induction hypothesis and get $\sigma(X) = C'[Y'] = C[C_0[\sigma(V_1)]]$. When processing V_1 in the second loop on i the assignment to Y is updated to $C_0[\sigma(V_1)]$ which is also the value of $\sigma(Y)$ and therefore $\sigma(X) = C[\sigma(Y)]$.

   Altogether the statement is proven.

The three cases above prove $\langle\!\langle \sigma(X) \rangle\!\rangle(\phi) = \langle\!\langle C[\sigma(Y)] \rangle\!\rangle(\phi)$ for the case that every right hand side of σ' contains exactly one variable $V \in \mathit{dom}(\sigma')$.

If $\sigma'(X)$ contains several Y_k they can be considered independently in ascending order according to $<_V$.

If $\sigma'(Y)$ contains several variables $Z_k \in dom(\sigma')$ then we just need to consider those $Z_l >_V Y$ because only these remain in $\sigma'(Y)$ until replacing Y in $\sigma'(X)$. The $Z_l > X$ do not influence each other. They are consistently replaced in X and Y during the second loop on i. $Z_l = X$ does not influence any other variables. Processing it just yields a difference in form of an unfolding step as stated above. The influences of several Z_l with $Y <_V Z_l <_V X$ on each other can be proven inductively considering the Z_l in ascending order according $<_V$.

Altogether we have proven the lemma. □

Lemma 5.3.11 *The algorithm GIS terminates for every input substitution with a finite domain $dom(\sigma)$.*

Proof of Lemma 5.3.11: By the precondition of the lemma $|V| = |dom(\sigma)|$ is finite. The component graph G' is generated from G just by merging nodes. Thus, $|V'|$ is also finite. Since every execution of the *while* loop changes the mark of one node $v' \in V'$ from 0 to 1 the *while* loop is executed a finite number of times. Inside of the *while* loop the *then* case as a sequence of terminating commands terminates. This is true for the *else* case, too, because $dom(\sigma') \subseteq dom(\sigma')$ and therefore $|dom(\sigma')|$ is finite. Thus, the call $SMR(\sigma')$ terminates by Lemma 5.3.9. □

Lemma 5.3.12 *Let σ' be a substitution fulfilling with the following properties:*

1. *σ' does not contain a variable binding $A \leftarrow B$ with $B \in dom(\sigma')$.*
2. *If σ' contains a variable binding $A \leftarrow C[B]$ with a context C and a variable $B \in dom(\sigma')$ then there exists a variable $B' \notin dom(\sigma')$ such that B is bound to B' or a union containing B' in σ'.*

Let v be a value fulfilling

$$\forall k \in \mathbb{N} \,.\, v \in {\sigma'}^k(t_1) \boxdot {\sigma'}^k(t_2)$$

and let $\sigma = GIS(\sigma')$. Then

$$v \in \sigma(t_1) \boxdot \sigma(t_2)\,.$$

Proof of Lemma 5.3.12: Let t_1 and t_2 be types and v a value fulfilling the precondition of the lemma. We can conclude

$$\forall k \in \mathbb{N}\ \exists \tau \,.\, v \in \langle\!\langle \tau \circ {\sigma'}^k(t_1) \rangle\!\rangle \cap \langle\!\langle \tau \circ {\sigma'}^k(t_2) \rangle\!\rangle \tag{B.6}$$

The substitution τ transforms the given argument terms to closed terms. Thus, the terms assigned to all variables by τ are closed terms. We can divide τ into two substitutions τ' and

ρ with $\tau = \tau' \circ \rho$ and $dom(\rho) = dom(\sigma') = dom(\sigma)$ where σ is the substitution generated from σ' by *GIS*.

Every non-cyclic value v has a finite representation and this also holds for all finite approximations of cyclic values. We can therefore find a $k' \in \mathbb{N}$ for every v such that v is in the intersection of $\langle\!\langle \tau' \circ \rho \circ \sigma'^{k'}(t_1) \rangle\!\rangle$ and $\langle\!\langle \tau' \circ \rho \circ \sigma'^{k'}(t_2) \rangle\!\rangle$ independently of ρ (with τ' chosen appropriately):

For variables instantiated with a base type, $\top$, a function type, a quantified variable, a frame type, an environment type or a type constructed by a free type constructor the statement is obvious, because structure is added to t_1 or t_2 and this is just possible a finite number of times.
If there is a cycle of n variables each containing its successor as element of a union type then $k' \geq n$ fulfills the statement: The prerequisites on σ' given in the lemma enforce that whenever a variable A is instantiated by σ' the instance is either A' or a union containing A' with $A' \notin dom(\sigma)$. By choosing $\tau'(A')$ appropriately every additional value introduced into such a chain of unions can also be introduced by τ'.
Variables occurring in intersections in $\sigma'^{k'}$ are obviously no problem when choosing a ρ that does not introduce any restrictions not introduced by σ'. This is especially the case for all ρ fulfilling

$$\forall X \in dom(()\sigma') \,.\, \langle\!\langle \rho(X) \rangle\!\rangle(\phi) = \langle\!\langle \rho \circ \sigma'(X) \rangle\!\rangle(\phi) \tag{B.7}$$

for every closed type substitution ϕ appropriate for $\rho(X)$ and $\rho \circ \sigma'(X)$.

Altogether we can now formulate the following statement:

$$\forall v \in \mathcal{V}\ \exists k' \in \mathbb{N}\ \forall \rho \text{ fulfilling (B.7)} \,.\, v \in \langle\!\langle \tau' \circ \rho \circ \sigma'^{k'}(t_1) \rangle\!\rangle \cap \langle\!\langle \tau' \circ \rho \circ \sigma'^{\prime k}(t_2) \rangle\!\rangle \tag{B.8}$$

Especially (B.7) is fulfilled for $\rho = \sigma := GIS(\sigma')$. We show this by arbitrarily choosing a fixed $X \in dom(\sigma')$ and distinguishing several cases on $\sigma'(X)$:

Case 1: $\sigma'(X)$ does not contain any variables $Y \in dom(\sigma')$. Then X has no predecessors in the graph G defined by *GIS*. Especially X is not a member of a strongly connected component with more than one node in G. Thus, when choosing X in the *while*-loop of *GIS* then the *then*-case is evaluated with $t' = t = \sigma'(X)$. Therefore, σ contains the assignment $X \leftarrow t$ and hence $\sigma(X) = \sigma'(X) = \sigma \circ \sigma'(X)$.

Case 2: $\sigma'(X)$ contains variables from $dom(\sigma')$ and X is not contained in a strongly connected component with more than one node in G. Then $\sigma'(X) = t$ and $\sigma \circ \sigma'(X) = \sigma(t)$. Now consider the variables $Y \in dom(\sigma')$ contained in t. For simplicity we assume t to contain just one variable. Several different variables can be proven by induction on their number processing the variables in the order they are chosen from *GIS*. When choosing X in the

while-loop all predecessors of X are marked with 1. Thus, either $Y = X$ or $Y \in \mathit{dom}(\sigma'')$. In the latter case $\sigma''(Y) = \sigma(Y)$ because the binding of Y is not changed after being inserted into σ'' and $\sigma = \sigma''$ at the end of the *forall* loop. Therefore, $\sigma \circ \sigma'(X) = \sigma'' \circ \sigma'(X) = \sigma(X)$. If $Y = X$ then $t = C[X]$ and $t' = \mu Z.C[Z]$.[5] Thus, $\sigma(X) = \mu Z.C[Z]$ and $\sigma \circ \sigma'(X) = C[\mu Z.C[Z]]$. Obviously, $C[\mu Z.C[Z]]$ can be achieved by unfolding $\mu Z.C[Z]$ one time and therefore both terms are semantically equivalent.

Case 3: X occurs in a strongly connected component K with more than one element in G. Let σ'' denote the state in the algorithm at the beginning of the processing of the strongly connected component containing X. During the calculation of *GIS* no change to σ'' affects the value of an already defined variable and therefore

$$\forall X \in \mathit{dom}(\sigma'') \,.\, \sigma(X) = \sigma''(X).$$

Since $\mathit{dom}(\sigma'') \subseteq \mathit{dom}(\sigma)$ we have $\sigma = \sigma \circ \sigma''$. Therefore,

$$\sigma \circ \sigma'(X) = (\sigma \circ \sigma'') \circ \sigma'(X) = \sigma \circ (\sigma'' \circ \sigma')(X) = \sigma \circ \sigma_r(X)$$

with σ_r as defined by *GIS*. By definition of σ_r the type $\sigma_r(X)$ just contains variables $Y \in \mathit{dom}(\sigma_r)$. Using this information we have

$$\sigma \circ \sigma_r(X) = \sigma|_{\mathit{dom}(\sigma_r)} \circ \sigma_r(X) = \tilde{\sigma} \circ \sigma_r(X)$$

Since $\tilde{\sigma} = \mathit{SMR}(\sigma_r)$, Lemma 5.3.10 implies

$$\tilde{\sigma} \circ \sigma_r(X) = \tilde{\sigma}(X)\,.$$

Since the assignment $X \leftarrow \tilde{\sigma}(X)$ is added to σ'' and not changed any further we have

$$\tilde{\sigma}(X) = \sigma(X)\,.$$

Considering the three cases together we have proven $\sigma \circ \sigma'(X) = \sigma(X)$.

We denote the smallest k' as given in (B.8) by k_v and prove

$$\forall i \in \{0, 1, \ldots, k_v\} \,.\, v \in \langle\!\langle \tau' \circ \sigma \circ \sigma'^{k_v - i}(t_1) \rangle\!\rangle \cap \langle\!\langle \tau' \circ \sigma \circ \sigma'^{k_v - i}(t_2) \rangle\!\rangle \tag{B.9}$$

by induction on i:

$\mathbf{i = 0}$**:** By choosing $\rho = \sigma$ we get the statement directly from (B.8).

$\mathbf{i \to i + 1}$**:** From the induction hypothesis we can conclude

$$v \in \langle\!\langle \tau' \circ \sigma \circ \sigma' \circ \sigma'^{k_v - (i+1)}(t_1) \rangle\!\rangle \cap \langle\!\langle \tau' \circ \sigma \circ \sigma' \circ \sigma'^{k_v - (i+1)}(t_2) \rangle\!\rangle\,.$$

[5] X may occur at several positions of t. We use the notation of a context C with X at one position for simplicity but do not make use of a restricted number of occurrences of X in t.

To get the intended statement

$$v \in \langle\!\langle \tau' \circ \sigma \circ {\sigma'}^{k_v-(i+1)}(t_1) \rangle\!\rangle \cap \langle\!\langle \tau' \circ \sigma \circ {\sigma'}^{k_v-(i+1)}(t_2) \rangle\!\rangle \, .$$

we just have to show $\langle\!\langle \sigma \circ \sigma'(t) \rangle\!\rangle(\phi) = \langle\!\langle \sigma(t) \rangle\!\rangle(\phi)$ for every type term t and every appropriate closed type substitution ϕ. In showing this we can restrict ourselves to the statement $\langle\!\langle \sigma \circ \sigma'(X) \rangle\!\rangle(\phi) = \langle\!\langle \sigma(X) \rangle\!\rangle(\phi)$ for $X \in \mathit{dom}(\sigma')$. This statement has already been proven above.

In (B.9) we can now choose $i = k_v$ and can conclude

$$\exists \tau \, . \, v \in \langle\!\langle \tau \circ \sigma(t_1) \rangle\!\rangle \cap \langle\!\langle \tau \circ \sigma(t_2) \rangle\!\rangle \, . \tag{B.10}$$

Besides (B.6), the precondition of the lemma implies that when dividing τ into τ_f for the free and τ_q for the quantified variables the following holds: We can replace τ_q by every τ'_q with the same domain that does not assign $\perp$ to any variable and still get a common element v'.

When choosing v' instead of v and $\tau' = \tau'_q \circ \tau_f$ instead of τ the proof of (B.10) goes through without a change. This is the case for every v' and especially every τ'_q chosen as above because

$$\forall k \in \mathbb{N} \, . \, v \in {\sigma'}^k(t_1) \boxdot {\sigma'}^k(t_2)$$

implies the existence of an appropriate v' for every τ'_q.

Altogether we have proven $v \in \sigma(t_1) \boxdot \sigma(t_2)$. □

Lemma 5.3.13 *If the algorithm S-CE terminates for every pair of terms in set normalized form (and empty free constraint set and empty recursion information) then CE terminates for every pair of input types in set normalized form.*

Proof of Lemma 5.3.13: The call to *S-CE* terminates (according to the precondition of the lemma) and provides a finite collection of substitutions. Thus, the *forall* loop is executed a finite number of times. For every $\sigma' \in \Sigma'$ $|\mathit{dom}(\sigma')|$ is finite. Since *GIS* terminates by Lemma 5.3.11 the termination of *CE* under the preconditions given in the lemma is proven. □

Theorem 5.3.14 *The algorithm CE terminates for every pair of input types in set normalized form.*

Proof of Theorem 5.3.14: The proof of lemma 5.3.13 carries over except of executions of the rules (*Comp1*) and (*Comp2*) in *S-CE*. This case might cause a recursive chain of the form

$$\textit{S-CE} \rightarrow \textit{ST} \rightarrow \textit{CE} \rightarrow \textit{S-CE}$$

After the first execution of the chain at least one argument t_1 or t_2 of *S-CE* does not contain any type variables (without loss of generality t_1). This is the case because t_1 is a subterm of

a type of the form $\mathcal{C}\tilde{t}_1$. By definition of the complement type constructor $\tilde{t}_1$ must not contain any type variables and there are no type variables introduced into t_1 during processing of *ST*, *S-CE* and *CE*. We have two distinguish two cases:

1. t_2 does not contain a complement type constructor that causes an execution of rule (*Comp2*). Then the number of executions of the recursive chain above is bounded by the number of complement type constructors in t_1: Since t_1 does not contain any variables this number cannot increase during the execution of the recursive chain (especially because there are no variables bound by μ that can generate new complement type constructors by unfolding) and in every execution it is decreased by one by *S-CE*.

2. t_2 contains complement type constructors. After the first execution of rule (*Comp2*) the following recursive call of *S-CE* is done with arguments t'_1 and t'_2 where both t'_1 and t'_2 do not contain any variables. (The property of t'_1 carries over from t_1. For t'_2 it follows from the definition of $\mathcal{C}$ as for t_1 above.) The number of further executions of the recursive chain discussed here is bounded by the total number of complement type constructors in t'_1 and t'_2.

In all cases the recursive chain is just executed a finite number of times. It can therefore not violate the termination of *CE*. □

Corollary 5.3.15 *The algorithm ST terminates for every pair of semi-closed input types in set normalized form.*

Proof of Corollary 5.3.15: According to 3.4.7 *ST* terminates as long as *CE* is a terminating function. Since *CE* is terminating due to Theorem 5.3.14 *ST* terminates for every input. □

Corollary 5.3.16 *The algorithm S-CE terminates for every pair of input types in set normalized form.*

Proof of Corollary 5.3.16: Lemma 5.2.16 gives the statement of the corollary under the condition that *ST* terminates. Since *ST* terminates for every input due to Cor. 5.3.15 *S-CE* terminates for every input. □

Theorem 5.3.17 *Let $t_1, t_2 \in \mathcal{T}$, both in set normalized form. Let there exist a value $v \neq \bot$ such that*

$$v \in t_1 \boxdot t_2 \,.$$

Then there exist substitutions $\sigma \in CE(t_1, t_2)$ such that

$$v \in \sigma(t_1) \boxdot \sigma(t_2) \,.$$

Proof of Theorem 5.3.17: Let t_1 and t_2 be types and v a value fulfilling the precondition of the theorem. To the initial call to *S-CE* calculating Σ' we can apply Lemma 5.2.20. (For applying the lemma we need Assumption 3.4.6. Due to Theorem 5.3.14 *CE* terminates and therefore there is just a finite number of recursive subcalls to *CE*. We can use induction on the number of recursive subcalls necessary to evaluate the current call to prove Assumption 3.4.6 for the subcalls.) Lemma 5.2.20 now gives us:

$$\exists \sigma' \in \Sigma' \; \forall k \in \mathbb{N} \,.\, v \in \mathit{subst}(\sigma')^k(t_1) \boxdot \mathit{subst}(\sigma')^k(t_2) \tag{B.11}$$

(Note that $\tilde{\sigma}$ from Lemma 5.2.20 is the empty substitution in the initial call and is therefore omitted here.) In the following we will identify σ' with $\mathit{subst}(\sigma')$.

Because of the definition of Σ the substitution $\sigma = \mathit{GIS}(\sigma')$ fulfills $\sigma \in \Sigma$. Due to Lemma 5.3.12 it furthermore fulfills

$$v \in \sigma(t_1) \boxdot \sigma(t_2) \,.$$

This proves the theorem. □

Appendix C

The Type Language for Scheme

In Cha. 3 the type language is defined on an abstract level just stating the existence of several base types and tuple like type constructors. In this appendix the definition is refined by introducing sets of base types and tuple like type constructors as they appear in the functional programming language Scheme (cf. [KCE98]).

C.1 Base Types

In this section we present several base types used by our type checker. The base types can be partitioned into different sets of types where the types from different partition sets denote disjoint sets of values. For every such set a subtype hierarchy on the base types is introduced by the function $STbase : \mathcal{B} \times \mathcal{B} \rightarrow \{\texttt{true}, \texttt{false}\}$. For base types b_1 and b_2 from different partition sets $STbase(b_1, b_2) = \texttt{false}$ always holds. (We will just introduce the cases that cause the result `true` for $STbase(b_1, b_2)$ which can be given for the different partition sets separately.) Furthermore we define the function $CEbase : \mathcal{B} \times \mathcal{B} \rightarrow \{\texttt{true}, \texttt{false}\}$ that returns true whenever the two argument base types have common elements. Again it suffices to define *CEbase* for types of the same partition and to set the result for arguments from different partitions to `false`.

When the needed sets of values are described we omit the non-termination $\perp$.

C.1.1 The Boolean Types

There are three boolean types `Ttrue`, `Tfalse` and `bool` where `Ttrue` just contains the value `#t`, `Tfalse` contains the value `#f` and `bool` contains both of these values.

Besides the trivial cases of two equal arguments the type hierarchy on these types is defined by:

- $STbase(\texttt{Ttrue}, \texttt{bool})$
- $STbase(\texttt{Tfalse}, \texttt{bool})$

CEbase on the boolean partition set can be given in terms of *STbase* as follows:

$$STbase(b_1, \texttt{bool}), STbase(b_2, \texttt{bool}) \Rightarrow \\ CEbase(b_1, b_2) = \texttt{true} \Leftrightarrow b_1 = b_2 \vee b_1 = \texttt{bool} \vee b_2 = \texttt{bool}\,.$$

Note that `Ttrue` and `Tfalse` are just used for external representation of types. Internally the value assignments $A(\#t)$ and $A(\#f)$ are used instead.

C.1.2 The Symbol Types

The type containing all symbols is denoted by `sym`. It has just the trivial subtype and common element properties $STbase(\texttt{sym}, \texttt{sym})$ and $CEbase(\texttt{sym}, \texttt{sym}) = \texttt{true}$.

C.1.3 The Number Types

The set of values expressed by any number type is also expressible as the union of value sets of the following types:

- `posint-e` (exactly represented positive integers)
- `posint-i` (inexactly represented positive integers)
- `zero-e` (0 in exact representation)
- `zero-i` (0 in inexact representation)
- `negint-e` (exactly represented negative integers)

- `negint-i` (inexactly represented negative integers)
- `posrat-no-int-e` (positive rational numbers that are not integers in exact representation)
- `posrat-no-int-i` (positive rational numbers that are not integers in inexact representation)
- `negrat-no-int-e` (negative rational numbers that are not integers in exact representation)
- `negrat-no-int-i` (negative rational numbers that are not integers in inexact representation)
- `posreal-no-rat-e` (exactly represented positive reals excluding rational numbers)
- `posreal-no-rat-i` (inexactly represented positive reals excluding rational numbers)
- `negreal-no-rat-e` (exactly represented negative reals excluding rational numbers)
- `negreal-no-rat-i` (inexactly represented negative reals excluding rational numbers)
- `comp-no-real-e` (exact complex numbers that are not real)
- `comp-no-real-i` (inexact complex numbers that are not real)
- `num-no-comp-e` (exact numbers that are not complex)
- `num-no-comp-i` (inexact numbers that are not complex)

This set of types completely covers the hierarchy of numbers according to [CE91]. In practical Scheme implementations some of these types might be empty.

The number types defined above are not comparable according to the type hierarchy, i.e. the function *STbase* yields `true` on these types just for the trivial cases of identical elements. The same holds for the common element test *CEbase*.

In the following we introduce several further number types that are equivalent to unions of the types above. The type hierarchy is defined accordingly.

By uniting several of these types supertypes are defined as follows:

- Number types not distinguishing between exact and inexact numbers are defined as follows: For every type `xyz-i` and the corresponding `xyz-e` there is a type `xyz`. Its semantics is given by $\langle\!\langle \texttt{xyz} \rangle\!\rangle := \langle\!\langle \texttt{xyz-e} \rangle\!\rangle \cup \langle\!\langle \texttt{xyz-i} \rangle\!\rangle$ and the type hierarchy function is extended by $STbase(\texttt{xyz-e}, \texttt{xyz}) = \texttt{true}$ and $STbase(\texttt{xyz-i}, \texttt{xyz}) = \texttt{true}$.

- There are types not distinguishing between positive and negative: For every two types *posabc* and *negabc* there is a type abc with semantics defined by $\langle\!|\texttt{abc}|\!\rangle = \langle\!|\texttt{posabc}|\!\rangle \cup \langle\!|\texttt{zero*}|\!\rangle \cup \langle\!|\texttt{negabc}|\!\rangle$ and the following extensions to *STbase*:
  - $STbase(\texttt{posabc}, \texttt{abc}) = \texttt{true}$
  - $STbase(\texttt{negabc}, \texttt{abc}) = \texttt{true}$
  - $STbase(\texttt{zero*}, \texttt{abc}) = \texttt{true}$

  zero* here stands for

  - zero-i if posabc and negabc have a suffix -i.
  - zero-e if posabc and negabc have a suffix -e.
  - zero if posabc and negabc have neither a suffix -e nor -i.

- Types of rational numbers are of the form sigrat* where sig can be pos, neg or empty and * can be -e, -i or empty. The semantics is given by

  $$\langle\!|\texttt{sigrat*}|\!\rangle := \langle\!|\texttt{sigint*}|\!\rangle \cup \langle\!|\texttt{sigrat-no-int*}|\!\rangle$$

  and the type hierarchy is extended by

  - $STbase(\texttt{sigint*}, \texttt{sigrat*}) = \texttt{true}$
  - $STbase(\texttt{sigrat-no-int*}, \texttt{sigrat*}) = \texttt{true}$

- Types of real numbers are of the form sigreal* with sig and * as above. The semantics is given by

  $$\langle\!|\texttt{sigreal*}|\!\rangle := \langle\!|\texttt{sigrat*}|\!\rangle \cup \langle\!|\texttt{sigreal-no-rat*}|\!\rangle$$

  and the type hierarchy is extended by

  - $STbase(\texttt{sigrat*}, \texttt{sigreal*}) = \texttt{true}$
  - $STbase(\texttt{sigreal-no-rat*}, \texttt{sigreal*}) = \texttt{true}$

- Types of complex numbers are of the form comp* with * as above. The semantics is given by $\langle\!|\texttt{comp*}|\!\rangle := \langle\!|\texttt{real*}|\!\rangle \cup \langle\!|\texttt{comp-no-real*}|\!\rangle$ and the type hierarchy is extended by

  - $STbase(\texttt{real*}, \texttt{comp*}) = \texttt{true}$
  - $STbase(\texttt{comp-no-real*}, \texttt{comp*}) = \texttt{true}$

- Further types of general numbers have the form num* with * as above, the semantics given by $\langle\!|\texttt{num*}|\!\rangle := \langle\!|\texttt{comp*}|\!\rangle \cup \langle\!|\texttt{num-no-comp*}|\!\rangle$ and the type hierarchy extended by

- $STbase(\texttt{comp*}, \texttt{num*}) = \texttt{true}$
- $STbase(\texttt{num-no-comp*}, \texttt{num*}) = \texttt{true}$

The function *CEbase* on the number types can now be defined in terms of *STbase* by:

$$STbase(b_1, \texttt{num}), STbase(b_2, \texttt{num}) \Rightarrow$$
$$CEbase(b_1, b_2) = \texttt{true} \Leftrightarrow \exists b \neq \perp . \, STbase(b, b_1) \wedge STbase(b, b_2) \,.$$

C.1.4 The Character Types

The type of all characters is denoted by `char`. There are the following subtypes of characters:

- `num-char` denotes the type of all characters denoting digits.
- `whitespace-char` is the type of all whitespace characters.
- With `upper-char` we denote the type of all uppercase alphabetic characters.
- `lower-char` is the type of all lowercase alphabetic characters.
- `sym-char` denotes the type of all characters not covered by any of the types above.

With respect to *STbase* the types listed above are pairwise not comparable with each other. They are all subtypes of `char`.

CEbase is defined by

$$STbase(b_1, \texttt{char}), STbase(b_2, \texttt{char}) \Rightarrow$$
$$CEbase(b_1, b_2) = \texttt{true} \Leftrightarrow b_1 = b_2 \vee b_1 = \texttt{char} \vee b_2 = \texttt{char} \,.$$

C.1.5 Input/Output and Ports

There are two types of ports: `in-port` is the type of all input type and `out-port` the type of all output ports. The supertype containing all input and output ports is denoted by `port`. Besides the trivial cases *STbase* is defined by:

- $STbase(\texttt{in-port}, \texttt{port}) = \texttt{true}$.

- $STbase(\texttt{out-port}, \texttt{port}) = \texttt{true}$.

CEbase is defined by:

$$STbase(p_1, \texttt{port}), STbase(p_2, \texttt{port}) \Rightarrow$$
$$CEbase(p_1, p_2) = \texttt{true} \Leftrightarrow p_1 = p_2 \vee p_1 = \texttt{port} \vee p_2 = \texttt{port}\,.$$

The type `eof` contains exactly those objects that can occur as end of file object. *STbase* and *CEbase* are defined in the trivial way.

C.1.6 *STbase* and *CEbase* on value assignments

The functions *STbase* and *CEbase* can be extended to value assignments as follows:

- If both arguments are value assignments the functions just return true if the arguments are equal: Let $t_1 = \mathcal{A}(c_1), t_2 = \mathcal{A}(c_2)$.
$$STbase(t_1, t_2) = \texttt{true} \Leftrightarrow CEbase(t_1, t_2) = \texttt{true} \Leftrightarrow c_1 = c_2\,.$$
- If one of the arguments is a value assignment $\mathcal{A}(c)$ and the other one is a type t the result depends on the test $c \in \langle\!|t|\!\rangle$:
$$STbase(\mathcal{A}(c), t) = \texttt{true} \Leftrightarrow CEbase(\mathcal{A}(c), t) = \texttt{true} \Leftrightarrow$$
$$CEbase(t, \mathcal{A}(c)) = \texttt{true} \Leftrightarrow c \in \langle\!|t|\!\rangle\,.$$

C.1.7 Correctness of *STbase* and *CEbase*

Lemma C.1.1 (termination and correctness of *STbase*) *Let b_1 and b_2 be base types or value assignments. Then*

$$STbase(b_1, b_2) = \textit{\texttt{true}} \Rightarrow \langle\!|b_1|\!\rangle \subseteq \langle\!|b_2|\!\rangle\,.$$

Proof: The lemma can be proven along the definition of *STbase*:

- For boolean types we obviously have:
$$\langle\!|\texttt{Ttrue}|\!\rangle = \{\bot, \#t\} \subseteq \{\bot, \#t, \#f\} = \langle\!|\texttt{bool}|\!\rangle$$
and
$$\langle\!|\texttt{Tfalse}|\!\rangle = \{\bot, \#f\} \subseteq \{\bot, \#t, \#f\} = \langle\!|\texttt{bool}|\!\rangle$$

- For the symbol type nothing has to be proven.
- The definitions of *STbase* on number types are given in the context of defining number types by unions of other number types. These defining unions directly prove the correctness of *STbase* on all number types.
  E.g. `int-e` is defined by $\langle\!|\texttt{int-e}|\!\rangle = \langle\!|\texttt{posint-e}|\!\rangle \cup \langle\!|\texttt{negint-e}|\!\rangle \cup \langle\!|\texttt{zero-e}|\!\rangle$. This definition directly gives a correctness proof for the following corresponding definitions of *STbase*:
  - $STbase(\texttt{posint-e}, \texttt{int-e}) = \texttt{true}$.
  - $STbase(\texttt{negint-e}, \texttt{int-e}) = \texttt{true}$.
  - $STbase(\texttt{zero-e}, \texttt{int-e}) = \texttt{true}$.
- The definition of *STbase* for character types is obviously correct because of

$$\langle\!|\texttt{char}|\!\rangle = \langle\!|\texttt{num-char}|\!\rangle \cup \langle\!|\texttt{whitespace-char}|\!\rangle \cup \langle\!|\texttt{upper-char}|\!\rangle \cup \\ \cup \langle\!|\texttt{lower-char}|\!\rangle \cup \langle\!|\texttt{sym-char}|\!\rangle \,.$$

□

C.2 Type Constructors

This section introduces the tuple like type constructors that can be processed by our type checker.

C.2.1 Pairs and lists

The pair type constructor $(A\ .\ B)$ takes two element types A and B as arguments and returns the type of all pairs with the first element of type A and the second element of type B.

The list type constructor (*list* A) just abbreviates the recursive type $\mu X.(\cup\ \texttt{nil}\ (A\ .\ X))$ where `nil` is a synonym for the value assignment $A(())$ of the empty list (). (*list* A) is primarily used for the external representation of types. Internally it is replaced by the recursive definition.

Note that in a type system with *list* defined independently from *pair* the type constructors are no longer free.

C.2.2 Strings

For every type A with $STbase(A, \texttt{char})$ the string type constructor *string* constructs the type ($string$ A) of strings where each character is of type A. `string` is an abbreviation of the type ($string$ `char`).

C.2.3 Vectors

For every element type A the type constructor *vector* generates the vector type ($vector$ A). `vector` is an abbreviation of the type ($vector$ $\top$).