Complete Type Inference in Functional Programming

Vom Fachbereich Informatik
der FernUniversität Hagen
angenommene

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

von
Dipl.-Inform.

Manfred Widera

Gutachter: Prof. Dr. Christoph Beierle
Prof. Dr. Martin Erwig


Hagen, Juni 2001
Abstract

Dynamically typed functional programming languages perform type checking of programs at runtime. Their language definition does not provide the opportunity of rejecting ill typed programs by static type checking at compile time. To make the benefits of static type checking available for dynamically typed languages, the approach of soft-typing was proposed. A soft-typing system raises warnings and introduces runtime type checks for all program parts that cannot be proven well typed. However, a disadvantage of soft-typing is its inability to reject any programs. This is due to the fact that there can be warnings risen by a soft typing system that are caused by the restricted power of the system and not by a type error in the program. Since the type checker is not part of the language definition, in the latter case the programs are still valid.

The work described in this thesis aims to extend soft-typing by the capability of rejecting programs via detection of provable type errors. For every function call it checks whether the type inferred for the argument and the expected input type of the called function have common elements. If they do not have common elements, no evaluation of this function call can succeed; the program is therefore rejected.

This common element property reverses the subtyping property of sound type inference systems: whereas sound type inference considers a function call as not provably well typed whenever it can fail, the type inference system presented here considers a function call as not provably ill typed whenever it can succeed for at least one element of the argument type. This type inference process is therefore complete in the following sense: all programs not containing a function call that must fail at runtime are accepted.

By the idea of complete type inference it is possible to use a very powerful type language without problems regarding the feasibility of type inference: by not inferring the most special types expressible in the type language for some of the program expressions, the system might miss certain type errors, but the completeness property is not violated and severe errors that necessarily lead to a program failure and that should be corrected first are still detected.

After defining an algorithm for checking types for common elements, a type inferencing algorithm is defined using abstract interpretation. The type language used is expressive enough to cover the full programming language Scheme. For Scheme we present an instantiation of the general approach, dealing with all language constructs except destructive updates and continuations; for covering these several extensions are already given in the thesis. Furthermore, the ideas of this work can also easily be transferred to other dynamically typed functional languages.
Acknowledgements

First of all, I would like to thank Prof. Dr. Christoph Beierle. His contributions to the success of my work are manifold: he aided me in extending my interest and knowledge in declarative and especially functional programming, gave me the opportunity and the time to pursue my ideas and supported my work by stimulating discussions, constructive criticism, and careful advise.

Many thanks go to Prof. Dr. Martin Erwig who was so kind to be the co-referee of this thesis and to Prof. Dr. Hans-Werner Six who served on my thesis committee.

Last but not least, I want to thank the members of our group for their helpfulness and the enjoyable working atmosphere.
# Contents

1 Introduction ........................................ 1  
1.1 Motivation ........................................ 1  
1.2 Sound Versus Complete Type Checking ...................... 3  
  1.2.1 Differences in Applications .............................. 3  
  1.2.2 Differences in Type Inference ........................... 6  
1.3 Outline ........................................ 8  

2 Preliminaries and Related Work .................. 11  
2.1 Set Operations ................................... 11  
2.2 Term Rewriting Systems .............................. 13  
  2.2.1 Terms ......................................... 13  
  2.2.2 Usual Term Rewriting Systems ...................... 15  
  2.2.3 Extended Term Rewriting Syntax ...................... 16  
2.3 Graph Theory .................................... 19  
2.4 Abstract Interpretation ........................... 20  
2.5 Type Inference .................................. 21  

3 The Type Language .............................. 25  
3.1 The Standard Types ................................ 25  
  3.1.1 Syntax of the Standard Types ...................... 25  
  3.1.2 Semantics of the Standard Types .................... 34  
  3.1.3 Set Normalized Standard Types ...................... 41  
  3.1.4 Equivalence Classes of Types ....................... 43  
3.2 Value Assignments ................................ 44  
3.3 I/O-Representations of Function Types ............... 44  
3.4 The Subtype Hierarchy on Semi-Closed Types ............. 48  
  3.4.1 The Algorithm $ST$ ................................ 48  
  3.4.2 Examples of $ST$ .................................. 51  
  3.4.3 Properties of $ST$ ................................ 51  

4 Checking Types for Common Elements .......... 53
# A Proofs of Chapter 3

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>219</td>
</tr>
<tr>
<td>A.2</td>
<td>222</td>
</tr>
</tbody>
</table>

# B Proofs of Chapter 4

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>229</td>
</tr>
<tr>
<td>B.2</td>
<td>243</td>
</tr>
</tbody>
</table>

# C Proofs of Chapter 6

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>251</td>
</tr>
<tr>
<td>C.2</td>
<td>257</td>
</tr>
<tr>
<td>C.3</td>
<td>268</td>
</tr>
<tr>
<td>C.4</td>
<td>271</td>
</tr>
</tbody>
</table>

# D The Type Language for Scheme

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>277</td>
</tr>
<tr>
<td>D.1.1</td>
<td>278</td>
</tr>
<tr>
<td>D.1.2</td>
<td>278</td>
</tr>
<tr>
<td>D.1.3</td>
<td>278</td>
</tr>
<tr>
<td>D.1.4</td>
<td>281</td>
</tr>
<tr>
<td>D.1.5</td>
<td>281</td>
</tr>
<tr>
<td>D.1.6</td>
<td>282</td>
</tr>
<tr>
<td>D.1.7</td>
<td>282</td>
</tr>
<tr>
<td>D.2</td>
<td>283</td>
</tr>
<tr>
<td>D.2.1</td>
<td>283</td>
</tr>
<tr>
<td>D.2.2</td>
<td>284</td>
</tr>
<tr>
<td>D.2.3</td>
<td>284</td>
</tr>
</tbody>
</table>

# E Abstract Semantics of Predefined Scheme Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1</td>
<td>285</td>
</tr>
<tr>
<td>E.2</td>
<td>286</td>
</tr>
<tr>
<td>E.3</td>
<td>286</td>
</tr>
<tr>
<td>E.4</td>
<td>287</td>
</tr>
<tr>
<td>E.5</td>
<td>287</td>
</tr>
<tr>
<td>E.6</td>
<td>292</td>
</tr>
<tr>
<td>E.7</td>
<td>294</td>
</tr>
<tr>
<td>E.8</td>
<td>295</td>
</tr>
<tr>
<td>E.9</td>
<td>295</td>
</tr>
<tr>
<td>E.10</td>
<td>296</td>
</tr>
<tr>
<td>E.10.1</td>
<td>296</td>
</tr>
<tr>
<td>E.10.2</td>
<td>297</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

In software engineering it is an important task to detect errors in software as soon as possible. Especially, it is of great interest to detect errors before the execution of a program.

Many errors occurring in software are type errors. In dynamically typed functional languages (as e.g. Scheme) these errors can only be detected at runtime. A static type checker could assist the programmer by detecting type errors without executing the program for many different input arguments.

The notion of type errors varies from system to system. In this work we say that a type error occurs whenever a function \( f \) is called with an argument tuple \( t \) that is not in the domain of \( f \). For instance, a call to the division function with 0 in the second argument is a type error; this is often not the case in other type systems since the type language is not powerful enough. When a type error must occur for some function call in a program \( P \) then we call \( P \) ill-typed.

Static type checking of functional programs has been analyzed in detail ([Mil78] and many extensions). It is well-known that no algorithm is able to decide for all lambda expressions correctly, whether a given expression is free of type errors. (This is quite obvious and e.g. stated in [AWL94].) Therefore, type checkers usually are restricted to soundness, i.e. every program that is type correct according to a sound type checker will not produce a type error for any input data. In other words, for every sound type checker Milner’s slogan [Mil78]

“Well-typed programs do not go wrong”
The approach to type soundness is useful in many contexts of programs that are believed to be correct, e.g. in proving the type correctness of critical software or in identifying runtime type checks that can be dropped without a risk in a dynamically typed language. In the context of debugging programs, sound type checkers show the following disadvantage: A sound type checker does not detect function calls containing type errors, but function calls whose type correctness cannot be proven. For statically typed languages as e.g. standard ML certain functions (e.g. those working on partial types [Tha88, Tha94]) are ruled out because they are not typable. In dynamically typed languages employing a sound type checker for generating warnings (this is called soft typing, see e.g. [CF91]), it is the user's task to determine whether a warning in the program results from a type error or just from an inaccuracy of the type checker.

In this work, we present a novel typing approach that avoids this disadvantage of sound type checkers. Its goal is not to show that a program is free of type errors. Instead, its goal is to prove that a program does contain a type error. Additionally, we want to point out the origination of all detected errors as exactly as possible to support the programmer in correcting them.

For such an approach to type checking it is sensible to require that only real type errors are reported. Equivalently, any program that is type correct must be reported as being type correct. Therefore, we call this type checking approach complete since every type correct program is accepted. Thus, completeness of a type checker guarantees that whenever an error is reported we are sure that the program is indeed ill-typed. Rephrasing Milner's slogan the motto of complete type checkers is:

"Ill-typed programs must go wrong."

However, it is well known that a type checker for functional languages cannot be sound and complete at the same time due to the UN-decidability of the type checking problem. The missing soundness of any complete system will cause some errors not to be caught. This is not critical in a usual debugging framework because debugging a program consists of further steps that will detect some of the type errors, too. When after debugging and correcting the program type correctness is crucial, a sound type checker can still be used to prove it.

To keep the inaccuracy of our system as low as possible, we use an expressive type language which contains type constructors for union types and recursive types among others. (These kinds of type constructors are essential for an expressive type language [CF91].) Furthermore, our type system includes intersection of types, type difference in a restricted form, and the base types and type constructors to express types of all constructs usually available in a functional language. The given type definition is influenced by the LISP dialect Scheme
[KCE98], but it should be easy adaptable to other functional languages. Partial types, as described in [Tha88, Tha94] are no problem at all in the complete type system described here, because union types can be used to express heterogeneous element types of structures, and an exact distinction between the possible element types is not necessary.

1.2 Sound Versus Complete Type Checking

It is well known that no type checker for functional languages can be sound and complete, i.e. can report every type error and only those. This is true because of two reasons:

- No type language can express every set of values exactly with a finite representation. For example, in type languages it is often impossible to express the type of all numbers without 0 which would be the input type of the division operator /.

- Some sets of values are not decidable. E.g. checking the input type of / involves calculating all zeros of an arbitrary function that may occur as argument of /. This is not possible even for simple classes of functions, as e.g. [Wan74] shows.

Therefore, one must decide for every application whether a sound or a complete type checker is more appropriate.

We can consider all the possible functional programs as given on an axis as shown in Fig. 1.1. This figure illustrates the duality between sound and complete type checking approaches. From left to right the difficulty to prove correctness increases and the difficulty to prove the existence of a type error decreases. Both sound and complete type checkers should try to hit the border between well-typed and ill-typed programs as exactly as possible. However, since no type checker can be sound and complete, a sound type checker can just guarantee to reject every ill-typed program but it may reject some well-typed ones as well. On the other hand, a complete type checker guarantees to report only real type errors. Therefore, it accepts every well-typed program but may fail to catch some of the type errors.

1.2.1 Differences in Applications

Dynamically typed functional languages do not carry the definition of a specific type checker in their language definition. Since they often incorporate language constructs that are hard to type precisely, soft typing and complete typing as presented in this work seem to be the
best ways to provide static typing information in dynamically typed languages without ruling out working programs.

The following example describes the behaviour of a soft typing system and the complete type checker presented in this work for different situations:

**Example 1.2.1** Consider the function call $c = (\text{vector-ref } v \ i)$ in the functional language Scheme [KCE98] with a vector expression $v$ and an index expression $i$. Assume that $k$ is an expression with inferred type $\text{posint}$ (the type of all positive integers). Depending on the expression $i$ the type checkers behave as follows:

- If $i = (+ k 3)$ then the type $\text{posint}$ denoting the set of all positive integers can be inferred for $i$. The function call $c$ is well-typed in each of the considered type checkers.

- For $i = (− k 3)$ just the type $\text{int}$ (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 $c$. The complete type checker that will be presented in this work accepts the call $c$.  

---

**Figure 1.1: Difference between soundness and completeness**
If $i = (\ast k - 2)$ the most special type that can be inferred for $i$ is $\text{negint}$ (the type of all negative integers). A soft typing system still just raises a warning. Our complete type checker raises an error for $c$ because it indeed infers $\text{negint}$ for $i$ and therefore, this call cannot succeed with a negative number as vector index.

Whenever the soft typing system raises a warning in the example above a sound type system of a strictly typed functional language would have to reject the program. (Both the type checker of the statically typed language and the soft typing system are sound type checkers. They just differ in the interpretation of their output.) Existing strictly typed languages (e.g., the type inference system used in Standard ML [Wik87]) often just consider a generalized type $\text{int}$ of all integer numbers. The errors not detectable with this abstraction (e.g., division by zero or negative index access of vectors) are excluded from the scope of the type checker.

The research on type checkers for functional languages from Hindley/Milner [Hin69], [Mil78] to current systems dealt with sound type checkers. As far as software engineering is concerned, soundness is the main property of a type checker since a successful type check of a program gives a partial proof of its correctness. Thus, sound type checkers are used to prove properties of correct programs.

Another benefit of a sound type checker is the opportunity of optimizing functional programs in dynamically typed languages by eliminating runtime type checks in such cases, where the type safety of the program can be statically proven before runtime. This approach is called soft typing. A description can be found e.g. in [WC94], [AWL94], [CF91].

However, sound type checkers are only adequate if the correctness of a program is of interest. As Ex. 1.2.1 shows, their usability for detecting errors in programs of a dynamically typed language is restricted, because real type errors may not be distinguished from inaccuracies of the system. This can confuse the users of the type checker. Complete type checkers in contrast have the advantage that every reported error is known to be a real type error. This helps to avoid confusing the user and may lead to better explanations of type errors.

When still in the development and debugging phases of software development, the disadvantage of missing some of the errors does not carry much weight, because usually these errors will be caught by further debugging processes. In fact, the use intended for the system presented in this work is in combination with a soft typing system. This combination yields a set of output messages for every program that is structured into

- **errors** marking program parts that must cause runtime type errors when executed. These are the program parts rejected by the complete type checker.

- **warnings** marking program parts that could not be proven to be well-typed or ill-typed. These are the program parts not accepted by the soft typing system but neither rejected
by the complete type checker.

By this message priority the programmer can concentrate on correcting the errors first, before he analyzes the program parts marked with warnings for their correctness.

### 1.2.2 Differences in Type Inference

The aim of a type checker is to compare the use of functions in function calls with the constraints given by their function definition. For every function call the input arguments are compared with the input valid for the function. Different type checkers differ in the constraints on the input arguments they check:

Let $A = f(a_1, \ldots, a_n)$ be an expression with some function $f$ and subexpressions $a_i$ at the argument positions $i$ of $f$. Let $t'_i$ be the type expected by $f$ at position $i$ and $t_i$ the inferred type of the expression $a_i$ if $a_i$ is typable. (We just consider eager evaluation of function calls here. This implies that whenever some $a_i$ is not typable the attempt to type $A$ fails, too.) The following conditions must hold for $A$ to be well-typed in different type systems:

- $A$ is well-typed in the Hindley/Milner system [Hin69, Mil78] if the inferred type $t_i$ of $a_i$ and the expected type $t'_i$ at argument position $i$ are equal for every $i$, i.e.
  \[ \forall i \in \{1, \ldots, n\} . t_i = t'_i. \]
  A set of such constraints can be checked using unification.

- $A$ is well-typed for a sound type checker with subtyping if every argument $a_i$ has an inferred type $t_i$ that is compatible with the type $t'_i$ expected at the $i$th argument position, i.e.
  \[ \forall i \in \{1, \ldots, n\} . \langle t_i \rangle \subseteq \langle t'_i \rangle \]
  i.e.
  \[ \forall i \in \{1, \ldots, n\} \forall x \in \langle t_i \rangle . x \in \langle t'_i \rangle^1 \]

  (see Figure 1.2).

Some sound type systems (e.g. [WC94]) restrict this kind of constraint in a way that allows a modification of the Hindley-Milner algorithm without increasing its complexity too much.

Another approach, as presented in [AW93], uses an alternative constraint solving algorithm for inclusion constraints whose complexity makes it hard to use (cf. [AW92]).

---

1 $\langle t \rangle$ denotes the set of all values denoted by a type $t$.  

A is well-typed for our complete type checker if the inferred type $t_i$ of every argument $a_i$ can be specialized to a type that is compatible with the type $t'_i$ expected at the $i^{th}$ argument position:

$$\forall i \in \{1, \ldots, n\} \ \exists x \in \langle t_i \rangle . x \in \langle t'_i \rangle$$

i.e.

$$\forall i \in \{1, \ldots, n\} . \langle t_i \rangle \cap \langle t'_i \rangle \neq \emptyset$$

(see Figure 1.2). Unfortunately, there seems to be no way of checking constraints of this kind efficiently, e.g. by unification.

No type inference system for a functional programming language can compute the exact set of possible values of every lambda expression: Some sets of possible values cannot be described exactly in the given type language. Even without this restriction, computing the exact set of possible values sometimes means evaluating the expression for every mapping of input data to the free variables of the expression. This is often impossible because of e.g. non terminating recursion or simply because of an infinite number of different input tuples that must be considered independently.

If the exact set of denoted values was inferable for every expression there would be no point in distinguishing sound and complete type checkers. Both properties could be reached by the same system.

However, type inference systems usually just guarantee to compute supertypes of the exact
type of the examined expression and try to keep the inaccuracy of the computation as low as possible. Sound and complete type checkers now differ in the effects of extra values that indeed cannot occur but are denoted by the computed type:

- In a sound type checker these extra values can cause a type error by violating the inclusion constraint between computed and expected type of an expression, but they cannot prevent a constraint violation in case of a real type error, i.e. they cannot cause the erroneous acceptance of an ill-typed program.

- The complete type checker described in this work will not falsely report a non-existing type error because of extra values in the computed type of some subexpression since these extra values do not violate the constraint of non-empty intersection of computed and expected type. The complete system can overlook type errors because of extra values in the computed type that can be also found in the expected type and thus give a non-empty intersection. But on the other hand, everything the complete type checker detects is a real error, i.e. the program must go wrong at this point.

Both complete and sound type checkers need an expressive type language and an accurate type inference algorithm to keep their inaccuracy as low as possible.

### 1.3 Outline

In the rest of this work a complete type checker and a type inference system appropriate for it are introduced and the main properties of the introduced system are stated. The work is organized as follows:

In Chapter 2 several tools and techniques used throughout the work are summarized. The topics mentioned there are:

- Set operations: the usual definitions of some set operations, e.g. union and intersection, are recalled. Taking into consideration the associativity and commutativity of these binary operations we define variants of arbitrary arity using a prefix notion influenced by Scheme.

- 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.
Abstract interpretation: the usual technique of abstraction and the relations between standard and abstract semantics are briefly summarized.

Type inference: techniques for type inference and type checking known from literature are briefly described. A discussion of related work on type inference is also done here.

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) denote common elements or not. A formal definition of common elements and an algorithm approximating a solution to this question are presented in Chapter 4. After presenting some preliminaries in Sec. 4.1, the problem is divided into two subproblems: In Sec. 4.2 we describe how constraints on the instances of the types can be collected by recursively traversing the types. Section 4.3 explains how the collected sets of constraints can be transformed into idempotent substitutions.

In order to define the type inference system, Chap. 5 gives a definition of the functional language programs can be expressed in. Essentially, the definitions given there cover the full functional language Scheme [KCE98], except for continuations and some syntactic sugar. Though the standard semantics defined in Chap. 5 is as usual, a quite detailed definition is given because many of the ideas used for the abstract semantics introduced in Chap. 6 can be motivated by the standard semantics.

The presentation of a type inference and type checking system is done in Chap. 6. The formalism used to describe the system is an abstract interpretation modeling the standard semantics presented in Chap. 5. The intended properties of the resulting type inference system are stated in Sec. 6.1. After the abstract domains have been defined in Sec. 6.2 a first abstract semantics is presented in Sec. 6.3. This semantics depends on a given main function and a given tuple of input types in order to perform type checking. The need for providing such an input typing is remedied in Sec. 6.4. The presentation of the abstract semantics is finished in Sec. 6.5 by introducing special methods for the processing of recursion in order to guarantee termination of the system for all input programs.

Hitherto, both the type language and the functional language are chosen with Scheme [KCE98] in mind, but the definitions are generic enough to cover different functional languages as
well. In Chap. 7 these generic definitions are instantiated with types and predefined function definitions that are specific for Scheme and the application of the resulting system to both well-typed and ill-typed programs is presented.

In Chap. 8 some conclusions are given and future work is discussed.

The proofs of all theorems stated throughout the work are given in the appendices. Appendix A contains the proofs of Chap. 3, App. B for Chap. 4 and App. C for Chap. 6.

The specializations of different objects that are defined throughout the work in order to fit the properties of Scheme are done in App. D for the type language and in App. E for the abstract semantics.

Preliminary versions of parts of this work have been published before: the main components of a complete type checker are identified in [WB98]. The test for common elements of types is presented in [WB00a] and [WB01a], and the complete type checking algorithm was presented in [WB99a] and [WB00b]. [WB99b] and [WB01b] develop an extension of term rewriting that is employed in this thesis.
Chapter 2

Preliminaries and Related Work

The aim of this chapter is to summarize definitions of different concepts and tools used throughout the rest of this work and to fix the used notions. This is done for sets and operations on sets in Sec. 2.1, for term rewriting systems in Sec. 2.2, for graphs in Sec. 2.3, for abstract interpretation in Sec. 2.4 and for type inference systems in Sec. 2.5. When we introduce new concepts, a discussion of related work is also given in the corresponding section.

2.1 Set Operations

Sets and set operation are used in this work in the usual manner. This section just recalls some definitions and fixes the notions used throughout the work.

Definition 2.1.1 (sets) There are two forms to denote sets. Finite sets defined by enumerating all elements are denoted by

\[ S_1 := \{e_1, e_2, \ldots, e_k\} \]

where the \( e_i \) for \( i = 1, \ldots, k \) are all elements of the set \( S_1 \).

When defining a set \( S_2 \) by selecting those elements of a given set \( S \) that fulfill a certain formula \( P \) we write

\[ S_2 := \{x \in S \mid P(x)\} \]

When \( x \in S \) is clear from the context we sometimes just write \( \{x \mid P(x)\} \).

As already used in the definition of \( S_2 \) we write \( x \in S \) to denote that \( x \) is an element of the set \( S \).
Unions and intersections of sets are defined in the usual manner:

**Definition 2.1.2 (unions, intersections)** Let $S_1$ and $S_2$ be sets. The union of $S_1$ and $S_2$ is defined as

$$S_1 \cup S_2 = \{ x \mid x \in S_1 \lor x \in S_2 \}$$

The intersection of $S_1$ and $S_2$ is analogously defined as

$$S_1 \cap S_2 = \{ x \mid x \in S_1 \land x \in S_2 \}$$

Since union and intersection are associative operations we can extend them to expect more than two argument sets. We introduce a LISP like notion as follows:

**Definition 2.1.3** For union and intersection of a variable number of sets we write

$$(\cup S_1 \ldots S_k) \text{ and } (\cap S_1 \ldots S_k)$$

with $k \geq 1$. These notions are defined by

$$(\cup A) := A$$

$$(\cup A B) := A \cup B$$

$$(\cup A_1 \ldots A_n A_{n+1}) := (\cup A_1 \ldots A_n) \cup A_{n+1} \text{ for } n \geq 2$$

$$(\cap A) := A$$

$$(\cap A B) := A \cap B$$

$$(\cap A_1 \ldots A_n A_{n+1}) := (\cap A_1 \ldots A_n) \cap A_{n+1} \text{ for } n \geq 2$$

**Definition 2.1.4 (set difference)** Let $S_1$ and $S_2$ be sets. The set difference of $S_1$ and $S_2$ is defined as

$$S_1 \setminus S_2 = \{ x \in S_1 \mid x \notin S_2 \}.$$  

**Definition 2.1.5 (complement set)** Let $S$ and $U$ be sets. The complement set of $S$ with respect to $U$ is defined as

$$\mathcal{C}_U S = \{ x \in U \mid x \notin S \}.$$ 

If the set $U$ of all considered values is clear from the context we often write $\mathcal{C} S$ instead of $\mathcal{C}_U S$.

For generating Cartesian products we define a LISP like constructor $\times$ with variable arity analogously to union and intersection:
Definition 2.1.6 (Cartesian product) Let \( S_1, \ldots, S_k \) with \( k \geq 1 \) be sets. Their Cartesian product is defined as
\[
(\times S_1 \ldots S_k) := \{ (e_1, \ldots, e_k) \mid e_1 \in S_1, \ldots, e_k \in S_k \}.
\]
For \( k = 1 \) we identify \((e)\) with \( e\) and therefore \((\times S)\) with \( S\).

A structure on sets is given by the subset relation defined as follows:

Definition 2.1.7 (subsets) Let \( S_1 \) and \( S_2 \) be sets. \( S_1 \) is called a subset of \( S_2 \) if every element of \( S_1 \) is also an element of \( S_2 \):
\[
S_1 \subseteq S_2 \iff (x \in S_1 \Rightarrow x \in S_2).
\]

Using the subset relation we can now define the power set of a given set:

Definition 2.1.8 (power set) Let \( S \) be a set. The power set of \( S \) is the set of all subsets of \( S \):
\[
\mathcal{P}(S) := \{ S' \mid S' \subseteq S \}
\]

2.2 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. 2.2.1 and summarize term rewriting systems and their main properties in Subsec. 2.2.2. 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. 2.2.3.

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

2.2.1 Terms

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

Definition 2.2.1 (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:
• $x \in T(F, \mathcal{X})$ for all $x \in \mathcal{X}$.

• If $t_i \in T(F, \mathcal{X})$ for $i = 1, \ldots, n$ and $f \in F_n$ then $f(t_1, \ldots, t_n) \in T(F, \mathcal{X})$.

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

**Definition 2.2.2 (positions and subterms)** Let $t \in T(F, \mathcal{X})$ be a term. A position is a list of natural numbers: $p = i_1.i_2.\ldots.i_k.\epsilon$ with the empty list denoted by $\epsilon$. The subterm of $t$ at position $p$ (denoted by $t|_p$) is defined as follows:

- $t|_{\epsilon} = t$ 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 $t|_p$ is undefined.

If $t$ is a term and $p$ a position with $t|_p$ 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 2.2.3 (term update at a position)** Let $t, t' \in T(F, \mathcal{X})$ 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 2.2.4 (substitutions, instances of terms)** A substitution is a function $\sigma : \mathcal{X} \to T(F, \mathcal{X})$ that differs from the identity for a finite number of inputs only. $\sigma$ 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 $\sigma(t|_p)$.

---

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

14
2.2.2 Usual Term Rewriting Systems

A term rewriting system is given by a set of rewriting rules \( l \rightarrow r \):

**Definition 2.2.5 (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 \( l \rightarrow r \) with \( l, r \in T(\mathcal{X}, \mathcal{F}) \) 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 \( \rightarrow_R \) on \( T(\mathcal{X}, \mathcal{F}) \):

**Definition 2.2.6 (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 \( \rightarrow_R \) induced by \( R \) is the binary relation containing all pairs \((t, t')\) of terms (written \( t \rightarrow_R t' \)) with the following property: there is a variable-renamed instance of a rewrite rule \( l \rightarrow r \in R \) (not containing any variables occurring in \( t \))^2, a substitution \( \sigma \) 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 \( \rightarrow_R \) is a rewrite relation then its reflexive, transitive closure is denoted by \( \rightarrow^*_R \).

A term \( t \) with \( t \not\rightarrow_R t' \) 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 2.2.7 (termination of term rewriting systems)** Let \( R \) be a term rewriting system. \( R \) is terminating if every chain \( t_0 \rightarrow_R t_1 \rightarrow_R t_2 \rightarrow_R \ldots \) leads to a term \( t_k \) with \( t_k \not\rightarrow_R t \) for every term \( t \) after a finite number of steps.

For a given term \( t \) there might be different positions where a rule from \( R \) is 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:

---

\(^2\)This situation can always be achieved by renaming the variables in \( l \rightarrow r \).
Definition 2.2.8 (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 $t_2 \to^*_R t'$.

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 2.2.9 (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 2.2.10 (critical pairs) Let $r = l_1 \to r_1$ and $r' = l_2 \to r_2$ 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 $\sigma$ with $\sigma((l_1)_p) = \sigma(l_2)$. ($\sigma$ should be the most general unifier, i.e. $\sigma$ 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.

2.2.3 Extended Term Rewriting Syntax

The terms we want to transform using term rewriting systems differ from the definition in Subsec. 2.2.1 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 [WB99b]. 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 \( i \in \{1,2,\ldots\} \) the pattern \(<i\ldots>\) represents an argument list of variable arity. A pattern \(<i\ldots>\) 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 \(<i\ldots><j\ldots>\). Note that the whole expression \(<i\ldots>\) 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 \(<i\ldots>\) and \(<j\ldots>\) with \(i = j\) denote the same list.

Example:

\[
(+ <1\ldots> (+ <2\ldots>) <3\ldots>) \rightarrow (+ <1\ldots> <2\ldots> <3\ldots>)
\]

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\ldots>\leftarrow [a,b,c], <2\ldots>\leftarrow [d,e], <3\ldots>\leftarrow [f]\}\}. The nested sum \((+ a b (+ c (+ d e)) f)\) of depth 2 can be flattened either by the sequence

\[
(+ a b (+ c (+ d e)) f) \\
\rightarrow (+ a b c (+ d e) f) \\
\rightarrow (+ a b c d e f)
\]

or by the sequence

\[
(+ a b (+ c (+ d e)) f) \\
\rightarrow (+ a b (+ c d e) f) \\
\rightarrow (+ a b c d e f).\]

- The pattern \(<a_i\ldots e_i>\) is used instead of \(<i\ldots>\) if the individual list elements are changed in a uniform manner. \(<a_i\ldots e_i>\) 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 \(<i\ldots>\) and \(<a_i\ldots e_i>\), and constructs of different kinds must not share the same index.

There is an alternative form \(<t(a_i)\ldots t(e_i)>\) where \(t(x)\) stands for any term containing a variable \(x\). This form expresses a list where all elements are instances of the pattern
The pattern \(<t(a) \ldots t(e)><\) matches every list \((v_1 \, v_2 \ldots v_k)\) where each \(v_j\) is of the form \(t(u_j)\), i.e.

\[
\forall j \in \{1, \ldots, k\} \exists u_j \cdot v_j = \sigma_j(t(x)) \text{ with } \sigma_j = \{ x \leftarrow u_j \}
\]

Please note the following correspondence between the patterns \(<t(a) \ldots t(e)><\) and \(<a \ldots e><\): if some substitution \(\tau\) assigns the list \((v_1 \, v_2 \ldots v_k)\) to \(<t(a) \ldots t(e)><\), it implicitly contains the assignment of \((u_1 \, u_2 \ldots u_k)\) to \(<a \ldots e><\) as well as the assignment of \((v'_1 \, v'_2 \ldots v'_k)\) \((v'_j := \sigma_j(t'(x)))\) to \(<t'(a) \ldots t'(e)><\) and vice versa. This ensures that all patterns \(<a \ldots e><\) and \(<t'(a) \ldots t'(e)><\) occurring on the right hand side of a rule are well-defined whenever any of the patterns \(<t(a) \ldots t(e)><\) or \(<a \ldots e><\) occurs on the left hand side of the rule.

Example: The rule

\[
(* (+ \, <a_1 \ldots e_1><\, t) \rightarrow (+ \, <*a_1\, t><\, (*\, e_1\, t)<\, t))
\]

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 \(<a_1 \ldots e_1><\) together with the corresponding \(<t(a) \ldots t(e)><\), 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 \(\varphi(R)\).

- A 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 stemming 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 [WB01b].
### 2.3 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 2.3.1 (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 2.3.2 (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 \( \alpha \) and \( \omega \) to paths \( w = r_1, \ldots, r_m \) and \( w' = r'_1, \ldots, r'_n \) 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 2.3.3 (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 2.3.4 (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

\[
G' = (V', R') \quad \text{with} \\
V' = \{ZK_1, \ldots, ZK_k\} \quad \text{and} \\
R' = \{(ZK, ZK') \in V' \times V' \mid ZK \neq ZK' \land \exists v \in ZK, v' \in ZK'. (v, v') \in R\}.
\]

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.

### 2.4 Abstract Interpretation

This section summarizes the main ideas of abstract interpretation used throughout this work. Introductions to abstract interpretation can be found in [JN95] and [CC92]. It was first defined for a class of imperative flow chart programs in [CC77]. The transfer to functional programs was first done by Mycroft [Myc80] for first order languages and flat domains. This work was extended to higher order functions in [BHA86b, BHA86a]. Several applications of abstract interpretation to declarative languages can be found in [AH87]. A reformulation of several type systems known from literature as abstract interpretations is given in [Cou97].

Abstract interpretation is a tool for program analysis. Consider a program \( P \) in a programming language \( L \). The programming language defines a set \( \text{Conc} \) of values (called the \textit{(concrete) semantic domain}) that can occur as output values or intermediate values during evaluating an arbitrary program given in the language.

When analyzing the program \( P \) one is usually not interested in the exact values occurring at certain program points, but just in certain properties common to all these values. These values are expressed by introducing an \textit{abstract value set} or an \textit{abstract semantic domain} \( \text{Abs} \) as well as an abstraction function \( \alpha : \mathcal{P}(\text{Conc}) \rightarrow \text{Abs} \) and a concretization function \( \gamma : \text{Abs} \rightarrow \mathcal{P}(\text{Conc}) \) defining the correspondence between \( \text{Conc} \) and \( \text{Abs} \). \( \text{Abs} \) and \( \text{Conc} \) should be complete lattices with partial orders \( \sqsubseteq_{\text{Conc}} \) and \( \sqsubseteq_{\text{Abs}} \).

\( \alpha \) and \( \gamma \) are supposed to have the following natural properties (see e.g. [CC77], [JN95]):

- \( \alpha \) and \( \gamma \) are monotonic.
- \( \forall a \in \text{Abs} . \ a = \alpha(\gamma(a)) \).
- \( \forall c \in \text{Conc} . \ c \sqsubseteq_{\text{Conc}} \gamma(\alpha(c)) \).

As the programming language \( L \) has a standard semantics \([\cdot]\) that assigns a value \( v \in \text{Conc} \) to every well-formed expression \( e \) in \( L \), we can now define an \textit{abstract semantics} of \( L \). This abstract semantics \([\cdot]_A \) assigns an \textit{abstract value} \( v_A \in \text{Abs} \) to every well-formed expression \( e \) in \( L \).

---

3 Usually both the standard and the abstract semantics besides the language expression depend on a further value, e.g. an environment or a state. This is omitted here for simplicity.
In order to be a useful approximation of the standard semantics $\llbracket \cdot \rrbracket$ an abstract semantics $\llbracket \cdot \rrbracket_A$ has to fulfill the following soundness property:

**Definition 2.4.1 (soundness in abstract interpretation)** Let $f : C \to D$ be a concrete semantic operation with domains $C$ and $D$, let $\alpha_{C,A} : C \to A$ and $\alpha_{D,B} : D \to B$ be abstraction functions mapping $C$ to $A$ and $D$ to $B$, and let $\gamma_{A,C}$ and $\gamma_{B,D}$ be the corresponding concretization functions.

An abstract function $f^\# : A \to B$ is sound for $f$ if

$$\forall x \in C. \alpha_{D,B} \circ f(x) \sqsubseteq_B f^\#(\alpha_{C,A}(x)).$$

A notion of completeness of an abstract interpretation as used in [GRS00] is defined as follows:

**Definition 2.4.2 (completeness in abstract interpretation)** Let $f, f^\#$, $\alpha_{C,A}$, $\alpha_{D,B}$, $\gamma_{A,C}$ and $\gamma_{B,D}$ be as in Def. 2.4.1. $f^\#$ is complete for $f$ if

$$\forall x \in C. \alpha_{D,B} \circ f(x) =_B f^\#(\alpha_{C,A}(x)).$$

In order to get useful results from an abstract interpretation soundness as defined in Def. 2.4.1 is needed. If the completeness property from Def. 2.4.2 holds for an abstract interpretation one can trust in the precision of the yielded results. As known from [GRS00] completeness can always be achieved in principle. However, this is done by either extending or restricting the abstract domains. This can cause abstracting away information we are interested in or the consideration of details we want to abstract from.

### 2.5 Type Inference

This section summarizes some essential approaches to type inference in functional programming and parts of the history.

The idea of type inference was independently developed by Hindley [Hin69] and Milner [Mil78] (where the notions used in [Mil78] better correspond to those used in current work on type inference).

The aim of type inference is to detect errors of applying functions to inappropriate input values before runtime. Therefore, one introduces *types*, i.e. syntactic entities that denote certain sets of values of the semantic domain of the functional language. Types are built of:

- A set of base types (each denoting simple values with common properties), e.g. `num` denoting all numbers.
A set of type constructors used to construct complex types. (The values denoted by constructed types are constructed values like lists and functions.) An example for constructed types is list num denoting all lists with numbers as elements.

[Mil78] requires the sets of values denoted by different types to be disjoint, i.e. for every value there is exactly one type.

In order to allow functions that are applicable to all values with a common structure function types are allowed to contain type variables. E.g. for the function reverse we have the type

\[ \text{list } a \rightarrow \text{list } a \]

with a type variable \( a \). This type expresses the fact that reverse is applicable to all lists of any (fixed) element type and returns a list of the same element type. Functions of such a type are called polymorphic. The kind of polymorphism introduced by allowing type variables is called parametric polymorphism.

The idea of static type inference is that the types of functions, the types of their arguments, and the types of the application results influence and constrain each other. A program is well typed if the set of constraints given in this way has a solution. A well typed program cannot cause a type error at runtime.

By the introduction of static type inference the set of functional programming languages is divided into:

- **dynamically typed languages.** These languages are not influenced by static type inference. They carry a type tag for every value occurring during program execution and use these tags for type checking at runtime. In a dynamically typed language type errors may occur at runtime. Examples for such languages are Scheme [KCE98], [FMK95] and Common LISP [Ste90].

- **statically/strictly typed languages.** The strictly typed languages incorporate a static type checker into the language definition: A program that does not pass the type check is not a valid program in the language and must not be compiled/executed. These languages detect type errors before runtime already. Examples are Standard ML [Wik87], Miranda [Hin92] and Haskell [Tho96], [HPF97].

In order to make the benefits of static type checking available for dynamically typed functional programming the notion of soft-typing was introduced [CF91, Fag92], [MW97]. For soft-typing systems a rejection of ill typed (i.e. not well typed) programs is not possible because the programming language does not contain any typing constraints. Dynamically typed languages
often allow the (error-free) execution of programs that are not typable by a static type checker, and rejecting such ill typed programs by the type checker would also reject programs that are valid according to the language definition. Instead of rejecting programs a soft-typing system raises warnings for all program parts that are not well typed. These warnings provide the following information:

- Program parts without warnings cannot go wrong. The runtime type checks on these parts of the program can be dropped enhancing the program performance.
- The warnings spot those parts of the program that might contain type errors. They guide the programmer to program parts that should be checked manually.

Besides the usual type checking approach there is work to set based analysis for Scheme programs [Fla97, FF99]. It is mentioned here because its output is quite similar to soft-typing.

The research of type systems and type inference produced several extensions to [Mil78]:

- **Semantics:** For the types given as syntactic entities a precise semantics is given in [MPS86] and [APP89] for different type languages.

- **Subtyping:** The notion of subtyping is found e.g. in [FP91], [AW92, AW93], and [AC93]. The subtyping relation on types models the subset relation on the sets of denoted values. By this it became possible to provide union types and intersection types (e.g. [Dam94]).

- **Conditional types:** The concept of conditional types presented in [AWL94] has the aim to enhance the precision in typing case-expressions: Instead of uniting the result types of all cases the conditional type checker leaves out those cases whose conditions cannot be met by the type of the conditional expression.

- **Partial types:** This notion given in [Tha88, Tha94] introduces structures with non-uniform element types to static type inference. E.g. lists containing both integer numbers and boolean values are not a problem in LISP, but they cannot be used e.g. in ML without defining a new element data type that hold both integers and booleans.

- **Dependent types:** These are types that can depend on a term as e.g. lists of a certain length or trees with a given height depending on a natural number. Examples of type languages with dependent types can be found in [XP98, XP99] and [Aug98]. They can be used to solve e.g. the vector range check problem given in Ex. 1.2.1 (cf. [XP98]). However, the increased expressive power of type systems with dependent types makes
type inference difficult. The cited approaches differ from the complete type checking approach of this work by expecting some kind of explicit type information (type annotations or even completely explicit typing). Thus, the programs checked with dependent types have to be augmented or changed in order to benefit from the increased expressive power of the type language.

A last piece of work that is quite related to the work presented in the following chapters is given in [WC94, Wri94]. It presents a soft-typing system that has the capability to raise type errors besides type warnings: An error is risen instead of a warning for all program parts that not only might go wrong but must go wrong. In [Wri94] the condition for a type error is implementation dependent since it is defined in terms of a special type representation used throughout that work. It seems that the type representation used there has not been considered any further, possibly because the inferred types were quite difficult to understand by the programmer. Furthermore, there is just an ad hoc semantics given for soft-typing errors that is not obvious e.g. in the use of function types.
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. Section 4 introduces an algorithm which tests two given types for common elements. This test will play a central role in the type inference process described in Chapter 6.

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 $T_S(V)$ of all standard type terms (or standard types for short) over $V$ is defined as the smallest set fulfilling the following properties:

1. $B \subseteq T_S(V)$ where $B$ is the set of all type constants or base types introduced in Def. 3.1.12. Especially $\text{sym} \in B$.
2. $V \subseteq T_S(V)$.
3. $c \in K, e_1, \ldots, e_{a(c)} \in T_S(V) \Rightarrow (c \ e_1 \ldots e_{a(c)}) \in T_S(V)$ where $K$ is the set of type constructors as introduced in Def. 3.1.13.
4. $e_1, \ldots, e_{k \geq 0} \in T_S(V) \Rightarrow (\cup e_1 \ldots e_k), (\cap e_1 \ldots e_k) \in T_S(V)$.
5. $e \in T_S(V)$, $e$ does not contain a subterm $e' \in V \Rightarrow \mathcal{C}e \in T_S(V)$.
6. $e_1, e_2 \in T_S(V), e_2$ does not contain a subterm $e' \in V \Rightarrow e_1 \setminus e_2 \in T_S(V)$.
7. $T_{\text{func}}, T_{\text{funcP}}, T_{\text{funcU}} \in T_S(V)$ with the function types $T_{\text{func}}, T_{\text{funcP}}, T_{\text{funcU}}$ given in Def. 3.1.16.
8. $s \in \text{sym}, t \in T_S(V) \Rightarrow (\text{bind} \ s \ t) \in T_S(V)$ with the binding type constructor bind described in Def. 3.1.17.
9. $u_i = (\text{bind} \ s_i \ t_i) \in T_S(V), s_i \neq s_j$ for $i \neq j \Rightarrow (\text{frame} \ u_1 \ldots u_n) \in T_S(V)$ with the frame type constructor frame given in Def. 3.1.17.
10. $e$ is an environment type and $f$ a frame type $\Rightarrow (\text{env} \ e \ f) \in T_S(V)$ with the environment type constructor env introduced in Def. 3.1.19.
11. If $X \in V_f$ is a free variable and $t \in T_S(V)$ contains $X$ then $\mu X.t \in T_S(V)$. 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 $\mathcal{T}_{SCS}(V) \subset \mathcal{T}_S(V)$ of semi-closed standard types consists of all types not containing any free type variables $X \in V_f$ not bound by $\mu$.

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 $\mu$ 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 for 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 $j^{th}$ frame one can specify the $j^{th}$ 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 $p = p_1.p_2....p_k$ with $p_i \in \mathbb{N}$. The empty list is denoted by $\epsilon$. For a type $t \in \mathcal{T}_S$ and a position $p$ the subterm of $t$ at $p$, written $t|_p$, is either $t|_p = \text{undefined}$ or $t|_p \in \mathcal{T}_S$ defined as follows:

- $t|_{\epsilon} = t$.
- For $k \geq 1$, $t|_p = t'|_p$ with $p := p_2....p_k$ and $t'$ calculated as follows:
  - If $t = (\cup t_1 ... t_j)$, $t = (\cap t_1 ... t_j)$ or $t = (c t_1 ... t_j)$ with $c \in \mathbb{K}$ and if $p_1 \in \{1, ..., j\}$ then $t' = t|_{p_1}$.
  - If $t = t_1 \setminus t_2$ and $p_1 \in \{1, 2\}$ then $t' = t|_{p_1}$.
  - If $t = C\tilde{t}$ and $p_1 = 1$ then $t' = \tilde{t}$.
  - If $t = (\text{bind} \ s \ \tilde{t})$ then:
    - If $p_1 = 1$ then $t' = s$.
    - If $p_1 = 2$ then $t' = \tilde{t}$.
  - If $t = (\text{frame} \ u_1 u_2 ... u_j)$ and $p_1 \in \{1, ..., j\}$ then $t' = u|_{p_1}$.
  - If $t = (\text{env} \ e \ f)$ then:
    - If $p_1 = 1$ then $t' = f$.
    - If $p_1 > 1$ and $e|_{(p_1-1)} \neq \text{undefined}$ 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 \( t = \text{undefined} \), then \( t' := \text{undefined} \).

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 \( p = p_1.p_2.\ldots .p_m \) be a position in \( t \) and \( p' = p'_1.p'_2.\ldots .p'_n \) a position in \( t|_p \). The concatenation of \( p \) and \( p' \) defined by \( p.p' = p_1.p_2.\ldots .p_m.p'_1.p'_2.\ldots .p'_n \) 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

\[
\text{subterms}(t) = \{ t' \mid \exists p . t' = t|_p \neq \text{undefined} \}.
\]

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

\[
\text{subterms}(t) = \text{subterms}(t|_2)
\]

because especially \( t \in \text{subterms}(t|_2) \), and therefore \( \text{subterms}(t|_2) \) also contains all subterms of \( t \).

\( \text{subterms}(t) \) 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 \( A \):

\[
t = \mu X. (\cup \text{nil} (A . X))
\]
This term has the following subterms:

\[
\begin{align*}
t_{|\epsilon} &= \mu X. (\cup \text{nil} (A \cdot X)) \\
t_{|1} &= X \\
t_{|2} &= (\cup \text{nil} (A \cdot t)) \\
t_{|2,1} &= \text{nil} \\
t_{|2,2} &= (A \cdot t) \\
t_{|2,2,1} &= A \\
t_{|2,2,2} &= t 
\end{align*}
\]

Calculating subterms of \( t_{|2,2,2} \) does not yield any new terms because \( t_{|2,2,2} = t \). Therefore

\[\text{subterms}(t) = \{ \mu X. (\cup \text{nil} (A \cdot X)), X, (\cup \text{nil} (A \cdot t)), \text{nil}, (A \cdot t), A \}.\]

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 \( t = \mu X. (\cup \text{nil} (A \cdot X)) \) 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 \( t_{|2,2} = (A \cdot t) \) and \( (A \cdot t) = t_{|2,2} \) 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 \( t = \mu X. (\cup \text{nil} (A \cdot X)) \) 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 type \( t \).

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 \( t = \mu X.(\cup \text{nil} (A . X)) \) just has \( X \) as proper subterm.
- For the term \( t' = (\cup \text{nil} (A . t)) \) with \( t \) as above we get \( \{\text{nil}, (A . t), t, X\} \) 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 \( t|_p \neq \text{undefined} \). 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 \( B = \{\bot, O, \text{sym}, b_1, \ldots, b_v\} \) for some \( v \in \mathbb{N} \) is called the set of base types or type constants. \( \bot \) is called the empty type, \( O \) is called the zero type, and \( \text{sym} \) is called the type of all symbols.

**Definition 3.1.13 (free type constructors)** The set \( K = \{c_1, \ldots, c_w\} \) for some \( w \in \mathbb{N} \) is called the set of free type constructors or tuple-like type constructors. Every free type constructor \( c_i \in K \) has one or several arities denoted by \( a(c_i) \).

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. D.

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

The set \( K \) of type constructors contains the pair type constructor \((\ldots)\).

**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 $\text{Tfunc}$ is called the function type. The two restricted function types $\text{Tfunc}_P$ and $\text{Tfunc}_U$ 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)** $\text{bind}$ is called the binding type constructor. The generated symbol bindings have the form $(\text{bind} \ s \ t)$ where $s \in \text{sym}$ and $t$ is a type.
frame is called the frame type constructor. The frame types generated by this constructor have the form 
(frame $u_1 \ldots u_k$) for every $k \in \mathbb{N}$ where the $u_i = (\text{bind } s_i t_i)$ are symbol bindings of pairwise disjoint symbols $s_i$. For

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

we often use the notation

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

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 $F = (\text{frame } s_1 : t_1 \ldots s_k : t_k)$ 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 $(\text{frame } s_1 : t_1 \ldots s_k : t_k)$ 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  

$$F_1 = (\text{frame } x : \text{int } s : \text{string } f : Tfunc)$$  
$$F_2 = (\text{frame } x : \text{posint } y : \text{posint})$$

or in the alternative notation

$$F_1 = [x \mapsto \text{int}, s \mapsto \text{string}, f \mapsto Tfunc]$$  
$$F_2 = [x \mapsto \text{posint}, y \mapsto \text{posint}]$$

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

The type of all environments is denoted by $\text{env}$.

By Def. 3.1.19 all environment types have the form

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

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

$$(F_1 \ F_2 \ldots \ F_n)$$
for the environment

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

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

\textbf{Example 3.1.20 (environment types)} An example for an environment type is

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

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 $\mu$ has to be used.

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

\textbf{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

$$\mu X. (\cup \nil \ (A \cdot X))$$

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 $\mu X \cdot t$ 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 $\text{unfold}$ that explicitly performs such changes on a given term:

\textbf{Definition 3.1.23 (unfolding recursive bindings)} Let $t \in T_S$ be type. We define the function $\text{unfold}$ as $\text{unfold}(t) = t'$ 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:

\[
\begin{align*}
\text{unfold}^0(t) &= t \\
\text{unfold}^{1}(t) &= \text{unfold}(t) = \text{unfold}(\text{unfold}^0(t)) \\
&\quad \vdots \\
\text{unfold}^{k}(t) &= \text{unfold}(\text{unfold}^{k-1}(t))
\end{align*}
\]

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:

\[
\begin{align*}
\text{unfold}^0(\mu X. (\cup \text{nil} (A . X))) &= \mu X. (\cup \text{nil} (A . X)) \\
\text{unfold}^1(\mu X. (\cup \text{nil} (A . X))) &= (\cup \text{nil} (A . \mu X. (\cup \text{nil} (A . X)))) \\
\text{unfold}^2(\mu X. (\cup \text{nil} (A . X))) &= (\cup \text{nil} (A . (\cup \text{nil} (A . \mu X. (\cup \text{nil} (A . X)))))
\end{align*}
\]

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

\[
\top := \mu X. (\cup B \bigcup_{c \in K} (c \underbrace{X X \ldots X}_{a(c)}) \text{ frame env Tfunc})
\]

3.1.2 Semantics of the Standard Types

Defining the semantics for most types is straightforward. 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:

\[ l := ("mon" . ("tue" . ("wed" . ("thu" . ("fri" . ("sat" . ("sun" . ()))))))) \]

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:

\[ l' := \mu v. ("mon" . ("tue" . ("wed" . ("thu" . ("fri" . ("sat" . ("sun" . v))))))) \]

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

The semantics of closed types is defined by a semantic function \( \{\cdot\}_{c} : T_{c} \rightarrow P(V) \) where \( V \)
denotes the semantic domain, i.e. the set of all values expressible in a functional language. \( V \)
is defined as follows:

Definition 3.1.27 (semantic domain of types) The semantic domain \( V \) of types consists
of the following (pairwise disjoint) subsets:

- A set \( V_{S} \) of simple values.
- Sets \( V_{c} = \{(c \ v_{1} \ldots v_{k}) \mid v_{i} \in V\} \) for every type constructor \( c \) with arity \( k \). Thus, every
tuple-like type constructor corresponds to a free data constructor of the same arity.
- \( PF\text{Func} \) is the set of predefined functions.
- \( LC \) (\( \text{lambda}_{\text{closures}} \)) 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 \( \text{emptyenv} \)
and all pairs \( (e' \ f) \) with \( e' \in \text{Env} \) and \( f \in \text{Frame} \).
- The set \( V_{\text{rec}} \) 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 \( \mu \) 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:

\[ v := \mu x. (1 \cdot (2 \cdot (3 \cdot x))) \]

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 \( \epsilon \)). The cyclic data constructor \( \mu \) 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 \( \text{unfold}_V \) transforms \( v \) to a value \( v' \) as follows: \( \text{unfold}_V(v) = v' \) where \( v \) and \( v' \) differ exactly on the positions \( p \) fulfilling:

- \( v|_p = \mu x. \bar{v} \) 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:

\[
\begin{align*}
\text{unfold}_V^0(v) &= v \\
\text{unfold}_V^1(v) &= \text{unfold}_V(v) = \text{unfold}_V(\text{unfold}_V^0(v)) \\
& \quad \vdots \\
\text{unfold}_V^k(v) &= \text{unfold}_V(\text{unfold}_V^{k-1}(v))
\end{align*}
\]

The function \( \text{unfold}_V \) implies a equivalence relation induced by

\[ v_1 = \text{unfold}_V(v_2) \Rightarrow v_1 \equiv v_2. \]

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:

\[
\begin{align*}
\text{unfold}_V^0(\mu x. (1 \cdot (2 \cdot (3 \cdot x)))) &= \mu x. (1 \cdot (2 \cdot (3 \cdot x))) \\
\text{unfold}_V^1(\mu x. (1 \cdot (2 \cdot (3 \cdot x)))) &= (1 \cdot (2 \cdot (3 \cdot \mu x. (1 \cdot (2 \cdot (3 \cdot x))))) \\
\text{unfold}_V^2(\mu x. (1 \cdot (2 \cdot (3 \cdot x)))) &= (1 \cdot (2 \cdot (3 \cdot (1 \cdot (2 \cdot (3 \cdot \mu x. (1 \cdot (2 \cdot (3 \cdot x))))))))
\end{align*}
\]
In the following we define the semantic function \([t]_c\) for the different standard types. We often use the notation \(e : t\) for \(e \in [t]_c\).\(^4\)

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

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 \(b_1, \ldots, b_v\) there are sets \([b_i]_c\). For our example type language the sets \([b_i]_c\) are explained in App. D.

**Definition 3.1.32 (intersection existence property)** Let \(\mathcal{B}\) a set of base types and \([t]_c\) a semantic function. \(\mathcal{B}\) has the intersection existence property if the following holds:

If \(b, b' \in \mathcal{B}\) with \([b]_c \cap [b']_c \neq \emptyset\) then there exists some \(b' \in \mathcal{B}\) with \([b']_c = [b]_c \cap [b']_c\).

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

**Definition 3.1.33 (type constructors)** The semantic function \([t]_c\) is extended to constructed types as follows:

- \([\bigcup t_1 \ldots t_k]_c = (\bigcup [t_1]_c \ldots [t_k]_c)\). For \(k = 1\), \([\bigcup t_1]_c = [t_1]_c\) and for \(k = 0\) \([\bigcup]_c = [\bot]_c\).

- \([\bigcap t_1 \ldots t_k]_c = (\bigcap [t_1]_c \ldots [t_k]_c)\). For \(k = 1\), \([\bigcap t_1]_c = [t_1]_c\) and for \(k = 0\) \([\bigcap]_c = [\top]_c\).

- \([t_1 \setminus t_2]_c = [t_1]_c \setminus [t_2]_c\).

- \([Ct]_c = \mathcal{V} \setminus [t]_c =: \mathcal{C}_\mathcal{V}[t]_c\).

- For every tuple-like type constructor \(c_i\) (with \(i \in \{1, \ldots, w\}\)) there is a corresponding data constructor \(d_i\) such that \([\{c_i t_1 \ldots t_{a(c_i)}\}]_c = \{(d_i e_1 \ldots e_{a(c_i)}) \mid e_1 : t_1, \ldots, e_{a(c_i)} : t_{a(c_i)}\}\).

---

\(^4\)The notation \([\cdot]_c\) (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 \( t := (\text{posint} \cdot \text{lower-char}) \) we have 
\[
\{t\}_c = \{(v_1 \cdot v_2) \mid v_1 \in \{\text{posint}\}_c, v_2 \in \{\text{lower-char}\}_c\} = \{(1 \cdot a), (1 \cdot b), \ldots, (2 \cdot f), \ldots\}.
\]

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 \( \text{PFunc} \) of predefined function definitions and a set \( \text{LC} \) of user-defined function definitions. The semantics of the function types is given by:

\[
\begin{align*}
\{Tfunc_P\}_c & = \text{PFunc}. \\
\{Tfunc_U\}_c & = \text{LC}. \\
\{Tfunc\}_c & = \{Tfunc_P\}_c \cup \{Tfunc_U\}_c.
\end{align*}
\]

In the following we will use \textit{function} as an abbreviation of \textit{function definition}.

Knowledge of the internal structure of the semantic domains of function types is not essential at the moment. We will defer the definition of the internal structure to Def. 5.1.5 for the predefined functions and Def. 5.1.8 for the user-defined ones in Chapter 6.

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 \( f = (\text{frame} s_1 : t_1 \ldots s_n : t_n) \) the set \( \{f\}_c \) is the set of those frames \( f_v \in \text{Frame} \) that are defined exactly on the symbols \( s_1, \ldots, s_n \) and assign a value \( v_i \in \{t_i\}_c \) to \( s_i \).

For an environment type \( e = (\text{env} e' f) \) with another environment type \( e' \) and a frame type \( f \), \( \{e\}_c \) contains exactly those environments \( (e'_v f_v) \) that consist of a parent environment \( e'_v \in \{e'\}_c \) and a frame \( f_v \in \{f\}_c \).

\( \{\text{env}_0\}_c \) contains just the empty environment \textit{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:

\[
\{F_1\}_c = \{[x \mapsto v_1, s \mapsto v_2, f \mapsto v_3] \mid v_1 \in \{\text{int}\}_c, v_2 \in \{\text{string}\}_c, v_3 \in \{Tfunc\}_c\} = \\
= \{[x \mapsto 42, s \mapsto \text{"Hello world!"}, f \mapsto f_+], [x \mapsto -3, s \mapsto \text{"abc"}, f \mapsto f_{\text{map}}], \ldots\}.\]
\[ \langle F_2 \rangle_c = \{ [x \mapsto v_1, y \mapsto v_2] \mid v_1 \in \langle \text{posint} \rangle_c, v_2 \in \langle \text{posint} \rangle_c \} = \{ [x \mapsto 1, y \mapsto 1], [x \mapsto 256, y \mapsto 42], \ldots \} \]

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:

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

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 \( \sigma \) 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 \( \text{dom}(\sigma) = \{X_1, \ldots, X_k\} \).

A substitution is called idempotent if the assigned values \( t_X \) do not contain any variables \( Y \in \text{dom}(\sigma) \) 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 \( \sigma \) to \( t \), i.e. the type generated from \( t \) by replacing every free occurrence of \( x_i \in \text{dom}(\sigma) \) 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 $\sigma$ is appropriate for a type $t$ if $\sigma$ 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 $\sigma$ be an appropriate closed type substitution for $t$.

The semantic function for types with variables is $\langle \cdot \rangle : T \times TS \rightarrow V$. It is based on the semantic function for ground types as follows:

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

In the following we write $\langle t \rangle$ instead of $\langle t \rangle(\sigma)$ 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 $t \in T_S(V)$ a type, and let $\odot$ denote an additional type that does not contain any values (even not the non-termination $\bot$). Then the $k$th approximation of $t$ is

$\text{approx}^k(t) = \text{cut}(\text{unfold}^k(t))$

where $\text{cut}$ replaces every occurrence of a recursive type in its argument by $\odot$.

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

$\text{approx}^0(\mu X.(\cup \text{nil} (A . X))) = \odot$

$\text{approx}^1(\mu X.(\cup \text{nil} (A . X))) = (\cup \text{nil} (A . \odot))$

$\text{approx}^2(\mu X.(\cup \text{nil} (A . X))) = (\cup \text{nil} (A . (\cup \text{nil} (A . \odot))))$

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

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

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

40
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. \( \forall t \in T. \star : t \) (even \( \star : \diamond \)). The finite approximations of \( v \) are defined as follows:

\[
\text{approx}^k_V(v) = \text{cut}_V(\text{unfold}^k_V(t))
\]

where \( \text{cut}_V \) 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:

\[
\begin{align*}
\text{approx}^0_V(\mu x. (1 . (2 . (3 . x)))) &= \star \\
\text{approx}^1_V(\mu x. (1 . (2 . (3 . x)))) &= (1 . (2 . (3 . \star))) \\
\text{approx}^2_V(\mu x. (1 . (2 . (3 . x)))) &= (1 . (2 . (3 . (1 . (2 . (3 . \star))))) )
\end{align*}
\]

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

\[
\{ \mu X.t \} = \{ v \in V | \exists k \in \mathbb{N}. v \in \{ \text{approx}^k(\mu X.t) \} \} \cup \\
\quad \cup \{ v \in V_{\text{rec}} | \forall i \in \mathbb{N} \exists j \in \mathbb{N}. \text{approx}^i_V(v) \in \{ \text{approx}^j(\mu X.t) \} \}
\]

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 \( t = \mu X.(\cup \text{nil} \text{ (int . X))} \) we have \( \forall i \in \mathbb{N}. \text{approx}^i_V(v) \in \{ \text{approx}^{2i+1}(t) \} \) and therefore \( v \in \{ t \} \). Analogously the list \( l' \) of all week days as defined in Ex. 3.1.26 fulfills \( l' \in \{ \mu X.(\cup \text{nil} \text{ (string . X))} \} \).

### 3.1.3 Set Normalized Standard Types

The type constructors \( \cup, \cap, \setminus \) and \( C \) 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 4 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 $\setminus$.

2. There are no directly nested occurrences of the complement type constructor of the form $\mathcal{C}Ct'$.

3. For every occurrence $Ct'$ 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. 2.2:

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

\[
\begin{align*}
\setminus (t_1 t_2) & \rightarrow (\cap t_1 C t_2) \\
\mathcal{C} (\cup <a_1 \ldots e_1>) & \rightarrow (\cap <\mathcal{C}a_1 \ldots \mathcal{C}e_1>) \\
\mathcal{C} (\cap <a_1 \ldots e_1>) & \rightarrow (\cup <\mathcal{C}a_1 \ldots \mathcal{C}e_1>) \\
\mathcal{C}Ct & \rightarrow t
\end{align*}
\]

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 219.

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

**Proof:** See App. A.1, Page 220.

**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 220.
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 \( \langle t \rangle(\sigma) = \langle t' \rangle(\sigma) \) for every appropriate closed type substitution \( \sigma \).

**Proof:** See App. A.1, Page 221.

Example 3.1.57 (set normalized form) Consider the type

\[
t = num \setminus (\cup Cposreal \ interset-e).
\]

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

\[
t \rightarrow_1 (\cap num (\cup Cposreal \ interset-e)) \rightarrow_2 (\cap num (\cap C\itesp posreal \ Cinterset-e)) \rightarrow_4
\]

\[
\rightarrow_4 (\cap num (\cap posreal \ Cinterset-e))
\]

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 \( t \).
- \( \{ t, (\cup t \bot), (\cup t t), (\cap t) \} \) 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:

\[
\text{(let ((x 5)
(y 3)
(z 2))
(/ (- x (+ y z))))}
\]

When using the type \text{posint-e} (type of exact positive integers) for \(x\), \(y\) and \(z\) we can only infer the type \text{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 (which is essentially a singleton type as described in [Hay91]) allows to work directly on constant values as long as this is possible:

Definition 3.2.2 (value assignment) Let \(v \in V\) be a value with \(v : t_b\) for some base type \(t_b\). The value assignment \(\mathcal{A}(v)\) is a type with \(\langle \mathcal{A}(v) \rangle = \{v\}\). We identify \(\langle \mathcal{A}(v) \rangle\) with its only element \(v\). The set of all value assignments is denoted by \(VA\).

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
• $\llbracket t_1 \rrbracket \subseteq \text{dom}(f)$

• $f(\llbracket t_1 \rrbracket) \subseteq \llbracket t_2 \rrbracket$

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 $\llbracket t \rrbracket \subseteq \llbracket t_1 \rrbracket$ approximated by a subtype relation $\subseteq$ with

$$t \sqsubseteq t_1 \Rightarrow \llbracket t \rrbracket \subseteq \llbracket t_1 \rrbracket \subseteq \text{dom}(f)$$

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 $\llbracket t \rrbracket \cap \text{dom}(f) \neq \emptyset$, 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 $\llbracket t \rrbracket \cap \text{dom}(f) \neq \emptyset$, but $\llbracket t \rrbracket \cap \llbracket t_1 \rrbracket = \emptyset$ raises a type error.

**Example 3.3.1** According to the usual definition of function types the unary division operator should have the type $\text{num} \setminus \text{zero} \rightarrow \text{num}$. 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.

The type checker of Chap. 6 based on the type language described here uses 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 $\text{IN}_i \rightarrow \text{OUT}_i$ with $\text{IN}_i, \text{OUT}_i \in T$ such that:

• $\text{dom}(f) \subseteq \bigcup_i \llbracket \text{IN}_i \rrbracket$
\[ \forall i. f(\langle IN_i \rangle) \subseteq \langle OUT_i \rangle \cup \{ \text{error} \} \] (where error denotes a type error caused by applying a function to an inappropriate argument).

The set of all I/O-representations is denoted by io.

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 \( v \in \text{dom}(f) \) 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-representation of the function add where every line represents one I/O-representation pair, e.g. one element of the I/O-representation set:

\[
\begin{align*}
\text{posint} \times \text{posint} & \rightarrow \text{posint} \\
\text{int} \times \text{int} & \rightarrow \text{int} \\
\text{num} \times \text{num} & \rightarrow \text{num} \\
\text{string} \times \text{string} & \rightarrow \text{string}
\end{align*}
\]

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 type \( PIN_i \). Furthermore, when an output type \( POUT_i \) is inferred for a given input \( PIN_i \), we can just expect the property \( f(\langle PIN_i \rangle) \cap \langle POUT_i \rangle \neq \emptyset \) 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 \( PIN_i \rightarrow POUT_i \) with \( PIN_i, POUT_i \in \mathcal{T} \) such that:

\[
\begin{align*}
\forall i. \text{dom}(f) \cap \langle PIN_i \rangle & \neq \emptyset \\
\forall i. f(\langle PIN_i \rangle) \cap \langle POUT_i \rangle & \neq \emptyset
\end{align*}
\]
The set of all PI/PO-representations is denoted by $\text{pipo}$.

**Example 3.3.5 (PI/PO-representation)** Consider the following function $\text{map1}$ 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)))))
```

A PI/PO-representation for $\text{usemap1}$ is

$$\{A \overset{\Rightarrow}{\rightarrow} \text{num} \times (\text{list } A) \rightarrow \text{num}\}$$

From the use of $l$-new in $\text{usemap1}$ we know that the output type of the first argument of $\text{usemap1}$ 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.

We can now refine the definition of the set $\mathcal{T}$ to contain I/O-representations and PI/PO-representations:

**Definition 3.3.6 (set of all types)** Definition 3.2.3 is refined as follows: $\text{io, pipo} \subset \mathcal{T}$. The subtypes $\mathcal{T}_{SC}(V)$, $\mathcal{T}_{C}(V)$ and $\mathcal{T}_{G}(V)$ are extended analogously. All types containing elements of $\mathcal{T}$ as arguments are changed correspondingly.

Note that I/O-representations as well as PI/PO-representations look similar to refinement types [FP91] and intersection types [Pie96]. Indeed comparable to these type definitions every I/O-representation pair or PI/PO-representation pair expresses a subset of the properties a denoted function must all have. They however differ in the following details:

- While the function types in [FP91] and [Pie96] have input types that denote a *subset* of the domain of every denoted function the input types of
  - an I/O-representation pair denotes a *superset* of the denoted function’s domains.
  - a PI/PO-representation pair denotes a set of values that must just have common elements with the denoted function’s domains.

47
The output type of a PI/PO-representation pair need not cover all possible output values but just some of them.

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 the restriction of 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 presence 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. 4 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. D 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:

\[
STbase(b_1, b_2) = \text{true} \Rightarrow \langle b_1 \rangle \subseteq \langle b_2 \rangle
\]

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. 4. 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 $ST(t_1, t_2) := \text{true}$.
2. If $t_1 = \bot$ or $t_2 = \top$ then $ST(t_1, t_2) := \text{true}$.
3. If $t_1$ and $t_2$ are base types or value assignments then $ST(t_1, t_2) := ST_{base}(t_1, t_2)$.
4. If $t_1 \in \{Tfunc_P, Tfunc_U\}$ and $t_2 = Tfunc$ then $ST(t_1, t_2) := \text{true}$.
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 $f_1 = (\text{frame}\ s_1 : t_1 \ldots s_k : t_k)$ and $f_2 = (\text{frame}\ s'_1 : t'_1 \ldots s'_{k'} : t'_{k'})$ with $\{s_1, \ldots, s_k\} = \{s'_1, \ldots, s'_{k'}\}$ then
   \[
   ST(f_1, f_2) := \forall_{i \in \{1, \ldots, k\}} \exists_{j \in \{1, \ldots, k'\}} s_i = s'_j \land ST(t_i, t'_j).
   \]
7. If $e_1 = (\text{env}\ e'_1 \ f_1)$ and $e_2 = (\text{env}\ e'_2 \ f_2)$ then
   \[
   ST(e_1, e_2) := ST(e'_1, e'_2) \land ST(f_1, f_2).
   \]
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 $ST(t_1, t_2) := \text{true}$.
   (b) If $\mathcal{NB}$ and $t_1 = \mu X. t'_1$ then $ST(t_1, t_2) := ST(\text{unfold}(t_1), t_2)$. 

49
(c) If \(N\mathcal{B}\) and \(t_2 = \mu X.t_2'\) then \(ST(t_1, t_2) := ST(t_1, \text{unfold}(t_2))\).

9. For the set operations the following correspondences hold:

(a) If \(N\mathcal{B}\) and \(t_1 = (\cup t_{1,1} \ldots t_{1,k})\) then

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

(b) If \(N\mathcal{B}\) and \(t_2 = (\cup t_{2,1} \ldots t_{2,k})\) then

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

(c) If \(N\mathcal{B}\) and \(t_1 = (\cap t_{1,1} \ldots t_{1,k})\) then

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

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

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

(e) If \(N\mathcal{B}\) and \(t_1 = Ct_1'\) and \(t_2 = Ct_2'\) then \(ST(t_1, t_2) := ST(t_2', t_1')\).

(f) If \(N\mathcal{B}\) and \(t_2 = Ct_2'\) and \(CE(\tau(t_1), t_2') = \emptyset\) then \(ST(t_1, t_2) := \text{true}\) where \(\tau\) instantiates every variable \(X \in V_q\) occurring in \(t_1\) with \(\top\).

10. If \(N\mathcal{B}\) then \(ST(t_1, t_2) = \text{false}\).

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:

\[
t_1 \sqsubseteq t_2 \iff ST(t_1, t_2).
\]

\(^7\text{There are no common elements between }\tau(t_1)\text{ and }t_2'\text{ detected by }CE.\)
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:

\[
\begin{align*}
\text{int} & \subseteq \text{num} \\
(int \ . \ string) & \subseteq (num \ . \ string) \\
(\cup \ env \ (int \ . \ string)) & \subseteq (\cup \ env \ (num \ . \ string))
\end{align*}
\]

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

Example 3.4.4 Consider the types

\[t_1 = ((\cup \ posint \ negint) \ . \ string) \text{ and } t_2 = (\text{num} \ . \ string).\]

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

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

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

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

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

\[ST(((\cup \ posint \ negint) \ . \ string), (\text{num} \ . \ string)) = \text{true}.\]

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. 4. More exactly there are chains of recursive calls of the form

\[ST \rightarrow CE \rightarrow S-CE \rightarrow ST.\]

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.

---

8We use $\doteq$ to denote syntactic equality of function calls (e.g. $ST$) in contrast to $=$ denoting the equality of the results.
Assumption 3.4.5 (termination assumption for CE) The algorithm CE terminates for every input.

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

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 222.

A proof for the termination of CE used in Lemma 3.4.7 is given in Theorem 4.3.14.

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

$$ST(t_1, t_2) = \text{true} \Rightarrow \langle \sigma(t_1) \rangle \subseteq \langle \sigma(t_2) \rangle$$

for every closed type substitution $\sigma$ appropriate for $t_1$ and $t_2$.

Proof: See App. A.2, Page 222.

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 $\sigma$ such that $\langle \sigma(t_1) \rangle \cap \langle \sigma(t_2) \rangle \neq \{\bot\}$? 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 $t_1 = (\text{int} \ . \ X_\forall)$ and $t_2 = (\text{posint} \ . \ Y)$ 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 $\rho_q = \{X_\forall \leftarrow \text{int}\}$ and a value $v = (5 \ . \ 42)$ and get

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

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

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

On the other hand $t_1$ and $t_3 = (\text{posint} \ . \ \text{negint})$ 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

$$\exists \rho_f \forall \rho_q (\text{dom}(\rho_f) \subseteq V_f \land \text{dom}(\rho_q) \subseteq V_q \land \forall X_\forall \in \text{dom}(\rho_q). \rho_q(X_\forall) \neq \bot \land$$

$$\land \rho_q \circ \rho_f \text{ is appropriate for } t_1, t_2) \Rightarrow$$

$$\Rightarrow \exists v \in \langle \rho_q \circ \rho_f(t_1) \rangle \cap \langle \rho_q \circ \rho_f(t_2) \rangle \land v \neq \bot \quad (4.1)$$
We call every \( v \) with the properties given in (4.1) a common element of \( t_1 \) and \( t_2 \) and denote the set of all common elements of \( t_1 \) and \( t_2 \) by \( t_1 \diamond t_2 \).

Note that quantified variables are treated in a special way in Def. 4.0.10: 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 4.1 contains the preliminaries used to define \( CE \). In Sec. 4.2 an algorithm \( S-CE \) is introduced that calculates constraints on the variable instantiations. The algorithm \( CE \) presented in Sec. 4.3 transforms these constraint sets to idempotent substitutions \( \sigma \).

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 \( CE_{base} \) 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 \( \sigma \) is a substitution returned by \( CE \), then \( \sigma \) just assigns types to free type variables. This motivates the following definition:

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

\[
\text{dom}(\sigma) \subseteq V_f.
\]

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 \(t \in T\). 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 \(\text{dom}(\sigma) = \{X_1, \ldots, X_n\}\) 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 \(\sigma\) be a free constraint set. A free type substitution \(\sigma'\) is compatible with \(\sigma\) if for every \(X\) with \(X \leftarrow t \in \sigma\) the following holds:

- \(X \in \text{dom}(\sigma')\)
- \(\llbracket t \rrbracket \subseteq \llbracket \sigma'(X) \rrbracket\)

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

For every free constraint set \(\sigma'\) there is the natural free type substitution \(\sigma\) of \(\sigma'\) defined as follows:

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

\[
\text{subst}(\sigma') = \{X \leftarrow \sigma'(X) \mid X \in \text{dom}(\sigma')\}.
\]

For a c-collection \(\Sigma'\) the natural s-collection of \(\Sigma'\) is

\[
\text{subst}(\Sigma') = \{\text{subst}(\sigma') \mid \sigma' \in \Sigma'\}.
\]
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 $\sigma_1$ and $\sigma_2$ be two free constraint sets. Their union is defined as

$$\sigma_1 \otimes \sigma_2 = \{ A \leftarrow (\cup t_1 t_2) \mid A \leftarrow t_1 \in \sigma_1 \land A \leftarrow t_2 \in \sigma_2 \} \cup \{ A \leftarrow t_1 \mid A \leftarrow t_1 \in \sigma_1 \land A \not\in \text{dom}(\sigma_2) \} \cup \{ A \leftarrow t_2 \mid A \leftarrow t_2 \in \sigma_2 \land A \not\in \text{dom}(\sigma_1) \}.$$ 

We can now define the union of c-collections:

**Definition 4.1.8 (union of c-collections)** Let $\Sigma_1$ and $\Sigma_2$ be two c-collections the union of $\Sigma_1$ and $\Sigma_2$ is defined as

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

We use the same operator $\otimes$ for both free constraint sets and c-collections. When identifying free constraint sets $\sigma$ 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 $CE_{base}$ to decide the common element question for base types and value assumptions.

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

$$\langle b_1 \rangle \cap \langle b_2 \rangle \neq \{\bot\} \Rightarrow CE_{base}(b_1, b_2) = true.$$ 

There is an algorithm also called $CE_{base}$ that terminates for every input and returns the value of the function $CE_{base}$.

Especially $CE_{base}$ fulfills

$$CE_{base}(b_1, b_2) = true \text{ if } (b_1 = \top \land b_2 \neq \bot) \lor (b_2 = \top \land b_1 \neq \bot)$$

but

$$CE_{base}(b_1, b_2) = false \text{ if } b_1 = \bot \lor b_2 = \bot.$$ 

App. D contains a definition of $CE_{base}$ 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 $S$-$CE(t_1, t_2, \sigma, r)$ where

- $t_1$ and $t_2$ are the types that are checked.
- $\sigma$ 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

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

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 \( \text{SCE-list-reduce} \) expects the following arguments:

- A list \( L \) of tuples \((t_1,t_2)\) where \( t_1 \) and \( t_2 \) are types.
- A c-collection \( \Sigma \).
- A recursion history \( r \).

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

\[
\begin{align*}
\text{SCE-list-reduce}((), \Sigma, r) & \quad \Sigma \\
\text{SCE-list-reduce}((t_i, t'_i, l_{i+1}, \ldots, l_k), \Sigma, r) & \quad \text{SCE-list-reduce}((l_{i+1}, \ldots, l_k), \bigcup_{\sigma \in \Sigma} S-CE(t_i, t'_i, \sigma, r), r)
\end{align*}
\]

\((SCE\text{-list-reduce1})\)

\((SCE\text{-list-reduce2})\)

**Example 4.2.2 (SCE-list-reduce)** Consider the following call to \( \text{SCE-list-reduce} \):

\[
\text{SCE-list-reduce}(((A, \text{string}), (\text{num}, \text{int})), \{\}, r)
\]

This call is processed by rule \((SCE\text{-list-reduce2})\). 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 \text{string} as arguments, i.e.

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

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

\[
\text{SCE-list-reduce}(((\text{num}, \text{int})), \Sigma', r)
\]

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

\[
\Sigma'' := \Sigma' = S-CE(\text{num}, \text{int}, \sigma', r)
\]

The resulting c-collection \( \Sigma'' \) is used in the recursive call

\[
\text{SCE-list-reduce}((), \Sigma'', r)
\]

to \( \text{SCE-list-reduce} \). Since the list of type pairs is empty rule \((SCE\text{-list-reduce1})\) applies and \( \Sigma'' \) is returned.
For changing a constraint in a given constraint set the functions \textit{extend-constraint} and \textit{replace-constraint} can be used:

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

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

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

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

Note that \textit{extend-constraint} and \textit{replace-constraint} behave equivalently if $X \not\in \text{dom}(\sigma)$.

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 \textit{constrain-all-free} and \textit{free-to-top}:

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

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

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

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

The following subsections describe the actions of \textit{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 is formalized in the rules \((Top1)\) and \((Top2)\):

\[
\begin{align*}
S-CE(\top, t_2, \sigma, r) \quad &\{\text{constrain-all-free}(t_2, \sigma)\} \\
t_2 \neq \bot & \quad \text{\((Top1)\)}
\end{align*}
\]

\[
\begin{align*}
S-CE(t_1, \top, \sigma, r) \quad &\{\text{constrain-all-free}(t_1, \sigma)\} \\
t_1 \neq \bot & \quad \text{\((Top2)\)}
\end{align*}
\]

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 is formalized by rule \((RecT)\).

\[
\begin{align*}
S-CE(t_1, t_2, \sigma, r) \quad &\{\sigma\} \\
(t_1, t_2) \in r & \quad \text{\((RecT)\)}
\end{align*}
\]

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

\[
\begin{align*}
S-CE(t_1 = \mu X. t'_1, t_2, \sigma, r) \quad &\text{combine-cs}(S-CE(unfold(t_1), t_2, \sigma, ((t_1, t_2) \cdot r))) \\
&(Rec1)
\end{align*}
\]
The function \( \text{combine-cs} \) used in (Rec1) and (Rec2) expects and returns a \( c \)-collection and is defined as follows:

\[
\text{combine-cs}(\emptyset) = \emptyset \quad \text{and} \quad \text{combine-cs}(\Sigma) = \left\{ \bigotimes_{\sigma \in \Sigma} \sigma \mid \emptyset \neq \Sigma' \subseteq \Sigma \right\} \quad \text{for } \Sigma \neq \emptyset
\]

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

The non-empty subsets of \( \Sigma \) are

\[
\begin{align*}
\Sigma_1 &= \{ \sigma_1 \} \\
\Sigma_2 &= \{ \sigma_2 \} \\
\Sigma_3 &= \Sigma.
\end{align*}
\]

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

\[
\begin{align*}
\bigotimes_{\sigma \in \Sigma_1} &= \sigma_1 \\
\bigotimes_{\sigma \in \Sigma_2} &= \sigma_2 \\
\bigotimes_{\sigma \in \Sigma_3} &= \sigma_3 \text{ with } \sigma_3 = \{ X \leftarrow (\cup \text{int bool}), Y \leftarrow \text{int}, Z \leftarrow \text{bool} \}.
\end{align*}
\]

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

The reason for applying \( \text{combine-cs} \) on the result of the recursive subcall becomes obvious in the following example:

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

\[
\begin{align*}
t_1 &= \mu X.((\cup f(\text{bool}) f(\text{int})) . (\cup \text{nil} X)) \\
t_2 &= \mu Y.(f(A) . (\cup \text{nil} Y))
\end{align*}
\]

\(^1\text{Some of the rules used for the example are defined afterwards. We just sketch the results here.} \)
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

$$S\text{-}CE(t'_1, t'_2, \emptyset, r) \text{ with } t'_1 = \text{unfold}(t_1), t'_2 = \text{unfold}(t_2) \text{ and } r = ((t'_1, t_2) (t_1, t_2))$$

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

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

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 $\Sigma_1$ is returned.

Unfortunately, there is a value $v = (\#t. (5 . \text{nil}))$ with $v \in t_1 \sqcap t_2$ (e.g. with $\rho_f = \{A \leftarrow (\cup \text{bool} \text{int})\}$ and $\rho_q = \emptyset$), but $v \not\in \sigma(t_1) \sqcap \sigma(t_2)$ for all $\sigma \in \Sigma_1$.

When we assume the rule (RecT) not to exist, the processing of the call in Example 4.2.8 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 $SCE\text{-}list\text{-}reduce$ generates the free constraint sets $\sigma_{1,1} = \{A \leftarrow (\cup \text{bool} \text{bool})\}$ and $\sigma_{1,2} = \{A \leftarrow (\cup \text{bool} \text{int})\}$ from $\{A \leftarrow \text{bool}\}$ and $\sigma_{2,1} = \{A \leftarrow (\cup \text{int} \text{bool})\}$ and $\sigma_{2,2} = \{A \leftarrow (\cup \text{int} \text{int})\}$ from $\{A \leftarrow \text{int}\}$. 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 $\sigma$ 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:

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

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

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 $\{\{X \leftarrow (\cup \text{bool} \ Z), Y \leftarrow (\cup \text{int} \ Z)\}\}$. with a new free variable $Z$.

Note that the introduction of $Z$ is necessary in order to preserve e.g. the common element $v = \#(42 \ 40\ 5)$ 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 $\sigma$ is updated to contain $t_2$:

$$S$-$CE(t_1, t_2, \sigma, r) \xrightarrow{\{\text{constrain-all-free}(t_2, \text{extend-constraint}(t_1, t_2, \sigma))\}} t_1 \in V_f, t_2 \notin V_f \text{ (Var1)}$$

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

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

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

Note that without applying $\text{constrain-all-free}$ the result in this example is $\Sigma'_2 = \{\sigma'_2\}$ with $\sigma'_2 = \{X \leftarrow (Y . \text{int}), Y \leftarrow \text{int}\}$. While the value $v = (((\#t . 168) . 42)$ is a common element of $t_1$ and $t_2$ under $\sigma_2$ it is not a common element when applying $\sigma'_2$ 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:

\[
S-CE((\cup t_{1,1} \ldots t_{1,k}), t_2, \sigma, r) \quad \subseteq \quad \bigcup_{i=1}^{k} S-CE(t_{1,i}, t_2, \sigma, r) \quad (U1)
\]

\[
S-CE(t_1, (\cup t_{2,1} \ldots t_{2,k}), \sigma, r) \quad \subseteq \quad \bigcup_{i=1}^{k} S-CE(t_1, t_{2,i}, \sigma, r) \quad (U2)
\]

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 \(SCE\text{-list-reduce}\) given in Def. 4.2.1 on page 58:

\[
S-CE((\cap t_{1,1} \ldots t_{1,k}), t_2, \sigma, r) \quad \subseteq \quad SCE\text{-list-reduce}(((t_{1,1}, t_2), \ldots, (t_{1,k}, t_2)), \{\sigma\}, r) \quad (I1)
\]

\[
S-CE(t_1, (\cap t_{2,1} \ldots t_{2,k}), \sigma, r) \quad \subseteq \quad SCE\text{-list-reduce}(((t_1, t_{2,1}), \ldots, (t_1, t_{2,k})), \{\sigma\}, r)) \quad (I2)
\]

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

**Example 4.2.11** Let \(t_1 = (A \cdot \text{nil})\) and \(t_2 = (\cap (\text{num} \cdot \text{nil}) (\text{bool} \cdot \text{nil}))\). For this pair of types \(S-CE\) returns the c-collection \(\{A \leftarrow (\cup \text{num bool})\}\) 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 \(((\cap \text{num bool}) \cdot \text{nil})\) with an additional check for emptiness of the resulting intersection in Ex. 4.2.11 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 \( \top \):

\[
S-CE(Ct'_1, t_2, \sigma, r) \quad \{\sigma'\} \quad \sigma'(t_2) \not\subseteq t'_1, \sigma' := \text{free-to-top}(t_2, \sigma) \quad (\text{Comp1})
\]

\[
S-CE(t_1, Ct'_2, \sigma, r) \quad \{\sigma'\} \quad \sigma'(t_1) \not\subseteq t'_2, \sigma' := \text{free-to-top}(t_1, \sigma) \quad (\text{Comp2})
\]

Note that \( \sigma'(t_2) \not\subseteq t'_1 \) abbreviates the test \( \neg ST(\sigma'(t_2), t'_1) \). \( \sigma' \) is generated from \( \sigma \) 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 \( \not\subseteq \)-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 \( t_1 = \text{Cint} \) 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 \( \sigma' = \text{free-to-top}(t_2, \emptyset) = \{X \leftarrow \top, Y \leftarrow \top\} \) and performs the test \( \neg ST((\top . \top), \text{int}) \). 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 \( SCE\text{-}list\text{-}reduce \):

\[
S-CE((c \ t_{1,1} \ldots t_{1,k}) (c \ t_{2,1} \ldots t_{2,k}), \sigma, r) \quad \overset{\text{Constr}}{\Rightarrow} \quad \text{SCE\text{-}list\text{-}reduce}(((t_{1,1}, t_{2,1}), \ldots, (t_{1,k}, t_{2,k})), \{\sigma\}, r))
\]

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:

\[
S-CE(F_1, F_2, \sigma, r), [s_1 \mapsto t_{1,i}, \ldots, s_k \mapsto t_{k,i}] \in fs(F_i) \text{ for } i \in \{1, 2\}
\]

\[
SCE-list-reduce(((t_{1,1}, t_{2,1}), \ldots, (t_{1,k}, t_{2,k})), \{\sigma\}, r))
\]

(\text{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:

\[
S-CE((F_{1,1} \ldots F_{1,k}), (F_{2,1} \ldots F_{2,k}), \sigma, r)
\]

\[
SCE-list-reduce(((F_{1,1}, F_{2,1}), \ldots, (F_{1,k}, F_{2,k})), \{\sigma\}, r))
\]

(\text{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:

\textbf{Definition 4.2.13} The function \(fq : T \times T \rightarrow \{\text{true, false}\}\) expects two types \(t_1\) and \(t_2\) and returns \text{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 \text{false}.

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

\[
S-CE(t_1, t_2, \sigma, r)
\]

\[
\{\sigma\}
\]

\(fq(t_1, t_2) = \text{true}
\]

(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 \( C\text{E}_{\text{base}} \) fulfilling Assumption 4.1.9.

The behaviour of \( S\text{-CE} \) is formalized in the following rule:

\[
\frac{S\text{-CE}(t_1, t_2, \sigma, r)}{\{\sigma\}} \quad CE_{\text{base}}(t_1, t_2) = \text{true} \quad (\text{Base})
\]

4.2.13 Integrating the Cases

In this subsection we will present the algorithm \( S\text{-CE} \) by integrating the previously introduced rules in a fixed order:

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

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

If none of these rules is applicable \( S\text{-CE} \) returns the empty c-collection \( \emptyset \).

4.2.14 An \( S\text{-CE} \) Example in Detail

The following example shows in detail how calls to \( S\text{-CE} \) are processed and how the use of unions by \textit{extend-constraint} enables \( S\text{-CE} \) to process heterogeneous lists correctly.

**Example 4.2.15 (\( S\text{-CE} \))** Consider the call \( S\text{-CE}(t_1, t_2, \sigma, r) \) with:

\[
\begin{align*}
t_1 &= \mu X. (\cup \text{nil}(A \cdot X)) \\
t_2 &= (\text{posint} \cdot (\text{string} \cdot \text{nil})) \\
\sigma &= \emptyset \\
r &= ()
\end{align*}
\]

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

\[
S\text{-CE}(\cup \text{nil}(A \cdot t_1), t_2, \emptyset, r' := ((t_1, t_2))).
\]
2. By rule (U1) the S-CE-call is recursively split into two subcalls:
   (a) \( S\text{-CE}(\text{nil}, t_2, \emptyset, r') = \emptyset \)
   (b) \( S\text{-CE}((A : t_1), t_2, \emptyset, r') \)

3. \( S\text{-CE}((A : t_1), t_2, \emptyset, r') \) is processed by rule (Constr). SCE-list-reduce performs the following subcalls (the result of subcall number \( i \) is denoted by \( \Sigma_i \)):
   (a) \( \Sigma_1 = S\text{-CE}(A, \text{posint}, \emptyset, r') = \{\{A \leftarrow \text{posint}\}\} \) by rule (Var1).
   (b) \( \Sigma_2 = S\text{-CE}(t_1, (\text{string} . \text{nil}), \sigma' := \{A \leftarrow \text{posint}\}, r') \)

4. \( S\text{-CE}(t_1, (\text{string} . \text{nil}), \sigma', r') \) is processed by rule (Rec1) resulting in the call
   \( S\text{-CE}((\cup \text{nil} (A : t_1)), (\text{string} . \text{nil}), \sigma', r'' = ((t_1, (\text{string} . \text{nil})), (t_1, t_2))) \).

5. By rule (U1) we have two subcalls for the contained call to S-CE:
   (a) \( S\text{-CE}(\text{nil}, (\text{string} . \text{nil}), \sigma', r'') = \emptyset \)
   (b) \( S\text{-CE}((A : t_1), (\text{string} . \text{nil}), \sigma', r'') \)

6. \( S\text{-CE}((A : t_1), (\text{string} . \text{nil}), \sigma', r'') \) is processed by rule (Constr):
   (a) \( \Sigma'_1 = S\text{-CE}(A, \text{string}, \sigma', r'') = \{\sigma'' := \{A \leftarrow (\cup \text{posint string})\}\} \)
   (b) \( \Sigma'_2 = S\text{-CE}(t_1, \text{nil}, \sigma'', r'') \)

7. The second call is processed by rule (Rec1) producing the call
   \( S\text{-CE}((\cup \text{nil} (A : t_1)), \text{nil}, \sigma'', r'' := ((t_1, \text{nil}), (t_1, (\text{string} . \text{nil})), (t_1, t_2))) \).

8. By rule (U1) the call to S-CE causes the following subcalls:
   (a) \( S\text{-CE}(\text{nil}, \text{nil}, \sigma'', r''' = \{\sigma''\} \) by rule (Base).
   (b) \( S\text{-CE}((A : t_1), \text{nil}, \sigma'', r''' = \emptyset \)

   The call containing the union \((\cup \text{nil} (A : t_1))\) in step (7) yields the result \(\{\sigma''\}\). This is also the result for \(\Sigma'_{2}\).

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 \(\Sigma_2\) 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 229.

The remaining condition on $ST$ for termination of $S$-$CE$ is removed later. The unrestricted termination result is given in Cor. 4.3.16.

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:

![Diagram](image-url)

Figure 4.1: open $S$-$CE$ calculations
Definition 4.2.17 (open S-CE calculation) Let
\[ \tilde{c} = S-CE(\tilde{t}_1, \tilde{t}_2, \tilde{\sigma}, \tilde{r}) \]
be a call to S-CE and
\[ c' = S-CE-list-reduce((t_1, t'_1), \ldots, (t_n, t'_n), \Sigma', r') \]
a call to S-CE-list-reduce occurring as (maybe indirect) subcall of \( \tilde{c} \). Let the first argument of S-CE-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
\[ c_{j,l} = S-CE(t_j, t'_j, \sigma_{j,l}, r) \]
be the calls to S-CE initiated by S-CE-list-reduce when processing the \( j \)th list element and let
\[ C_j = \{c_{j,1}, \ldots, c_{j,m_j}\} \]
be the calls to S-CE initiated by S-CE-list-reduce when processing the \( j \)th list element and let
\[ \Sigma_j = S-CE(t_j, t'_j, \emptyset, r) \]
for every \( j \in \{1, \ldots, n\} \).

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. 4.1). 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 S-CE-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 S-CE-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. 4.2.18 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 of all corresponding open S-CE calculations are enumerated from 1 to \( n \).

Definition 4.2.18 (implicit constraint sets) Let \( c := S-CE(t, t', \sigma, r) \) be a call to S-CE steaming from an initial call \( \tilde{c} := S-CE(\tilde{t}, \tilde{t}', \emptyset, ()) \). 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 \( c_{j,l} = S-CE(t_j, t'_j, \sigma_{j,l}, r) \) and let \( \Sigma_j = S-CE(t_j, t'_j, \emptyset, r) \) for every \( j \in \{1, \ldots, n\} \).

Then the implicit constraint set of \( c \) with respect to \( \tilde{c} \) is
\[ \text{implicit-cs}_{\tilde{c}}(c) := \bigotimes_{j=1,\ldots,n} \Sigma_j. \]
If the set of open S-CE calculations of $c$ with respect to $\tilde{c}$ is empty, we define

$$\text{implicit-cs}_c(\tilde{c}) := \{\emptyset\}.$$ 

The following Lemma 4.2.19 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. 4.2.1 is equivalent to calculating the implicit constraint set of the initial call. The constraints generated by the subcalls to $SCE$ 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 $SCE$ and $c'$ the first call to $SCE$-list-reduce performed when processing $\tilde{c}$. Let $\Sigma$ be the result of a call

$$c' = SCE$-list-reduce$((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)$$

and let $C_j$ denote the set of recursive calls to $SCE$ when processing the $j$th list element. Let $\Sigma_j$ be the c-collection used in the recursive call

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

for all $j$ and $\Sigma'_j = SCE(t_j, t'_j, \emptyset, r)$ then:

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

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

**Proof:** See App. B.1, Page 231.

The following lemma states the main property of $SCE$’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 $SCE$ with $t_1, t_2$ and $\tilde{\sigma}$ as arguments and a recursion history $r$ that is not arbitrary chosen but generated by $SCE$ starting with an initial call with empty recursion history. Then the result contains a free constraint set $\sigma$ that essentially restrict $t_1$ and $t_2$ to types containing $v$.

But the free constraint sets $\sigma$ (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 $\sigma$ 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 4.2.20: if during evaluating the call $S$-CE$(\tilde{t}_1, \tilde{t}_2, \emptyset, ())$ there is a subcall $S$-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 $\sigma \in \Sigma = S$-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 $t_1, t_2 \in T$, both in set normalized form and let $c := S$-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

$$\forall k \in \mathbb{N}. v \in \text{subst}(\tilde{\sigma})^k(t_1) \square \text{subst}(\tilde{\sigma})^k(t_2). \quad (4.2)$$

Then there exist a free constraint set

$$\sigma \in \text{combine-cs-cond}_{(c, \tilde{c})}(S$-CE$(t_1, t_2, \tilde{\sigma}, r) \otimes \text{implicit-cs}(c)) \tag{4.3}$$

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

$$\forall k \in \mathbb{N}. v \in \sigma'^k(t_1) \square \sigma'^k(t_2). \quad (4.3)$$

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

$$\{\tilde{t}_X\}(\tau) \subseteq \{t_X\}(\tau) \quad (4.4)$$

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

**Proof:** See App. B.1, Page 235.

Note that the second part of (4.2) in Lemma 4.2.20 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 $(\text{list } X_Y)$ has common elements with $(\text{list } \top)$ but not with $(\text{list } Y_Y)$.

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

Though Lemma 4.2.20 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

\[ S-CE(t_1, t_2, \emptyset, ()) \]

with \( t_1 = (\cup \text{ int bool}), t_2 = (\cap \text{ int bool}) \).

When decomposing the union first, the result is given by the union of the results of the two subcalls \( S-CE(\text{int}, t_2, \emptyset, ()) \) and \( S-CE(\text{bool}, t_2, \emptyset, ()) \). 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

\[ \text{SCE-list-reduce}(((t_1, \text{int}), (t_1, \text{bool})), \emptyset, ()) \]

is returned. \( \text{SCE-list-reduce} \) performs the subcalls

\[ S-CE(t_1, \text{int}, \emptyset, ()) \text{ and } S-CE(t_1, \text{bool}, \emptyset, ()) \].

Both of these calls return \( \Sigma_0 = \{\emptyset\} \). The result of \( \text{SCE-list-reduce} \) is \( \Sigma_0 \). 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 4.2.22:

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 too late, constructors that have to be decomposed early can be hidden by the \( \mu \)-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 \( v \in \langle \tau(t_1) \rangle \cap \langle \tau(t_2) \rangle \) for an arbitrary substitution \( \tau \) 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 \( t_1 = (\cup t_{1,1} \ldots t_{1,k}) \) 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

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

Decomposing the union first unifies 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

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

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 \((\text{Rec1}), (\text{Rec2}) \rightarrow (\text{Var1}), (\text{Var2}) \rightarrow (\text{U1}), (\text{U2}) \rightarrow (\text{I1}), (\text{I2}) \rightarrow (\text{Comp1}), (\text{Comp2})\) in which \(S-\text{CE}\) applies the cases that just depend one one of the argument types. The order of the other cases can be chosen arbitrarily (except of \((\text{RecT})\) that must be checked before \((\text{Rec1})\) and \((\text{Rec2})\) in order to guarantee termination.

### 4.2.17 Further Optimizations

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

In the proof of Lemma 4.2.20 the fact that \((\text{Rec1})\) and \((\text{Rec2})\) apply \(\text{combine-cs}\) was just used for proving the correctness of \((\text{RecT})\). Indeed, when a recursive subcall generated by \((\text{Rec1})\) or \((\text{Rec2})\) returns without applying \((\text{RecT})\), 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 \((\text{Rec1})\) or \((\text{Rec2})\) and therefore the application of \(\text{combine-cs}\) is unnecessary.

**Example 4.2.23** Consider the types

\[
\begin{align*}
t_1 &= \mu X. ((\cup f(\text{bool}) f(\text{int})) \cdot (\cup \text{nil} X)) \\
t_2 &= (f(A) \cdot (f(A) \cdot \text{nil})).
\end{align*}
\]

The call \(S-\text{CE}(t_1, t_2, \emptyset, ())\) is processed by rule \((\text{Rec1})\) initiating the subcall \(S-\text{CE}(t'_1, t_2, \emptyset, r)\) with \(t'_1 = \text{unfold}(t_1)\) and \(r = (t_1, t_2)\). This call is decomposed by rule \((\text{Constr})\) and the first subcall \(S-\text{CE}((\cup f(\text{bool}) f(\text{int})), f(A), \emptyset, r)\) processed by \((\text{U1})\) and \((\text{Constr})\) and \((\text{Var2})\) for every subcall yields \(\Sigma_1 = \{\{A \leftarrow \text{bool}\}, \{A \leftarrow \text{int}\}\}\).

For every free constraint set \(\sigma \in \Sigma_1\) the function \(SCE\text{-list-reduce}\) initiates a subcall

\[
S-\text{CE}(\cup \text{nil} t_1), (f(A) \cdot \text{nil}), \sigma, r).
\]

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

\[
S-\text{CE}(t'_1, (f(A) \cdot \text{nil}), \sigma_1, r') \text{ with } r' = ((t_1, (f(A) \cdot \text{nil})), (t_1, t_2))\).
\]

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

\[
\Sigma_{2,1} = \{\{A \leftarrow (\cup \text{bool} \cdot \text{bool})\}, \{A \leftarrow (\cup \text{bool} \cdot \text{int})\}\}.
\]
The two subcalls $S-CE((\cup \text{nil} \ t_1), \text{nil}, \sigma, r')$ for $\sigma \in \Sigma_{2,1}$ just return $\sigma$. Thus, $\Sigma_{2,1}$ is returned.

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

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

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 4.2.8. 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 $S-CE(t'_1, t'_2, \emptyset, ((t'_1, t_2), (t_1, t_2)))$. 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 $\Sigma$, $S-CE$ has to return a pair $(\Sigma, \mathcal{R})$ where $\Sigma$ 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 $S-CE(t_1, t_2, \sigma, r)$ 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
    $$(\text{combine-cs}(\Sigma), \mathcal{R} \setminus \{(t_1, t_2)\}).$$
  - 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.

\[\text{Note: } t_1 = \mu X.((\cup f(\text{bool}) \ f(\text{int})).(\cup \text{nil} \ X)), \quad t_2 = \mu Y.(f(A).(\cup \text{nil} \ Y)), \quad t'_i = \text{unfold}(t_i) \text{ as in Example 4.2.8.}\]
Example 4.2.25 Consider the following two types

\[
t_1 = \mu X.(\cup \text{nil} ((\text{int} \cdot A) \cdot X))
\]
\[
t_2 = ((\cup (\text{int} \cdot \text{posint}) (\text{int} \cdot \text{bool})) \cdot \text{nil})
\]

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

1. \text{nil} and \(t_2\)
2. \(((\text{int} \cdot A) \cdot t_1)\) 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:

- \((\text{int} \cdot A)\) and \((\cup (\text{int} \cdot \text{posint}) (\text{int} \cdot \text{bool}))\)
- \(t_1\) and \(\text{nil}\)

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

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

\[
\Sigma' = \{\sigma_1, \sigma_2, \{A \leftarrow (\cup \text{posint bool})\}\}
\]

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 \( \mu \) 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 \( \{ X ← f(X) \} \) 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 \( \{ X ← \mu Y.f(Y) \} \).

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 \( \Sigma = \{ \sigma \} \)

\[
\{\{X_1 ← f(X_2), X_2 ← g(X_3), X_3 ← h(X_1)\}\}.
\]

The dependencies between the variables are given in Fig. 4.2 where an arrow from a variable \( V \) to a type \( t \) expresses that \( V \) is constrained to \( t \).
The intended result of CE is an s-collection consisting of the idempotent free type substitution
\[
\{X_1 \leftarrow f(g(\mu V_3 \cdot h(f(g(V_3))))),
X_2 \leftarrow g(\mu V_3 \cdot h(f(g(V_3)))),
X_3 \leftarrow \mu V_3 \cdot h(f(g(V_3)))\}.
\]

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 \( \sigma' \) in the result \( \Sigma' \) of S-CE into a free type substitution. All these free type substitutions are collected in an s-collection \( \Sigma \).

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

**Algorithm:** CE

**Input:** Two terms \( t_1 \) and \( t_2 \).

**Output:** An s-collection \( \Sigma \).

\[
\Sigma' := S-CE(t_1, t_2, \emptyset, ())
\]
\[
\Sigma := \emptyset
\]

forall \( \sigma' \in \Sigma' \) do (* Transform free constraint sets into free type substitutions *)
\[
\sigma'' := GIS(\sigma') (* generate idempotent substitution from \( \sigma' \) *)
\]
\[
\Sigma := \Sigma \cup \{\sigma''\} (* collect result substitutions in s-collection \( \Sigma \) *)
\]
end (* forall *)

return \( \Sigma \)

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 $\sigma$
Output: An idempotent substitution $\tilde{\sigma}$.

$G := (V, R)$ with $V = \text{dom}(\sigma')$ and $R = \{(y, x) \mid x \neq y, x \leftarrow t \in \sigma' \text{ and } y \text{ is subterm of } t\}$
$G' := (V', R')$ is the component graph of $G$. (* cf. Def. 2.3.4 *)
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 $\sigma'$ *)
    $t' := \sigma''(t)$ (* apply already known substitution $\sigma''$ *)
    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 $v'$.
    $\sigma_r := \{X \leftarrow t \mid X \in \tilde{V}, X \leftarrow \tilde{t} \in \sigma', t = \sigma''(\tilde{t})\}$ (* $\sigma_r = \sigma'' \circ \sigma'|_{\tilde{V}}$ *)
    $\tilde{\sigma} := \text{SMR}(\sigma_r)$
    $\sigma'' := \sigma'' \cup \tilde{\sigma}$
  end (* if-then-else *)
end (* while *)
return $\sigma''$

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 $\prec_V$ 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 $\prec_V$ 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 $X \prec Y$. 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 $X_i \prec_V X_{i+1}$ 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 $t' := t$
  $\sigma := \sigma \setminus \{X_i \leftarrow t\} \cup \{X_i \leftarrow t'\}$ (* $\sigma := \sigma|_{\text{dom}(\sigma)\setminus\{X_i\}} \cup \{X_i \leftarrow t'_{X_i}\} *$
  for $j = i + 1$ to $k$ do (* remove $X_i$ from all $t_{X_j}$ with $j > i *$
    $\sigma := (\sigma|_{\{X_i\}} \circ \sigma|_{\{X_{i+1},\ldots,X_k\}}) \cup \sigma|_{\{X_1,\ldots,X_i\}}$ *
    $t_{X_j} := \sigma(X_j)$
    $t'_{X_j} := t_{X_j}[X_i/\sigma(X_i)]$
    $\sigma := \sigma \setminus \{X_j \leftarrow t_{X_j}\} \cup \{X_j \leftarrow t'_{X_j}\}$
  end (* for $j$ *)
end (* for $i$ *)
end (* for $k$ downto $1$ *)

In every $t_{X_j}$ replace every $\mu\tilde{X}_i.t$ in a position where $\tilde{X}_i$ is already bound by some $\mu$ constructor.

Example 4.3.6 (SMR) Consider a call to SMR with the input substitution $\sigma$ 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:

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

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

$$\{X_1 \leftarrow f(X_2), X_2 \leftarrow g(\mu V_3 \cdot h(f(g(V_3)))), X_3 \leftarrow \mu V_3 \cdot h(f(g(V_3)))\}.$$
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 4.2.15 we get the following example for CE:

**Example 4.3.7 (CE)** Consider $t_1$ and $t_2$ as defined in Ex. 4.2.15 and a call CE($t_1, t_2$). This call causes the subcall to $S$-CE discussed in Ex. 4.2.15 and yields the result calculated there:

$$\Sigma' := \{A \leftarrow (\cup \text{ string posint})\}$$

For $\sigma' = \{A \leftarrow (\cup \text{ string posint})\}$, 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, $\sigma'' := \{A \leftarrow (\cup \text{ string posint})\}$ and $\sum := \{\sigma''\}$ is returned.

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

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

$$t_1 = (X \cdot (Y \cdot (Z \cdot \text{nil})))$$

and

$$t_2 = ((Y \cdot X) \cdot (Z \cdot ((\text{num} \cdot X) \cdot \text{nil}))).$$

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

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

The graph $G$ calculated in GIS for the only element $\sigma'$ of $\Sigma'$ is given in Fig. 4.3. 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 $\sigma'$ as argument. The individual iterations of the first $i$-loop performs the following changes:
1. $X$ is recursively bound by a $\mu$ constructor in $t_X = (Y . X)$ yielding $t'_X = \mu X.(Y . X)$.
   In $t_Z$ $X$ is replaced by $t'_X$ yielding $(num . \mu X.(Y . X))$.

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 $(num . \mu X.(Z . X))$.

3. The occurrence of $Z$ in $t_Z$ from the step before is recursively bound by $\mu$. The resulting term is $\mu Z.(num . \mu X.(Z . X))$

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

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

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 $t_Z = \mu Z.(num . \mu X.(Z . X))$.

2. $Y$ is replaced by $t_Y$ from the step before in $t_X$ yielding $\mu X.(\mu Z.(num . \mu X.(Z . X)).X)$.

1. Nothing to do.

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

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

The resulting substitution is the only element of the s-collection \(\Sigma\) returned by CE, i.e.:

\[
\Sigma = \{ \{ X \leftarrow \mu X. (\num (Z . X)) . X \}, \\
Y \leftarrow \mu Z. (\num . \mu X (Z . X)), \\
Z \leftarrow \mu Z. (\num . \mu X (Z . X)) \}
\]

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

\[
\Sigma = \{ \{ X \leftarrow \mu V. (\num (W . V)) . V \}, \\
Y \leftarrow \mu W. (\num . \mu V (W . V)), \\
Z \leftarrow \mu W. (\num . \mu V (W . V)) \}
\]

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

### 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 \(\text{dom}(\sigma)\).

**Proof:** See App. B.2, Page 243.

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

\[
\langle [\sigma \circ \sigma'(t)](\phi) = \langle [\sigma(t)](\phi)
\]

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

**Proof:** See App. B.2, Page 243.
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 245.

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

1. σ′ does not contain a variable binding A ← B with B ∈ dom(σ′).
2. If σ′ contains a variable binding A ← C[B] with a context C and a variable B ∈ dom(σ′), then there exists a variable B’ ∈ dom(σ′) such that B is bound to B’ or a union containing B’ in σ′.

Let v be a value fulfilling

\[ \forall k \in \mathbb{N} . v \in \sigma^k(t_1) \boxplus \sigma^k(t_2) \]

and let σ = GIS(σ′). Then

\[ v \in \sigma(t_1) \boxplus \sigma(t_2) . \]

Proof: See App. B.2, Page 245.

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.


We can now prove the unrestricted termination of CE. With Lemma 4.3.13 given, the proof consists of proving the termination of the loop

\[ CE \rightarrow S-CE \rightarrow ST \rightarrow CE \]

(4.5)

between the mutually dependent algorithms CE, S-CE and ST. Figure 4.4 shows the termination dependencies. An arrow from an Algorithm 1 to an Algorithm 2 expresses that the termination of Alg. 2 depends on termination of Alg. 1. The corresponding lemma is given under every arrow.
Informally, the termination proof for \textit{CE} (Theorem 4.3.14) uses the fact that every execution of the loop given in (4.5) 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 \textit{CE})** The algorithm \textit{CE} terminates for every pair of input types in set normalized form.

**Proof:** See App. B.2, Page 248.

Given Theorem 4.3.14 we can now easily conclude the termination of \textit{ST} for every input. This extends the termination proof given in Lemma 3.4.7.

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

**Proof:** See App. B.2, Page 249.

Analogously, we can extend Lemma 4.2.16 as follows:

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

**Proof:** See App. B.2, Page 249.

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

$$v \in t_1 \boxtimes t_2.$$  

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

$$v \in \sigma(t_1) \boxtimes \sigma(t_2).$$

**Proof:** See App. B.2, Page 249.
Chapter 5

Syntax and Standard Semantics of the Functional Language

In this chapter syntax and standard semantics of a functional programming language are defined. This programming language will later be used as the source language for input programs to a type checking process presented in Chap. 6. Though the standard semantics presented here is well known from the programming language Scheme [KCE98] we will give a detailed description of it. Many of the principles given in this description will carry over to an abstract semantics implementing the type checker in Chap. 6.

The chapter starts by presenting a core component of the functional programming language in Sec. 5.1. The syntax of this core language is given in Subsec. 5.1.1. The standard semantics follows in Sec. 5.1.2. In Sec. 5.2 syntax and standard semantics of several extensions to the core language are presented. These extensions are destructive updates in Subsec. 5.2.1, structured data in Subsec. 5.2.2, and environments as first class values in Subsec. 5.2.3.

5.1 Defining the Source Language

In this section we define the main subset $CS^0$ of the functional language $CS$ that serves as the source language for our analysis task. It is essentially given as a core part of the functional language Scheme.
5.1.1 The Syntax

**Definition 5.1.1 (syntax of CS^0)** The syntax of CS^0 is given by the following sets of expressions, constants and symbols:

\[
\begin{align*}
\text{Expression}^0 & \quad \rightarrow \quad c \\
& \quad | \quad x \\
& \quad | \quad (e_0 \ e_1 \ldots \ e_k) \\
& \quad | \quad (\text{if } e_0 \ e_1 \ e_2) \\
& \quad | \quad (\lambda(x_1 \ldots \ x_k) \ e)
\end{align*}
\]

\[
\begin{align*}
\text{ConstT} & \quad (\text{Basic Constant Terms}) \\
x \in \text{Sym} & \quad (\text{Symbols})
\end{align*}
\]

The sets ConstT and Sym are disjoint.

An expression in CS^0 is either a constant term, a symbol, an application, an if-expression, or a lambda-expression. As usual in Scheme, lambda expressions can bind more than one symbol at once.

All occurrences of a symbol \(x_i\) in the lambda-expression \((\lambda(x_1 \ldots \ x_k) \ e)\) is called bound by \(\lambda\). All other occurrences of symbols are called free. We call an expression closed if it does not contain any free symbols.

To simplify the representation of this work CS^0 does not contain the Scheme constructs cond, and and or. Essentially, expressions built by these constructs can be replaced by iterated use of if-expressions as follows:

\[
\begin{align*}
\text{(cond (test1 result1)} \\
\quad \text{(test2 result2)} \\
\quad \ldots \\
\quad \text{(testk resultk)} \\
\quad \langle(\text{else result-else}\rangle \\
\text{)}
\end{align*}
\]

(with optional expression parts enclosed in \(<\ldots>\)) can be rewritten as:

\[
\begin{align*}
\text{(if test1 result1}
\end{align*}
\]
The and-expression

(and arg1 arg2 ... argk)

can be rewritten as

(if arg1
    (if arg2 ...
        (if argk-1
            argk
            #f)
        #f)
    #f)

or-expressions can be transformed analogously.

It is sometimes important to precisely identify certain expressions or a certain occurrence of an expression. To do this easily we follow [Fla97] in introducing a label for every expression:

**Definition 5.1.2 (labels of expressions)** Let $e \in Expression^0$ be an expression. Every subexpression $e'$ of $e$ carries a unique label $l \in Label$. Especially, if $e'$ and $e''$ are subexpressions of $e$ that are syntactically equal but occur at different positions in $e$ then $e'$ and $e''$ carry different labels.

For an expression $e$ carrying the label $l$ we write $l : e$. The set of all labels contained in an expression $e$ (including the label of $e$ itself) is denoted by $Label(e)$.

For simplicity of the notation we omit the labels of expressions when they are not needed.

The language $CS^0$ will be extended in several ways in Sec. 5.2. Each of the extensions will carry a new superscript instead of 0. Superscripts as e.g. in $Expression^0$ denote the current language a subdefinition belongs to. If no such superscript occurs, then the current definition is used (even extended by redefinitions done later).
5.1.2 The Standard Semantics

We give a denotational semantics of the functional language syntactically defined by $CS^0$ using the notations of [Sch86]. First of all we need a some semantic domains:

First of all there are several primitive domains that will be used to denote constant terms.

**Definition 5.1.3 (primitive domains)** $PD = \{D_1, \ldots, D_p\}$ denotes the set of all primitive domains. (We do not specify the primitive domains any further at the moment.) The union of all primitive domains is denoted by $Const$. The values of $Const$ are called constant values.

Some primitive domains have special properties or are represented in a special way. These domains are now defined in more detail:

**Definition 5.1.4 (zero value)** The primitive domain $O$ just contains the zero value also denoted by $O$. It denotes a meaningless dummy value.

The zero value will especially be used by functions performing side effects in Sec. 5.2.1. It is used when no return value is needed but the functions work is done by side effects.

The predefined function terms will be denoted by predefined function definitions that are given by a special primitive domain called the predefined function domain. (In the following we will sometimes speak of functions instead of function definitions.)

**Definition 5.1.5 (predefined functions)** $PFunc$ is a special primitive domain called the domain of predefined functions. The elements of $PFunc$ are denoted by $f_{<xyz>}$ where $<xyz>$ represents the symbol $f_{<xyz>}$ is initially bound to. For every $f \in PFunc$ the domain of $f$ is denoted by $\text{dom}(f)$.

Another primitive domain with special properties is the domain of boolean values defined as follows:

**Definition 5.1.6 (domain of booleans)** The domain $\text{bool}$ containing the values $\#t$, $\#f$, and $\bot$ is called the domain of boolean values. ($\bot$ stands for a non-terminating computation and is introduced in Def. 5.1.9.)

The set $Const$ of all primitive domain values is used by the function $\text{denoteconst}$ that assigns a value from a primitive domain to every constant term:

**Definition 5.1.7 (denoting constant terms)** The function

$$\text{denoteconst} : ConstT \rightarrow Const$$
assigns a constant value to every constant term.

To denote lambda-expressions we need a domain of lambda-closures that is given by the following definition:

**Definition 5.1.8 (lambda closures)** LC denotes the domain of lambda-closures where a lambda-closure is of the form \( lc(e(x_1, \ldots, x_k), E) \) with an expression \( e \in \text{Expression}^0 \), symbols \( x_1, \ldots, x_k \in \text{Sym} \), and an environment \( E \in \text{Env} \) (see Def. 5.1.14).

To get more complex domains we need some domain constructors that generate compound domains from simpler element domains. The domain constructors used here are given by the following definition:

**Definition 5.1.9 (domain constructors)** Domain constructors are the product constructor, the sum constructor and the constructor for lifted domains defined as follows:

- For domains \( A_1, \ldots, A_k \) the product domain constructor generates a domain of the form \(( A_1 \ldots A_k)\) that contains all tuples \((a_1 \ldots a_k)\) where \( a_i \in A_i \) for all \( i \in \{1, \ldots, k\} \).

- For domains \( A_1, \ldots, A_k \) the sum domain \((+ A_1 \ldots A_k)\) contains all pairs \((m(A_i), a_i)\) with \( a_i \in A_i \) for some \( i \in \{1, \ldots, k\} \). \( m(A_i) \) is a unique name for the domain \( A_i \).

- For every domain \( A \) the lifted domain \( A_\perp \) contains all values from \( A \) and additionally the value \( \perp \) denoting non-termination of a computation (or an unknown computation result).

In the following we assume a value \((m(A_i), a_i) \in (+ A_1 \ldots A_k)\) to behave exactly like \( a_i \in A_i \). Thus, we can omit functions for disassembling values when working on sum domains. Furthermore, we assume every domain to be lifted and identify the \( \perp \)-values steaming from different argument domains in a sum domain.

The semantics of symbols and terms containing symbols depends on an environment mapping symbols to values. To define environments we first explain what an assignable value is:

**Definition 5.1.10 (assignable values)** The set of all assignable values (expressible by \( CS^0 \)) is

\[
\text{AssValue}^0 := \text{PFunc} \cup \text{LC} \cup \text{Const}
\]

The set \( \text{Value} \) of values that is needed during evaluating an expression of \( CS^0 \) is defined as follows:
Definition 5.1.11 (set of all values) The set \( \text{Value} \) of all values is defined by
\[
\text{Value} := \text{AssValue} \cup \{\text{error}, \bot\}
\]
i.e. \( \text{Value} \) contains all values from \( \text{AssValue} \) and the additional values \( \text{error} \) to denote erroneous evaluations and \( \bot \) to denote non-terminating evaluations.
This definition of \( \text{Value} \) yields \( \text{Value}^0 \) and several extensions corresponding to extensions of \( \text{AssValue}^0 \).

In Scheme not only \#t and \#f can occur as results of a test of e.g. if. Every value \( v \in \text{AssValue} \) can occur and is interpreted as a boolean value. This boolean interpretation is defined as follows:

Definition 5.1.12 (boolean interpretation) The boolean interpretation of values is given by a function
\[
b_i : \text{AssValue} \cup \{\bot\} \rightarrow \text{bool} \cup \{\bot\}
\]
with \( b_i(\bot) = \bot \).

Example 5.1.13 (boolean interpretation) According to [KCE98] we have
\[
b_i(\#f) = \text{false} \quad \text{and} \quad b_i(v) = \text{true}
\]
for all \( v \in \text{AssValue} \setminus \{\#f\} \). (Alternatively \( b_i \) can be defined to interpret \( () \) as false.)

Using the set of assignable values we can define the set of environments as follows:

Definition 5.1.14 (environments) A clean environment \( E' \in \text{EnvC} \) is a partial function
\[
E' : \text{Sym} \rightarrow \text{AssValue}
\]
from symbols to assignable values. For every clean environment \( E' \in \text{EnvC} \) the domain \( \text{dom}(E') \subset \text{Sym} \) is finite.

For every clean environment \( E' \) there is a (simple) environment which is a total function \( E : \text{Sym} \rightarrow \text{AssValue} \cup \{\text{undef}\} \) defined as follows:
\[
E(x) = \begin{cases} E'(x) & \text{if } x \in \text{dom}(E') \\ \text{undef} & \text{otherwise} \end{cases}
\]
The set of all (simple) environments is denoted by \( \text{Env}^0 \).

For \( E \in \text{Env} \) we define \( E[y_1 \mapsto v_1, \ldots, y_k \mapsto v_k] \in \text{Env} \) as the environment behaving like \( E \) except \( E[y_1 \mapsto v_1, \ldots, y_k \mapsto v_k](y_i) = v_i \). i.e.
\[
E[y_1 \mapsto v_1, \ldots, y_k \mapsto v_k](x) = \begin{cases} v_i & \text{if } x = y_i \text{ for } i \in \{1, \ldots, k\} \\ E(x) & \text{otherwise} \end{cases}
\]
Now we can explain how expressions of \( CS^0 \) are transformed to values from \( Value^0 \):

**Definition 5.1.15 (the standard semantics of \( CS^0 \))** The semantics of expressions \( e \in Expression \) under an environment \( E \in Env \) is given by the semantic function

\[
\llbracket \cdot \rrbracket_0 : Expression^0 \times Env^0 \to Value.
\]

For the semantics of \( e \in Expression^0 \) under \( E \) we write \( \llbracket e \rrbracket_0(E) \).

The definition of \( \llbracket e \rrbracket_0(E) \) is done by case distinction on the arguments \( e \) and \( E \). The results for the individual cases are given in Tab. 5.1 on Page 94 and Tab. 5.2 on Page 95.

As the definition shows the semantics of a constant term is given by the low level function \( denoteconst \) mapping the terms to constant values. The semantics of a symbol \( x \) is given by the environment \( E \) passed to \( \llbracket \cdot \rrbracket \) as the second argument. If \( x \) is defined in \( E \) then the meaning of \( x \) is given by the value \( v \) that is assigned to \( x \) in \( E \) by \( (Inst) \). If \( x \) is not bound in \( E \) then the meaning of \( x \) is the special value \( error \) (by \( (Inst-Err) \)) indicating an error situation.

The meaning of an \( if \)-expression is given by the rules \( (If-True) \), \( (If-False) \), \( (If-Err) \) and \( (If-Strict) \). \( (If-True) \) and \( (If-False) \) reduce the \( if \)-expression to the meaning of its second or third argument, respectively, depending whether the first argument \( v \) is interpreted as \( \#t \) or \( \#f \). \( (If-Err) \) is applied whenever the interpretation of the first argument fails due to an error and \( (If-Strict) \) defines the expressions \( (if \ e \ e_1 \ e_2) \) as strict in the first argument.

\( (Lambda) \) evaluates lambda-expressions to lambda closures. A lambda-closure is written as \( lc(e(x_1, \ldots, x_k), E) \) and consists of the body \( e \) of the lambda-expression, the tuple \( (x_1, \ldots, x_k) \) of symbols used as formal parameters of the lambda-expression and the definition environment the lambda-expression was evaluated in. \( (App-Lambda) \) reduces applications of lambda-closures by reducing the body \( e \) of the lambda-expression in the definition environment modified by assigning the semantic meaning of the argument expressions \( e_i \) of the application to the symbols \( x_i \) given as parameters in the lambda-closure.

Applications to predefined functions are processed like shown by \( (App-Pre) \) and \( (MisApp-Pre) \) where \( (App-Pre) \) applies a predefined function \( f \) (that is the semantic meaning of the first element of the application expression) to a value tuple \( (\llbracket e_1 \rrbracket_0(E), \ldots, \llbracket e_k \rrbracket_0(E)) \) in the domain of \( f \) and \( (MisApp-Pre) \) generates an error for applying a predefined function with an inappropriate argument tuple. If there is an argument that evaluates to \( error \) then the whole application expression evaluates to \( error \) as given by \( (App-Err) \). \( (App-Strict) \) expresses the strictness of the application in all argument positions.
\[
c \in ConstT \Rightarrow [c]_0(E) = \text{denoteconst}(c) \quad \text{(Const)}
\]
\[
x \in Sym, E(x) \neq \text{undef} \Rightarrow [x]_0(E) = E(x) \quad \text{(Inst)}
\]
\[
x \in Sym, E(x) = \text{undef} \Rightarrow [x]_0(E) = \text{error} \quad \text{(Inst-Err)}
\]
\[
b_i([e]_0(E)) = \text{true} \Rightarrow [(if \ e e_1 e_2)]_0(E) = [e_1]_0(E) \quad \text{(If-True)}
\]
\[
b_i([e]_0(E)) = \text{false} \Rightarrow [(if \ e e_1 e_2)]_0(E) = [e_2]_0(E) \quad \text{(If-False)}
\]
\[
[e]_0(E) = \text{error} \Rightarrow [(if \ e e_1 e_2)]_0(E) = \text{error} \quad \text{(If-Err)}
\]
\[
b_i([e]_0(E)) = \bot \Rightarrow [(if \ e e_1 e_2)]_0(E) = \bot \quad \text{(If-Strict)}
\]
\[
x_1, \ldots, x_k \in Sym \Rightarrow \begin{align*}
[(\lambda (x_1 \ldots x_k) \ e)]_0(E) &= \text{lc}(e(x_1, \ldots, x_k), E) \\
\end{align*} \quad \text{(Lambda)}
\]
\[
\exists i. x_i \notin Sym \Rightarrow \begin{align*}
[(\lambda (x_1 \ldots x_k) \ e)]_0(E) &= \text{error} \\
\end{align*} \quad \text{(Lambda-Err)}
\]
\[
[e_0]_0(E) = \text{lc}(e(x_1, \ldots, x_k), E'), \quad \begin{align*}
\{ [e_i]_0(E) \mid i \in \{1, \ldots, k\} \} & \Rightarrow [e_0]_0(E'[x_1 \mapsto [e_1]_0(E), \ldots, x_k \mapsto [e_k]_0(E)]) \quad \text{(App-Lambda)}
\end{align*}
\]
\[
[e_0]_0(E) = \text{lc}(e(x_1, \ldots, x_k), E'), \quad \begin{align*}
n \neq k, \bot \notin \{ [e_i]_0(E) \mid i \in \{1, \ldots, n\} \}, & \Rightarrow [e_0]_0(E') = \text{error} \quad \text{(LApp-Err)}
\end{align*}
\]
\[
[e_0]_0(E) = f \in \text{PFunc}, \quad \begin{align*}
([e_1]_0(E), \ldots, [e_k]_0(E)) & \in \text{dom}(f), & \Rightarrow [e_0]_0(E) &= f([e_1]_0(E), \ldots, [e_k]_0(E)) \quad \text{(App-Pre)}
\end{align*}
\]
\[
[e_i]_0(E) = \bot \notin \{ [e_i]_0(E) \mid i \in \{1, \ldots, k\} \}, \quad \begin{align*}
[e_0]_0(E) &= \text{error} \quad \text{(MisApp-Pre)}
\end{align*}
\]

| Table 5.1: semantic function \([ \cdot ]_0\) of CS\(^0\) (Part 1) |
5.2 Extending Syntax and Standard Semantics of $CS^0$

In this section some extensions of $CS^0$ are presented. These extensions are destructive updates in Subsec. 5.2.1, structured data in Subsec. 5.2.2 and environments as first class values in Subsec. 5.2.3.

5.2.1 Destructive Updates in the Core Language

In this section we extend $CS^0$ to a functional language $CS^S$ in order to express destructive updates on symbols. The syntax of $CS^0$ is extended by some more expressions yielding the syntax of $CS^S$. To define the semantics of the new syntactic elements we need a refined notion of environments.

5.2.1.1 Extending the Syntax for Destructive Updates

The following definition extends the syntax of $CS^0$ by adding notions of binding a value to a symbol, of updating such a binding and of sequences of expressions (that usually contain side effects).

**Definition 5.2.1 (extended syntax of $CS^S$)** The syntax of $CS^S$ is achieved by extending the syntax of $CS^0$ as follows:

| $\exists i. [e_i]_0(E) = \bot$, $\not\exists j < i. [e_j]_0(E) = \text{error}$ | $[(e_0 \ e_1 \ ... \ e_k)]_0(E) = \bot$ | (App-Strict) |
| $\exists i. [e_i]_0(E) = \text{error}$, $\not\exists j < i. [e_j]_0(E) = \bot$ | $[(e_0 \ e_1 \ ... \ e_k)]_0(E) = \text{error}$ | (App-Err) |
| $[e_0]_0(E) \not\in \text{PFunc} \cup \text{LC}$ | $[(e_0 \ e_1 \ ... \ e_k)]_0(E) = \text{error}$ | (MisApp) |

Table 5.2: semantic function $[\cdot]_0$ of $CS^0$ (Part 2)
\[
e \in \text{Expression}^S \rightarrow e^0 \in \text{Expression}^0 \quad \text{(Expressions)}
\]
\[
| \quad (\text{define } x \ e)
| \quad (\text{set! } x \ e)
| \quad (\text{begin } e_1 \ e_2 \ldots)
\]

Every expression in \(CS^0\) is also an expression in \(CS^S\). The \textit{define}-syntax binds the value of an expression to a symbol, \textit{set!}-expressions destructively update symbol bindings and \textit{begin}-expressions denote sequences of expressions with side effects.

### 5.2.1.2 The Standard Semantics of Updates

To express updates on symbols as described here and on parts of structures (see Sec. 5.2.2) more uniformly we associate stores with values. More precisely, we introduce different occurrences of values and associate a unique store with every occurrence of a value. Binding a symbol \(x \in \text{Sym}\) to a value (more precisely, to an occurrence) \(v\) then always means binding the symbol to the store \(s\) that corresponds to \(v\). Formally, this situation is given by the following definition:

**Definition 5.2.2 (stored assignable values)** The set of stored assignable values is defined as

\[
\text{AssValue}^{st} = \{(v, s) \mid v \in \text{AssValue}^0, s \in \text{Store}\}
\]

where \text{Store} denotes a set of stores.

We often identify an element \(w = (v, s) \in \text{AssValue}^{st}\) with \(v \in \text{AssValue}\) and use the function

\[
\text{getstore} : \text{AssValue}^{st} \rightarrow \text{Store} \quad \text{with} \quad \text{getstore}(w) = s
\]

to denote the store of \(w\). From now on we always use stored assignable values instead of assignable values, i.e. every following extension of \(\text{AssValue}^0\) indeed is an extension of \(\text{AssValue}^{st}\).

Besides introducing stores we have to change the structure of environments as used before. This is motivated and defined in the following.

In the definition of the semantics of \(CS^0\) the environment used for reducing an expression was constant. For evaluating subexpressions an extended environment could be used, but these extensions to environments where discarded after the subexpression was completely evaluated.
When destructive updates can occur during the evaluation of subexpressions some structure in the environments is needed to determine the scope of the side effect. Structured environments are constructed as lists of frames where frames can informally be understood as sets of assignments of values to symbols.

**Definition 5.2.3 (frame)** A frame $F$ is a partial function $F : \text{Sym} \rightarrow \text{AssValue}$ from symbols to assignable values. The domain of $F$ is denoted by $\text{dom}(F)$.

$[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$ denotes the frame that is defined exactly on the set $\{x_1, \ldots, x_k\}$ of symbols and binds the symbol $x_i \in \text{Sym}$ to the value $v_i \in \text{AssValue}$.

$F[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$ denotes the frame that is generated from $F$ by binding the symbols $x_i$ to the values $v_i$. The former bindings of $x_i$ in $F$ (if $x_i$ was bound before) are discarded.

The set of all frames is denoted by Frame.

**Example 5.2.4 (frame)** Let $x, y, z \in \text{Sym}$. A frame binding $x$ to 5, $y$ to $\#f$ and $z$ to “hello world” is defined as $F = [x \mapsto 5, y \mapsto \#f, z \mapsto \text{“hello world”}]$.

Frames can change their bindings during the execution of a program. We therefore associate a store with every frame that remains constant during executing the program. The formal tools are introduced in the following definition:

**Definition 5.2.5 (stored frames)** The set of stored frames is given by

$\text{Frame}^{st} = \text{Frame} \times \text{Store}$

The function $\text{getstore} : \text{AssValue}^{st} \cup \text{Frame}^{st} \rightarrow \text{Store}$ is extended to determine the unique store of every stored frame $F$.

In the following we identify $\text{Frame}$ with $\text{Frame}^{st}$, i.e. we assume every frame to be a stored one.

Frames can now be used to define structured environments that are essentially lists of frames:

**Definition 5.2.6 (structured environments)** A structured environment is a finite list $E = (F_1 F_2 \ldots F_k)$ of frames. The value of a symbol $x$ in an environment $E$ is determined as follows:

$$E(x) = \begin{cases} F_i(x) & \text{if } i \text{ is the smallest index with } x \in \text{dom}(F_i) \\ \text{undef} & \text{if no such } i \text{ exists} \end{cases}$$
The set of all structures environments is denoted by $\text{Env}_{SF}$.\(^1\)

If $E = (F_1 \ldots F_k) \in \text{Env}_{SF}$, $x_1, \ldots, x_n \in \text{Sym}$ and $v_1, \ldots, v_n \in \text{Value}$ then we define the extension of $E$ as follows:

$$E[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] := ([x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \ F_1 \ldots F_k)$$

**Example 5.2.7 (structured environment)** Let $F_1 = [x \mapsto 5, y \mapsto \#f]$ and $F_2 = [x \mapsto 42, z \mapsto \text{“hello world”}]$ be frames. Then $E = (F_1 F_2)$ is a structured environment with

$$E(x) = 5$$
$$E(y) = \#f$$
$$E(z) = \text{“hello world”}$$

Note that the binding of $x$ in $F_2$ is not accessible in $E$.

**Definition 5.2.8 (stored environments)** The set of stored environments is defined by

$$\text{Env}_{SF}^{st} = \text{Env}_{SF} \times \text{Store}.$$  

The function getstore is extended to

$$\text{getstore : AssValue}^{st} \cup \text{Frame}^{st} \cup \text{EnvS}_{SF}^{st} \rightarrow \text{Store}$$

defined by $\text{getstore}((v, s)) = s$ for every stored assignable value, stored frame or stored environment $(v, s)$.

From now on we assume every environment to be stored and we identify $\text{Env}_{SF}$ with $\text{Env}_{SF}^{st}$.

The stores assigned to every assignable value, frame and environment informally can be understood as memory addresses of the stored values. They are used to express the change of stored values during program execution. The set of current values assigned to certain stores are called a *state* and are defined by a function lookup-store that is inverse to getstore and defined as follows:

**Definition 5.2.9 (looking up stores in states)** A state is given by a function

$$\text{lookup-store : Store} \rightarrow \text{AssValue}^{st} \cup \text{Frame}^{st} \cup \text{EnvS}_{SF}^{st}$$

that fulfills

$$\text{getstore}(\text{lookup-store}(S)) = S \text{ for all } S \in \text{Store}.$$  

\(^1\)The subscript $SF$ marks these environments as state free. In the following environments will be extended by states.
We can now define a set $State$ of all states and define a semantic function that besides an expression $e$ and an environment $E$ expects a state $St$. Since changes in an environment and in a state often correspond we want to extend environments to hold all information that can be found in a pair taken from $EnvS_{sf} \times State$.

To simplify the needed notions we consider stores as special symbols that do not occur explicitly in the program. Every program state given by a function $lookup-store \in State$ is represented by a frame $FS$ with $FS(S) = lookup-store(S)$. The set of all these frame states is denoted by $FS$. $State$ and $FS$ both denote the same set of functions, but from an informal point of view $FS$ additionally spots on a certain representation of functions as frames.

Stored structured environments are extended to contain a frame state $FS \in FS$ as most general frame. For every stored structured environment $E$ the function

$$getframestate : EnvS^{st} \to FS$$

returns the frame state given in $E$.

A first attempt to give a definition of the set of environments containing frame states is the following:

$$EnvS = EnvS_{sf} \times FS$$

with $FS$ defined by

$$FS = Store \to AssValue \cup Frame \cup EnvS.2$$

(We use a product in the definition of $EnvS$ instead of extending the list of frames given by an environment, but the resulting set is isomorphic to the intended set of extended lists of frames.)

The definitions of $EnvS$ and $FS$ are mutually recursive. We can insert the definition of $FS$ into the definition of $EnvS$ and get:

$$EnvS = EnvS_{sf} \times (Store \to AssValue \cup Frame \cup EnvS).$$

By changing from extended environments to stored extended environments we get

$$EnvS = EnvS_{sf} \times (Store \to AssValue \cup Frame \cup EnvS) \times Store.$$

With this new directly recursive definition of $EnvS$ the set of all frame states is given as

$$FS = Store \to AssValue \cup Frame \cup EnvS$$

---

2Here and in the following we often omit the superscript st for stored objects. All objects are considered as stored ones unless explicitly stated otherwise.
without any mutual recursion.

The solution of the recursive definition of $EnvS$ above is obtained according to [Sch86, Chap. 11]. The construction of the defined sets is not described here in detail. We rather explain the result for the following simple example definition taken from [Sch86]:

$$Alist = (\text{nil} \cup (A \cdot Alist))_\bot$$

The domain $Alist$ of lists given by this definition contains exactly the following elements:

- $Alist_0 = \{\bot\} \subset \text{Alist}$
- $Alist_{\text{fin}} = \bigcup_{i \in \mathbb{N}_0} Alist_i \subset \text{Alist}$ contains all finite lists.
- In addition to the finite lists $Alist$ contains the infinite lists given as least upper bounds of chains in $Alist_{\text{fin}}$.

Comparable to this example $EnvS$ contains environments $E$ with a frame state mapping the store of $E$ to $E$, i.e. environments that are given by a recursive definition. In fact we are especially interested in this kind of environment.

Since environments and state are now combined in this extension of structured environments we have to make sure that an environment is always consistent with its frame state. We demand each stored structured environment to fulfill the state condition defined as follows:

**Definition 5.2.10 (state condition of environments)** Let $E = ((F_1, F_2, \ldots, F_k, FS), S) \in EnvS^{st}$ be a stored structured environment. $E$ fulfills the state condition if the following holds:

1. $FS(S) = E$.
2. $\text{getframestate}(FS(S')) = FS$ for every $S' \in \text{Store}$ with $FS(S') \in EnvS^{st}$.
3. If $FS(S') = E' \in EnvS^{st}$ and $F \in \text{Frame}^{st}$ occurs in $E'$ then $FS(\text{getstore}(F)) = F$.

Informally, the state condition states that

- an environment $E$ is consistent with its own frame state (1).
- the frame state is consistent with all environments occurring in it, i.e. for each of these environments $E'$ the frame state $FS$ is given by $FS = \text{getframestate}(E')$ (2).
- a frame occurring $F$ occurring in a frame $E'$ is also stored in the frame state of $E'$ (3).
**Example 5.2.11** Consider the environment \( E = (F_1 F_2) \) with \( F_1 = [x \mapsto 5, y \mapsto \#f] \) and \( F_2 = [x \mapsto 42, z \mapsto \text{“hello world”}] \) as given in Ex. 5.2.7. Implicitly this environment is defined as \( E = (F_1 F_2 FS) \) where FS is the only frame binding the special symbols denoting stores.

When side effects occur on a value, the store of that value is not changed. For a given stored value \( w \) and an environment \( E \) we can therefore use the store to get the current version \( w' \) of \( w \) with respect to \( E \), i.e. the value that is given by the store of \( w \) in \( \text{getframestate}(E) \).

**Definition 5.2.12 (looking up stored values)** The function

\[
\text{get-stored-value} : \text{Store} \times \text{Env}^{st} \rightarrow \text{AssValue}^{st}
\]

is defined by:

\[
\text{get-stored-value}(S, E) = \text{getframestate}(E)(S).
\]

The side effects considered in this section effect the environments they are evaluated in. Thus, from the time an environment \( E \) was defined to the time it is used there could have been destructive updates changing the bindings in \( E \). To make this explicit we define a function \( \text{update} \) as follows:

**Definition 5.2.13 (updating frames and environments)** Let \( F = (\tilde{F}, S) \in \text{Frame}^{st} \) be a (stored) frame, i.e. \( \text{getstore}(F) = S \) and let \( \tilde{E} \) be an environment. The bindings of \( F \) corresponding to the current state of \( \tilde{E} \) are given by the frame

\[
\text{update}(F, \tilde{E}) = \text{get-stored-value}(S, \tilde{E}) = \text{get-stored-value}(\text{getstore}(F), \tilde{E}).
\]

\( \text{update} \) for environments is analogously defined by

\[
\text{update}(E, \tilde{E}) = \text{get-stored-value}(\text{getstore}(E), \tilde{E}).
\]

In our semantics environments and their frame states will be generated from each other without changing frames and environments to different values. i.e. the stored environments occurring during program execution are state compatible with all previously defined stored frames and stored environments:

**Definition 5.2.14 (state compatibility)** Let \( \tilde{E} \) and \( E = ((F_1 \ldots F_k FS), S) \) be stored structured environments and \( F \) a stored structured frame. \( \tilde{E} \) is state compatible with \( F \) and \( E \) if

\[
\text{update}(F, \tilde{E}) \in \text{Frame}^{st}
\]

\[
\text{update}(E, \tilde{E}) = ((\text{update}(F_1, \tilde{E}) \ldots \text{update}(F_k, \tilde{E}) \text{getframestate}(\tilde{E})), S)
\]
\begin{align*}
c \in \text{ConstT} \Rightarrow & \quad EUS(c, E) = E \quad \text{(Const)} \\
x \in \text{Sym} \Rightarrow & \quad EUS(x, E) = E \quad \text{(Inst)} \\
b([e]_S(E)) = \text{true} \Rightarrow & \quad EUS((\text{if } e, e_1, e_2), E) = EUS(e_1, EUS(e, E)) \quad \text{(If-True)} \\
b([e]_S(E)) = \text{false} \Rightarrow & \quad EUS((\text{if } e, e_1, e_2), E) = EUS(e_2, EUS(e, E)) \quad \text{(If-False)} \\
x_1, \ldots, x_k \in \text{Sym} \Rightarrow & \quad EUS((\text{lambda } (x_1 \ldots x_k) e), E) = E \quad \text{(Lambda)} \\
\llbracket e_0 \rrbracket_S(E) = f \in \text{PFunc}, & \quad EUS((\text{begin } e_0), E), \ldots, \llbracket e_k \rrbracket_S(EUS((\text{begin } e_0 \ldots e_{k-1}), E))) \in \text{dom}(f) \\
\Rightarrow & \quad EUS((\text{begin } e_0 \ldots e_k), E) = \quad \text{(App-Pre)} \\
\llbracket e_0 \rrbracket_S(E) \in \text{dom}(f), & \quad EUS((\text{begin } e_0 \ldots e_k), E) = \quad \text{(Begin-1)} \\
\Rightarrow & \quad EUS((\text{begin } e_1 e_2 \ldots e_k), E) = EUS((\text{begin } e_1 e_2 \ldots e_k), EUS(e_1, E)) \quad \text{(Begin-k)} \\
k > 1 \Rightarrow & \quad EUS((\text{begin } e_1 e_2 \ldots e_k), E) = EUS((\text{begin } e_1 e_2 \ldots e_k), EUS(e_1, E)) \quad \text{(Begin-k)}
\end{align*}

Table 5.3: environment update function $EUS$ for $CS^S$ without (direct) side effects

For the semantic function $\llbracket \cdot \rrbracket_S$ as given in Def. 5.1.15 it was essential that the environment was constant during evaluating subexpressions. In the context of destructive updates the evaluation of subexpressions can change the environment. Thus, the following definition not only defines the standard semantics of the new expressions of $CS^S$, but also refines the definition of the semantic function $\llbracket \cdot \rrbracket_S$ for the constructs of $CS^0$ in order to process the change of environments correctly.

The function $EUS$ defined in the following expresses the environment update that is performed during the evaluation of certain expressions. It is used in the definition of the semantic function of $CS^S$. $EUS$ already uses the definition of $\llbracket \cdot \rrbracket_S$ that is given in Def. 5.2.16. Note that $EU$ is not defined for those cases with $\llbracket \cdot \rrbracket_S$ yielding error or $\perp$.

**Definition 5.2.15 (environment updates)** When evaluating an expression $e$ under an environment $E$ the resulting environment $E'$ can differ from $E$. The function

$$EUS : \text{Expression}^S \times \text{EnvS} \rightarrow \text{EnvS}$$

expresses these changes. Table 5.3 defines $EU$ for those expressions that do not perform side effects on their own (But many of them must hand over side effects from their subexpressions to the context).

The definition of $EU$ for those expression directly performing side effects is given in Tab. 5.4.
Table 5.4: environment update function $EU^S$ for $CS^S$ with side effects (The definition of $FS'$ is given in Def. 5.2.15.)
For the rules (Def) and (Set) the frame state $FS'$ is generated from $FS$ as follows: Let $F'_i$ be the frame that was generated by modifying $Fi$ by (Def) or (Set):

$$FS'(S) = \begin{cases} 
F'_i & \text{if } S = \text{getstore}(F_i) \\
((\tilde{F}'_1 \ldots \tilde{F}'_n) S), S) & \text{if } FS(S) = ((\tilde{F}'_1 \ldots \tilde{F}'_n) S), S) \\
FS(S) & \text{otherwise}
\end{cases}$$

For the second case the frames $\tilde{F}'_j$ are given by

$$\tilde{F}'_j = \begin{cases} 
F'_i & \text{if } \tilde{F}_j = F_i \\
\tilde{F}_j & \text{otherwise}
\end{cases}$$

The rules of the definition of $EU$ that are of special interest are (App-Lambda), (Def) and (Set):

- The environment update for the application of lambda-closures as described in (App-Lambda) has to be read as follows: first, the current environment $E$ is updated by possible side effects of $e_1, \ldots, e_k$ during their evaluation in $E$. The resulting environment is again updated by further evaluating the body $e$ of the lambda-closure in an environment that results from the definition environment $E'$ of the lambda-closure by adding a new frame with bindings of the formal parameters. $update(E)$ in $[e_i]_{S}(update(E))$ denotes the environment $E$ updated by the $e_j$ before.

- The environment update performed by define takes place in the first frame of the current environment. The symbol given in the first argument is bound to a new store holding the value given as second argument. The binding is performed in the environment resulting from the evaluation of the second argument.

- Analogously to define, the environment update performed by set! consists of the environment update performed by evaluating the second argument and changing a binding: In the first frame of the current environment the symbol $x$ given as first argument occurs its binding is changed to a new store holding the value given as second argument.

The update of the frame state changing $FS$ to $FS'$ updates the store of $F_i$ to $F'_i$, and in every environment changes the frame $F_i$ and the frame state to the current values. This is the minimal change that reenforces the state condition to hold for the result of $EU^S$.

Note that the recursive definition of $FS'$ is a valid one. On page 100 the set $\mathcal{FS}$ is defined in terms of $EnvS$ which is recursively defined. Thus, recursive structures as needed for $FS'$ are contained in $\mathcal{FS}$.

Now the semantic function of $CS^S$ with changing environments is defined as follows:
Definition 5.2.16 (semantic function of \( CS^S \)) The semantic function
\[
\cdot : Expression^S \times EnvS \rightarrow Value^S
\]
is defined by the case distinction in Tab. 5.5 and Tab. 5.6.

Besides the rules already known from Def. 5.1.15 the new rules (Def), (Set), (Set-Err), (Begin-1) and (Begin-k) explain how expressions built by \textit{define}, \textit{set!} and \textit{begin} have to be interpreted:

- The return value of \textit{define} is unspecified (denoted by \( O \)). The semantic meaning becomes clear from the environment update function \( EU \) that is given in Def. 5.2.15.

- \textit{set!} essentially behaves like \textit{define}: it changes the environment (see Def. 5.2.15) and returns an unspecified value. When the symbol given in the first argument was not bound before, an error results.

- \textit{begin} sequentially evaluates its arguments. For the evaluation of every argument it uses the environment that contains all changes performed by the arguments before.

In the context of destructive updates it is important to know how the predefined functions contained in \( PFunc \) are accessed: for every top level expression \( e \) that is not a subexpression of another expression \( e' \) the evaluation of \( e \) takes place in the so called standard environment that just contains assignments of the predefined function terms to the symbols representing their standard names according to [KCE98]. During the evaluation of \( e \) the symbols defined in the standard environment are possible targets of redefinitions or destructive updates.

5.2.2 Structured Data

An important part of the definition of our functional language is the opportunity to express structured data. In the following we define how structured data can be constructed, destructed and updated.

Structured data occurs in two different kinds: structures of arbitrary length are constructed by means of a \textit{pair}-constructor. The \textit{vector}-constructor on the other hand is used when the length of a structure is known and fast access to the individual elements is needed.\(^3\) Defining syntax and semantics (also for the abstract semantics given in Def. 6.3.36) is quite similar for both types of structured data. We therefore restrict the presentation here to pairs, which are the most often used structuring tools in functional languages like Scheme.

\(^3\)Scheme as described in the R\(^5\)RS-standard additionally contains strings as structured data. We do not consider this any further here because strings behave like vectors except of the element type being restricted to characters.
In the following rules we use the abbreviations $a_i$ for $i = 1, \ldots, k$ with

$$a_i = \llbracket e_i \rrbracket_S(EU((\text{begin } e_0 \ldots e_{i-1}), E)).$$

| $c \in \text{ConstT}$ ⇒ | $\llbracket c \rrbracket_S(E) = \text{denote const}(c)$ | (Const) |
| $x \in \text{Sym}, E(x) \neq \text{undef}$ ⇒ | $\llbracket x \rrbracket_S(E) = E(x)$ | (Inst) |
| $x \in \text{Sym}, E(x) = \text{undef}$ ⇒ | $\llbracket x \rrbracket_S(E) = \text{error}$ | (Inst-Err) |

| $\text{bi}(\llbracket e \rrbracket_S(E)) = \text{true}$ ⇒ | $\llbracket (\text{if } e_1 e_2) \rrbracket_S(E) = \llbracket e_1 \rrbracket_S(EU(e, E))$ | (If-True) |
| $\text{bi}(\llbracket e \rrbracket_S(E)) = \text{false}$ ⇒ | $\llbracket (\text{if } e_1 e_2) \rrbracket_S(E) = \llbracket e_2 \rrbracket_S(EU(e, E))$ | (If-False) |
| $\llbracket e \rrbracket_S(E) = \text{error}$ ⇒ | $\llbracket (\text{if } e_1 e_2) \rrbracket_S(E) = \text{error}$ | (If-Err) |

| $\text{bi}(\llbracket e \rrbracket_S(E)) = \bot$ ⇒ | $\llbracket (\text{if } e_1 e_2) \rrbracket_S(E) = \bot$ | (If-Strict) |

| $x_1, \ldots, x_k \in \text{Sym}$ ⇒ | $\llbracket (\text{lambda } (x_1 \ldots x_k) e) \rrbracket_S(E) = lc(e(x_1, \ldots, x_k), E)$ | (Lambda) |

| $\exists i. x_i \not\in \text{Sym}$ ⇒ | $\llbracket (\text{lambda } (x_1 \ldots x_k) e) \rrbracket_S(E) = \text{error}$ | (Lambda-Err) |

| $\llbracket e_0 \rrbracket_S(E) = lc(e(x_1, \ldots, x_k), E') \in \text{LC}, \bot, \text{error} \not\in \{a_1, \ldots, a_k\}$ | $\llbracket (e_0 \ldots e_k) \rrbracket_S(E) = \llbracket e \rrbracket_S(\text{update}(E', EU(\text{begin } e_0 e_1 \ldots e_k), E))[$ $x_1 \mapsto a_1, \ldots, x_k \mapsto a_k)]$ | (App-Lambda) |

| $\llbracket e_0 \rrbracket_S = lc(e(x_1, \ldots, x_k), E'), n \neq k, \not\exists i. a_i = \bot$ | $\llbracket (e_0 \ldots e_n) \rrbracket_S(E) = \text{error}$ | (LApp-Err) |

| $\llbracket e_0 \rrbracket_S(E) = f \in \text{PFunc},$ $(a_1, \ldots, a_k) \in \text{dom}(f), \Rightarrow$ | $\llbracket (e_0 \ldots e_k) \rrbracket_S(E) = f(a_1, \ldots, a_k)$ | (App-Pre) |

| $\bot, \text{error} \not\in \{a_1, \ldots, a_k\}$ |

Table 5.5: The semantic function $\llbracket \cdot \rrbracket_S$ of CS$^S$ (Part 1)
In the following rules we use the abbreviations $a_i$ for $i = 1, \ldots, k$ with

$$ a_i = [e_i]_{S}(EU((\text{begin } e_0 \ldots e_{i-1}), E)). $$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[c_0]_{S}(E) = f \in PFunc,$</td>
<td>$([e_0 e_1 \ldots e_k])_{S}(E) = \text{error}$ (MisApp-Pre)</td>
</tr>
<tr>
<td>$(a_1, \ldots, a_k) \notin \text{dom}(f),$</td>
<td></td>
</tr>
<tr>
<td>$\not\exists i. a_i = \bot$</td>
<td></td>
</tr>
<tr>
<td>$[e_0]_{S}(E) \notin {\bot, \text{error}}$</td>
<td>$([\text{define } x e])_{S}(E) = \emptyset$ (Def)</td>
</tr>
<tr>
<td>$(x \notin \text{Sym} \lor [e]_{S}(E) = \text{error})$</td>
<td>$([\text{define } x e])_{S}(E) = \text{error}$ (Def-Err)</td>
</tr>
<tr>
<td>$[e]_{S}(E) \neq \bot$</td>
<td></td>
</tr>
<tr>
<td>$[e]_{S}(E) = \bot$</td>
<td>$([\text{define } x e])_{S}(E) = \bot$ (Def-Strict)</td>
</tr>
<tr>
<td>$x \in \text{Sym}$ and $E(x) \neq \text{undef},$</td>
<td></td>
</tr>
<tr>
<td>$[e]_{S}(E) \notin {\bot, \text{error}}$</td>
<td>$([\text{set } x e])_{S}(E) = \emptyset$ (Set)</td>
</tr>
<tr>
<td>$(x \notin \text{Sym} \lor E(x) = \text{undef})$</td>
<td>$([\text{set } x e])_{S}(E) = \text{error}$ (Set-Err)</td>
</tr>
<tr>
<td>$[e]_{S}(E) = \text{error}$</td>
<td></td>
</tr>
<tr>
<td>$[e]_{S}(E) = \bot$</td>
<td>$([\text{set } x e])_{S}(E) = \bot$ (Set-Strict)</td>
</tr>
<tr>
<td>$[e]_{S}(E) \notin {\text{error, } \bot}$</td>
<td>$([\text{begin } e])<em>{S}(E) = [e]</em>{S}(E)$ (Begin-1)</td>
</tr>
<tr>
<td>$k &gt; 1,$ $[e_1]_{S}(E) \notin {\bot, \text{error}}$</td>
<td>$([\text{begin } e_1 e_2 \ldots e_k])<em>{S}(E) = ([\text{begin } e_2 \ldots e_k])</em>{S}(EU(e_1, E))$ (Begin-k)</td>
</tr>
<tr>
<td>$[e_1]_{S}(E) = \text{error}$</td>
<td>$([\text{begin } e_1 \ldots e_k])_{S}(E) = \text{error}$ (Begin-Err)</td>
</tr>
<tr>
<td>$[e_1]_{S}(E) = \bot$</td>
<td>$([\text{begin } e_1 \ldots e_k])_{S}(E) = \bot$ (Begin-Strict)</td>
</tr>
</tbody>
</table>

Table 5.6: The semantic function $[\cdot]_{S}$ of $CS^{S}$ (Part 2)
5.2.2.1 The Syntax of Structured Data

There are no further syntactic constructs needed to express operations on structured data as these operations are given by ordinary predefined functions. The introduction of structured data however allows to express the behaviour of the additional syntactic keyword \textit{quote}.

\textbf{Definition 5.2.17 (extended syntax of }CS^L\textit{)} The syntax of }CS^L\textit{ is achieved by extending the syntax of }CS^S\textit{ as follows:

\begin{equation}
\begin{array}{c}
e \in Expression^L \\
\rightarrow e^S \in Expression^S \\
\mid (quote \ e)
\end{array}
\end{equation}

Every expression in }CS^S\textit{ is still an expression in }CS^L\textit{}. Furthermore, expressions can be quoted by using the new keyword \textit{quote}.

5.2.2.2 Standard Semantics of Structured Data

First of all we need a new domain constructor generating pairs. This constructor is given by the following definition:

\textbf{Definition 5.2.18 (pair domain constructor)} For two domains }A\textit{ and }B\textit{ the pair domain constructor generates the domain \((A . B)\) of pairs \((a . b)\) with assignable values }a \in A\textit{ and }b \in B\textit{.

The set of assignable values is extended to

\[ AssValue^L = AssValue^S \cup (AssValue^L . AssValue^L) . \]

The meaning of this recursive definition is again given as stated in [Sch86].

A special case of infinite values covered by this definition are cyclic values.

\textbf{Example 5.2.19 (cyclic value)} The infinite list just containing the value 42 as element can be understood as cyclic value of the form presented in Fig. 5.1.

Recalling Def. 3.1.28 on page 36 we use the notation }\mu x.C[x]\textit{ for cyclic values where }x\textit{ is a variable (not a symbol) and }C[x]\textit{ denotes a context of }x\textit{, i.e. a structure with }x\textit{ occurring as a substructure.

Using this notion the cyclic structure given in Ex. 5.2.19 can be written as }\mu x.(42 . x).\textit{
Every pair is uniquely identified by its store. It is important to note that the values $a$ and $b$ are stored values.

Furthermore, every store $S$ that is bound to a structured value in a frame state $FS$ must fulfill the structured state condition defined as follows:

**Definition 5.2.20 (structured state condition)** Let $FS$ be a frame state and $S$ a store with $FS(S) = v$ where $v = (a \cdot b)$ and $a, b \in \text{AssValue}^L$. $S$ fulfills the structured state condition with respect to $FS$ if every subvalue $w$ of $v$ fulfills

$$w = FS(\text{getstore}(w)) .$$

$FS$ fulfills the structured state condition if every store $S'$ with $FS(S') = (a' \cdot b')$ fulfills the structured state condition with respect to $FS$.

To define the semantics of *quote* expressions we need a new domain $\text{Sym}_V \subset \text{Const}$ to express symbols as values:

**Definition 5.2.21 (symbols as values)** We define a new set $\text{Sym}_V \subset \text{Const}$ of symbol values such that for each symbol $s \in \text{Sym}$ there is a symbol value $s_v \in \text{Sym}_V$ denoted in the same way. The function

$$\text{sym-to-const} : \text{Sym} \rightarrow \text{Sym}_V$$

returns for every symbol the corresponding symbol value.

The semantics of *quote*-expressions is defined as follows:

**Definition 5.2.22 (semantics of *quote*-expressions)** The function

$$\text{quotesem} : \text{Expression} \rightarrow \text{AssValue}$$
is defined as follows:

- If \( e \in ConstT \) then \( \text{quotesem}(e) = \text{denoteconst}(e) \).\(^4\)
- If \( e \in Sym \) then \( \text{quotesem}(e) = \text{sym-to-const}(e) \).
- If \( e = (e_1 \ldots e_k) \) then \( \text{quotesem}(e) = \text{make-list}(\text{quotesem}(e_1), \ldots, \text{quotesem}(e_k)) \) with
  - \( \text{make-list}() = \text{nil} \in \text{Const} \).
  - \( \text{make-list}(v_1, v_2, \ldots, v_k) = (v_1 . \text{make-list}(v_2, \ldots, v_k)) \).

The operations on pairs are given by a set \( \text{PairOp} \) of predefined functions defined as follows:

**Definition 5.2.23 (extended semantics of \( CS^L \))** Let

\[ \text{PFunc} \supset \text{PairOp} := \{ f_{\text{cons}}, f_{\text{car}}, f_{\text{cdr}}, f_{\text{set-car}!}, f_{\text{set-cdr}!} \} \].

For \( f_{\text{cons}}, f_{\text{car}} \) and \( f_{\text{cdr}} \) the semantics is defined by:

\[ f_{\text{cons}} : A \times B \to (A \cdot B) \]
\[ f_{\text{cons}}(a, b) = (a \cdot b) \]

\[ f_{\text{car}} : (A \cdot B) \to A \]
\[ f_{\text{car}}((a \cdot b)) = a \]

\[ f_{\text{cdr}} : (A \cdot B) \to B \]
\[ f_{\text{cdr}}((a \cdot b)) = b \]

These definitions can be used in Rule (App-Pre) of \([\cdot]_S\).

Since the functions \( f_{\text{set-car}!} \) and \( f_{\text{set-cdr}!} \) perform side effects we explicitly give the semantics of these functions in Tab. 5.7 together with the semantics of quote-expressions.

The function \( \text{struct-update} : Env^{st} \times AssValue^{st} \to AssValue^{st} \) used by (Set-Car-EU) and (Set-Cdr-EU) is defined as follows:

\[ \text{struct-update}(((F_1 \ldots F_k \ FS), S), v, v') = FS'(S) \]

\(^4\)Constant expressions in \( CS \) are self evaluating expressions, i.e. they are evaluated to constants denoted in the same way.
In the following rules we use the abbreviations
\( a_f = [ef]_L(E) \)
\( a_1 = [e_1]_L(EU^L(ef, E)) \)
\( a_2 = [e_2]_L(EU^L((\text{begin } ef e_1), E)) \)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_f = f_{\text{set-car!}}, a_1 = v, v = (v_1', v_2') \Rightarrow EU^L((ef e_1 e_2), E) = \text{struct-update}(EU^L(\text{begin } ef e_1 e_2), E), v, (a_2 . v_2')) )</td>
<td>(Set-Car-EU)</td>
</tr>
<tr>
<td>( a_f = f_{\text{set-cdr!}}, a_1 = v, v = (v_1', v_2') \Rightarrow EU^L((ef e_1 e_2), E) = \text{struct-update}(EU^L(\text{begin } ef e_1 e_2), E), v, (v_1' . a_2)) )</td>
<td>(Set-Cdr-EU)</td>
</tr>
<tr>
<td>( EU^L((\text{quote } e), E) = E )</td>
<td>(Quote-EU)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_f = f_{\text{set-car!}}, a_1 = (v_1 . v_2), a_2 \notin {\bot, \text{error}} \Rightarrow [(ef e_1 e_2)]_L(E) = \mathcal{O} )</td>
<td>(Set-Car)</td>
</tr>
<tr>
<td>( a_f = f_{\text{set-car!}}, a_1 \neq \bot, (a_1 \neq (v_1 . v_2) \lor a_2 = \text{error}), \Rightarrow [(ef e_1 e_2)]_L(E) = \text{error} )</td>
<td>(Set-Car-Err)</td>
</tr>
<tr>
<td>( a_f = f_{\text{set-car!}}, a_1 = \bot \lor (a_2 = \bot \land a_1 \neq \text{error}), \Rightarrow [(ef e_1 e_2)]_L(E) = \bot )</td>
<td>(Set-Car-Strict)</td>
</tr>
<tr>
<td>( a_f = f_{\text{set-cdr!}}, a_1 = (v_1 . v_2), a_2 \notin {\bot, \text{error}} \Rightarrow [(ef e_1 e_2)]_L(E) = \mathcal{O} )</td>
<td>(Set-Cdr)</td>
</tr>
<tr>
<td>( a_f = f_{\text{set-cdr!}}, a_1 \neq \bot, (a_1 \neq (v_1 . v_2) \lor a_2 = \text{error}), \Rightarrow [(ef e_1 e_2)]_L(E) = \text{error} )</td>
<td>(Set-Cdr-Err)</td>
</tr>
<tr>
<td>( a_f = f_{\text{set-cdr!}}, a_1 = \bot \lor (a_2 = \bot \land a_1 \neq \text{error}), \Rightarrow [(ef e_1 e_2)]_L(E) = \bot )</td>
<td>(Set-Cdr-Strict)</td>
</tr>
</tbody>
</table>

\( [(\text{quote } e)]_L(E) = \text{quotesem}(e) \) | (Quote) |

Table 5.7: extensions of the environment update function \( EU \) and the semantic function \([\cdot]\) for \( CS^k \) (structured data)
where $FS'$ is generated from $FS$ as follows:

$$FS'(S) = \begin{cases} 
  v' & \text{if } S = \text{getstore}(v) \\
  C[v'] & \text{if } FS(S) = C[v] \text{ for a context } C \\
  (\tilde{F}_1' \ldots \tilde{F}_n' FS'), S & \text{if } FS(S) = ((\tilde{F}_1 \ldots \tilde{F}_n FS), S) \\
  FS(S) & \text{else}
\end{cases}$$

where $\tilde{F}_i'$ is generated from $F_i'$ according to the second case. A context $C[v]$ of a value $v$ in this definition can be a structure containing $v$ as a substructure or a frame binding $v$ (or a structure containing $v$) to a symbol.

Informally, a call $\text{struct-update}(E, v, v')$ returns the environment $E$ after replacing $v$ by $v'$ in every structure containing $v$ and every frame or environment binding $v$ or a containing structure.

As we can see from the rules $(\text{Set-Car})$ and $(\text{Set-Cdr})$ the functions $\text{set-car!}$ and $\text{set-cdr!}$ do not return a meaningful value in the error-free case. The real work is done by $(\text{Set-Car-EU})$ and $(\text{Set-Cdr-EU})$ changing the store by updating the first or second element of the pair their first argument evaluates to.

Note that a structured value $v$ can occur as substructure in several different structures. Changing a subvalue of $v$ effects all the values that contain the store of $v$.

**Example 5.2.24 (generating structured data)** Consider the following data definition:

$$(\text{cons} \ 5 \ (\text{cons} \ #t \ (\text{cons} \ 42 \ ())))$$

It yields the value $(5 \ . \ (#t \ . \ (42 \ . \ ())))$ that is often abbreviated as $(5 \ #t \ 42)$. Denoting this list by $v$ we can select the individual elements as follows:

$$(\text{car} \ v) \rightarrow 5$$

$$(\text{car} \ (\text{cdr} \ v)) \rightarrow #t$$

$$(\text{car} \ (\text{cdr} \ (\text{cdr} \ v))) \rightarrow 42$$

**Example 5.2.25 (updating structured data)** Consider the list $(5 \ . \ (#t \ . \ (42 \ . \ ())))$ (abbreviated as $(5 \ #t \ 42)$) defined in Ex. 5.2.24. Updates on this list are possible as follows:

$$(\text{set-car!} \ v \ 6) \text{ changes } v \text{ to } (6 \ #t \ 42)$$

$$(\text{set-cdr!} \ (\text{cdr} \ (\text{cdr} \ v)) \ (\text{cons} \ 50 \ ())) \text{ changes } v \text{ to } (6 \ #t \ 42 \ 50)$$

112
The sequence

\[
\begin{align*}
&\text{(define 'week (cons mon (cons tue (cons wed (cons thu (cons fri (cons sat (cons sun nil)))))))))} \\
&(\text{set-cdr! (cdr (cdr (cdr (cdr (cdr week))))))}) week
\end{align*}
\]

binds week to the cyclic structure \(\mu x.(\text{mon . (tue . (wed . (thu . (fri . (sat . (sun . x))))))}).\) A graphical representation of this structure is given in Fig. 5.2.

![Figure 5.2: days of week](image)

### 5.2.3 Making Environments First Class

The extensions introduced in this section provide tools for explicitly controlling the environment an expression is evaluated in. It is e.g. possible to use different environments as different modules defining sets of functions with equal names. In choosing the environment a complex expression is evaluated in one can choose the appropriate set of functions and can hence modify the meaning of the expression by modifying the context given by the expression.

**Example 5.2.26 (biological environments\(^5\))** Consider a biological framework where we want to define several different environments (e.g. an arctic and an oceanic environment). Several items (e.g. the most dangerous animal) should be accessible in a uniform manner after choosing the environment. We can implement this using Scheme environments, especially the make-environment function as e.g. defined in [ASS85], as follows:

\(^5\)This example is taken from [BGH98].
Using the function eval we can now look up the most dangerous animal in both of these environments:

\[ 1 \Rightarrow (\text{eval } '\text{most-dangerous-animal } \text{arctic-environment}) \]
;Value: polar-bear

\[ 1 \Rightarrow (\text{eval } '\text{most-dangerous-animal } \text{oceanic-environment}) \]
;Value: shark

The following subsections define the functions needed to implement first class environments. We just choose the function names according to [KCE98] and give a semantics to the function \text{scheme-report-environment} that is similar to \text{make-environment} in the example.

### 5.2.3.1 The Syntax of First Class Environments

There is no need to extend the syntax of \text{CS}^S to process environments as first class values. The operations needed for this purpose are expressed by ordinary predefined functions.

### 5.2.3.2 Standard Semantics of First Class Environments

The set \text{PFunc} of predefined functions is extended by the following new elements:

- \text{f}_{\text{scheme-report-environment}}, \text{f}_{\text{null-environment}} and \text{f}_{\text{interaction-environment}} initially bound to the symbols \text{scheme-report-environment}, \text{null-environment} and \text{interaction-environment}, respectively, are used to generate environments as values.

- \text{f}_{\text{eval}} initially bound to \text{eval} evaluates a given expression in a given environment. \text{f}_{\text{eval}} exploits the fact that Scheme expressions can be expressed as values.
The semantics of the functions mentioned above can informally be described as follows: Every call to one of the environment generating expressions yields an environment with appropriate bindings. This is always the same environment for $f_{\text{interaction-environment}}$ and $f_{\text{null-environment}}$ and always a new copy of an environment for $f_{\text{scheme-report-environment}}$.$^6$ An eval expression evaluates the expression represented by its first argument in the environment given by the second argument and returns the result.

The following definition provides a master copy for every kind of environment provided by one of the environment generating expressions:

**Definition 5.2.27 (master copies of environments)** $E_0, E_{SR}, E_I \in \text{EnvS}$ are predefined environments with the following properties:

- $E_0 = ([])$ where $[]$ denotes a frame containing no bindings. $[]$ cannot occur in any other environment.
- $E_{SR} = (F_{SR})$ where $F_{SR}$ is a frame that just occurs in $E_{SR}$ and contains the bindings to all predefined functions according to [KCE98].
- $E_I = (F_I)$ where $F_I$ is the top level frame of the current read-evaluate-print loop. $E_I$ initially contains all the bindings of $E_{SR}$ (and maybe some additional ones) and can be updated.

Note that these master copies do not change during program execution. For $E_0$ and $E_{SR}$ this is the case because these environments are not changeable (see Def. 5.2.28). $E_I$ does only occur in form of environment copies (see Def. 5.2.29).

In [KCE98] null-environment is defined to return an unchangeable environment. Thus, we need a function mutable? that distinguishes between changeable and unchangeable environments. It is given by the following definition:

**Definition 5.2.28 (mutability of environments)** For every environment $E \in \text{EnvS}$ the function mutable?: $\text{EnvS} \rightarrow \{m+, m-\}$ yields $m+$ if $E$ is changeable and $m-$ otherwise.

For interaction-environment we want to define a semantics that reflects the semantics of make-environment in MIT-Scheme [H+$95$]. I.e. every call to that function returns a changeable environment with the set of bindings given by $E_I$. Therefore we need a procedure envcopy for copying the bindings of environments. envcopy is given by the following definition:

$^6$[KCE98] does not specify the behaviour of the interpreter when updating an environment returned by scheme-report-environment. We define a semantics here that returns a mutable copy for every call.
Definition 5.2.29 (copying environments) Let $E \in EnvS$ be an environment with the domain $\text{dom}(E) = \{x_1, \ldots, x_k\}$ and let $v_i := E(x_i)$ for all $i$. Then we define the procedure $\text{envcopy} : EnvS \rightarrow EnvS$ as

\[ \text{envcopy}(E) := (\text{genF}[x_1 \mapsto v_1, \ldots, x_k \mapsto x_k]) \]

where $\text{genF}[x_1 \mapsto v_1, \ldots, x_k \mapsto x_k]$ stands for a newly generated frame containing exactly the bindings of $x_i$ to $v_i$ for every $i$.

Note that a result environment $E'$ of $\text{envcopy}$ exactly matches the bindings of the corresponding argument $E$, but $E'$ just contains one frame and not the frame structure of $E$.

As environments become first class now we have to refine the set of assignable values to contain environments. A first attempt to do this is

\[ \text{AssValue}^E = \text{AssValue}^L \cup EnvS \]

with $\text{AssValue}^L$ as given by Def. 5.2.18 on page 108 and $EnvS$ as given on page 100.

More precisely, pairs have to be extended to contain environments as well. Thus, we recall the definition of $\text{AssValue}^L$ and replace the recursive occurrences of $\text{AssValue}^L$ by $\text{AssValue}^E$:

\[ \text{AssValue}^E = \text{AssValue}^S \cup (\text{AssValue}^E \cdot \text{AssValue}^E) \cup EnvS. \]

We have to refine this definition by replacing $EnvS$ by $EnvS^E$ that can bind environments beside other values to symbols. Consider the old definition

\[ EnvS = EnvS_{SF} \times FS \]

where $FS$ is defined as

\[ FS = \text{Store} \rightarrow \text{AssValue}^L \cup \text{Frame} \cup EnvS \]

and $EnvS_{SF}$ can be considered as defined by

\[ EnvS_{SF} = \bigcup_{k \in \mathbb{N}} (\text{Sym} \rightarrow \text{AssValue}^L)^k. \]

When replacing every occurrence of $\text{AssValue}^L$ be $\text{AssValue}^E$ we can omit the union element $EnvS$ in the definition of $FS$ since it is already part of $\text{AssValue}^E$. We get

\[ EnvS^E = \bigcup_{k \in \mathbb{N}} (\text{Sym} \rightarrow \text{AssValue}^E)^k \times (\text{Store} \rightarrow \text{AssValue}^E \cup \text{Frame}). \]

Inserting the definition of $EnvS^E$ into that of $\text{AssValue}^E$ yields the following definition:
Definition 5.2.30 (assignable values) The set of assignable values is refined as follows:

\[ \text{AssValue}^E = \text{AssValue}^S \cup (\text{AssValue}^E \cdot \text{AssValue}^E) \cup \bigcup_{k \in \mathbb{N}} (\text{Sym} \to \text{AssValue}^E)^k \times (\text{Store} \to \text{AssValue}^E \cup \text{Frame}) \].

Again the meaning of this recursive definition is obtained according to [Sch86].

With \( \text{AssValue}^E \) from Def. 5.2.30 we can use the extended definition of \( \text{EnvS}^E \) as given above. In the following we will identify \( \text{EnvS} \) with \( \text{EnvS}^E \).

Besides an environment as defined above the function \( \text{eval} \) needs an expression that should be evaluated in the given environment. Since \( \text{eval} \) is not a special form it evaluates its arguments before changing to the given environment. The first argument available at this point is no longer a syntactic expression but a value. The function \( \text{valueexpression} \) given in Def. 5.2.31 takes a value and returns the expression that corresponds to the output-form of this value whenever such an expression exists. I.e. the function \( \text{valueexpression} \) from Def. 5.2.31 fulfills \( [\text{valueexpression}(v)](E) = v \) for every environment \( E \).

Definition 5.2.31 (expressions available through values) The function

\[ \text{valueexpression} : \text{AssValue} \to \text{Expression} \]

transforms a value back to an expression:

1. For every constant \( c \in \text{Const} \setminus \text{Sym} \), \( \text{valueexpression}(c) := ct \) where \( ct \in \text{ConstT} \) is a constant term with \( [ct](E) = c \) for an arbitrary environment \( E \). (This case covers especially numbers and symbols for function names and syntactic keywords.)

2. For a symbol value \( s_v \in \text{Sym} \), \( \text{valueexpression} \) returns the symbol \( \text{valueexpression}(s_v) := s \) with \( \text{sym-to-const}(s) = s_v \).

3. If \( l \) is a list with elements \( e_1, \ldots, e_k \) and \( et_i = \text{valueexpression}(e_i) \) then

\[ \text{valueexpression}(l) := (et_1 \ldots et_k). \]

4. In every other case \( \text{valueexpression}(v) = \text{express}(v) \) with \( \text{express}(v) \in \text{Expression} \) and \( [\text{express}(v)](E) = v \) for every \( v \in \text{AssValue} \) and every \( E \in \text{EnvS} \). The set \( \text{Expression} \) is extended to contain the results of \( \text{express} \).
The function \textit{express} allows to transform every arbitrary value that is not a list or a self evaluating expression into an expression. The result expression \textit{express}(v) might not have an external representation (e.g. lambda closures and predefined functions). The need of such a function is explained by the following example:

\textbf{Example 5.2.32} Consider the following piece of Scheme code:

\begin{verbatim}
(eval (quasiquote (,+ 3 4)) (null-environment 5))
\end{verbatim}

The quasiquote expression essentially yields its argument expression without evaluating it except of + that is unquoted by ",". The call to \textit{null-environment} returns the environment $E_0$ without any bindings. The argument 5 states that the null environment according to the Scheme report no. 5 [KCE98] is selected.

Do to [KCE98] the use of quasiquote forms an expression and therefore a valid input to \textit{eval}. In the example the result of quasiquote is a list with the function $f_+$ usually bound to + as first and the constant expressions 3 and 4 as second and third element.

Motivated by this example we can consider every value as an expression. This corresponds to the behaviour of e.g. DrScheme v101:

\begin{verbatim}
> (eval + (null-environment 5))
#<primitive:+>
\end{verbatim}

The result \#<\textit{primitive}:+> is a print representation of $f_+$ which is the result of evaluating \textit{express}(f_) in the null environment.

For those values with no expressions defined up to now the function \textit{express} yields these (non-printable) expressions.

In Def. 5.2.33 we define the standard semantics of the predefined functions introduced before. Note that the functions $f_{\text{scheme-report-environment}}$ and $f_{\text{null-environment}}$ expect one argument. They are just defined for the value 5 denoting the number 5 of the scheme report [KCE98].

\textbf{Definition 5.2.33 (semantics of environments)} The standard semantics of CSL is extended to operate on

- $f_{\text{scheme-report-environment}}$
- $f_{\text{null-environment}}$
as stated in Tab. 5.8.

In a context where environments might be unchangeable we furthermore have to refine the semantics of several functions and expressions that change environments.

**Definition 5.2.34 (semantics of unchangeable environments)**  
*The semantic rules of define and set! in Tab. 5.6 on Page 107 and of set-car! and set-cdr! in Tab. 5.7 on Page 111 have to be extended by the additional condition*

\[
\text{mutable?}(E) = m^+.
\]

*The error cases for \text{mutable?}(E) = m^- are presented in Tab. 5.9.*
In the following rules we use the abreviations

\[ a_f = [e_f]_L(E), \quad a_1 = [e_1]_L(EU^L(e_f, E)), \quad a_2 = [e_2]_L(EU^L((\text{begin} \ e_f \ e_1), E)) \]

<table>
<thead>
<tr>
<th>( a_f )</th>
<th>( a_1 )</th>
<th>( a_2 )</th>
<th>( \text{Rule} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_{\text{scheme-report-environment}} )</td>
<td>( 5 \in \text{Const} )</td>
<td></td>
<td>(SRE-EU)</td>
</tr>
<tr>
<td>( f_{\text{null-environment}} )</td>
<td>( 5 \in \text{Const} )</td>
<td></td>
<td>(S0-EU)</td>
</tr>
<tr>
<td>( f_{\text{interaction-environment}} )</td>
<td></td>
<td></td>
<td>(IE-EU)</td>
</tr>
<tr>
<td>( f_{\text{eval}}, a_1 = v, a_2 = E' \in \text{EnvS} )</td>
<td></td>
<td></td>
<td>(Eval-EU)</td>
</tr>
<tr>
<td>( f_{\text{eval}}, a_1 = v, a_2 \notin \text{EnvS} )</td>
<td></td>
<td></td>
<td>(Eval-Err)</td>
</tr>
<tr>
<td>( \forall f \in { f_{\text{scheme-report-environment}}, f_{\text{null-environment}}, f_{\text{interaction-environment}}, f_{\text{eval}} } )</td>
<td></td>
<td></td>
<td>(Arg-Err)</td>
</tr>
</tbody>
</table>

Table 5.8: extended environment update function \( EU \) and semantic function \([\cdot]\) for \( CS^E \) containing first class environments
<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
<th>Result</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in \text{Sym}, \text{mutable?}(E) = \text{m-}$</td>
<td>$[(\text{define } x \ v)]_E(E) = \text{error}$</td>
<td>(Def-NoChange)</td>
<td></td>
</tr>
<tr>
<td>$x \in \text{Sym}, \text{mutable?}(E) = \text{m-}$</td>
<td>$[(\text{set! } x \ v)]_E(E) = \text{error}$</td>
<td>(Set-NoChange)</td>
<td></td>
</tr>
<tr>
<td>$[e_f]_E(E) = \text{f_set-car}$, $\bot \not\in {[[e_1]]_E(EU(e_f, E)), [\text{set-car!}](EU((\begin{begin e_f e_1}), E)), \ldots}$</td>
<td>$[(e_f \ v_1 \ v_2)]_E(E) = \text{error}$</td>
<td>(Set-Car-NoChange)</td>
<td></td>
</tr>
<tr>
<td>$[e_f]_E(E) = \text{f_set-cdr}$, $\bot \not\in {[[e_1]]_E(EU(e_f, E)), [\text{set-cdr!}](EU((\begin{begin e_f e_1}), E)), \ldots}$</td>
<td>$[(e_f \ v_1 \ v_2)]_E(E) = \text{error}$</td>
<td>(Set-Cdr-NoChange)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.9: updated semantics for unchangeable environments
Chapter 6

Type Inference

In this chapter a type inference system for the functional language $CS$ defined in Chap. 5 is presented. This language is defined generally enough to fit the properties of different existing programming languages. Indeed it is expressible enough such that it can be instantiated easily to Scheme according to the official language report [KCE98]. In Chap. 7 and App. D and E we will do this instantiation, dealing with all language constructs (except of destructive updates and continuations).

In order to simplify the presentation of the type checker we will consider a restricted language $CS$ that differs from $CS^E$ from Chap. 5 by not considering destructive updates. i.e. the functions $f_{set-car!}$ and $f_{set-cdr!}$ and the syntactic keyword $set!$ are not considered and $define$ must not be used to overwrite already bound symbols. Although destructive updates are not considered, our type checker already has several properties preparing it for the extension to destructive updates.

We define the type inference algorithm in terms of an abstract interpretation (see e.g. [CC92], [JN95]). Section 6.2 introduces the different abstract values used for the abstract interpretation in Subsec. 6.2.1. Essentially, the abstract values are given by the type language presented in Chap. 3. The correspondence between abstract values and standard values is presented in Subsec. 6.2.4.

A first version of the abstract semantics of $CS$ is presented in Sec. 6.3. The abstract semantics defined there is not yet able to cope with type variables. Therefore, it is just applicable if a so-called input typing (i.e. a tuple of abstract input values for the main function) can be provided. Furthermore, it fails to terminate for most programs using recursion. We therefore present extensions to the abstract semantics to handle free type variables in Sec. 6.4 and to enforce termination in Sec. 6.5.

The examples presented throughout this chapter are just simple ones illustrating only special
details. The detailed discussion of a larger example is deferred to Chap. 7.

6.1 The Goal of Complete Type Inference

In the rest of this chapter a type inference system is introduced that expects a program in the functional language $CS$, that is guaranteed to terminate for every such input, and that generates an output with the following properties:

- For every function definition $f$ in a program that can be executed without a runtime type error an I/O-representation is generated, i.e. input and output types are inferred such that the input types cover the whole domain of $f$ and the output types cover the whole corresponding set of output values.

- Whenever a program expression is marked with `error` there is an execution path in the program that must fail at this expression.

As usual for many type inference systems an additional input besides a program (i.e. type annotations for certain functions) is not necessary.

A type inference system with the above properties just rejects a program (by marking one of its expressions with `error`) when an execution path in the program exists that must fail whenever followed. This implies that such a type inference system is complete in the sense that type correct programs are never rejected [WB99a, WB00b].

The completeness of the type inference system described in the rest of this chapter corresponds to the soundness of the implementing abstract interpretation according to Def. 2.4.1 as follows: if a sound abstract interpretation yields `error` as result of an expression, then the only result for this expression in the standard semantics is an error. By rejecting just programs containing such expressions the type inference system is complete.

Completeness of the abstract interpretation as defined in Def. 2.4.2 is not reached for the abstract semantics implementing our type inference system for the following reason: distinguishing e.g. `posint`, `negint` and `zero` precisely involves determining the sign of the result of a call e.g. to the addition function. This is just possible when using abstract domains which allow to compare the absolut values of two arguments to +. We accept the loss of information here in order to allow e.g. the abstraction from the absolut values of numbers.
6.2 Domains of Abstract Interpretation

In this section we present a first version of the type inference process that is essentially understandable as the application of a program on tuples of types as abstract values. The set of abstract values used in this context consists of the elements of the type language presented in Chap. 3 with some extensions defined in the following.

The abstract domains are defined in Subsec. 6.2.1. Some additional operations on the domain of abstract environments is given in Subsec. 6.2.3. The correspondence between standard and abstract domains is given in Subsec. 6.2.4 and some semantical properties of the operations on abstract environments are stated in Subsec. 6.2.5.

6.2.1 Abstract Domains

This section describes the domains of the abstract interpretation, i.e. the set of abstract values that serve as input and output values for the abstract interpreter. Essentially, every type described in Chap. 3 can be considered as an abstract value needed for the abstract interpretation. In the following we recall these definitions and define some extensions:

Definition 6.2.1 (primitive abstract values) The primitive abstract values directly correspond to the base types as presented in Def. 3.1.12, i.e. every $b \in B$ is a primitive abstract value. Every primitive abstract domain is ordered by the type hierarchy $\sqsubseteq$ and lifted, i.e. contains the type $\bot$.

The domain of all primitive abstract values (i.e. the set of all base types) is denoted by $\text{Const}_A$ and is partially ordered by $\sqsubseteq$ from Def. 3.4.2. Its elements are called abstract constants.

We extend the set of primitive abstract values by the value $\text{error}$ indicating an error in program execution:

Definition 6.2.2 (error value) The abstract value $\text{error}$ is called the abstract error value.

As for the standard semantics there is a special domain of abstract predefined functions that will be used to denote the predefined function terms. The set of abstract predefined functions is defined as follows:

Definition 6.2.3 (abstract predefined functions) $\text{PFunc}_A$ is called the abstract domain of predefined functions. The abstract predefined functions are denoted by $f^A_{<xyz>}$ where $<xyz>$ is the symbol the function is usually bound to.
For the abstract interpreter just certain properties of the abstract predefined functions are of interest. These properties are given in I/O-representations.

**Definition 6.2.4** Every abstract predefined function $f$ is given by an I/O-representation as given in Def. 3.3.2.

If $r$ is an I/O-representation for a function $f$, this is denoted by $r = D(f)$.

**Example 6.2.5 (abstract predefined functions)** Consider the following I/O-representations of the functions **cons** and **null?**.

\[
\begin{align*}
D(\text{cons}) &= \{A \times B \rightarrow (A \cdot B)\} \\
D(\text{null?}) &= \{\text{nil} \rightarrow \text{Ttrue}, \text{Tfalse} \mid \text{nil} \rightarrow \text{Tfalse}\}
\end{align*}
\]

where **Ttrue** and **Tfalse** represent the boolean abstract values for true and false, respectively, according to App. D.1.1.

**cons** expects exactly two input values of arbitrary types $A$ and $B$ and returns a pair type with the first element of type $A$ and the second of type $B$.

The function **null?** expects exactly one input argument and returns **Ttrue** if this argument is **nil** and **Tfalse**, otherwise.

**Lambda-closures** in the standard semantics combine information about the body of a lambda-expression, the names and order of formal parameters, and the environment in which the lambda-closure was created. When changing from standard to abstract interpretation neither the body of a lambda-expression nor the names and order of formal parameters change as they express syntactic information. All we have to do is to replace structured environments by abstract environments to get the following definition of abstract lambda-closures:

**Definition 6.2.6 (abstract lambda-closures)** The set $LC_A$ called the domain of abstract lambda-closures contains all values of the form $lc_A(e(x_1, \ldots, x_k), FL_A)$ where $e \in \text{Expression}$, $x_1, \ldots, x_k \in \text{Sym}$, $k \in \mathbb{N}_0$ and $FL_A \in \text{FrameList}_A$ is an abstract frame list as defined in Def. 6.2.15.

For every abstract lambda closure $l$ an I/O-representation $D(l)$ is defined analogously to Def. 6.2.4.

For abstract domains essentially the same domain constructors are defined as for the standard domains. The abstract domain constructors are given in the following definition:
Definition 6.2.7 (abstract domain constructors) Comparable to the domain constructors for standard semantics there are

- an abstract domain constructor for product-domains as defined in Def. 5.1.9 for the standard semantics. It is denoted by the product type constructor $\times$ that is one of the tuple like type constructors from Def. 3.1.13.
- union-domains (constructed by the union type constructor from Def. 3.1.13). They correspond to the sum domains of Def. 5.1.9 from the standard semantics, but need not be disjoint unions. A union element occurring in several of the argument domains carries tags of all of its domains.
- free domain constructors: every free type constructor from Def. 3.1.13 is an abstract domain constructor. One of them is the pair type constructor (· ·) corresponding to the pair domain defined in Def. 5.2.18.
- All abstract domains are lifted, i.e. contain the abstract bottom element $\bot$. The bottom elements of different domains are identified.

For all abstract domain constructors there are abstract value constructors denoted in the same way.

Example 6.2.8 (constructed abstract values) Using the type constructors of App. D one can e.g. define the value $t$ of all lists of three numbers as:

$$t := (\text{num} . (\text{num} . (\text{num} . \text{nil}))).$$

The principle of abstract interpretation is to simulate the execution of a program on abstract values. The abstract values that can occur during the abstract execution without an error are called assignable abstract values and defined as follows:

Definition 6.2.9 (assignable abstract values) The set $\text{AssValue}_A$ of assignable abstract values is defined by

$$\text{AssValue}_A = \text{Const}_A \cup \text{PFunc}_A \cup \text{LC}_A \cup \text{Env}_A \cup \bigcup_{j=1}^{w}\{(c_j t_1 \ldots t_{a(c_j)}) | c_j \in \mathcal{K} \text{ from Def. 3.1.13}, t_i \in \text{AssValue}_A\} \cup \bigcup\{(\bigcup t_1 \ldots t_m) | m \in \mathbb{N}_0, t_1, \ldots, t_m \in \text{AssValue}_A\} \cup \bigcup\{(\bigcap t_1 \ldots t_m) | m \in \mathbb{N}_0, t_1, \ldots, t_m \in \text{AssValue}_A\} \cup \{\mathcal{C} t | t \in \text{AssValue}_A\}$$
where Env_A is the set of abstract environments as given in Def. 6.2.15. The semantics of this recursive definition is determined according to [Sch86]. For cyclic values we recall the notion \(\mu X.t\) with \(X\) occurring in \(t\).

To express updates of abstract structures correctly we have to emulate the existence of stores from the standard semantics. The abstract store of an abstract value is defined as follows:

**Definition 6.2.10 (stored abstract values)** The set of stored abstract values is defined by

\[\text{AssValue}^{st}_A = \text{AssValue}_A \times \text{Store}\]

where Store is the same set of stores that is used in the standard semantics.

For every abstract stored value \(v\) the function \(\text{getstore}_A: \text{AssValue}^{st}_A \to \text{Store}\) yields the store of \(v\).

We identify \(\text{AssValue}_A\) with \(\text{AssValue}^{st}_A\) by assuming every abstract value to be a stored one.

Informally, as in the standard semantics a store can be considered as the memory position the stored value occurs in. As in the standard semantics we will consider stores as special symbols that do not occur explicitly in a program. These stores are bound exclusively in the most general frame (called abstract frame state) occurring in every abstract environment.

Abstract environments essentially will be defined analogously to the structured environments in Def. 5.2.6. As the main difference abstract environments bind symbols to abstract values.

In some situations we will have to work on several environments that just differ in the bindings for certain symbols. To express such sets of environments we define environments as pairs of

- a frame list that can bind a symbols to the value \(\text{undef}\) explicitly and that can bind symbols to abstract variables taken from a new set \(\text{Var}_A\).
- a set of substitutions for the abstract variables.

To generate new variables \(X \in \text{Var}_A\) the procedure \(\text{newvar}\) is used that returns an unused variable whenever called. To simplify the handling of abstract environments in the presence of free type variables in Sec. 6.4 we refine the definition of the set \(V_f\) of free type variables in Def. 3.1.1 to contain the set \(\text{Var}_A\).

An abstract frame is defined quite analogously to the definition of frames for the standard semantics. In order to generate frame lists as described above we just need an extension of abstract frames called partial abstract frames that allows to bind symbols to an extended set of values called partial abstract values. The set \(\text{AssValue}_{PA}\) of partial abstract values contains
the value `undef` and variables taken from `Var_A` and values containing these as subterms. Furthermore, instead of the set `Env_A` of abstract environments occurring in `AssValue_A` it contains the set `FrameListP_A` of abstract partial frames that need an additional set of substitutions in order to become environments. The set `AssValueP_A` is defined as follows:

**Definition 6.2.11 (partial abstract values)** The set of all partial abstract values is defined by

\[
AssValueP_A = Const_A \cup PFunc_P_A \cup LC_A \cup FrameListP_A \cup \\
\cup ConstructP_A \cup UnP_A \cup IntP_A \cup CompP_A \cup Var_A \cup \{undef\}
\]

where

- `PFunc_P_A` is defined analogously to `PFunc_A` with the additional options `IN_i \in Var_A`, `OUT_i \in Var_A` and the sets of pairs `(IN_j, OUT_j)` may also contain variables from `Var_A`.
- `FrameListP_A` is a partial abstract frame list as defined in Def. 6.2.12.
- `ConstructP_A := \bigcup_{j=1}^w \{ (c_j \ t_1 \ldots \ t_{a(c_j)}) \mid t_i \in AssValueP_A \}`.
- `UnP_A = \{ (\cup t_1 \ldots t_m) \mid m \in \mathbb{N}_0, t_1, \ldots, t_m \in AssValueP_A \}`.
- `IntP_A = \{ (\cap t_1 \ldots t_m) \mid m \in \mathbb{N}_0, t_1, \ldots, t_m \in AssValueP_A \}`.
- `CompP_A = \{ C(t) \mid t \in AssValueP_A, t does not contain variables \}`.

Abstract frame lists can be understood as abstract environments according to Def. 6.2.15 containing variables. We can therefore consider `AssValue_A` as the subset of `AssValueP_A` of all partial abstract values that cannot be instantiated by a substitution any further.

**Definition 6.2.12 (abstract frame, partial abstract frame, abstract frame list)**

An abstract frame `F \in Frame_A` is a frame type as given by Def. 3.1.17, i.e. `Frame_A = frame`.

As an additional notation `F[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]` denotes the abstract frame that is generated from `F` by binding the symbols `x_i` to the abstract values `v_i`. The former bindings of `x_i` in `F` are discarded.

A partial abstract frame is defined as an abstract frame except of binding symbols `x` to partial abstract values instead of ordinary abstract values. The set of all partial abstract frames is denoted by `FrameP_A`.

A list of abstract frames is called an abstract frame list and is denoted by `FrameList_A`. A list of partial abstract frames called an partial abstract frame list and is denoted by `FrameListP_A`. 

129
Again we consider the set $Frame_A$ of abstract frames as a subset of the set $FrameP_A$ of partial abstract frames.

**Example 6.2.13 (abstract frames)** Let $F_1$, $F_2$, $F_3$ and $F_4$ be defined as follows:

- $F_1 = [x \mapsto \text{int}, s \mapsto \text{string}]$
- $F_2 = [x \mapsto \text{posint}, y \mapsto \text{posint}]$
- $F_3 = F_1[z \mapsto A] = [x \mapsto \text{int}, s \mapsto \text{string}, z \mapsto A]$ with $A \in \text{Var}_A$
- $F_4 = F_2[x \mapsto \text{undef}] = [x \mapsto \text{undef}, y \mapsto \text{posint}]$

Then $F_1$ and $F_2$ are abstract frames and $F_3$ and $F_4$ are partial abstract frames.

$L_1 := (F_1 F_2) \in \text{FrameList}_A$ and $L_2 := (F_3 F_4) \in \text{FrameListP}_A$.

When defining abstract environments we want to group a frame list $L$ with a set $\Sigma$ of substitutions that are appropriate closed type substitutions according to Def. 3.1.41 after being transformed into idempotent substitutions by GIS presented in Def. 4.3.4. This is formally expressed by the following definition:

**Definition 6.2.14 (GIS appropriate substitution)** Let $\sigma$ be a type substitution and $t$ a type term (or more precisely a partial assignable value). $\sigma$ is GIS appropriate for $t$ if $\text{GIS}(\sigma)$ is an appropriate closed type substitution for $t$.

The set of abstract environments can now be defined as follows:

**Definition 6.2.15 (abstract environments)** An abstract environment $E = (L, \Sigma)$ consists of a list $L = (F_1 F_2 \ldots F_k) \in \text{FrameListP}_A$ of partial abstract frames $F_i$ and a finite set $\Sigma \subset \text{TS}$ of type substitutions $\sigma : \text{Var}_A \rightarrow \text{AssValue}_A$ as defined in Def. 3.1.39. For an abstract environment $(L, \Sigma)$ the elements of $\Sigma$ have to be GIS appropriate for $L$. The substitutions $\sigma_j \in \Sigma$ are indexed by elements $j$ taken from an index set $J$. The set of all abstract environments is denoted by $\text{Env}_A$. The set $\Sigma$ of substitutions is called the environment instantiation of $E$.

The store of a stored abstract environment is associated with the environment’s frame list, not with the substitution set.

**Definition 6.2.16 (frame states in abstract environments)** Every abstract environment is extended to contain an abstract frame state $FS_A$ as most general frame. The recursive definition of environments containing frame states is completely analogous to the standard semantics and is omitted here.

---

1This restriction will be dropped in Sec. 6.4 in order to allow free type variables.
For an abstract environment \( E = (F_1 \ldots, F_k, FSA, \Sigma) \) the abstract frame state of \( E \) is represented by

\[
\text{getframestate}_A(E) = (FSA, \Sigma).
\]

Looking up a store in such a frame state is done with respect to a \( \sigma \in \Sigma \) analogously as defined for environments.

The Definitions 6.2.6 and 6.2.9 are mutually recursive with Def. 6.2.15 and Def. 6.2.11 is mutually recursive with Def. 6.2.12. As in Chap. 5 the solution to these recursive definitions is obtained according to [Sch86].

We sometimes write \( E[x \mapsto t] \) to denote the environment that is generated from \( E \) by additionally binding \( x \) to \( t \). The frame the binding of \( x \) is added to is either made clear in the context or does not matter if different bindings of \( x \) cannot occur at the same time in different frames.

For abstract first order environments we use the notions \([\ldots, x \mapsto E', \ldots], F[\ldots, x \mapsto E', \ldots] \) and \( E[\ldots, x \mapsto E', \ldots] \) with \( E' = (FL', \Sigma') \in Env_A \) to denote the binding of \( FL' \) to \( x \).

In contrast to the usual definition of environments, a symbol \( x \) can have several values in an environment \( E = (L, \Sigma) \) in our definition depending on the choice of \( \sigma_j \in \Sigma \). The \( \sigma_j \) are usually not idempotent and therefore must be transformed into idempotent substitutions by the function \( GIS \) from Def. 4.3.4.

**Definition 6.2.17 (abstract environment application)** Let \( x \in Sym \) be a symbol, \( E = (L, \Sigma) \in Env_A \) an abstract environment and \( j \) an index in \( \Sigma \). The value of \( x \) in \( E \) under the index \( j \) is denoted by \( E^{(j)}(x) \) and is determined as follows:

\[
E^{(j)}(x) = \begin{cases} 
GIS(\sigma_j)(F_i(x)) & \text{if } L = (F_1 \ldots F_k) \text{ and } i \leq k \text{ is the smallest index with} \\
\text{undef} & \text{if no such } i \text{ exists}
\end{cases}
\]

If \( F_i(x) = \tilde{L} \in \text{FrameList}_{PA} \) we refine this definition to return \((\tilde{L}, \{\sigma_j\})\) instead of \(GIS(\sigma_j)(\tilde{L})\).

The value set of \( x \) in \( E \) (with substitutions indexed in \( J \)) is denoted by \( E(x) \) and defined as follows:

\[
E(x) = \left( \bigcup \{E^{(j)}(x) \mid j \in J\} \right)
\]

i.e. \( E(x) \) a union type containing all values of \( x \) under a valid index \( j \in J \).
In Def. 6.2.17 above union types \( E(x) \) containing exactly one union element or several identical elements are identified with this element.

**Example 6.2.18 (abstract environments)** Let \( E_1 = (L_1, \{\emptyset\}) \) and \( E_2 = (L_2, \{\sigma_1, \sigma_2\}) \) with \( L_1 \) and \( L_2 \) as in Ex. 6.2.13 and \( \sigma_1 = \{A \leftarrow \text{num}\}, \sigma_2 = \{A \leftarrow \text{string}\}. \)

The application of \( E_1 \) and \( E_2 \) to \( x \) and \( z \) yields:

\[
\begin{align*}
E_1(x) &= \text{int} \\
E_1(z) &= \text{undef} \\
E_2(x) &= \text{int} \\
E_2(z) &= (\cup \text{num} \text{ string})
\end{align*}
\]

### 6.2.2 Stores and Abstract Frame States

As for the standard semantics, there are some abstract state conditions necessary for the values bound in an abstract frame state. The definition of the abstract state condition is comparable to Def. 5.2.10 and Def. 5.2.20 in the standard semantics.

**Definition 6.2.19 (abstract state condition)** An environment

\[
E = ((F_1 \ldots F_k \ FS_A), \{\sigma_1, \ldots, \sigma_1\})
\]

fulfills the abstract state condition if every

\[
E_i = (((GIS(\sigma_i)(F_1)) \ldots GIS(\sigma_i)(F_k) \ GIS(\sigma_i)(FS_A)), \{\emptyset\})
\]

fulfills the following conditions (with \( FS_i \) abbreviating \( GIS(\sigma_i)(FS_A) \)):

- \( FS_i(getstore_A(E_i)) = E_i. \)
- \( getframestate_A(FS_i(S')) = FS_i \) for every \( S' \in \text{Store} \) with \( FS_i(S') \in Env_A \).
- If \( FS_i(S') = C[v] \) then \( FS_i(getstore_A(v)) = v \) where \( C[v] \) denotes a context of \( v \), i.e. a structure containing \( v \) or a context \( C'[v] \) as substructure or an abstract frame or environment binding \( v \) or some \( C'[v] \) to a symbol.

In the following we assume every abstract environment to fulfill the abstract state condition. When bindings of an environment have been changed the abstract state condition for the new environment can be reenforced by the following function \( FS\text{-env-update} \):

\[\text{We omit the frame states occurring in } E_1 \text{ and } E_2 \text{ here.}\]
**Definition 6.2.20 (state condition enforcement)** The function 

\[ FS\text{-}env\text{-}update : \text{Env}_A \rightarrow \text{Env}_A \]

updates the frame state of its argument in order to fulfill the state conditions as follows: For \( E := ((F_1 \ldots F_k \text{FS}_A), \{\sigma_1, \ldots, \sigma_n\}) \) we get 

\[ FS\text{-}env\text{-}update(E) = E' \text{ with } E' := ((F_1 \ldots F_k \text{FS}'_A), \{\sigma_1, \ldots, \sigma_n\}) \]

with \( \text{FS}'_A \) is defined by

\[
\text{FS}'_A(S) = \begin{cases} 
  E' & \text{if } S = \text{getstore}_A(E), \text{FS}_A(S) \in \text{Env}_A \\
  C[E'] & \text{if } \text{FS}_A(S) = C[E] \text{ for a context } C[] \\
  v & \text{if } S = \text{getstore}_A(v), \text{FS}_A(S) = \text{undef} \text{ and } v \text{ occurs in } E' \\
  \text{FS}_A(S) & \text{else}
\end{cases}
\]

A context \( C[E] \) in this definition can be a structure containing \( E \) or a frame or environment binding \( E \) of a context \( C'[E] \) of \( E \).

If \( \text{FS}_A(\text{getstore}_A(E)) \not\in \text{Env}_A \) then \( FS\text{-}env\text{-}update(E) = \text{error} \).

As in the standard semantics frames and environments can change after their definition.\(^3\) When using a frame or environment that has been defined some time before it has to be updated according to the current frame state. This is done by the abstract version of \( \text{update} \) defined as follows:

**Definition 6.2.21 (updating frames and environments)** Let \( F = (\tilde{F}, S) \in \text{Frame}_A^\text{st} \) be a (stored) abstract frame, i.e. \( \text{getstore}_A(F) = S \) and let \( \tilde{E} \) be a simple abstract environment. The bindings of \( F \) corresponding to the current state of \( \tilde{E} \) are given by the frame

\[
\text{update}_A(F, \tilde{E}) = \text{getframestate}_A(\tilde{E})(S) = \text{getframestate}_A(\tilde{E})(\text{getstore}_A(F)) .
\]

\( \text{update}_A \) for (partial) abstract frame lists \( L \) and abstract environments \( E = (L, \Sigma) \) is analogously defined by

\[
\text{update}_A(L, \tilde{E}) = \text{getframestate}_A(\tilde{E})(\text{getstore}_A(L)) .
\]

and

\[
\text{update}_A(E, \tilde{E}) = (\text{getframestate}_A(\tilde{E})(\text{getstore}_A(L)), \Sigma) .
\]

where \( \text{getframestate}_A \) returns the abstract frame state of a given abstract environment.

\(^3\)This is due to destructive updates the abstract interpreter is prepared for.
In the abstract semantics there are situations where several abstract environments have to be combined to a single one. We have to give a condition on the frame states occurring in the two environments. The idea behind this condition is the following: Two environments can be combined only if they have an identical history, i.e. they steam from the same environment but were updated during processing different parts of the program.

We assume that every store is used at most once during one type inference task and therefore stores that are defined in both frame states have denoted that same value which structure is still unchanged. Thus, stores defined in both abstract frame states must contain equal values or at least values with a comparable structure. This condition is formally introduced as follows:

**Definition 6.2.22 (joinable abstract frame states)** Let $FS_A$ and $FS'_A$ be abstract frame states. $FS_A$ and $FS'_A$ are called joinable if every store $S \in \text{dom}(FS_A) \cap \text{dom}(FS'_A)$ fulfills one of the following conditions:

1. If $FS_A(S) \in \text{Frame}_A$ then $FS'_A(S) \in \text{Frame}_A$

2. If $FS_A(S) = ((F_1 \ldots F_k FS_A), \Sigma) \in \text{Env}_A$ then $FS'_A(S) = ((F'_1 \ldots F'_k FS'_A), \Sigma') \in \text{Env}_A$ and

   \[ \text{getstore}_A(F_i) = \text{getstore}_A(F'_i) \] for all $i = 1, \ldots, k$.

3. If $FS_A(S) = (c v_1 \ldots v_{a(c)})$ then $FS'_A(S) = (c v'_1 \ldots v'_{a(c)})$ for every free type constructor $c \in \mathcal{K}$.

4. $FS_A(S) = FS'_A(S)$ in every other case.

Note that the cases of Def. 6.2.22 yield a complete and disjoint case distinction on $FS_A(S)$ and that for every case on $FS'_A(S)$ there is exactly one valid case on $FS_A(S)$. The property of joinability of frame states is therefore commutative.

### 6.2.3 Properties and Operations for Abstract Environments

In this section simple abstract environments and operations on abstract environments are defined. A simple abstract environment is an environment $E = (L, \Sigma)$ not making use of the possibilities given by $\Sigma$:

**Definition 6.2.23 (simple abstract environments)** We call an abstract environment $E = (L, \Sigma)$ simple if $|\Sigma| = 1$, i.e. $\Sigma = \{\sigma\}$, and if $\text{GIS}(\sigma)(L) \in \text{FrameList}_A$, i.e. it just contains abstract frames instead of partial abstract frames. For a simple abstract environment
we sometimes identify \( E \) with \( E' = (L', \{ \emptyset \}) \) or even with \( L' := GIS(\sigma)(L) \). The set of all simple abstract environments is denoted by \( SEnv_A \).

In the definition above \( GIS(\sigma)(L) \) cannot contain variables from \( Var_A \) because \( \sigma \) must be \( GIS \) appropriate for \( L \). The condition \( GIS(\sigma)(L) \in FrameList_A \) additionally states that explicit bindings to \( \texttt{undef} \) are not allowed.

Note that simple abstract environments are essentially the same as environment types from Def. 3.1.19 with bindings of symbols to abstract values as an extension of types.

**Example 6.2.24 (simple abstract environments)** Let \( E_1 \) and \( E_2 \) be as in Ex. 6.2.18, i.e.

\[
E_1 = (([x \mapsto \text{int}, s \mapsto \text{string}] [x \mapsto \text{posint}, y \mapsto \text{posint}]), \{\emptyset\})
\]

and

\[
E_2 = (([x \mapsto \text{int}, s \mapsto \text{string}, z \mapsto A] [x \mapsto \texttt{undef}, y \mapsto \text{posint}]),
\quad \{\{A \leftarrow \text{num}\}, \{A \leftarrow \text{string}\}\}).
\]

Then \( E_1 \in SEnv_A \) is a simple abstract environment, but \( E_2 \in Env_A \) is not because it contains two substitutions and an explicit binding of \( \texttt{undef} \) in the second frame.

Abstract environments express sets of simple abstract environments with a common structure. The meaning of abstract environments in terms of simple abstract environments is formalized in the following definition:

**Definition 6.2.25 (denoted sets of simple abstract environments)** Every abstract environment \( E = (L, \Sigma) \) expresses a set \( S(E) \) of simple abstract environments defined by

\[
S(E) = \{ (CAE(L, \sigma), \{\sigma\}) \mid \sigma \in \Sigma \}
\]

where \( CAE : FrameList_A \times TS \rightarrow FrameList_A \) (\( CAE \) stands for clean up abstract environment) eliminates all bindings of symbols \( x \) with \( GIS(\sigma)(F_i)(x) = \texttt{undef} \) from the frames \( F_i \) occurring in its first argument \( L \).

The following lemma states that looking up the bindings of a symbol yields the same result both in an abstract environment and in the corresponding set of simple abstract environments:

**Lemma 6.2.26** Let \( E \) be an environment and \( x \) a symbol. Then

\[
E(x) = (\cup \{ E'(x) \mid E' \in S(E) \}).
\]
Example 6.2.27 (simplify abstract environments) Consider the abstract environment $E_2$ from Ex. 6.2.18, i.e.

$$E_2 = ((\begin{array}{l} x \mapsto \text{int}, \ y \mapsto \text{string}, \ z \mapsto A \\ x \mapsto \text{undefined}, \ y \mapsto \text{posint} \end{array}) \ \ {\{A \leftarrow \text{num}\}, \{A \leftarrow \text{string}\}}).$$

Then $S(E_2)$ contains the following two simple abstract environments:

- $E' = ((\begin{array}{l} x \mapsto \text{int}, \ y \mapsto \text{posint} \end{array}) \ \ {\{A \leftarrow \text{num}\}})$
- $E'' = ((\begin{array}{l} x \mapsto \text{int}, \ y \mapsto \text{posint} \end{array}) \ \ {\{A \leftarrow \text{string}\}})$

When evaluating an expression under an abstract environment $E = (L, \Sigma)$ several different return values may result from the different substitutions in $\Sigma$. To handle every substitution and the corresponding return value uniformly we extend the definition of abstract environments to hold an output value of an expression, too:

Definition 6.2.28 ((simple) extended abstract environment) An extended abstract environment is an abstract environment binding a special symbol $\text{return}$ (that does not occur in the program code) in the most general frame $FS$. The set of all extended abstract environments is denoted by $\text{Env}_E$. An extended abstract environment that is also a simple abstract environment is called simple extended abstract environment. $\text{SEnv}_E$ is the set of all simple extended abstract environments.

We consider abstract frame states additionally binding $\text{return}$ as special abstract frame states and get $\text{Env}_E \subset \text{Env}$ and $\text{SEnv}_E \subset S\text{Env}$. 

Example 6.2.29 (extended abstract environment) An extended abstract environment is

$$E = ((\begin{array}{l} x \mapsto \text{int}, \ s \mapsto \text{string} \end{array}) \ \ {\{A \leftarrow \text{num}\}}), \{\emptyset\})$$

with some abstract frame state $FS_E$. $E$ is a simple extended abstract environment, too.

The definition of abstract environments as given in Def. 6.2.15 does not provide a unique representation for every abstract environment: If $E = (L, \Sigma)$ is an abstract environment and a symbol $x \in \text{Sym}$ is bound to the same abstract value $v \in \text{AssValue}_E$ according to all substitutions $\sigma \in \Sigma$ one can choose whether to express the binding of $x$ to $v$ directly in $L$ or to bind $x$ to some new $X \in \text{Var}_E$ and to add $X \leftarrow v$ to all $\sigma \in \Sigma$. A normal form for these cases is provided by the following definition of canonical abstract environments:
Definition 6.2.30 (canonical abstract environments)  A canonical abstract environment is an abstract environment $E = (L, \Sigma) \in \text{Env}_A$ with $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ such that there is no $X \in \text{Var}_A$ and no $v \in \text{AssValue}_A$ with $\forall i \in \{1, \ldots, k\} . X \leftarrow v \in \sigma_i$.

The goal of the following algorithm $\text{canonical}$ is to transform every abstract environment into an equivalent canonical abstract environment.

Definition 6.2.31 (generating canonical abstract environments) The following algorithm $\text{canonical}$ defines the function $\text{canonical} : \text{Env}_A \rightarrow \text{Env}_A$:

Algorithm: canonical  
Input: An abstract environment $E = ((F_1 \ldots F_k \text{FS}_A), \{\sigma_1, \ldots, \sigma_n\}) \in \text{Env}_A$  
Output: An abstract environment $E' \in \text{Env}_A$

$\text{Vars} := \bigcap_{i=1}^{n} \{x \in \text{Var}_A \mid x \text{ instantiated in } \sigma_i\}$

for $x \in \text{Vars}$ do
  $\text{Val} := \bigcup_{i=1}^{n} \{v \in \text{AssValue}_A \mid x \leftarrow v \in \sigma_i\}$
  if $\text{Val} = \{v'\}$ for some $v' \in \text{AssValue}_A$ then
    $\sigma' := \text{GIS}(\{x \leftarrow v'\})$
    $E := ((\sigma'(F_1) \ldots \sigma'(F_k) \sigma'(\text{FS}_A)), \{\sigma' \circ \sigma_1|_{\text{dom}(\sigma_1) \setminus \{x\}}, \ldots, \sigma' \circ \sigma_n|_{\text{dom}(\sigma_n) \setminus \{x\}}\})$
  return $\text{FS-env-update}(E)$

with $\text{FS-env-update}$ as presented in Def. 6.2.20.

The property of transforming every abstract environment into a canonical abstract environment is formally proven in the following lemma:

Lemma 6.2.32 For every abstract environment $E \in \text{Env}_A$ the result $E' := \text{canonical}(E)$ is a canonical abstract environment.

Proof: See App. C.1, Page 252. 

For every environment $E$ the function $\text{canonical}$ returns an equivalent environment $\text{canonical}(E)$.

This will be stated in a semantical context in Lemma 6.2.50.

When information from several branches of a program comes together the problem of combining abstract environments with a common number of frames to a single abstract environment arises. The combination of two abstract environments is done by the following algorithm:
Definition 6.2.33 (combining abstract environments) The function

\[ \text{combine} : \text{Env}_A \times \text{Env}_A \rightarrow \text{Env}_A \]

combines two abstract environments. It is defined by the following algorithm:

**Algorithm:** combine

**Input:** Two abstract environments

\[ E_1 = ((F_{1,1} \ldots F_{1,k} F_{S1}), \Sigma_1), \quad E_2 = ((F_{2,1} \ldots F_{2,n} F_{S2}), \Sigma_2) \in \text{Env}_A \]

with joinable abstract frame states \( F_{S1} \) and \( F_{S2} \).

**Output:** An abstract environment \( E' \in \text{Env}_A \)

if \( k \neq n \) then return \( \text{undef} \) (* input environments must have same number of frames *)

else

\[ \text{for } i \in \{1, \ldots, k+1\} \text{ do} \]

\[ \text{Symbols} := \text{dom}(F_{1,i}) \cup \text{dom}(F_{2,i}) \]

\[ \text{for } s \in \text{Symbols} \text{ do} \]

\[ \text{if } s \notin \text{dom}(F_{1,i}) \text{ then} \]

\[ \text{add the binding } s := \text{undef} \text{ to } F_{1,i}. \]

\[ \text{if } s \notin \text{dom}(F_{2,i}) \text{ then} \]

\[ \text{add the binding } s := \text{undef} \text{ to } F_{2,i}. \]

(* Now s is bound in both frames *)

\[ \text{if } F_{1,i}(s) \neq F_{2,i}(s) \text{ then} \]

\[ X := \text{newvar}() \] (* introduce new variable for s *)

\[ \text{Add } X \leftarrow F_{1,i}(s) \text{ to every } \sigma \in \Sigma_1 \]

\[ \text{Add } X \leftarrow F_{2,i}(s) \text{ to every } \sigma \in \Sigma_2 \]

\[ \text{Change the binding of } s \text{ in } F_{1,i} \text{ and } F_{2,i} \text{ to } X. \]

return \( \text{FS-env-update}(((F_{1,1} \ldots F_{1,k} F_{S1}), \Sigma_1 \cup \Sigma_2)) \)

The following lemma states the commutativity of \( \text{combine} \).

**Lemma 6.2.34 (commutativity of \( \text{combine} \))** The function \( \text{combine} \) is commutative, i.e. for two abstract environments \( E_1, E_2 \in \text{Env}_A \) we have:

\[ \text{combine}(E_1, E_2) = \text{combine}(E_2, E_1). \]

**Proof:** See App. C.1, Page 252.

Besides commutativity the function \( \text{combine} \) is associative and by combining two abstract environments the set of denoted standard environments does not change. These statements will be proven as a semantical basis in Subsec. 6.2.5.
Example 6.2.35 (combine) Consider the environments $E_1$ and $E_2$ from Ex. 6.2.18 on Page 132 (again not considering the frame states), i.e.

$$E_1 = ((F_1 = [x \mapsto \text{int}, s \mapsto \text{string}] F_2 = [x \mapsto \text{posint}, y \mapsto \text{posint}]), \{\sigma_1\})$$

$$E_2 = ((F_3 = [x \mapsto \text{int}, s \mapsto \text{string}, z \mapsto A] F_4 = [x \mapsto \text{undef}, y \mapsto \text{posint}]), \{\sigma_2,\sigma_3\})$$

with:

- $\sigma_1 = \{\}$
- $\sigma_2 = \{A \leftarrow \text{num}\}$
- $\sigma_3 = \{A \leftarrow \text{string}\}$

The calculation of combine($E_1, E_2$) is done as follows:

- The frames $F_1$ and $F_3$ are combined to a frame $F'$ binding the symbols $x$, $s$ and $z$.
  - For $x$ and $s$ there is nothing to do because these symbols are bound to the same type in both frames.
  - For $z$ we add the binding $z \leftarrow \text{undef}$ to $F_1$. We then introduce a new variable $X$ and update $\sigma_1$ by adding $X \leftarrow \text{undef}$. $\sigma_2$ and $\sigma_3$ are extended by the binding $X \leftarrow A$, respectively. Then we change the bindings of $z$ to $X$ in both frames.

As a result we get the frame

$$F' = ([x \mapsto \text{int}, s \mapsto \text{string}, z \mapsto X]$$

and the intermediate substitutions

- $\sigma_1 = \{X \leftarrow \text{undef}\}$,
- $\sigma_2 = \{A \leftarrow \text{num}, X \leftarrow A\}$,
- $\sigma_3 = \{A \leftarrow \text{string}, X \leftarrow A\}$.

- $F_2$ and $F_4$ are combined to $F''$ binding the symbols $x$ and $y$.
  - For $y$ there is nothing to do.
  - For $x$ we introduce a new variable $Y$ and add the binding $Y \leftarrow \text{posint}$ to $\sigma_1$ and $Y \leftarrow \text{undef}$ to $\sigma_2$ and $\sigma_3$. Then the binding of $x$ is replaced by $Y$ in both frames.
The result frame is

\[ F'' = [x \mapsto Y, y \mapsto \text{posint}] \]

with the intermediate substitutions

- \( \sigma_1 = \{ X \leftarrow \text{undef}, Y \leftarrow \text{posint} \} \),
- \( \sigma_2 = \{ A \leftarrow \text{num}, X \leftarrow A, Y \leftarrow \text{undef} \} \),
- \( \sigma_3 = \{ A \leftarrow \text{string}, X \leftarrow A, Y \leftarrow \text{undef} \} \).

The combined frames together with the updated substitutions yield the following result environment:

\[ \text{combine}(E_1, E_2) = \left( ([x \mapsto \text{int}, s \mapsto \text{string}, z \mapsto X] [x \mapsto Y, y \mapsto \text{posint}]), \right. \\
\left. \{\{X \leftarrow \text{undef}, Y \leftarrow \text{posint}\}, \{A \leftarrow \text{num}, X \leftarrow A, Y \leftarrow \text{undef}\}, \right. \\
\left. \{A \leftarrow \text{string}, X \leftarrow A, Y \leftarrow \text{undef}\} \right) \).

In certain situations (e.g. as a result of the algorithm \text{combine}) we can get extended abstract environments of the following form: \( E = (L, \{\sigma_1, \ldots, \sigma_k\}) \) with at least one pair \( \sigma_i, \sigma_j \) of substitutions that just differ in one variable \( X \) where \( X \) just occurs as the value bound to \text{return} in \( L \). We will combine such substitutions to one substitution that instantiates \( X \) with the union of its former values. This is formalized in the following definition:

**Definition 6.2.36 (compressing extended abstract environments)** The following algorithm \text{compress} defines a function \text{compress} : \text{EnvE}_A \rightarrow \text{EnvE}_A:

**Algorithm:** \text{compress}

**Input:** An extended abstract environment \( E \in \text{EnvE}_A \)

**Output:** An extended abstract environment \( E' \in \text{EnvE}_A \)

Let \( E = (L, \Sigma) \) with \( \Sigma = \{\sigma_1, \ldots, \sigma_k\} \)

\[ \text{if } \text{return} \text{ is not bound to a variable } \text{then return } E. \]

Let \( X \) be the value bound to \text{return}

\[ \text{if there is a further occurrence of } X \text{ in } L \text{ besides } \text{return} \text{ or at a right hand side of } \Sigma \text{ then return } E. \]

\text{while there are two different substitutions } \sigma_i \text{ and } \sigma_j \text{ that are equal except on } X \text{ do}

\[ \sigma' = \sigma_i|_{\text{dom}(\sigma_i) \setminus \{X\}} \circ \{X \leftarrow (\cup \sigma_i(X) \sigma_j(X))\} \]

\[ E := (L, \Sigma \setminus \{\sigma_i, \sigma_j\} \cup \sigma') \]

\text{return } \text{FS-env-update}(E)
with $FS$-env-update defined in Def. 6.2.31.

The equivalence of the argument and the result of $compress$ will be stated in Lemma 6.2.53 after defining the semantics of the abstract domains.

**Example 6.2.37** ($compress$) Consider the extended abstract environment $E$ defined as follows:

$$E = (([x \mapsto \text{num}, y \mapsto \text{num}] [s \mapsto \text{string}] \ FS_A[\text{return} \mapsto X]),$$

$$\{\{X \leftarrow \text{num}\}, \{X \leftarrow \text{string}\}, \{X \leftarrow \text{bool}\}\}$$

with an arbitrary frame state $FS_A$. Then $compress(E) = E'$ with

$$E' = (([x \mapsto \text{num}, y \mapsto \text{num}] [s \mapsto \text{string}] \ FS'_A[\text{return} \mapsto X]),$$

$$\{X \leftarrow (\cup \text{num} \text{ string} \text{ bool})\}$$

with $FS'_A$ calculated by $FS$-env-update.

### 6.2.4 Correspondence between Abstract and Standard Values

In principle abstract interpretation executes a given program on a set of abstract values. These abstract values approximate the program’s behaviour on those standard values that are denoted by the abstract ones. This section defines the sets of standard values that are denoted by an abstract value.

In literature concerning abstract interpretation [CC92, Cou97, JN95] the function mapping abstract values to sets of standard values is called concretization map and is denoted by $\gamma$. On the other hand, in type theory [MPS86], [Dam94] types are understood as syntactic elements, and the set of denoted values is given by a semantic function often written similar to $\llbracket \cdot \rrbracket$ or $f[\cdot]$ for a function name $f$. In this work we use a notation spotting on the syntactic properties of types. But since $\llbracket \cdot \rrbracket$ is used for the semantics of functional programs the semantics of types is denoted by $\langle \llbracket \cdot \rrbracket \rangle$ already used in Chap. 3.

Essentially abstract values are given by the types defined in Chap. 3. The meaning of types is given by their semantics:

**Remark 6.2.38** (meaning of types as abstract values) For all types $t \in T$ (which all fulfill $t \in \text{AssValue}_A$ the set of standard values represented by $t$ is given by $\langle \llbracket t \rrbracket \rangle$ as presented in Subsec. 3.1.2 and extended in Sec. 3.2.
Remark 6.2.38 motivates to extend the semantic function \( \langle \cdot \rangle \) from types to the other abstract values as well. Therefore, \( \langle \cdot \rangle \) needs some extensions.

The semantics of stored abstract values is a set of stored values with the same store defined as follows:

**Definition 6.2.39 (semantics of stored abstract values)** Let \( v_A = (v'_A, S) \) be a stored abstract value. The semantics of \( v_A \) is given by

\[
\langle v_A \rangle = \{ (v', S) \mid v' \in \langle v'_A \rangle \}.
\]

Every abstract predefined function represents exactly one predefined function that is identified by the symbol it is initially bound to:

**Definition 6.2.40 (meaning of abstract predefined functions)** For a symbol \(<xyz>\) that is initially bound to a predefined function the semantics of the corresponding abstract predefined function is defined by:

\[
\langle f_A^{<xyz>} \rangle = \{ f^{<xyz>} \}.
\]

The semantics of an I/O-representation is given by the set of all functions whose domain is covered by the \( IN_i \) (where each \( IN_i \) indeed covers a part of the domain) and every \( OUT_i \) covers all results for a valid input taken from the corresponding \( IN_i \):

**Definition 6.2.41 (semantics of I/O-representations of abstract functions)** Let \( f \) be an abstract function (predefined or lambda-closure) expressed by the I/O-representation

\[
D(f) = \{ (IN_1, OUT_1), \ldots, (IN_k, OUT_k) \}.
\]

The semantics of \( f \) or equivalently \( D(f) \) is defined as

\[
\langle D(f) \rangle := \{ g \in PFunc \cup LC \mid \forall i \in \{1, \ldots, k\}. dom(g) \cap \langle IN_i \rangle \neq \emptyset, dom(g) \subseteq \bigcup_{i=1}^{k} \langle IN_i \rangle, \forall i. g(dom(g) \cap \langle IN_i \rangle) \subseteq \langle OUT_i \rangle \}.
\]

For the I/O-representations used to describe the predefined functions we furthermore need the following assumption:

**Assumption 6.2.42** Let \( f^{A}_{<xyz>} \) be an abstract predefined function expressed by the I/O-representation \( D(f^{A}_{<xyz>}) \). Then \( f^{<xyz>} \in \langle D(f^{A}_{<xyz>}) \rangle \).
This assumption holds for the abstract predefined functions presented in App. E. This is easily provable by considering these predefined functions one by one.

Before discussing the semantics of frames and environments partial abstract values are considered. For the partial abstract values $v_A$ that are not contained in $\text{AssValue}_A$ the semantics depends on the reason why $v_A \notin \text{AssValue}_A$ holds:

**Definition 6.2.43 (meaning of partial abstract values)**

- If $v_A$ contains $\text{undef}$ then $\{v_A\} = \{\text{undef}\}$.

- If $v_A$ contains variables taken from $\text{Var}_A$ (but not $\text{undef}$) then the semantics of $v_A$ is given under a set $\Sigma$ of substitutions $\sigma \in \Sigma$ that are GIS appropriate for $v_A$:

$$\{v_A\}_\Sigma = \bigcup_{\sigma \in \Sigma} \{\text{GIS}(\sigma(v_A))\}.$$ 

To explain the meaning of abstract environments we start with the values represented by abstract frames and simple abstract environments:

**Definition 6.2.44 (meaning of abstract frames and abstract frame lists)**

Every abstract frame $F^A = [x_1 \mapsto v_1^A, \ldots, x_k \mapsto v_k^A] \in \text{Frame}_A$ with $v_i^A \in \text{AssValue}_A$ exactly represents the set of frames mapping the symbols $x_i$ to values $v_i$ represented by $v_i^A$, i.e.:

$$\{(x_1 \mapsto v_1^A, \ldots, x_k \mapsto v_k^A)\} := \{[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] \in \text{Frame} \mid v_i \in \{v_i^A\}\}.$$ 

An abstract frame list $L = (F_1^A \ldots F_n^A) \in \text{FrameList}_A$ represents all structured environments $E \in \text{EnvS}$ built as lists of frames $F_i$ denoted by $F_i^A$, i.e.:

$$\{(F_1^A \ldots F_n^A)\} := \{(F_1 \ldots F_n) \in \text{EnvS} \mid F_i \in \{F_i^A\} \text{ for all } i\}.$$ 

The meaning of a partial abstract frame or an abstract frame list again depends on a set of GIS appropriate substitutions:

**Definition 6.2.45 (meaning of partial abstract frames and partial abstract frame lists)**

Let $F_A \in \text{Frame}_A$ be a partial abstract frame, $L_A \in \text{FrameList}_A$ a partial abstract frame list and $\Sigma$ an environment instantiation that is GIS appropriate for $F_A$ and $L_A$. Then the meaning of $F_A$ under $\Sigma$ is the union of sets represented by $F_A$ after applying an idempotent substitution generated from an element of $\Sigma$, i.e.:

$$\{F_A\}_\Sigma := \bigcup_{\sigma \in \Sigma} \{\text{GIS}(\sigma)(F_A)\}.$$
Analogously the meaning of \( L_A \) is defined by applying all substitutions from \( \Sigma \) and uniting the resulting meanings:

\[
\langle L_A \rangle_{\Sigma} := \bigcup_{\sigma \in \Sigma} \langle GIS(\sigma)(L_A) \rangle.
\]

The meaning of an abstract environment is directly given by the meaning of its partial abstract frame list under its environment instantiation:

**Definition 6.2.46 (meaning of abstract environments)** For an abstract environment \( E = (L, \Sigma) \in Env_A \) the semantics is

\[
\langle E \rangle := \langle L \rangle_{\Sigma}.
\]

Analogously to partial abstract frame lists the meaning of abstract lambda closures has to be given under an environment instantiation appropriate for the abstract frame list stored as definition environment:

**Definition 6.2.47 (meaning of abstract lambda closures)** Consider an abstract lambda closure \( l := lc_A(e(x_1, \ldots , x_k), FL_A) \in LC_A \) and let \( \Sigma \) be an environment instantiation appropriate for \( FL_A \). The meaning of \( l \) under \( \Sigma \) is the set of all lambda closures with expression \( e \), formal parameters \( x_1, \ldots , x_k \) and a definition environment represented by \( FL_A \) under \( \Sigma \), i.e.:

\[
\langle lc_A(e(x_1, \ldots , x_k), FL_A) \rangle_{\Sigma} := \{ lc(e(x_1, \ldots , x_k), E) \mid E \in \langle FL_A \rangle_{\Sigma} \}.
\]

We have now defined the meaning of abstract values given by \( AssValue_A \) with some of them depending on an environment instantiation. For all those abstract values which meaning is independent from an environment instantiation we can define the meaning under an arbitrary environment instantiation \( \Sigma \) by

\[
\langle v_A \rangle_{\Sigma} := \langle v_A \rangle.
\]

This makes it possible to use \( \langle \cdot \rangle_{\Sigma} \) instead of \( \langle \cdot \rangle \) in every occurrence. When the environment instantiation \( \Sigma \) is clear from the context or does not matter we often use the notion \( \langle \cdot \rangle \) again instead of \( \langle \cdot \rangle_{\Sigma} \).

Up to now we have defined the semantics of abstract values that can express every successful execution of a program. In the case of errors during program execution we need the abstract error value that denotes exactly the standard error value:

**Definition 6.2.48 (error value)** The correspondence between the standard and the abstract error value is given as \( \langle error \rangle = \{ error \} \).
We extend the semantics of every abstract value to include the standard value *error*. Informally, we are not interested whether the standard semantics *may* fail for certain inputs, but whether it *must* fail for all inputs.

As stated in Def. 6.2.7 every abstract domain is lifted and contains the abstract value ⊥. The semantics of this bottom element just contains the bottom element of the standard semantics, i.e.

\[ \llbracket \bot \rrbracket = \{ \bot \}. \]

Furthermore, the bottom element of the standard semantics is included in the semantics of every abstract value:

\[ \bot \in \llbracket t \rrbracket \text{ for every type } t. \]

### 6.2.5 Semantic Properties of Abstract Environment Operations

With the semantics of the abstract values defined in Subsec. 6.2.4 we can now prove some semantical properties of operations on the abstract environments.

The correspondence between abstract environments and sets of simple abstract environments is stated in the following lemma:

**Lemma 6.2.49** Let \( E \) be an abstract environment. Then

\[ \llbracket E \rrbracket = \bigcup_{E' \in S(E)} \llbracket E' \rrbracket. \]

**Proof:** See App. C.1, Page 253.

The function *canonical* does not change the set of denoted environments:

**Lemma 6.2.50** For every abstract environment \( E \in Env_A \) the result \( \text{canonical}(E) \) expresses the same set of environments in the standard semantics as \( E \), i.e.

\[ \forall E \in Env_A. \llbracket E \rrbracket = \llbracket \text{canonical}(E) \rrbracket. \]

**Proof:** See App. C.1, Page 254.

The following lemma states that the set of simple abstract environments described by the two abstract environments \( E_1, E_2 \) remains unchanged in the result \( E = \text{combine}(E_1, E_2) \) of the function *combine* presented in Def. 6.2.33:
Lemma 6.2.51 (equivalence of combined abstract environments) Let $E_1, E_2 \in \text{Env}_A$, $E = \text{combine}(E_1, E_2)$. The function combine leaves the set of denoted environments unchanged, i.e. $\langle E \rangle = \langle E_1 \rangle \cup \langle E_2 \rangle$.

Proof: See App. C.1, Page 255.

With respect to semantical equality the function combine is associative:

Lemma 6.2.52 (associativity of combine) The function combine is associative with respect to the denoted sets of environments in the standard semantics, i.e. for three abstract environments $E_1, E_2, E_3 \in \text{Env}_A$ we have:

$$\langle \text{combine}(E_1, \text{combine}(E_2, E_3)) \rangle = \langle \text{combine}(\text{combine}(E_1, E_2), E_3) \rangle.$$

Proof: See App. C.1, Page 256.

By the commutativity proven in Lemma 6.2.34 combine does not prefer the information given in one of its two arguments. By the associativity we can apply combine to more then two arguments in an arbitrary order. Thus, in the following we will sometimes write $\text{combine}(E_1, E_2, \ldots, E_j)$ for an arbitrary sequence of calls to combine with arguments $E_1, \ldots, E_j$ in an arbitrary order.

For the function compress we can state that it always returns a semantically equivalent environment:

Lemma 6.2.53 (equivalence of compressed environments) Let $E \in \text{Env}_{EA}$ be an extended abstract environment. Then

$$\langle E \rangle = \langle \text{compress}(E) \rangle.$$

Proof: See App. C.1, Page 256.

6.3 The Abstract Semantic Function

We now define an abstract semantic function that covers most of the language syntax introduced in Sec. 5.2 and is therefore expressible enough to be instantiated to almost the full expressive power of Scheme according to [KCE98]. Just the destructive updates described in Sec. 5.2.1 are not considered here. Some helper functions used by the abstract semantic function are already prepared for coping with destructive updates and an outlook towards an abstract semantics with destructive updates is given in Chap. 8.
6.3.1 Auxiliary Functions of the Abstract Semantics

The abstract semantic function defined in this section depends on some functions that declare the semantics of low level syntactic objects of $CS$. These functions are defined first.

Constant terms occurring in a program are essentially denoted by the value assignments (cf. Sec. 3.2) of themselves:

**Definition 6.3.1 (abstract semantics of constant terms)** The abstract semantics of constant terms $ct \in \text{ConstT}$ is given by the function $\text{denoteconst}_A : \text{ConstT} \rightarrow \text{Const}_A$ defined by $\text{denoteconst}_A(ct) = A(\text{denoteconst}(ct)) \in \text{Const}_A$.

As explained in Sec. 5.2.3 the application of $f_{\text{eval}}$ in the standard semantics enforces the transformation of values to expressions. When we emulate this behaviour in the abstract semantics, we must transform abstract constants into expressions. Therefore, the abstract semantics is defined for an extended set of expressions called abstract expressions that are defined as follows:

**Definition 6.3.2 (abstract expressions)** The set $\text{Expression}_A$ of abstract expressions is defined as the smallest set fulfilling:

1. All expressions $e \in \text{Expression}$ are also elements of $\text{Expression}_A$.
2. A set $\text{ConstT}_A$ of abstract constant expressions with $|\text{ConstT}_A| = |\text{Const}_A|$ is a subset of $\text{Expression}_A$.
3. The definition of $\text{Expression}_A$ allows subexpressions according to (2) where subexpressions occur in the definition of Expression in Def. 5.2.1.

**Example 6.3.3 (abstract constant expressions)** Considering e.g. the abstract constants given in App. D there is a abstract constant expression for every abstract constant denoted in the same way, e.g. int, bool and sym.

For the new abstract constant expressions (or abstract constant terms) the semantics is given by a function $\text{denoteabsconst}_A$:

**Definition 6.3.4 (abstract semantics of abstract constant terms)** The semantics of the abstract constant terms is defined by a function

$$\text{denoteabsconst}_A : \text{ConstT}_A \rightarrow \text{Const}_A.$$
For every abstract constant term the function denoteabsconst\(_A\) returns the abstract constant denoted in the same way.

**Example 6.3.5 ((abstract) constant terms)** For \(5 \in \text{Const}\(_T\) and \(\text{int} \in \text{Const}\(_T\)\(_A\) we have

\[
\text{denoteconst}\(_A\)(5) = A(5), \quad \text{denoteabsconst}\(_A\)(\text{int}) = \text{int}.
\]

For conditional expressions we define two functions \(AT\) and \(AF\) that will be used to express the boolean interpretation of abstract values. Informally, \(AT\) returns true if all values denoted by the given abstract value are interpreted as \text{true} in the standard semantics. \(AF\) yields true for all abstract values just denoting values that are interpreted as \text{false}. The definition of \(AT\) and \(AF\) is as follows:

**Definition 6.3.6 (abstract boolean interpretation)** \(AT : \text{AssValue}\(_A\) \rightarrow \text{bool}\) approximates the test whether all values denoted by an abstract value are true, i.e. \(AT\) fulfills the property

\[
\forall v_A \in \text{AssValue}\(_A\). AT(v_A) = \text{true} \Rightarrow \forall v \in \langle v_A \rangle . \text{bi}(v) = \text{true}.
\]

Analogously, \(AF : \text{AssValue}\(_A\) \rightarrow \text{bool}\) approximates the test whether all denoted values are interpreted as \text{false}. \(AF\) fulfills the property

\[
\forall v_A \in \text{AssValue}\(_A\). AF(v_A) = \text{true} \Rightarrow \forall v \in \langle v_A \rangle . \text{bi}(v) = \text{false}.
\]

**Example 6.3.7 (abstract boolean interpretation)** For the types \(\text{int}\), \(A(\#f)\) and \(\text{bool}\) with the definition of \(\text{bi}\) given in Ex. 5.1.13 according to [KCE98] we get:

- \(AT(\text{int}) = \text{true}, AF(\text{int}) = \text{false}\).
- \(AT(A(\#f)) = \text{false}, AF(A(\#f)) = \text{true}\).
- \(AT(\text{bool}) = \text{false}, AF(\text{bool}) = \text{false}\).

### 6.3.1.1 Auxiliary Functions for Environments

To allow abstract environments as first class values we need definitions of master copies of abstract environments generated by the environment generation functions. This definition is given analogously to Def. 5.2.27:

**Definition 6.3.8 (master copies of environments)** \(AE_0, AE_{SR}, AE_I \in \text{Env}\(_A\)\) are predefined abstract environments with the following properties:
• $AE_0 = (\emptyset_A, \emptyset)$ where $\emptyset_A$ denotes an abstract frame containing no bindings. $\emptyset_A$ cannot occur in any other abstract environment.

• $AE_{SR} = (AF_{SR}, \emptyset)$ where $AF_{SR}$ is an abstract frame that just occurs in $AE_{SR}$ and contains the bindings to all predefined functions according to App. E.\textsuperscript{5}

• $AE_I = (AF_I, \emptyset)$ where $AF_I$ is the top level frame of the read-evaluate-print loop inside of the abstract interpreter. For the moment $AE_I$ has the same bindings as $AE_{SR}$.\textsuperscript{6}

The abstract frame of all abstract stores is common to all these environments.

Informally, $AE_0$, $AE_{SR}$ and $AE_I$ are the abstract versions of the environments $E_0$, $E_{SR}$ and $E_I$ given in Def. 5.2.27. Since some of the abstract environments correspond to unchangeable environments we need an analogous definition of mutability:

**Definition 6.3.9 (mutability of environments)** For every abstract environment $E \in Env_A$ the function $\text{mutable}_A : Env_A \to \{m+, m-\}$ yields $m+$ if $E$ is changeable and $m-$ otherwise.

Analogously to the standard semantics of `interaction-environment` providing a copy of $E_I$ we want to define its abstract semantics to provide a copy of $AE_I$. Thus we need a copy function for abstract environments or more precisely for (partial) abstract frame lists. This is presented in the following definition:

**Definition 6.3.10 (copying abstract frame lists)** Let $FL \in \text{FrameListP}_A$ be a (partial) abstract frame list, let $\{x_1, \ldots, x_k\} := \{x \in \text{Sym} \mid FL(x) \neq \text{undef}\}$ and let $v_i := FL(x_i)$ for all $i$. Then we define the procedure $\text{frame-list-copy}_A : \text{FrameListP}_A \to \text{FrameListP}_A$ as

$$\text{frame-list-copy}_A(FL) := (\text{genF}_A[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k])$$

where $\text{genF}_A[x_1 \mapsto v_1, \ldots, x_k \mapsto x_k]$ stands for a newly generated (partial) abstract frame containing exactly the bindings of $x_i$ to $v_i$ for every $i$.

The frame holding the abstract stores is not copied by $\text{frame-list-copy}_A$, but is automatically updated during the abstract interpretation process.

\textsuperscript{5}In contrast to the result of the function `scheme-report-environment` as defined in [KCE98] some of the bindings defined there are missing in $AE_{SR}$. This is e.g. the case for the symbols `set-car!` and `set-cdr!`.

\textsuperscript{6}$AE_I$ is allowed to contain bindings not defined in [KCE98]. This is of interest for instantiating the type inference system according to a Scheme implementation with additional functionality. We do not consider such extensions here for the moment.
Example 6.3.11 (copying abstract frame lists) Consider the abstract frame list \( L_1 \) given in Ex. 6.2.13, i.e.

\[
E_1 = ([x \mapsto int, s \mapsto string, x \mapsto posint, y \mapsto posint]).
\]

Then frame-list-copy \( A(L_1) \) is the following abstract frame list

\[
\text{frame-list-copy}_A(E_1) = ([x \mapsto int, s \mapsto string, y \mapsto posint], \emptyset).
\]

6.3.1.2 Abstract Applications

The application of predefined functions or abstract closures to an argument tuples during the abstract interpretation process is done in two steps: First, the subexpressions of the application (i.e. the function expression and the argument expressions) are evaluated. Then the application is performed. The first step of evaluating all subexpressions of an application expression yields a pre-application environment. It is presented in Def. 6.3.12 which is mutually recursive with Def. 6.3.36 defining the function

\[
[\cdot]^A(\cdot) : \text{Expression}_A \times \text{Env}_A \to \text{Env}_A.
\]

\([\cdot]^A(\cdot)\) integrates an abstract environment update function and the abstract semantic function. The return value of an evaluation can be found as the binding of return in the resulting environment.

Informally, the pre-application environment with respect to \( E \) and \( e \) is the environment generated by evaluating all subexpressions of \( e \) in \( E \) and binding the results of these evaluations to special symbols in \( E \). The order of these applications is fixed from left to right to simplify the presentation.

Definition 6.3.12 (pre-application environment) For an abstract environment \( E = ((F_1 \ldots F_n), \Sigma) \) and an application expression \( e := (e_0 \; e_1 \ldots \; e_k) \) the pre-application environment with respect to \( E \) and \( e \) (written \( \text{pre-app}(E, e) \)) is the abstract environment generated as follows:

\[
E_0 = E'_0[\text{func} \mapsto L'_0(\text{return})] \quad \text{with} \quad E'_0 = (L'_0, \Sigma'_0) = [e_0]^A(E)
\]
\[
E_1 = E'_1[\text{arg}_1 \mapsto L'_1(\text{return})] \quad \text{with} \quad E'_1 = (L'_1, \Sigma'_1) = [e_1]^A(E_0)
\]
\[
E_2 = E'_2[\text{arg}_2 \mapsto L'_2(\text{return})] \quad \text{with} \quad E'_2 = (L'_2, \Sigma'_2) = [e_2]^A(E_1)
\]
\[
\vdots
\]
\[
E_k = E'_k[\text{arg}_k \mapsto L'_k(\text{return})] \quad \text{with} \quad E'_k = (L'_k, \Sigma'_k) = [e_k]^A(E_{k-1})
\]
where $\text{func}, \text{arg}_1, \ldots \text{arg}_k$ are symbols not occurring in the program or as predefined symbol which are bound in a new frame $F'$. i.e. $E_k = ((F'_1 F'_2 \ldots F'_n), \Sigma')$ where each $F'_i$ is generated from the corresponding $F_i$ by side effects during evaluating $e_0, \ldots, e_k$. $\Sigma'$ is generated from $\Sigma$ during these evaluations and $F'$ contains exactly the bindings of $\text{func}$ and $\text{arg}_1, \ldots, \text{arg}_k$.

The set of pre-application environments is denoted by $\text{PreEnv}_A$.

The special symbols are denoted by $\text{func}$ for the result of evaluating the function expression and some $\text{arg}_i$ for the evaluation results of the the argument expressions. In a framework where side effects are possible the order in which the arguments are evaluated may get significant. The evaluation of every arguments has to be done in an environment containing all updates performed by the arguments evaluated before.

**Example 6.3.13 (pre-application environment)** Consider the the expression

$$ e = (\text{cons} \ x \ y) $$

and the environment

$$ E = ([x \mapsto \text{int}, s \mapsto \text{string}] [x \mapsto \text{posint}, y \mapsto \text{posint}] [\text{cons}\mapsto f^A_{\text{cons}}], \emptyset). $$

Then the pre-application environment with respect to $E$ and $e$ is

$$ E' := \text{pre-app}(E,e) = ([\text{func}\mapsto f^A_{\text{cons}}, \text{arg}_1 \mapsto \text{int}, \text{arg}_2 \mapsto \text{posint}] [x \mapsto \text{int}, s \mapsto \text{string}] [x \mapsto \text{posint}, y \mapsto \text{posint}] [\text{cons}\mapsto f^A_{\text{cons}}], \emptyset). $$

The second step of an evaluation is done for all simple pre-application environments independently. It primarily depends on the abstract value the symbol $\text{func}$ is bound to, but the bindings of the $\text{arg}_i$ have an effect on the further processing as well. The different tasks are performed for the corresponding simple abstract environments by the following functions:

- **apply-lambda** presented in Def. 6.3.14 performs the application of abstract lambda closures.

- **apply-strict** given in Def. 6.3.16 enforces strictness by returning $\bot$ if one of the arguments evaluates to $\bot$.

- Different error cases are covered by **apply-error** defined in Def. 6.3.17.

---

As long as no side effects are considered $F'_i = F_i$ holds.
• *apply-pre* performs the application of predefined functions. Its definition is developed in Sec. 6.3.2.

A single pre-application environment can contain simple pre-application environments corresponding to different cases. The results of all these cases for a pre-application environment will be collected and combined by the function *apply* which will be introduced in Def. 6.3.32.

If the symbol *func* is bound to a lambda closure in a pre-application environment the application is performed by the function *apply-lambda* presented in Def. 6.3.14.

Informally, the function *apply-lambda* takes all the simple abstract environments of its argument with *func* bound to an abstract lambda-closure, deletes the bindings of *func* and the *arg*, and changes the resulting abstract environment $E'$ as follows:

- If the number of arguments expected by the lambda-closure and the number of evaluated arguments are equal then lambda-closure is applied as usual: Its definition environment is extended by bindings of the formal parameters to the arguments stored in the *arg*, the expression $e$ of the lambda-closure is evaluated in the resulting environment and the result is bound to *return* in the environment $E'$ possibly modified by side effects during the evaluation of $e$.

- If the number of provided arguments is too large or too small for the lambda-closure then *return* is bound to *error* in $E'$.

**Definition 6.3.14 (abstract lambda application)** The abstract lambda application function $apply-lambda : PreEnvA \rightarrow \mathcal{P}(EnvE_A)$ is defined as follows:

$$apply-lambda(E) = \text{lambda-correct}(E) \cup \text{lambda-err}(E)$$

---

8In this work we do not consider side effects in detail. Nevertheless, when extending the abstract semantic function to elements of the programming language performing side effects this definition of *apply-lambda* can still be used.

9In order to evaluate the body of an abstract lambda closure this definition makes use of the abstract semantic function $[\ ]^A(\cdot)$ given in Def. 6.3.36.
with

\[ \text{lambda-correct}(E) = \{ \hat{E} \in \text{Env}_{E_A} \mid \exists E' = (L, \{\tilde{\sigma}\}) \in S(E). \]

\[ E'(\text{func}) = \text{lc}_A(e(x_1, \ldots, x_k), \hat{L}) \in LC_A, \]

\[ E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef}, \]

\[ E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i, \]

\[ \hat{E} = (\hat{L}, \hat{\Sigma}) := [e]^A((\text{update}_A(\hat{L}, E') [x_1 \mapsto L'(\text{arg}_1), \ldots, x_k \mapsto L'(\text{arg}_k)]), \{\tilde{\sigma}\}), \hat{E}) \]

and

\[ \text{lambda-err}(E) = \{ \hat{E} \in S\text{Env}_{E_A} \mid \exists E' \in S(E). \]

\[ E'(\text{func}) = \text{lc}_A(e(x_1, \ldots, x_k), \hat{L}) \in LC_A, \]

\[ (E'(\text{arg}_k) = \text{undef} \lor E'(\text{arg}_{k+1}) \neq \text{undef}), \]

\[ E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i, \]

\[ \hat{E} = \text{clearapp}(E')[\text{return} \mapsto \hat{L}(\text{return})]] \}

The function clearapp used above removes the frame binding \text{func} and the \text{arg}_i from the given environment. It is analogously defined on frame lists.

**Example 6.3.15 (abstract lambda application)** Consider the following pre-application environment

\[ E = ([[\text{func} \mapsto \text{lc}_A(e, (x, y), L')], \text{arg}_1 \mapsto \text{int}, \text{arg}_2 \mapsto \text{int}]\]

\[ [x \mapsto \text{string}, y \mapsto \text{string}], \{\emptyset\}) \]

with \( e = (+ (* x x) (* y y)) \) and \( L' = ([x \mapsto \text{string}, y \mapsto \text{string}] ) \). apply-lambda(E) adds a new frame to L', binds x to \text{arg}_1 and y to \text{arg}_2 and evaluates e in the environment \( E'' \) that consists of the resulting frame list \( L'' \) and the substitution list \( \emptyset \) taken from E, i.e.

\[ E'' = ([[x \mapsto \text{int}, y \mapsto \text{int}] [x \mapsto \text{string}, y \mapsto \text{string}], \{\emptyset\}) . \]

\([e]^A(E'') \) is calculated according to Def. 6.3.36. The result is \text{int} and therefore apply-lambda returns the environment

\( ([[x \mapsto \text{string}, y \mapsto \text{string}], F_{S_A}[\text{return} \mapsto \text{int}]], \{\emptyset\}) \)

with some frame state \( F_{S_A} \).
A situation not covered by \textit{apply-pre} for the predefined functions as defined in Subsec. 6.3.2 and \textit{apply-lambda} occurs if the function position or one of the argument positions evaluates to \(\bot\). In this case the returned result is \(\bot\):

\textbf{Definition 6.3.16 (strictness of applications)} For some pre-application environment \(E\) the strictness function of application \(\text{apply-strict} : \text{PreEnv}_A \rightarrow \mathcal{P}(S\text{Env}E_A)\) is defined as follows:

\[
\text{apply-strict}(E) = \{ \tilde{E} \in S\text{Env}E_A \mid \exists E' \in S(E), \]
\[
E'(\text{func}) \in \text{PFunc}_A \cup \text{LC}_A \cup \{\bot\},
\]
\[
\exists i. \bot \in \{E'(\text{func}), E'(\text{arg}_i)\} \text{ (with func identified with arg}_0),
\]
\[
\text{error} \notin \{E'(\text{arg}_j) \mid j < i\},
\]
\[
\tilde{E} = \text{clearapp}(E')[\text{return} \mapsto \bot].
\]

Besides abstract predefined functions and abstract lambda closures a pre-application environment can bind the symbol \textit{func} to every other abstract value, too. In all these cases the application result is an error.

Informally, \textit{apply-error} takes all the simple abstract environments that are denoted by the argument that were not selected by \textit{apply-pre} or \textit{apply-lambda} (i.e. with \textit{func} bound neither to an abstract predefined function nor to an abstract lambda-closure or with an error already occurring in one of the arguments), clears the bindings of \textit{func} and \textit{arg}_i and binds \textit{return} to \textit{error}.

\textbf{Definition 6.3.17 (abstract application errors)} The abstract application error function \(\text{apply-error} : \text{PreEnv}_A \rightarrow \mathcal{P}(S\text{Env}E_A)\) is defined as follows:

\[
\text{apply-error}(E) = \{ \tilde{E} \in S\text{Env}E_A \mid \exists E' \in S(E),
\]
\[
(E'(\text{func}) \notin \text{PFunc}_A \cup \text{LC}_A \lor
\]
\[
\lor (E'(\text{arg}_i) = \text{error} \text{ for some } i \land E'(\text{arg}_j) \neq \bot \text{ for all } j < i),
\]
\[
\tilde{E} = \text{clearapp}(E')[\text{return} \mapsto \text{error}]\}.
\]

\section*{6.3.2 Abstract Semantics of Predefined Functions}

The new idea of completeness of the type checker can especially be found in the way applications of predefined functions are processed. The abstract application of predefined functions is done by the function \textit{apply-pre} that is defined step-by-step throughout this section. For the main case the work of checking applicability of a function to an argument tuple and the
calculation of the corresponding abstract output is done by the function \( PAF \). This function specifies the main difference to sound type checkers. \( PAF \) essentially combines the elements of the result set of the function \( PPAF \). \( PPAF \) checks the input type of every I/O-representation pair of a given abstract predefined function for common elements of the given argument type and returns those results that are not more restrictive than necessary.

6.3.2.1 The Standard Case of Predefined Functions

In the standard case of applying a predefined function the needed calculations are done by the function \( PAF \). The aim of \( PAF \) is to check whether a given abstract predefined function \( f \) is applicable to an abstract value and to calculate the abstract result value in case of applicability. The function \( PPAF \) that is used by \( PAF \) performs this check individually for the elements of the I/O-representation of \( f \). It is defined as follows:

**Definition 6.3.18 (the partial predefined application function (\( PPAF \)))** Let \( f \in PFunc_A \) be an abstract predefined function and let \( t' \) be either an abstract value or a product of abstract values with \( k \) elements. The partial predefined application function \( PPAF(f, t') \) is the set of all pairs \((s, \sigma) \in \text{AssValue}_A \times \text{FTS} \) of a value \( s \) and a free type substitution \( \sigma \) with:

1. \((t, s, \sigma) \in \text{CEP}(f, t') \) where \( \text{CEP} \) (common element pairs) is defined by

   \[
   \text{CEP}(f, t') = \{(t, s, \sigma) \mid (t, s) \in D(f), \sigma \in \text{CE}(t, t')\}.
   \]

2. If \((\tilde{t}, \tilde{s}, \tilde{\sigma}) \in \text{CEP}(f, t') \) then the following holds for at least one closed type substitution \( \tau \) appropriate for \( \sigma((\cap t t')) \) and \( \tilde{\sigma}((\cap \tilde{t} t')) \):

   \[
   \begin{align*}
   & (a) \quad \tau \circ \sigma((\cap t t')) \not\sqsubseteq \tau \circ \tilde{\sigma}((\cap \tilde{t} t')). \\
   & (b) \quad \tau \circ \sigma((\cap t t')) \triangleq \tau \circ \tilde{\sigma}((\cap \tilde{t} t')) \Rightarrow \tau \circ \tilde{\sigma}(\tilde{t}) \not\sqsubseteq \tau \circ \sigma(t).
   \end{align*}
   \]

   where \( \triangleq \) stands for an approximation of the semantic equality with the following property:

   \[
   t_1 \triangleq t_2 \Rightarrow \langle t_1 \rangle = \langle t_2 \rangle.
   \]

Informally, \( \text{CEP}(f, t') \) contains the set of all instances of I/O-representation pairs of \( f \) with common elements with \( t' \). From these we select the output types \( s \) with the corresponding input types \( t \) being the most special ones (2b) that yield a maximal intersection with the provided input \( t' \) (2a). In (2b) the exactness of \( \triangleq \) just affects the number of unnecessary
values in the results returned by PPAF and hence the runtime of the system and the size of the output messages provided by the system.

\[ f \] is applicable to \( t' \) if \( CEP(f, t') \) is not empty. The further conditions enforce a result that is not more general than necessary.

**Example 6.3.19 (partial predefined application function)** Consider the function \( f = \text{zero?} \) (as given in App. E) and the input parameter \( t' = A \). The function PPAF first calculates \( CEP(t', f) \) as follows:

\[
CEP(t', f) = \{(\text{zero, Ttrue}, \{A \leftarrow \text{zero}\}), \\
(\text{posreal, Tfalse}, \{A \leftarrow \text{posreal}\}), \\
(\text{negreal, Tfalse}, \{A \leftarrow \text{negreal}\}), \\
(\text{num, bool}, \{A \leftarrow \text{num}\})\}
\]

Since \((\cap \text{num num})\) given by the last element of \( CEP(t', f) \) is strictly more general than the intersections given by the other elements \( PPAF(t', f) \) contains exactly one element:

\[ PPAF(t', f) = (\text{bool}, \{A \leftarrow \text{num}\}). \]

The given definition of \( PPAF \) calculates the most general output type of a call that does not contain unnecessary values. In a different approach one can additionally consider output types that are more restricted and are generated from more restricted input types. This approach could be useful in a framework where further restrictions of the input types can be inferred after the application. E.g. one can additionally carry the output types dropped in Ex. 6.3.19 until e.g. the input type is known to be \text{posreal}.

For every predefined function \( f \) and every abstract value (or product of abstract values) \( t' \) the predefined application function essentially returns a union type of all elements in \( PPAF(f, t') \). This is formalized in the following definition:

**Definition 6.3.20 (the predefined application function (\textit{PAF}))** Let \( f \) and \( t' \) be as in Def. 6.3.18. The predefined application function is defined as

\[
P\text{AF}(f, t') := \begin{cases} 
\bigcup_{(s, \sigma) \in PPAF(f, t')} \sigma(s) & \text{if } PPAF(f, t') \neq \emptyset \\
\text{error} & \text{else}
\end{cases}
\]

**Example 6.3.21 (predefined application function)** Let \( t' \) and \( f \) be as in Ex. 6.3.19. Then \( PAF(t', f) = \text{bool} \). (More precisely the result is \((\bigcup \text{bool})\) which is semantically equivalent to \text{bool}.)

156
In the first step of defining an abstract semantics we give this definition just on closed types. Therefore, we have to get rid of unbound variables introduced by $D(f)$ in the definition of $PPAF$ and remaining in the result of $PAF$. We define the closed predefined application function as follows:

**Definition 6.3.22 (the closed predefined application function ($PAF_c$))** The closed predefined application function $PAF_c$ takes the result of $PAF$ and replaces every variable not bound by $\mu$ by $\top$:

$$PAF_c(f, t') = PAF(f, t')[\mathcal{X}|\top]$$

where $t[\mathcal{X}|\top]$ denotes the type $t$ with every variable (not bound by $\mu$) replaced by $\top$.

**Example 6.3.23 (closed predefined application function)** Consider the function reverse (as given in App. E) and the input argument $t' = \text{nil}$. We get:

$$CEP(t', f) = (((\text{list } A), (\text{list } A), \emptyset).$$
$$PPAF(t', f) = \{(((\text{list } A), \emptyset)\}$$
$$PAF(t', f) = \text{list } A$$
$$PAF_c(t', f) = \text{list } \top$$

The closed predefined application function $PAF_c$ is used by the function $apply-pre$ that expects a pre-application environment $E$ as presented in Def. 6.3.12 and returns the corresponding result environments to all simple abstract environments in $S(E)$ with $\text{func}$ bound to a predefined function.

**Definition 6.3.24 (abstract predefined application (1))** The abstract predefined application function $apply-pre : \text{PreEnv}_A \rightarrow \mathcal{P}(\text{SEnv}_{E_A})$ is defined as follows:

$$apply-pre(E) = \text{pre-paf}_1(E)$$

with

$$\text{pre-paf}_1(E) = \{\tilde{E} \in \text{SEnv}_{E_A} | \exists E' \in S(E).$$

$$E'(\text{func}) =: f \in \text{PFunc}_A \setminus \text{PFunc}_A^{\text{special}},$$
$$E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef},$$
$$E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i,$$

$$\tilde{E} = \text{clearapp}(E')[\text{return} \mapsto PAF_c(f, (\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)))]\}.$$

$clearapp : \text{PreEnv}_A \rightarrow \text{Env}_A$ deletes the frame binding the symbols $\text{func}$ and all $\text{arg}_i$ in the argument environment.

The set $\text{PFunc}_A^{\text{special}} \subset \text{PFunc}_A$ denotes all abstract predefined functions that are marked with $\text{special}$ instead of an I/O-representation pair in App. E.
The function \textit{apply-pre} processes all simple abstract environments \( E' \) denoted by the argument \( E \) with \texttt{func} bound to a predefined function in \( E' \). In all of these \( E' \) \texttt{return} is bound to the result of applying the binding of \texttt{func} to the arguments bound in \texttt{arg}, as given by \( PAF_c \). The set of the resulting simple extended abstract environments \( \bar{E} \) can be returned in form of one extended abstract environment (that is implicitly calculated by \textit{combine}). We drop the application of \textit{combine} here because it will be called on a superset of the result of \textit{apply-pre}.

In order to simplify the notions we write \( E[\texttt{return} \mapsto \ldots] \) when binding the symbol \texttt{return} in the frame state. \texttt{return} will never be bound in any other frame.

\subsection*{6.3.2.2 Applications on Value Ascriptions}

The definition of \textit{apply-pre} from Def. 6.3.24 looses lots of information when all arguments of the function application are given as value ascriptions or types constructed from value ascriptions. In this case the function can be applied in the standard semantics. To express this case precisely we define ascription constructed types.

\begin{definition} \textbf{(ascription constructed types)} \ \ A type \( t \) is called an ascription constructed type if one of the following conditions holds:

\begin{itemize}
  \item \( t \in VA \) (introduced in Def. 3.2.2)
  \item \( t = (c \ t_1 \ldots \ t_k) \) with a free type constructor \( c \in \mathcal{K} \) and ascription constructed types \( t_1, \ldots, t_k \).
\end{itemize}

The set of all ascription constructed types is denoted by \( T_{AC} \).

For an ascription constructed type \( t \) we write \( \mathcal{V}(t) \) to denote the only value \( v \) with \( v \in \langle t \rangle \).

When applying predefined functions according to the standard semantics we have to avoid the application of a predefined function with an argument tuple causing a runtime error. We therefore need an application test function for every abstract predefined function \( f \) that checks whether \( f \) is applicable to a certain argument tuple in the standard semantics.

\begin{assumption} \textbf{(application test function)} \ \ Let \( f^A_{xyz} \) be an abstract predefined function, let \( f^A_{<xyz>} \in \langle f^A_{xyz} \rangle \) and let \( t_1, \ldots, t_k \) be ascription constructed types. There is a function \( \text{test-apply}(f^A_{<xyz>}): T_{AC} \to \{\text{true}, \text{false}\} \) checking the applicability of \( f_{<xyz>} \) to an argument tuple with the property

\[ \text{test-apply}(f^A_{<xyz>})(\times t_1 \ldots t_k) = \text{true} \Rightarrow \]

\[ \Rightarrow \ \text{The application } (f_{<xyz>} \mathcal{V}(t_1) \ldots \mathcal{V}(t_k)) \text{ returns without an error.} \]

\end{assumption}
We can now refine the abstract predefined application to allow a special treatment of ascription constructed types:

**Definition 6.3.27 (abstract predefined application (2))** The abstract predefined application function apply-pre : PreEnvA → P(SEnvE) known from Def. 6.3.24 is refined as follows:

\[
\text{apply-pre}(E) = \text{pre-paf}_2(E) \cup \text{pre-std}(E)
\]

where pre-paf$_2$ differs from pre-paf$_1$ in an additional test that pre-std is not applicable, i.e.

\[
\text{pre-paf}_2(E) = \{ \bar{E} \in SEnvE \mid \exists E' \in S(E). \quad E'(\text{func}) =: f \in PFunc_A \setminus PFunc_A^{\text{special}}, \\
E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef}, \\
E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i, \\
(\exists i. E'(\text{arg}_i) \notin \mathcal{T}_\text{AC} \lor \\
\lor \text{test-apply}(f)(\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)) = \text{false}), \\
\bar{E} = \text{clearapp}(E')[\text{return} \mapsto \text{PAF}_c(f, (\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)))]\}
\]

and pre-std(E) is defined by

\[
\text{pre-std}(E) = \{ \bar{E} \in SEnvE \mid \exists E' \in S(E). \quad E'(\text{func}) =: f \in PFunc_A \setminus PFunc_A^{\text{special}}, f' \in \{f\}, \\
E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef}, \\
E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i, \\
\forall i. E'(\text{arg}_i) \in \mathcal{T}_\text{AC}, \\
\text{test-apply}(f)(\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)) = \text{true}, \\
\bar{E} = \text{clearapp}(E')[\text{return} \mapsto \mathcal{A}((f' \forall (E'(\text{arg}_1)) \ldots \forall (E'(\text{arg}_k)))])\}
\]

### 6.3.2.3 Predefined Special Semantics

Some of the predefined functions have a special semantics that cannot be expressed by the predefined application function PAF. These functions are marked by special instead of an I/O-representation in App. E. Precisely, these are the functions $f^A_{\text{eval}}$ and $f^A_{\text{interaction–environment}}$.

**Definition 6.3.28 (abstract semantics of special predefined functions)** The set of abstract predefined functions marked with special in App. E is denoted by $PFunc_A^{\text{special}} \subset PFunc_A$. The abstract semantics of the elements of $PFunc_A^{\text{special}}$ is given by the function specialsem : SPreEnvA → EnvA where SPreEnvA stands for the set of simple pre-application environments defined straightforward. The definition of specialsem is given in Table 6.1.

159
In the following we use the notion $E := (L, \Sigma)$

$$E(func) = f^A_{eval},$$

valuation$_A(E(arg)) =: e \neq undef,$

update$_A(E(arg), E) =: E' \in Env_A,$

$E(arg) = undef,$

$E'' = (L'', \Sigma'') = \left[e\right]^A(E')$

special$_sem(E) =$

(update$_A(clearapp(L), E'')$

$\Rightarrow$

[return $\mapsto E''(\text{return})], \Sigma'')$

$$E(func) = f^A_{eval},$$

valuation$_A(E(arg)) = undef \lor \Rightarrow$

$E(arg) \notin Env_A \lor E(arg) \neq undef$

special$_sem(E) =$

clearapp($E$)[return $\mapsto$ error]

$\Rightarrow$

(Eval-Err)

$$E(func) = f^A_{interaction-environment},$$

$E(arg) = undef \Rightarrow$

special$_sem(E) =$

clearapp($E$)[return $\mapsto$

(frame-list-copy($AF_I$, $\Sigma$)]

$\Rightarrow$

(IE)

$$E(func) = f^A_{interaction-environment},$$

$E(arg) \neq undef \Rightarrow$

special$_sem(E) =$

clearapp($E$)[return $\mapsto$ error]

$\Rightarrow$

(IE-Err)

<table>
<thead>
<tr>
<th>$E(func) = f^A_{eval}$, $E(arg) = undef$</th>
<th>special$_sem(E) =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$valuation_A(E(arg)) =: e \neq undef,$</td>
<td>$update_A(clearapp(L), E'')$</td>
</tr>
<tr>
<td>$update_A(E(arg), E) =: E' \in Env_A,$</td>
<td>$\Rightarrow$</td>
</tr>
<tr>
<td>$E(arg) = undef,$</td>
<td>$[return \mapsto E''(\text{return})], \Sigma'')$</td>
</tr>
<tr>
<td>$E'' = (L'', \Sigma'') = \left[e\right]^A(E')$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E(func) = f^A_{eval}$, $E(arg) \notin Env_A \lor E(arg) \neq undef$</th>
<th>special$_sem(E) =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$valuation_A(E(arg)) = undef \lor \Rightarrow$</td>
<td>clearapp($E$)[return $\mapsto$ error]</td>
</tr>
<tr>
<td>$E(arg) \notin Env_A \lor E(arg) \neq undef$</td>
<td>$\Rightarrow$</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>(Eval-Err)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E(func) = f^A_{interaction-environment}$, $E(arg) = undef$</th>
<th>special$_sem(E) =$</th>
</tr>
</thead>
</table>
| $\Rightarrow$ | clearapp($E$)[return $\mapsto$

(frame-list-copy($AF_I$, $\Sigma$)] |
| $\Rightarrow$ | (IE) |

<table>
<thead>
<tr>
<th>$E(func) = f^A_{interaction-environment}$, $E(arg) \neq undef$</th>
<th>special$_sem(E) =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Rightarrow$</td>
<td>clearapp($E$)[return $\mapsto$ error]</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>(IE-Err)</td>
</tr>
</tbody>
</table>

Table 6.1: special abstract semantics of predefined functions
Informally, \( f_{\text{interaction-environment}}^A \) generates and returns a fresh copy of the master copy \( AF_I \) of the interaction environment’s frame list. It just fails in rule \( (IE-Err) \) if an invalid number of arguments was given. \( f_{\text{eval}}^A \) transforms the abstract value the first argument evaluates to back into an expression and evaluates it in the environment given by the second argument. The transformation of an abstract value into an (extended) expression is done by the function \( \text{valueexpression}_A \) defined as follows:

**Definition 6.3.29 (expressions available through abstract values)** The function

\[
\text{valueexpression}_A : \text{AssValue}_A \to \text{Expression}_A
\]

transforms an abstract value back into an abstract expression:

1. For every value assignment \( A(c) \) \( \text{valueexpression}_A(A(c)) := \text{valueexpression}(c) \).

2. For every abstract constant \( c \in \text{Const}_A \setminus \{\text{sym}\} \) there is an abstract constant term \( ct \in \text{ConstT}_A \) with \( c = \text{denoteabsconst}_A(ct) \). Then \( \text{valueexpression}_A(c) := ct \).

3. For \( c = \text{sym} \) (or every type containing \text{sym} as a subterm) we have \( \text{valueexpression}_A(c) := \text{undef} \).

4. For a list \( l \) with elements \( e_1, \ldots, e_k \) and \( et_i = \text{valueexpression}_A(e_i) \) we define

\[
\text{valueexpression}_A(l) := (et_1 \ldots et_k).
\]

5. In every other case \( \text{valueexpression}_A(v) = \text{express}_A(v) \) where \( \text{express}_A(v) \) is taken from the set \( \text{Expression}_A \). Precisely, \( \text{express}_A \) is defined such that \( [\text{express}_A(v)]^A(E) = v \) for every \( v \in \text{AssValue}_A \) and every \( E \in \text{EnvE}_A \).

With the semantics of the functions in \( \text{PFunc}_{\text{special}}^A \) presented in Def. 6.3.28 we can refine the abstract predefined application function to cover the special predefined functions as follows:

**Definition 6.3.30 (abstract predefined application including special functions)** The abstract predefined application function \( \text{apply-pre} : \text{PreEnv}_A \to \mathcal{P}(\text{SEnvE}_A) \) from Def. 6.3.27 is refined as follows:

\[
\text{apply-pre}(E) = \text{pre-paf}_2(E) \cup \text{pre-std}(E) \cup \text{pre-special}(E).
\]
where pre-paf₂ and pre-std are given in Def. 6.3.27 and

\[
\text{pre-special}(E) = \{ \hat{E} \in S\text{Env}_A | \exists E' \in S(E). \\
E'(\text{func}) = : f \in P\text{Func}_A^{\text{special}}, \\
E'(\text{arg}_i) \notin \{ \text{error}, \perp \} \text{ for all } i, \\
\hat{E} = \text{specialsem}(E') \}.
\]

The first two elements of the union above correspond to Def. 6.3.27. The last union element covers the function with special semantics according to Def. 6.3.28.

**Example 6.3.31 (abstract predefined application)** Consider the pre-application environment \(E'\) given in Ex. 6.3.13, i.e.

\[
E' := \text{pre-app}(E, e) = ([\text{func} \mapsto f^A_{\text{cons}}, \text{arg}_1 \mapsto \text{int}, \text{arg}_2 \mapsto \text{posint}] \\
[\text{x} \mapsto \text{int}, \text{s} \mapsto \text{string}] \\
[\text{x} \mapsto \text{posint}, \text{y} \mapsto \text{posint}] \\
[\text{cons} \mapsto f^A_{\text{cons}}] \text{FS}_A, \{\emptyset\})
\]

The function apply-pre calculates the result \((\text{int}. \text{posint})\) of applying the abstract function \(f^A_{\text{cons}}\) to the arguments \(\text{int}\) and \(\text{posint}\) and binds this result to \text{return} in the most general frame. The resulting environment is

\[
E'' := \text{apply-pre}(E') = ([\text{x} \mapsto \text{int}, \text{s} \mapsto \text{string}] \\
[\text{x} \mapsto \text{posint}, \text{y} \mapsto \text{posint}] \\
[\text{cons} \mapsto f^A_{\text{cons}}] \text{FS}'_A[\text{return} \mapsto (\text{int}. \text{posint})], \{\emptyset\})
\]

where \(\text{FS}_A\) and \(\text{FS}'_A\) are correct frame states for their environments, respectively.

### 6.3.3 Assembling the Abstract Semantic Function

For the use of apply-pre, apply-lambda, apply-strict and apply-error a pre-application environment must be calculated and the resulting extended abstract environments must be combined. This is done by the function apply defined as follows:

**Definition 6.3.32 (abstract application)** The abstract application function

\[
\text{apply} : \text{Application}_A \times \text{Env}_A \rightarrow \text{EnvE}_A
\]
Application_A \subset Expression_A denotes the application expressions is defined as follows:

\[
\begin{align*}
\text{apply}(e, E) &= \text{compress}(\text{combine}(\text{apply-pre}(\text{pre-app}(E, e))), \\
&\quad \cup \text{apply-lambda}(\text{pre-app}(E, e)), \\
&\quad \cup \text{apply-strict}(\text{pre-app}(E, e)), \\
&\quad \cup \text{apply-error}(\text{pre-app}(E, e)))
\end{align*}
\]

Analogously to applications the definitions of symbol bindings depend on the environment to be simple after evaluating the second argument. Thus, we define a special semantic function for this kind of expressions:

**Definition 6.3.33 (abstract semantics of define)** Let \( E \) be an abstract environment. If \( \text{ed} = (\text{define } x \ e) \) with a symbol \( x \) and an expression \( e \) then the abstract semantics of \( \text{ed} \) under \( E \) is given by:

\[
\text{define}(\text{ed}, E) := \text{define-correct}(\text{ed}, E) \cup \text{define-err}(\text{ed}, E) \cup \text{define-strict}(\text{ed}, E)
\]

with

\[
\begin{align*}
\text{define-correct}(\text{ed}, E) &= \{ \tilde{E} \in SEnv_A | \exists E' = ((F'_1 \ldots F'_k \ FS'_A), \{\sigma\}) \in S([e]^A(E)) . \\
&\quad \exists [e]^A(E)(\text{return}) \notin \{\bot, \text{error}\} \\
&\quad F'_1(x) = \text{undef}, \\
&\quad \tilde{E} = \text{FS-env-update}((F'_1 \ F'_2 \ldots F'_{k-1} \ F'_k \ FS'_A[\text{return} \mapsto \emptyset]), \{\sigma\}) \}\}
\end{align*}
\]

\[
\begin{align*}
\text{define-err}(\text{ed}, E) &= \{ \tilde{E} \in SEnv_A | \exists E' = ((F'_1 \ldots F'_k \ FS'_A), \{\sigma\}) \in S([e]^A(E)) . \\
&\quad ([e]^A(E)(\text{return}) = \text{error} \lor \\
&\quad F'_1(x) \neq \text{undef}), \\
&\quad \tilde{E} = \text{FS-env-update}((F'_1 \ F'_2 \ldots F'_{k-1} \ F'_k \ FS'_A[\text{return} \mapsto \text{error}]), \{\sigma\}) \}\}
\end{align*}
\]

\[
\begin{align*}
\text{define-strict}(\text{ed}, E) &= \{ \tilde{E} \in SEnv_A | \exists E' = ((F'_1 \ldots F'_k \ FS'_A), \{\sigma\}) \in S([e]^A(E)) . \\
&\quad ([e]^A(E)(\text{return}) = \bot, \\
&\quad F'_1(x) = \text{undef}, \\
&\quad \tilde{E} = \text{FS-env-update}((F'_1 \ F'_2 \ldots F'_{k-1} \ F'_k \ FS'_A[\text{return} \mapsto \bot]), \{\sigma\}) \}\}
\end{align*}
\]

If \( \text{ed} = (\text{define } e' \ e) \) with \( e' \notin \text{Sym} \) then

\[
\text{define}(\text{ed}, E) := E[\text{return} \mapsto \text{error}].
\]
If \( x \) is a symbol the first union element in the above definition expresses the error free case. In the second union element several error cases are combined. The cases of rebinding an already bound \( x \) causes an error in order to prevent destructive updates by \textit{define}.\(^{11}\) The third union element expresses the strictness of \textit{define} in the second argument. If the first argument to \textit{define} is not a symbol an error occurs.

**Example 6.3.34 (\textit{define})** Consider the environment

\[
E = (\{\{x \mapsto \texttt{int}\}, \{s \mapsto \texttt{string}\}\}, \varnothing)
\]

and the call \textit{define}((\textit{define} \ y \ 5), E) to define introduced in Def. 6.3.33. The result of this call is the following environment

\[
E' = \text{FS-env-update}((\{x \mapsto \texttt{int}, y \mapsto \mathcal{A}(5)\}, \{s \mapsto \texttt{string}\}, \mathcal{F}_A[\text{\texttt{return} \mapsto O}]), \varnothing).
\]

Analogously to the standard semantics \textit{quote}-expressions are transformed to abstract values precisely denoting the corresponding expression:

**Definition 6.3.35 (abstract semantics of \textit{quote}-expressions)** The function

\[
\text{quotesem}_A : \text{Expression} \rightarrow \text{AssValue}
\]

is defined as follows:

- If \( e \in \text{ConstT} \) then \( \text{quotesem}_A(e) = \mathcal{A}(\text{denoteconst}(e)) \).
- If \( e \in \text{Sym} \) then \( \text{quotesem}_A(e) = \mathcal{A}(\text{sym-to-const}(e)) \).\(^{12}\)
- If \( e = (e_1 \ldots e_k) \) then \( \text{quotesem}_A(e) = \text{make-list}_A(\text{quotesem}_A(e_1), \ldots, \text{quotesem}_A(e_k)) \) with
  - \( \text{make-list}_A() = \texttt{nil} \).
  - \( \text{make-list}_A(v_1, v_2, \ldots, v_k) = (v_1 \cdot \text{make-list}_A(v_2, \ldots, v_k)) \).

By the definition of extended abstract environments the whole semantics of an expression is expressible as an environment update. Thus, the abstract semantic function defined in the following combines the functionality of an environment update function (cf. Def. 5.2.15) and a semantic function (cf. Def. 5.2.16).

\(^{11}\)We assume the standard semantics to be changed analogously.

\(^{12}\)Note that quoted syntactic keywords are not distinguished from ordinary symbols and are processed by this rule.
Definition 6.3.36 (abstract semantic function)  The abstract semantic function

\[
\llbracket \cdot \rrbracket^A : \text{Expression}_A \times \text{Env}_E \rightarrow \text{Env}_E
\]

is defined by:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_0 \not\in {\text{define, if, lambda, begin, quote}} )</td>
<td>(App) [((e_0 \ e_1 \ldots \ e_k)^A(E) = \text{apply}((e_0 \ e_1 \ldots \ e_k), E))]</td>
</tr>
<tr>
<td>([\text{define } x \ e]^A(E) = \text{define}((\text{define } x \ e), E))</td>
<td>(Def)</td>
</tr>
<tr>
<td>([\text{quote } e]^A(E) = E[\text{return} \mapsto \text{quotesem}_A(e)])</td>
<td>(Quote)</td>
</tr>
<tr>
<td>else</td>
<td>(Std) [([e]^A(E) = \text{compress}(\text{combine}({[\cdot]^A_S(E') \mid E' \in S(E)})))]</td>
</tr>
</tbody>
</table>

The definition of the abstract semantic function is essentially based on the simple abstract semantic function \([\cdot]^A_S\) as given in Def. 6.3.37 except of the semantics of applications, and definitions of symbol bindings. The simple abstract semantic function expresses the semantics of expressions in simple abstract environments. It is defined as follows:

Definition 6.3.37 (simple abstract semantic function)  The simple abstract semantic function

\[
\llbracket \cdot \rrbracket^A_S : \text{Expression}_A \times S\text{Env}_E \rightarrow \text{Env}_E
\]

is defined by the rules in Tab. 6.2 (we always assume \( E = (L, \emptyset) \) with \( L = (F_1 \ldots F_k \ F_S) \) where \( F_S \) binds \text{return} and often identify \( L \) with \( E \)).

Note that constructing abstract lambda-closures without considering simple abstract environments yields the same result, but might lead to the more efficient implementation. Constructing lambda-closures in simple abstract environments according to rule (Lambda) yields a simpler representation of the abstract interpreter.

Example 6.3.38 (abstract semantic function)  Consider the expression \( e \) and the environment \( E := (L, \Sigma) \) defined as follows:

\[
e = ((\lambda x (\ast \ 2 \ x)) \ \text{int}), \quad L = (\ast \mapsto f^A \ F_S), \quad \Sigma = \{\emptyset\}.
\]

We evaluate \([e]^A(E)\) step by step.

- In the first step \( e \) is identified as application expression and the pre-application environment is calculated. Because of

\[
([\lambda x (\ast \ 2 \ x)]^A(E) = E[\text{return} \mapsto \text{lc}_A((\ast \ 2 \ x)(x), L)]
\]
| $c \in \text{ConstT} \Rightarrow$ | $[c]^A_S(E) = (F_1 \ldots F_{k-1}$ $F_k \text{FS}_A[\text{return} \mapsto \text{denoteconst}_A(c)])$ | (Const) |
| $c \in \text{ConstT}_A \Rightarrow$ | $[c]^A_S(E) = (F_1 \ldots F_{k-1}$ $F_k \text{FS}_A[\text{return} \mapsto \text{denoteabsconst}_A(c)])$ | (AbsConst) |
| $x \in \text{Sym}, x \text{ bound in } L \Rightarrow$ | $[x]^A_S(E) = (F_1 \ldots F_{k-1}$ $F_k \text{FS}_A[\text{return} \mapsto L(x)])$ | (Inst) |
| $x \in \text{Sym}, x \text{ not bound in } L \Rightarrow$ | $[x]^A_S(E) = (F_1 \ldots F_{k-1}$ $F_k \text{FS}_A[\text{return} \mapsto \text{error}])$ | (Inst-Err) |
| $\text{AT}([e]^A_S(E)(\text{return})) \Rightarrow$ | $[\text{if } e_1 e_2]^A_S(E) = [e_1]^A([e]^A(E))$ | (If-True) |
| $\text{AF}([e]^A_S(E)(\text{return})) \Rightarrow$ | $[\text{if } e_1 e_2]^A_S(E) = [e_2]^A([e]^A(E))$ | (If-False) |
| $\neg \text{AT}([e]^A_S(E)(\text{return})), \neg \text{AF}([e]^A_S(E)(\text{return})) \Rightarrow$ | $[\text{if } e_1 e_2]^A_S(E) = \text{compress(combine}([e_1]^A([e]^A(E)), [e_2]^A([e]^A(E))))$ | (If) |
| $x_1, \ldots, x_n \in \text{Sym} \Rightarrow$ | $[(\lambda x_1 \ldots x_n \ e)]^A_S(E) =$ $E[\text{return} \mapsto lc_A(e(x_1, \ldots, x_n), E)]$ | (Lambda) |
| $\exists i \in \{1, \ldots, n\}. x_i \notin \text{Sym} \Rightarrow$ | $[(\lambda x_1 \ldots x_n \ e)]^A_S(E) =$ $E[\text{return} \mapsto \text{error}]$ | (Lambda-Err) |
| $k > 1 \Rightarrow$ | $[(\text{begin } e_1 e_2 \ldots e_k)]^A_S(E) =$ $[(\text{begin } e_2 \ldots e_k)]^A([e_1]^A(E))$ | (Begin-k) |

Table 6.2: simple abstract semantic function
and

\[ \text{int}^A(E) = E[\text{return} \mapsto \text{int}]^{13} \]

the pre-application environment is

\[ E' := \text{pre-app}(E, e) = ((func \mapsto lc_A((\ast 2 x)(x), L), \text{arg}_1 \mapsto \text{int} \ [\ast \mapsto f^A_\ast ] \ FS'_A), \{\emptyset\}) \]

where \( FS'_A \) differs from \( FS_A \) by additionally binding the store of the new lambda-closure.

- Since \( func \) is bound to a lambda closure the application is processed by apply-lambda.\(^{14} \)
  A new environment \( \hat{E} = (([x \mapsto \text{int}] L), \{\emptyset\}) \) is generated and the body \( e' := (\ast 2 x) \) of the abstract lambda closure is evaluated in \( \hat{E} \).

- Since \( e' \) is an application the pre-application environment with respect to \( \hat{E} \) and \( e' \) is calculated. The result is
  \[ \hat{E}' = (([func \mapsto f^A_\ast, \text{arg}_1 \mapsto A(2), \text{arg}_2 \mapsto \text{int} \ [x \mapsto \text{int}] \ FS'_A), \{\emptyset\}). \]

- \( \hat{E}' \) is a simple abstract environment and \( \hat{E}'(func) \in \text{PFunc}_A \). Therefore, apply-pre is responsible for processing the result. The value bound to \( \text{return} \) is calculated by \( \text{PAF}_c(f^A_\ast, (\times A(2) \ \text{int})). \)

- The calculation of \( \text{PPAF}(f^A_\ast, (\times A(2) \ \text{int}) \) is done as follows:
  - Since \( D(f^A_\ast) \) consists of all I/O-representation pairs of the form \( A \times A \rightarrow A \) with a number type \( A \), the function CEP returns all tuples of the form \( (A \times A, A, A) \) with a number type \( A \).
  - All number types \( A \) with \( A \sqsubseteq \text{int} \) are ruled out by condition (2a) in Def. 6.3.18. The tuples with \( \text{int} \sqsubseteq A \) are ruled out by (2b). The only remaining tuple from the result of CEP is \( (\text{int} \times \text{int}, \text{int}, \emptyset) \) and therefore \( \text{PPAF} \) returns \( \{(\text{int}, \emptyset)\} \).

- By further processing the result of \( \text{PPAF} \) we get
  \[ \text{PAF}(f^A_\ast, (\times A(2) \ \text{int}) = \text{int} = \text{PAF}_c(f^A_\ast, (\times A(2) \ \text{int})) \]

and therefore the result of applying apply-pre to \( \hat{E}' \) is

\[ \hat{E}'' := (([x \mapsto \text{int}] [\ast \mapsto f^A_\ast ] \ FS'_A[\text{return} \mapsto \text{int}]), \{\emptyset\}). \]

\(^{13}\)To be precise \( \text{int} \) is evaluated in an environment with additional bindings of \( \text{return} \) and \( \text{func} \). Since these bindings do not change the result they are omitted here.

\(^{14}\)\( E' \) already is a simple abstract environment. We can therefore drop the consideration of the calls to apply-pre and apply-error performed by apply.
The return value of $E''$ is carried over as the result of applying the lambda closure and therefore
\[
[e]^A(E) = \left(\left(\ast \mapsto f_*^A\right) \mathcal{F}S'_A[\text{return} \mapsto \text{int}], \{\emptyset\}\right).
\]

The following lemma gives us a statement about the reconstruction of expressions from values. It will be used to prove the correctness of the abstract interpreter, especially the evaluation of calls to the abstract predefined function $f^A_{\text{eval}}$.

**Lemma 6.3.39 (expression reconstruction lemma)** Let $v_A \in \text{AssValue}_A$ be generated by abstract evaluation of an expression of the form $e_q = (\text{quote} \ldots) \in \text{Expression}$ and let $v$ be generated by evaluating $e_q$ in the standard semantics. Then

1. $v \in \mathcal{V}_A$.
2. If $e_A = \text{valueexpression}_A(v_A)$, $e = \text{valueexpression}(v)$, $E_A \in \text{Env}_A$ and $E \in \mathcal{E}_A$ then
\[
[e](E) \in \mathcal{V}_A^A(E_A)(\text{return})\).
\]

**Proof:** See App. E, Page 257. \qed

Informally, the expression reconstruction lemma states that the result of transforming the result value $v$ of a quote-expression into an expression $e$ and evaluating $e$ in an environment $E$ is not lost by performing the analogous tasks on the abstractions of $v$, $e$, and $E$.

In Lemma 6.3.41 we will prove the completeness of the abstract semantics defined so far. To do this we need to define the compatibility of an abstract expression $e_A \in \text{Expression}_A$ with an expression $e \in \text{Expression}$:

**Definition 6.3.40 (expression compatibility)** Let $e \in \text{Expression}$ and $e_A \in \text{Expression}_A$. $e_A$ is compatible with $e$ if one of the following cases holds:

- $e_A = e$.
- $e_A \in \text{Const}_A \setminus \{\text{sym}\}$ and $e \in \text{Const}$ such that
  \[
denoteconst(e) \in \mathcal{V}_A^A(\text{denoteabsconst}_A(e_A)).
  \]
• $e_A = (e_1 \ldots e_k)$, $e_A \notin \{if, \text{begin}, \lambda, \text{define}, \text{quote}\}$, $e = (e_1 \ldots e_k)$ and $e_i$ is compatible with $e'$ for every $i$.

The following lemma states that evaluating an expression $e$ in the standard semantics is approximated by evaluating a compatible abstract expression $e_A$ in the abstract semantics in the following sense: The evaluation result of $e$ is contained in the abstract evaluation result of $e_A$. The evaluation result of a given $e_A$ contains the result for every compatible $e$ that can be evaluated without an error. The system is therefore complete in the sense that it just returns error if no such $e$ exists.

A technical condition needed for Lemma 6.3.41 avoids the application of the function $f^A_{\text{eval}}$ to an expression that was abstracted too much during transformation to an abstract value and back to an expression to make sense to the abstract semantics:

We call an evaluation $[e_A]^A(E)$ of an abstract expression $e_A$ in an abstract environment $E$ free of eval abstraction if during evaluating $[e_A]^A(E)$ the abstract predefined function $f^A_{\text{eval}}$ is just applied to values generated by evaluating an expression of the form $(\text{quote} \ldots)$ in the first argument.

Especially this condition avoids the abstraction of a symbol or a syntactic keyword to the abstract constant term $\text{sym}$.

**Lemma 6.3.41 (completeness)** Let $e \in \text{Expression}$, $e_A \in \text{Expression}_A$ such that $e_A$ is compatible with $e$ and let $E \in \text{Env}_A$ such that $[e_A]^A(E)$ is free of eval abstraction. Suppose that $[e_A]^A(E) = \tilde{E} = (\tilde{L}, \tilde{\Sigma})$ can be calculated without an infinite loop.

Then for every $E' \in \{E\}$ such that $[e](E') \notin \{\bot, \text{error}\}$ we have

$$[e](E') \in ([e_A]^A(E)(\text{return}))$$

and

$$EU(e, E') \in ([e_A]^A(E)[\text{return} \leftarrow \text{undef}]).$$

**Proof:** See App. E, Page 259.

The lemma above states that abstract interpretation of an (abstract) expression always yields an abstract value that denotes all values that can result from evaluating a compatible expression in the standard semantics. We can conclude that the abstract value error is just returned if no compatible expression can be evaluated with a meaningful result.

The goal formulated in Sec. 6.1 is not yet met by this system because of two reasons:

• The system needs an (abstract) expression $e_A$ as input. One will usually choose the following form of an abstract expression: A begin expression containing the program
expressions (i.e. all definitions of user defined functions occurring in the program) followed by a call to the main function with abstract values as arguments. These abstract arguments are called input typing and should cover all input values the main function is intended for.

- The system fails to terminate for most recursive programs, i.e. for most programs of practical use in CS.

These restrictions will be eliminated in the following sections.

### 6.4 Type Variables and Type Inference

#### 6.4.1 Motivation

The abstract semantics defined in Sec. 6.3 implements a type checker that (up to now) has the disadvantage that besides of a program it expects an abstract call to one of the functions as input. Comparable to an interpreter for a functional language that starts acting when given a call to a function with an argument tuple we need the same for our type checker. For the user this means that in addition to the program (s)he has to provide the following objects:

1. A main function of the program must be identified, i.e. a function that is called at top level to invoke the program.

2. A so called input typing for the main function is needed, i.e. a tuple of types denoting the set of all valid input tuples of interest for the main function.

**Example 6.4.1 (input typing)** Consider the following definition of cons3:

```lisp
(define cons3 (lambda (x y z)
   (cons x (cons y z)))
```

In order to type check this function the abstract interpreter must be started with a begin expression as input where

- The first argument is the definition of cons3 as given above.
- The second argument is a call to cons3 with an input typing, i.e.

  `(cons3 bool posint negint)`
For the example input typing the abstract evaluation succeeds with the result

\[(bool \cdot (posint \cdot negint)).\]

A call with standard arguments is also possible, e.g. changing the second argument of the begin expression above to \((\text{cons}3\ 3\ 5\ 7)\) yields \((\mathcal{A}(3) \cdot (\mathcal{A}(5) \cdot \mathcal{A}(7))).\)

The need of an input typing is the more restrictive one in practical use. If we can get rid of this restriction, we do not need to know a main function any more but can check all the functions given in a program one by one.

In this section we will eliminate the need for input typings by extending the abstract interpretation to work on free type variables. Free type variables or types containing free type variables as subterms can occur in the following places during type inference:

1. They can occur in the types given as arguments to function calls.
2. Function calls can return variables or types containing variables.
3. Variables or type terms with variables as subterms can occur in the bindings of symbols given in environments. This is especially possible in simple abstract environments which are redefined in a straightforward manner to allow variables.

Example 6.4.2 (free variables) Reconsider the definition of \(\text{cons}3\) in Def. 6.4.1. When checking this function definition without providing an input typing all three cases occur:

1. Since the system has no information about the arguments of an abstract call to \(\text{cons}3\) three new variables \(V_x, V_y\) and \(V_z\) are chosen and the system checks the abstract call \((\text{cons}3\ V_x\ V_y\ V_z).\)

2. The result of an abstract evaluation of \((\text{cons}3\ V_x\ V_y\ V_z)\) in the usual top level abstract environment is \((V_x \cdot (V_y \cdot V_z)).\) This type is constructed using the pair type constructor and contains the three variables \(V_x, V_y\) and \(V_z.\)

3. Variables in symbol bindings occur e.g. in the most special frame of the environment the body of \(\text{cons}3\) is evaluated in when evaluating the abstract call \((\text{cons}3\ V_x\ V_y\ V_z).\) This frame is defined as \([x \mapsto V_x, y \mapsto V_y, z \mapsto V_z].\)

In order to incorporate variables into the abstract interpretation we analyze those rules of the abstract semantics presented in Sec. 6.3 that have to be changed.

First, we sort out those rules of Def. 6.3.36 and 6.3.37 that cannot return or restrict type variables in the new semantics and therefore need not be changed. These are the rules \((\text{Const}),\)
(AbsConst), (PFunc) and (Lambda) of Def. 6.3.37 that just generate an output exclusively depending on the expression they evaluate. The rule (Inst-Err) additionally depends on the environment, but it does not return or restrict any variables.

There are further rules that just pass through the values taken from an environment or the result of evaluating a subexpression. These rules do not restrict any variables and do not care for variables they possibly pass through. In detail, these are the rules (Inst), (If-True), (If-False), (If), (Begin-1) and (Begin-k). The rule (Def) behaves quite similar, but it does not return the value given by evaluating the second argument, but binds it to a symbol.

The only remaining rule (App) processes the application of functions. Here the following cases have to be processed independently:

1. The function position of an application is given by a free variable.
2. The function position is given by a PI/PO-representation. (PI/PO-representations (cf. Def. 3.3.4) are generated when processing calls of case 1.)
3. A lambda closure is applied.
4. An abstract predefined function is applied.

These cases will be discussed individually in Subsec. 6.4.3 (for Cases (1), (2) and (3)) and Subsec. 6.4.4 (Case (4)).

A refined definition of the set AssValueP of partial abstract values is given analogously to Def. 6.2.11 with changing the set $Var_A \subseteq V_f$ to $V_f$.

The refined set AssValueA differs from AssValueP as described above as follows:

- Elements of AssValueA must not contain undef at any position.
- Instead of partial abstract frame lists taken from FrameListP abstract environments from EnvA can occur.

These definitions are straightforward and not presented here in detail.

### 6.4.2 Indexed Variables

A further problem occurring in the context of variables is the following: A variable can occur at several positions in a term and each of these occurrences can denote a different value. This
is especially the case when unfolding a recursively defined abstract value which contains free
variables.

In order to maintain some precision of the system it is important to detect those occurrences
of a variable that denote the same value. We use the fact that every variable $A$ occurs in
a certain value for the first time. If this introducing value $v$ contains $A$ several times then
the different positions of $A$ in $v$ can be used to identify the different versions of $A$. Different
occurrences of $A$ steaming from the same position $p$ in $v$ always denote the same value. We
call $A$ an indexed variable and the pair $(v, p)$ an index of $A$.

For variables occurring free in a recursively defined abstract value $v$ an index is given by $v$ and
the position $p$ of $A$ after unfolding $v$ often enough such that there is no prefix of $p$ denoting
a recursively defined value in $v$.

Every unfolding step of $v$ must memorize the position of the recursively bound variable $X$
that was replaced. For every occurrence of $A$ inside a recursive binding we then can calculate
all indices this occurrence of $A$ can generate. An occurrence of a free variable inside of a
recursive binding is called open indexed. The restriction of an open indexed variable is given
by the union of the restrictions of the indexed occurrences it can generate.

**Example 6.4.3 (indexed variables)** Consider the recursive type
\[ t = \mu X. (\bigcup \text{nil} (A \cdot X)) \]
as introducing value of $A$. The only occurrence of $A$ is an open indexed one. Now consider
\[
\text{unfold}^2(t) = \bigcup \text{nil} (A_{2,1} \cdot (\bigcup \text{nil} (A_{2,2,2,1} \cdot \mu X.(\bigcup \text{nil} (A_{\text{open}} \cdot X))))). 
\]

A substitution $\sigma$ binding $A_{2,1}$ to int and $A_{2,2,2,1}$ to bool implicitly also binds the original $A$
to $(\bigcup \text{int bool} B)$ where $B$ is a new variables standing for the further possible bindings given
by $A_{\text{open}}$ in the unfolded term.

When transforming a given recursive type into a new recursive type the free variables occurring
in the new type have to be replaced by new ones and the appropriate variable bindings have
to be introduced. This is necessary in order to avoid confusion between the positions of the old and the new type.

**Example 6.4.4 (new index variables)** Consider the type $t$ and the substitution $\sigma$ from
Def. 6.4.3. Evaluating a call to the predefined function reverse with argument $t$ yields the
type $t' := \mu Y.(\bigcup \text{nil} (A \cdot Y))$. To avoid confusion between the indices of $A$ in $t$ and $t'$ we
introduce a new variable $C$, change $A$ in $t'$ to $C$, i.e. $t' = \mu Y.(\bigcup \text{nil} (C \cdot Y))$ and add the
binding $C \leftarrow (\bigcup \text{int bool} B)$ to $\sigma$.

In the following we assume every variable to be indexed. When speaking of identical variables
this means identical indices of a variable.
6.4.3 Applications with Variables

In case (1) an application expression \((e_0 \; e_1 \; \ldots \; e_k)\) is evaluated where \(e_0\) evaluates to a value \(v_0\) of unknown type that must be applied to a set of arguments. The result can just be valid if \(v_0\) is a function. Both the input type and the output type of this function are just partially known: The input type must have common elements with the provided argument type. For the output type a new variable is introduced. From the use of the application result we can conclude a type the function’s output type must have common elements with.

The application of a free variable (case 1) is formalized by the function \(\text{apply-var}\) that is defined as follows:

**Definition 6.4.5 (application of type variables)** The abstract application of free variables is performed by the function \(\text{apply-var}: \text{PreEnv}_A \rightarrow \mathcal{P}(\text{SEnv}_A)\) defined by:

\[
\text{apply-var}(E) = \{ \tilde{E} \in \text{Env}_E \mid \exists E' = (L', \sigma) \in S(E). \\
E'(\text{func}) = A \in V_f, \\
E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef}, \\
E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i, \\
\tilde{E} = (\text{clearapp}(L')| \text{return} \leftarrow B_E') , \\
\sigma \cup \{A \leftarrow (E'(\text{arg}_1) \times \ldots \times E'(\text{arg}_k) \rightarrow B_E')\}
\]

with \(\text{PI/PO-representation pairs enclosed in } \langle \ldots \rangle \text{ to improve readability.}\)

An application according to case (2) occurs when an PI/PO-representation introduced by \(\text{apply-var}\) (Def. 6.4.5) is applied again. Both the input type and the output type in general do not cover all possible input and output values, respectively. Thus, we unite the already known input type with the current argument type and unite the output type with a new free variable. An error occurs if the number of input arguments of the actual call and the number of arguments expected by the PI/PO-representation do not match.

**Definition 6.4.6 (application of PI/PO-representations)** The abstract application of a
PI/PO-representation is done by the function \( \text{apply-pipo}: \text{PreEnv}_A \rightarrow \mathcal{P}(\text{Env}_A) \) defined by:

\[
\text{apply-pipo}(E) = \{ \tilde{E} \in \text{Env}_A \mid \exists E' = (L', \sigma) \in S(E). \]

\[
L'(\text{func}) = A \in V_f,
\]

\[
E'(\text{func}) = \in \text{pipo}, \text{with}^{15}
\]

\[
\sigma^l(A) = A' \text{ for some } A' \in V_f \text{ and some } l \in \mathbb{N},
\]

\[
\sigma(A') = \{(t_{1,1} \times \ldots \times t_{1,n_1} \Rightarrow t'_{1}), \ldots, \}
\]

\[
(t_{m,1} \times \ldots \times t_{m,n_m} \Rightarrow t'_{m})\},
\]

\[
E'(\text{func}) = \text{GIS}(\sigma)(\sigma(A')),
\]

\[
E'(\text{arg}_n) \neq \text{undef}, E'(\text{arg}_{n+1}) = \text{undef},
\]

\[
E'(\text{arg}_i) \notin \{\bot, \text{error}\} \text{ for all } i,
\]

\[
\tilde{E} = (\text{clearapp}(L')[\text{return} \leftarrow B_{E'}], \{\sigma|_{\text{dom}(\sigma)} \setminus \{A'\} \cup
\]

\[
\{A' \leftarrow \{(t_{1,1} \times \ldots \times t_{1,n_1} \Rightarrow t'_{1}), \ldots, \}
\]

\[
(t_{m,1} \times \ldots \times t_{m,n_m} \Rightarrow t'_{m})\},
\]

\[
\{E(\text{arg}_1) \times \ldots \times E(\text{arg}_n) \Rightarrow B_{E'}\}\} \})
\]

with a new variable \( B_{E'} \) for every \( E' \)\)

Note that by applying PI/PO-representations no checks leading to an error can be done because a function \( f \) represented by a PI/PO-representation can employ ad-hoc polymorphism and can be defined for variable arity. We can therefore just collect the requirements on \( f \) given by the individual calls, and the user has to check whether such a function exists and was intended.

For the remaining cases variables can just occur at the argument positions of the call.

If a lambda closure is applied, the results of evaluating the argument terms are just bound in a new frame before evaluating the body of the lambda closure. It does not matter whether evaluating the argument terms yields closed types or not.

In order to restrict variables correctly we have to update the substitutions of the result environment by changing them to the result substitutions of evaluating the closure body.

This is already done by the function \( \text{apply-lambda} \) as given in Def. 6.3.14. A change of this function is not needed.

\[^{15}\text{The following lines select the last variable } A' \text{ in a chain of variable renamings starting with } A. \text{ Because } E'(\text{func}) \in \text{pipo} \text{ this variable } A' \text{ must be bound to a PI/PO-representation } \sigma(A'). \text{ The variables of } \text{dom}(\sigma) \text{ occurring in } \sigma(A') \text{ are eliminated by } \text{GIS}(\sigma) \text{ in } E'(\text{func}).\]
Strictness is also given for applications with variables. The function $apply-strict_V$ refines Def. 6.3.16 as follows:

**Definition 6.4.7 (strictness of applications with variables)** For a pre-application environment $E$ the strictness function of application $apply-strict_V : PreEnv_A \rightarrow EnvE_A$ is defined as follows:

$$apply-strict_V(E) = \{ \tilde{E} \in SEnvE_A \mid \exists E' \in S(E).$$

$$E'(func) \in PFunct_A \cup LC_A \cup V_f \cup pipo \cup \{ \perp \},$$

$$\perp \in \{ E'(func), E'(arg_i) \},$$

$$error \notin \{ E'(arg_j) \mid j < i \},$$

$$\tilde{E} = \text{clearapp}(E')[\text{return} \mapsto \perp]$$

where $pipo$ denotes the set of all PI/PO-representations as given in Def. 3.3.4.

The error case for an application function with variables is processed by the modified function $apply-error_V$. In contrast to $apply-error$ from Def. 6.3.17 the error free cases have to be extended by free type variables and PI/PO-representation at the function position:

**Definition 6.4.8 (abstract application errors with variables)** The abstract application error function with variables $apply-error_V : PreEnv_A \rightarrow EnvE_A$ is defined as follows:

$$apply-error_V(E) = \{ \tilde{E} \in SEnvE_A \mid \exists E' \in S(E).$$

$$(E'(func) \notin PFunct_A \cup LC_A \cup V_f \cup pipo \lor$$

$$\lor E'(arg_i) = error \text{ for some } i),$$

$$\tilde{E} = \text{clearapp}(E')[\text{return} \mapsto error] \}.$$
For type inference with variables it is possible that a variable that has been restricted before occurs in the input type \( t' \). In this case we are just interested in values that are denoted by \( t' \) after applying the restriction. Additional restrictions of the current application have to be added. We therefore use a refined definition of the partial predefined application function:

**Definition 6.4.9 (partial predefined application function with variables)** Let \( f \in PFunc_A \) be an abstract predefined function, let \( t' \) be either an abstract value or a product of abstract values with \( k \) elements and let \( \rho \) by a type substitution. The partial predefined application function with variables \( PPAF_V(f,t',\rho) \) is the set of all pairs \( (s,\sigma) \in AssValue_A \times TS \) with:

1. \( (t,s,\sigma) \in CEP_V(f,t',\rho) \) where \( CEP_V \) is defined by
   
   \[
   CEP_V(f,t',\rho) = \{(t,s,\sigma) | (t,s) \in D(f), \sigma' \in CE(t,t'),
   A \in \text{dom}(\sigma') \cap \text{dom}(\rho) \Rightarrow CE(\sigma'(A),GIS(\rho)(A)) \neq \emptyset,
   \sigma \in \text{intersect-ts}(\rho,\sigma') \}\,
   
   \]

   where \( \text{intersect-ts} : TS \times TS \rightarrow TS \) is defined by

   \[
   \text{intersect-ts}(\sigma',\rho) = \{A \leftarrow (\cap tt') | t = \sigma'(A), t' = \rho(A)\} \cup
   \{A \leftarrow t | t = \sigma'(A), A \notin \text{dom}(\rho)\} \cup
   \{A \leftarrow t' | A \notin \text{dom}(\sigma'), t = \rho(A)\}.\n   
   \]

2. If \( (\tilde{t},\tilde{s},\tilde{\sigma}) \in CEP(f,t') \) then the following holds for at least one closed type substitution \( \tau \) appropriate for \( GIS(\sigma)((\cap t t')) \) and \( GIS(\tilde{\sigma}((\cap \tilde{t} t'))) \):

   \[
   (a) \quad \tau \circ GIS(\sigma)((\cap t t')) \n   \quad \n   \n   \[
   \n   (b) \quad \tau \circ GIS(\tilde{\sigma}((\cap \tilde{t} t'))) \n   \quad \n   \n   \[
   \n   \]

   \]

   where \( \triangleq \) stands for an approximation of the semantic equality with the following property:

   \[
   t_1 \triangleq t_2 \Rightarrow \langle t_1 \rangle = \langle t_2 \rangle.
   
   \]

**Example 6.4.10 (partial predefined application function with variables)** Consider the call \( PPAF_V(f^A_{\text{car}},R,\{R \leftarrow \top \setminus \text{nil}\}) \) that could e.g. steam from analyzing the else expression of

177
(if (null? l)
  ()
  (car l))

with l bound to the variable R:

• PPAF\_V calls CEP\_V with the same arguments and the result

\[
\text{CEP}_V(f_{\text{car}}, R, \{R \leftarrow \top \ \text{\textbackslash} \ \text{nil}\}) = \{(A \cdot B), A, \{R \leftarrow (\cap (\top \ \text{\textbackslash} \ \text{nil}) (A \cdot B)), A \leftarrow V_A, B \leftarrow V_B\}\}.
\]

• None of the restrictions to the results of CEP\_V invalidates the only element, i.e. (2) in Def. 6.4.9 has no effect.

The result of the analyzed call is:

\[
\text{PPAF}_V(f_{\text{car}}, R, \{R \leftarrow \top \ \text{\textbackslash} \ \text{nil}\}) = \{(A, \{R \leftarrow (\cap (\top \ \text{\textbackslash} \ \text{nil}) (A \cdot B)), A \leftarrow V_A, B \leftarrow V_B\}\}.
\]

Note that V\_A and V\_B just occur in the returned substitution. Bindings of this kind could be deleted in a simplification stage.

The application of those predefined functions not having a special semantics is done by the function apply-pre\_V given in the following definition:

**Definition 6.4.11 (abstract predefined application with variables)** The abstract predefined application function with variables apply-pre\_V : PreEnv\_A → P(EnvE\_A) is defined as follows:

\[
\text{apply-pre}_V(E) = \text{pre-ppaf}_V(E) \cup \text{pre-err}_V(E) \cup \text{pre-std}(E)
\]

where

\[
\text{pre-ppaf}_V(E) = \{\tilde{E} \in \text{EnvE}_A \mid \exists E' = (L', \{\rho\}) \in S(E).
\]

\[
f = E'(\text{func}) \in \text{PFunc}_A \setminus \text{PFunc}_A^{\text{special}},
\]

\[
E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef},
\]

\[
E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i,
\]

\[
(\exists i. E'(\text{arg}_i) \notin T_{AC} \lor \text{test-apply}(f)(\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)) = \text{false},
\]

\[
(s, \sigma) \in \text{PPAF}_V(f, (\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)), \rho),
\]

\[
\tilde{E} = \text{clearapp}(L'[\text{return} \mapsto s], \{\sigma\})\}
\]
\[ \text{pre-err}_V(E) = \{ \tilde{E} \in S\text{Env}_{E_A} \mid \exists E' = (L', \rho) \in S(E). \]

\[ f = E'(\text{func}) \in P\text{Func}_{E_A} \setminus P\text{Func}_{E_A}^{\text{special}}, \]

\[ E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef}, \]

\[ E'(\text{arg}_i) \notin \{ \text{error}, \bot \} \text{ for all } i, \]

\[ (\exists i. E'(\text{arg}_i) \notin T_{AC} \lor \text{test-apply}(f)(\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)) = \text{false}), \]

\[ \text{PPAF}_V(f, (\times E'(\text{arg}_1) \ldots E'(\text{arg}_k)), \rho) = \emptyset, \]

\[ \tilde{E} = \text{clearapp}(E'[\text{return} \mapsto \text{error}]) \]

and \( \text{pre-std}(E) \) as presented in Def. 6.3.27.

In the definition of \( \text{apply-pre} \) above the first union element covers the error free abstract applications of predefined functions, and the second union element covers those applications where an empty result of \( \text{PPAF}_V \) denotes an error. The third union element processes applications with ascription constructed types as arguments. This part of the definition did not change compared with Def. 6.3.27 and Def. 6.3.30.

As for \( \text{apply-pre} \) in Def. 6.3.30 we need to define a special semantics for the functions bound to \text{eval} and interaction-environment. The definition of \( \text{specialsem}_V \) is completely analogous to \( \text{specialsem} \) in Def. 6.3.28.\(^{16}\) Just the function \( \text{valueexpression}_A \) from Def. 6.3.29 has to be extended in order to return \( \text{undef} \) for types containing free type variables.

We can now extend \( \text{apply-pre}_V \) to the predefined functions with special semantics:

**Definition 6.4.12 (abstract predefined application with variables including special functions)** The abstract predefined application function with variables \( \text{apply-pre}_V : \text{PreEnv}_{E_A} \rightarrow \mathcal{P}(\text{Env}_{E_A}) \) is redefined as follows:

\[ \text{apply-pre}_V(E) = \text{pre-ppaf}_V(E) \cup \text{pre-err}_V(E) \cup \text{pre-std}(E) \cup \text{pre-special}_V(E) \]

where \( \text{pre-special}_V(E) \) differs from \( \text{pre-special}(E) \) from Def. 6.3.30 just in performing a call \( \text{specialsem}_V(E') \) instead of \( \text{specialsem}(E') \). \( \text{pre-ppaf}_V(E), \text{pre-err}_V(E) \) and \( \text{pre-std}(E) \) are as presented in Def. 6.4.11.

In Def. 6.4.12 the first union element represents error free executions of predefined functions without special semantics, the second union elements represents application errors of functions without special semantics and the third one covers all applications of predefined functions with special semantics.

\(^{16}\) A new case for \( f_{\text{eval}}^A \) can occur when the second argument evaluates to a free type variable. To be precise we need a variable environment here whose required properties are restricted during evaluating \( f_{\text{eval}}^A \). We believe this case to occur quite seldom and do not discuss it here. The completeness property of the system is maintained by returning \( \top \) for this kind of calls to \( f_{\text{eval}}^A \).
6.4.5 The Abstract Semantic Function with Variables

As \textit{apply} from Def. 6.3.32 the function \textit{apply}_V calculates a pre-application environment and calls the auxiliary application functions defined before:

\textbf{Definition 6.4.13 (abstract application with variables)} The abstract application function with variables \textit{apply}_V : \textit{Expression}_A \times \textit{Env}_A \rightarrow \textit{EnvE}_A is defined as follows:

\[
\textit{apply}_V(e, E) = \text{compress}\left(\text{combine}\left(\text{apply-var}\left(\text{pre-app}(E, e)\right),
\bigcup \text{apply-pipo}\left(\text{pre-app}(E, e)\right),
\bigcup \text{apply-pre}_V\left(\text{pre-app}(E, e)\right),
\bigcup \text{apply-lambda}\left(\text{pre-app}(E, e)\right),
\bigcup \text{apply-strict}_V\left(\text{pre-app}(E, e)\right),
\bigcup \text{apply-error}_V\left(\text{pre-app}(E, e)\right)\right)\right).
\]

Altogether the abstract semantics with variables is defined analogously to the abstract semantics given in Def. 6.3.36 except of using the function \textit{apply}_V instead of \textit{apply} for applications:

\textbf{Definition 6.4.14 (abstract semantics with variables)} The abstract semantics with variables consists of all rules of Def. 6.3.36 and Def. 6.3.37 except of Rule (App) replaced by Rule (AppV) defined by:

\[
e_{0} \not\in \{\text{define}, \text{lambda}, \text{if}, \text{begin}, \text{quote}\} \Rightarrow [(e_{0} \ e_{1} \ldots \ e_{k})]^{A}(E) = \text{apply}_V((e_{0} \ e_{1} \ldots \ e_{k}), E) \quad (\text{AppV})
\]

Theorem 6.4.16 will state a property comparable to Lemma 6.3.41 for the extended abstract semantics. In order to formulate this property the definition of expression compatibility of an abstract expression \(e_A\) with an expression \(e\) from Def. 6.3.40 has to be extended to cover variables. Free variables are introduced in the set of abstract expressions and an abstract expression \(e_A\) given by a single free variable is compatible with every expression \(e \in \text{Expression}\):

\textbf{Definition 6.4.15 (expression compatibility with variables)} Definition 6.3.40 is extended by the following case:

- \(e_A \in \text{TV}_f\)

where \(\text{TF}_f\) is a set of free variable expressions and contains an element for every variable \(x \in \text{V}_f\) denoted in the same way. The set \(\text{ConstT}_A\) of abstract constant terms is extended to contain abstract variable expressions and the set \(\text{Expression}_A\) of abstract expressions is extended correspondingly.
The main completeness statement for the abstract semantics with variables informally is the following: let \( E \) be an abstract environment and \( E' \) a structured environment denoted by (an instance of) \( E \). Let furthermore \( e \) be an expression and \( e_A \) and abstract expression compatible with \( e \). Then the result of evaluating \( e \) in \( E' \) is contained in the result of performing abstract interpretation of \( e_A \) in \( E \). The same inclusion holds for the environments updated by evaluating \( e \) in \( E' \) or by evaluating \( e_A \) in \( E \). This statement is formally proven in the following theorem that extends Lemma 6.3.41 to variables:

**Theorem 6.4.16 (complete type inference)** Let \( e \in \text{Expression} \) and \( e_A \in \text{Expression}_A \) such that \( e_A \) is compatible with \( e \) and let \( E = (L, \Sigma) \in \text{Env}_A \) such that \([e_A]^A(E)\) is free of eval abstraction. Suppose that \([e_A]^A(E) = \tilde{E} = (\tilde{L}, \tilde{\Sigma})\) can be calculated without an infinite loop. Then for every \( E' \in \{\tau \circ \sigma(L)\} \) for a \( \sigma \in \Sigma \) and a closed substitution \( \tau \) appropriate for \( \sigma(L) \) with \([e](E') \notin \{\bot, \text{error}\}\) there is some \( \tilde{\sigma} \in \tilde{\Sigma} \) and some \( \tilde{\tau} \) appropriate for \( \tilde{\sigma}(\tilde{L}) \) such that

\[
[e](E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L}(\text{return}))\}
\]

and

\[
EU(e, E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L}[\text{return} \leftarrow \text{undef}]\}.
\]

**Proof:** See App. E, Page 268. \( \square \)

Compared to the end of Sec. 6.3 we now have the following situation. Since the abstract expression given as input to the abstract interpreter can contain variables we need no longer provide a call explicitly. We can rather process all function definitions in the program one by one and generate calls with the correct number of argument variables automatically.

The system presented so far hence fits the goal of a complete type checker that just needs a program as input argument. When an output is provided it meets the conditions of Sec. 6.1. The system still fails to terminate for most programs of practical use. This restriction is eliminated in the next section.

### 6.5 Recursion in Abstract Interpretation

The type checker presented in the preceding sections has the disadvantage that it will not terminate for most recursive input programs. The reason becomes obvious in the following example:

**Example 6.5.1** Consider the factorial function defined as follows:
(define (fac n)
  (if (<= n 0)
      1 ; return 1 for n <= 0
      (* n (fac (- n 1)))) ; calculate (fac n) recursively
)

Obviously, the function fac terminates for every integer number. But if considering the abstract semantics, the situation is different:

Calculating \([(\text{fac int})]^{A}(E)\) causes the evaluation of the body of fac in its abstract definition environment with \(n\) bound to \(\text{int}\). Since the result of checking \((<= n 0)\) with \(n\) bound to \(\text{int}\) does not fulfill AT or AF both paths of the if-expression are evaluated. Evaluating the else-expression yields a call to fac with \(arg_1 = [(- n 1)]^{A}(E[n \mapsto \text{int}]) = \text{int}\) for some \(E\). This again evaluates the body of fac in its abstract definition environment with \(n\) bound to \(\text{int}\) and causes an infinite loop.

As shown in Ex. 6.5.1 even functions that are terminating in the standard semantics need not terminate in the abstract semantics. This is because the value set denoted by an abstract value is often too large to fulfill either \(\text{AT}\) or \(\text{AF}\) as given in Def. 6.3.6. Hence, in evaluating an if-expression the abstract interpreter often has to evaluate both branches, i.e. in case of an if-expression testing a termination condition the non-recursive (and usually terminating) and the recursive branch must be processed.

Obviously, for the abstract interpretation a new way of enforcing termination must be found. The problem of non-termination of abstract interpretations has already been discussed in [JC87]. Our solution is quite similar to the fixpoint iteration described there. However, it must be adapted because in contrast to their approach it is not possible to calculate the abstraction of a function completely for our quite large abstract domains.

Principally, we enforce termination by an iterative process that often yields the wanted result as fixpoint (cf. e.g. [NNH99]). This process called iterated type inference is presented in Subsec. 6.5.1. If for some input program the algorithm does not converge, additional steps presented in Subsec. 6.5.2 are taken to enforce termination.

### 6.5.1 Iterated Type Inference

Iterated type inference is based on the following idea: when a call to a lambda closure \(f\) with argument \(a\) causes a recursive call \((f a)\) (more precisely a call with the function expression evaluating to \(f\) and the argument expressions evaluating to \(a\)) then the result of the original call or these recursive calls cannot be calculated directly. We rather use an iterated process that memorizes all calls to user defined functions as a pair consisting of an abstract lambda
closure and an argument. When a recursive call of the same function with the same argument is reached, evaluation of this call is suppressed, but the system returns the following result for the recursive call:

- \( \bot \) in the first iterative step.
- The result generated for the initial call in the previous iterative step if the current one is not the first.

Using this as result for the recursive call an intermediate result for the initial call can be calculated. The set of values denoted by the intermediate result increases from step to step.

**Example 6.5.2** Reconsider the program given in Ex. 6.5.1. Iterated type inference proceeds as follows:

1. When calling fac with an argument of type \( \texttt{int} \) both branches of the if-expression are evaluated. The then-case yields the value ascription \( A(1) \) (this is also true for every further evaluation of the then-case). The else-case evaluates the recursive call to fac to \( \bot \) and generates the result \( \bot \) for the whole branch. The result of the if-expression is the union \( (\cup A(1) \bot) \) that can be simplified to \( A(1) \).

2. In the second iteration step the recursive call to fac yields \( A(1) \) and the result of abstractly multiplying \( \texttt{int} \) with \( A(1) \) is \( \texttt{int} \). The new intermediate result of fac is \( (\cup A(1) \texttt{int}) \) that can be simplified to \( \texttt{int} \).

3. In the third iteration step the result of the recursive call to fac is \( \texttt{int} \), the else case yields \( \texttt{int} \) and the if-expression yields \( (\cup A(1) \texttt{int}) \) that can be simplified to \( \texttt{int} \). Since there was no change in the intermediate result by the last step the abstract result of fac with input argument \( \texttt{int} \) is \( \texttt{int} \).

Iterated type inference essentially is a fixed point generation as described in [NNH99, Chap. 4]. In the following we formalize this process in our framework.

Iterated type inference will be introduced in three steps with each step extending the application rule for lambda closures yielded by the previous step:

1. The abstract semantics with \( \lambda \)-history memorizes every open call to a lambda closure, i.e. every call whose evaluation was started but not yet finished.

2. The recursion interrupting semantics uses this information to detect recursive calls with an identical call (i.e. same lambda closure with same arguments) already memorized as open. In these cases it returns an intermediate value.
3. The \( \lambda \)-iterative semantics calculates an intermediate result for those calls that caused the interruption of a recursive call. The process of evaluating the initial call is repeated until the intermediate value stabilizes.

In order to detect those calls that must not be evaluated directly we need the abstract semantics with \( \lambda \)-history given in Def. 6.5.3 below. This is an abstract semantics with all “open” applications of lambda-expressions known. A lambda-expression is called open if the evaluation of its body has been started, but not yet finished:

The abstract semantics with \( \lambda \)-history is defined analogously to Def. 6.4.14 except of managing an additional symbol \( \lambda \text{-hist} \) bound in the frame state. When starting a type inference process \( \lambda \text{-hist} \) is bound to the empty list.

\( \text{apply-lambda} \) is changed to perform the following additional steps to all simple abstract environments \( E' \) with \( E'(\text{func}) = lc_A \in LC_A \):

- \( E'(\lambda \text{-hist}) \) is bound to \(((E'(\text{func}) E'(\text{arg}_1) ... E'(\text{arg}_n)) \downarrow \#f). E'(\lambda \text{-hist})\) before evaluating the body of \( lc_A \). i.e. a new element
  \((E'(\text{func}) E'(\text{arg}_1) ... E'(\text{arg}_n)) \downarrow \#f\)

is appended to the front of the list formerly bound to \( \lambda \text{-hist} \).

- \( E'(\lambda \text{-hist}) \) is bound to \( h\text{-rest} \) (where the previous value of \( E'(\lambda \text{-hist}) \) is \((h-1 \cdot h\text{-rest})\) after evaluating the body of \( lc_A \). (Here \( h-1 \) is the history entry generated for the just processed call and \( h\text{-rest} \) is the list rest containing the entries of open calls.) Informally, the first element of the list bound to \( \lambda \text{-hist} \) is discarded. This is the entry of the call currently finished.

i.e. as long as a call to a lambda closure is open there is an entry in \( \lambda \text{-hist} \) for this call containing the following elements:

- The lambda closure \( E'(\text{func}) \) and the arguments \( E'(\text{arg}_i) \) it was applied to.
- An intermediate value for the output type. At the beginning this value is initialized to \( \perp \).
- A so called iteration flag (initially \( \#f \)) indicating whether a recursive call to the same lambda closure with identical arguments appeared during evaluating the closure’s body. This flag will be used by the recursion interrupting semantics presented in Def. 6.5.4.

Formally, the abstract semantics with \( \lambda \)-history is defined as follows:
Definition 6.5.3 (abstract semantics with λ-history)  For a pre-application environment \(E = (L, \Sigma)\) the abstract semantics with λ-history differs from the abstract semantics with variables just in changing the function apply-lambda from Def. 6.3.14 to apply-lambda\(_H\). The changed lambda application function apply-lambda\(_H\) is defined as follows:

\[
\text{apply-lambda}_H(E) = \text{lambda-correct}_H(E) \cup \text{lambda-err}(E)
\]

where

\[
\text{lambda-correct}_H = \{ \tilde{E} \in \text{Env}_A | \exists E' = (L, \{\tilde{\sigma}\}) \in S(E).
\]

\[
E'(\text{func}) = lc_A(e(x_1, \ldots , x_k), \tilde{L}) \in LC_A,
\]

\[
E'(\text{arg}_k) \neq \text{undef}, E'(\text{arg}_{k+1}) = \text{undef},
\]

\[
E'(\text{arg}_i) \notin \{\text{error}, \bot\} \text{ for all } i,
\]

\[
H' := E'(\lambda\text{-hist}),
\]

\[
H'' := (((E'(\text{func}) E'(\text{arg}_1) \ldots E'(\text{arg}_n)) \perp \downarrow f). H')
\]

\[
\hat{E} = (\hat{L}, \hat{\Sigma}) := [e]^A((\text{update}_A(L[\lambda\text{-hist} \mapsto H''], E')[x_1 \mapsto L'(\text{arg}_1), \ldots ,
\]

\[
x_k \mapsto L'(\text{arg}_k)], \{\tilde{\sigma}\})),
\]

\[
\tilde{E} = \text{update}_A(\text{clearapp}(L[\text{return} \mapsto \hat{L}(\text{return}),
\]

\[
\lambda\text{-hist} \mapsto \text{rest}(\hat{E}(\lambda\text{-hist}))[\hat{\Sigma}], \hat{E}))
\]

with rest returning the rest of a given list, i.e. rest behaves like \(f_{\text{cdr}}^A\).

For lambda-err\(_E\) the definition presented in Def. 6.3.14 is used.

The next step of defining a iterated type inference is a recursion interrupting semantics: for every recursive call that does already occur in the history list an intermediate value is returned and the corresponding iteration flag is changed to \(#t\).

Definition 6.5.4 (recursion interrupting semantics) The recursion interrupting semantics is defined analogously to Def. 6.5.3 except for the following change of apply-lambda\(_H\) yielding apply-lambda\(_R\):

The following test is introduced into lambda-correct\(_H\): Let \(H' = E'(\lambda\text{-hist}) = (a_1 \ldots a_l)\). Is there a \(j \in \{1, \ldots , l\}\) such that

\[
a_j = ((R\text{-func} R\text{-arg}_1 \ldots R\text{-arg}_k) v \# f)
\]

and

\[
R\text{-func} = E'(\text{func}), R\text{-arg}_i = E'(\text{arg}_i) \text{ for all } i?
\]

185
• If this condition is fulfilled, the new function apply-lambda
\[ R \]
returns
\[ \tilde{E} = \text{clearapp}(E')[\text{return} \mapsto v, \lambda \text{-hist} \mapsto (a_i \ldots a_{j+1} a'_j a_{j-1} \ldots a_1)] \]
where
\[ a'_j = ((R-\text{func} \ R-\text{arg}_1 \ldots R-\text{arg}_k) v \# t). \]
• Otherwise, apply-lambda
\[ R \]
behaves like apply-lambda
\[ H \].

The recursion flag and the intermediate result value can now be used by the \( \lambda \)-iterative semantics that implements the iterated type inference process. The \( \lambda \)-iterative semantics differs from Def. 6.5.4 in the following detail: if after evaluating a closure body the corresponding iteration flag equals \# t (i.e. during evaluating the closure body a recursive call to the same function with the same arguments was suppressed), then iterated type inference is performed. It consists of several iterated steps of evaluating the lambda closure body with each time updating the intermediate result for the recursive subcalls of the same form. After every step \( i \) we set \( t'_i = \tilde{E}(\text{return}), \) i.e. \( t'_i \) is the current approximate return value. The sequence \((t_j)_j\) is defined as follows:

• \( t_0 = \bot \).

• \( t_i = (\cup t'_i t_{i-1}) \) for \( i > 0 \) where \( t'_i \) is the result of the call under the assumption that identical recursive calls return \( t_{i-1} \).

In the definition above we use \( \cup \) as an upper bound operator as described in [NNH99]. In contrast to the description there we do not modify a completely given sequence \((\tilde{t}_j)_j\), but every \( t'_i \) is based on the already modified \( t_{i-1} \) instead on \( t'_{i-1} \). The resulting sequence \((t_j)_j\) is nevertheless an ascending chain.

**Definition 6.5.5 (\( \lambda \)-iterative abstract semantics)** The \( \lambda \)-iterative abstract semantics is defined analogous to Def. 6.5.4 except of changing apply-lambda
\[ R \] to apply-lambda
\[ I \] in the following way:

Let \( \hat{E} \) be the environment after evaluating the lambda-closure’s body\(^{17}\) and let \( a = (cl \ v \ f) \) be the first element of \( \hat{E}(\lambda \text{-hist}) \).

• If \( f = \# f \) then apply-lambda
\[ I \] behaves like apply-lambda
\[ R \].

• If \( f = \# t \) then \( t_{\text{tmp}} = (\cup \hat{L}(\text{return}) v) \) and every \( \hat{\sigma} \in \hat{\Sigma} \) is analyzed independently as follows:

\(^{17}\)i.e. apply-lambda
\[ I \] just refines those cases where the additional condition of apply-lambda
\[ R \] does not hold.
If $\rho \circ \sigma(t_{tmp}) \sqsubseteq \rho \circ \sigma(v)$ for every closed type substitution $\rho$ appropriate for $\sigma(t_{tmp})$ and $\sigma(v)$ holds then apply-lambda$_t$ returns

\[
\tilde{E} = \text{update}_A(\text{clearapp}((L[\text{return} \mapsto t_{tmp}, \\
\lambda\text{-hist} \mapsto \text{rest}(\tilde{E}(\lambda\text{-hist})), \{}\hat{\sigma}\}), \hat{E}).
\]

- Otherwise,

\[
\begin{align*}
E'_{\text{new}} &= (L, \{}\hat{\sigma}\), \\
H''_{\text{new}} &= (((E'_{\text{new}}(\text{func}) E'_{\text{new}}(\text{arg}_1) \ldots E'_{\text{new}}(\text{arg}_n)) t_{\text{tmp}} \neq f) \cdot H') \\
\hat{E}_{\text{new}} &= [e]^A((\text{update}_A(L[\lambda\text{-hist} \mapsto H''_{\text{new}}], \hat{E})[x_1 \mapsto L'(\text{arg}_1), \ldots, \\
x_k \mapsto L'(\text{arg}_k)], \{}\hat{\sigma}\)))
\end{align*}
\]

and the result $\hat{E}_{\text{new}}$ is processed as $\hat{E}$ before.

For the definition above it is important to detect the subtype property of $t_{tmp}$ and $v$. For the types generated during iterated type inference this can be quite time consuming, and in certain cases the subtype property might not be detected. Checking $t_{tmp} \sqsubseteq v$ might be much easier when $t_{tmp}$ is simplified in a semantics preserving manner beforehand where many type simplification stages are straightforward (e.g. simplifying $(\cup \text{int posint})$ to int). Such a transformation can furthermore reduce the size of many of the types passed around the system.

### 6.5.2 Termination Enforcement for Iterated Type Inference

By introducing iterated type inference in Sec. 6.5.1 we did a step toward termination of type inference, but termination has not yet been reached for every input. (Note that we used an upper bound operator for enforcing $(t_j)_j$ to be an ascending chain, but no widening operator as described in [NNH99].) Furthermore, we have not yet described how to cope with variables in the recursive case. We will discuss several problematic situations and possible solutions independently in the following subsections, yielding a terminating type inference procedure.

#### 6.5.2.1 Value Ascriptions

When a call to a recursive function can be evaluated with value ascriptions for all types then the situation of calls with the same arguments may not occur. Nevertheless, an infinite execution is possible.
Example 6.5.6 Consider the factorial function defined in Ex. 6.5.1 and a call \((\text{fac } x)\) where \(x\) is bound to \(\mathcal{A}(500)\) in the abstract semantics. When evaluating the body of \(\text{fac}\) in an environment with \(n \mapsto \mathcal{A}(500)\) just the else-case is evaluated. It yields to a recursive call to \(\text{fac}\) with the argument \(\mathcal{A}(499)\).

In this case the evaluation will terminate after 500 recursion steps yielding the correct result. But we cannot be sure that user-defined functions terminate for every input. In the given example the test \((\leq n 0)\) could be replaced by \((= n 0)\) defining a function \(\text{fac2}\). Then a call to \(\text{fac2}\) with the argument \(\mathcal{A}(-3)\) does not terminate as \((\text{fac2 } -3)\) does not terminate in the standard semantics.

Since evaluating a program with value ascriptions may not terminate (because the program itself is not terminating) we introduce a threshold for the number of calls to a function with value ascriptions in the arguments. When exceeding this threshold the next recursive call is performed with every value ascription generalized to its most special base type.

6.5.2.2 Input Types Containing Variables

When the input types of a function call are not completely known input types containing (or consisting of) variables are used. These variables can be processed in the following ways during iterated type inference:

1. A variable in the input type is fully instantiated during the evaluation.

2. A variable in the input type is instantiated with a structure containing further variables.
   A recursive call is performed for a substructure that is similar to the original structure except of the occurrence of different variables.

3. A variable in the input type is not restricted at all.

In case (1) the variable does not cause any problems for termination. After the variable has been fully instantiated the calculation goes on as for a closed value. No non-standard actions are necessary in order to enforce termination. Case (3) indicates that the argument type represented by the variable does not matter at all. This is e.g. the case for polymorphic functions. Special actions are not necessary in this case either.

Case (2) causes the following problem: in every recursive call a variable occurring in a structure in one of the input arguments is instantiated with a structure of the same form containing a new variable. This yields an infinite number of calls with argument types that just differ in the variables used, but that are not identical.
Example 6.5.7 Consider the function reverse for reversing lists defined as follows:

\[
\text{(define (reverse l)}
\begin{align*}
&\quad \text{(rev l ())}) \quad ; \text{Call rev with empty accumulator} \\
&\quad \text{(define (rev l acc)}
\begin{align*}
&\quad \text{(if (null? l) \quad ; if complete list is processed}
&\quad \quad \text{acc} \quad \quad ; \text{return accumulated result}
&\quad \quad \text{(rev (cdr l) (cons (car l) acc)))})
\end{align*}
\end{align*}
\]

Consider the abstract evaluation of a call to reverse with the type variable \(A\) as argument. It causes a call to rev with \(A\) as first argument and () as second. (In this example we will not consider the argument acc any further). The condition of the if-expression of rev restricts \(A\) to () for the then-case and to a type \(t \neq ()\) for the else-case. In the else-case \(A\) is further restricted to \((B \cdot A')\) by the predefined functions cdr and car. Evaluating the else-expression results in a recursive call to rev with \(A'\) as argument. \(A'\) is restricted to \((B' \cdot A'')\) in the else case causing a call to rev with argument \(A''\). This causes an infinite loop with different arguments (i.e. \(A, A', A'', \ldots\) generated by the variable restrictions \(A \leftarrow (B \cdot A'), A' \leftarrow (B' \cdot A''), A'' \leftarrow (B'' \cdot A'''), \ldots\)) to reverse in every call.

Obviously, in such a case all the variables (\(A\) and \(A'\) in the example) express the same smallest type covering all values the analyzed function (reverse in the example) is applicable to. We can therefore use the same variable \(A\) for both. Since the binding of \(A\) is now defined in terms of \(A\) itself we need a recursive binding (maybe with subsequent renaming of the bound variable).

Example 6.5.8 Consider the situation in Ex. 6.5.7 again. \(A\) is restricted to \(( \cup () (B \cdot A'))\). By using \(A'\) as first argument to a recursive call to rev \(A'\) and \(A\) both represent the type of all valid input values to rev at argument position one. (Again acc is not considered for the moment.) We can set \(A' = A\) and therefore \(A = (\cup () (B \cdot A))\). The least fixed point of this equation is the recursive type \(A = \mu X_A.((\cup () (B \cdot X_A)) (with A renamed to X_A inside the recursive type)).

6.5.2.3 Base Types

Base types can occur as input or output arguments. Base types occurring as input arguments of a lambda closure \(f\) may change from call to call, but since the number of base types is finite the same base type must occur a second time after a finite number of recursive calls to \(f\). This is the standard case for iterated type inference.
When a base type occurs as output value of a recursive function definition an ascending chain of intermediate output types is calculated by iterated type inference. Since for every base type $t$ the number of base types $t'$ with $t \sqsubseteq t'$ is finite the chain must stabilize after a finite number of steps. Thus, there are no special steps to take in order to enforce termination for base types.

6.5.2.4 Structured Types

In general, and especially for structured types, ascending chains are possible that do not stabilize. In iterated type inference such a chain can either occur for the output type or it can cause an infinite number of calls to the same function all with different input types.

Example 6.5.9 Consider the definition of reverse in Ex. 6.5.7. In Ex. 6.5.8 the input type $\mu X. (\cup () (B \cdot X))$ was inferred. For the second input argument of rev the following sequence of types is inferred.

$t_1 = \perp$
$t_2 = (B \cdot \perp)$
$t_3 = (B \cdot (B \cdot \perp))$
$t_4 = (B \cdot (B \cdot (B \cdot \perp)))$
$t_5 = (B \cdot (B \cdot (B \cdot (B \cdot \perp))))$

In order to detect the infinite loops we introduce two thresholds for

- the maximal number of steps of iterated type inference.
- the maximal number of calls to the same function with different input arguments.

The thresholds should be chosen large enough that all ascending sequences of base types should be able to stabilize before the thresholds cause interruption of the process.

When one of the thresholds stops the process we need to generate a type that is a supertype of all the types given in the sequence generated so far. For this generated type we have to check again whether the process of iterated type inference stabilizes. Such a type can be generated as follows:

1. We can try to detect a repeated structural change in the sequence of types calculated so far. This uniform change can be used to construct a recursive type.
2. We can use $\top$ as a valid (though very weak) approximation for every type.

Method (2) can be applied either if (1) fails to return a recursive type $t$ or if this type $t$ fails the following test. One possibility to define method (1) is now explained in detail.

**Definition 6.5.10 (recursive type generation)** The function $\text{rec-gen}$ expects a list $l$ of types of the form $(t_1 \ t_2 \ \ldots \ t_n)$ and returns a type $t$ with $t_i \sqsubseteq t$ for all $i$ or $\bot$:

**Algorithm:** $\text{rec-gen}$

**Input:** A list $l = (t_1 \ t_2 \ \ldots \ t_n)$ of types.

**Output:** A type $t$.

1. Calculate the set $S = \{l_1, l_2, \ldots, l_m\}$ of all lists $l_i = (t_{i_1} \ \ldots \ t_{i_k_i})$ of maximal number of elements with the following properties:
   - All $t_{i_j}$ with $j > 1$ have the same top level type constructor.
   - If $k_i > 1$ then there is a position $p_i$ such that $t_{i_j} \approx t_{i_{j+1}|p_i}$ where $\approx$ denotes syntactic equality after identifying all free variables.
   - $t'_{i_1} = t'_{i_2} = \ldots t'_{i_k_i}$ where $t'_{i_j} = (t_{i_j}|p_i|\bot]$ with $t[p|t']$ as defined in Def. 3.1.11.

2. Set $\tilde{t}_i = \begin{cases} t_{i_i} & \text{for } k_i = 1 \\ \mu X_i. (\bigcup \tilde{t}_i \ | \tilde{t}_i[p_i|X_i]) & \text{else} \end{cases}$ where for every type $t$ the new type $\tilde{t} := \varphi (t)$ is generated a variable renaming $\varphi$ replacing every free variable in $t$ by a new variable.

3. Return $t = (\bigcup \tilde{t}_1 \ \tilde{t}_2 \ \ldots \ \tilde{t}_m)$.

Note that for every list $l_i$ the type $t_{i_i}$ does not follow the construction rules of $l_i$. $t_{i_i}$ rather acts as an initial type for the type transformation given by $l_i$. In the definition of $\tilde{t}_i$ we principally give a recursive binding for the uniform type transformation building $t_{i_2}, \ldots$ from their predecessor, respectively. This type transformation is given by $t_{i_2}[p_i|X_i]$. (Every $k$ with $2 \leq k \leq n$ is possible here instead of 2.) The additional union argument $t_{i_1}$ provides that after every number of unfoldings of $\tilde{t}_i$ the initial type occurs.

**Example 6.5.11** Consider the list $l = (t_1 \ \ldots \ t_5)$ with the $t_i$ as defined in Ex. 6.5.9. A call to $\text{rec-gen}$ with $l$ as argument is calculated as follows:

1. The set $S$ consists of exactly one list $l_1 = l$ with $p_1 = 2.e$. 

191
2. \( \tilde{t}_1 = \mu X_1. (\cup \perp (B \cdot X_1)) \)

3. \( t = (\cup \tilde{t}_1) \) can be simplified to \( \tilde{t}_1 \).

**Lemma 6.5.12 (termination of rec-gen)** The algorithm rec-gen terminates for every input.

**Proof:** See App. E, Page 271.

**Lemma 6.5.13** Let \( l = (t_1 \ t_2 \ldots \ t_n) \) be the input of rec-gen. There exists a closed substitution \( \sigma \) appropriate for every \( t_i \) and for \( t = \text{rec-gen}(l) \) such that \( t \) fulfills \( \langle \sigma(t_i) \rangle \subseteq \langle \sigma(t) \rangle \) for every \( i \in \{1,\ldots,n\} \).

**Proof:** See App. E, Page 272.

`rec-gen` should be called with the list of types fully instantiated according to the current substitution. This is necessary because the relation to the substitution is lost by the variable renaming performed by `rec-gen`. We expect `rec-gen` to return useful results when all types \( t_i \) from the input list except of a prefix belong to a list \( l_j \) with more than one element and when the number of these lists does not further increase by choosing a larger \( n \).

The algorithm `rec-gen` can be used for iterated type inference as follows: whenever the iterated type inference is stopped by one of the thresholds the corresponding list of types generated so far is generalized by `rec-gen`. A further step of iterated type inference is performed with the generated type \( t \) used where the types from the input list were used before. When this step succeeds \( t \) is taken instead of the types from the list. Otherwise, we further generalize \( t \) to \( \top \).

### 6.5.2.5 Using Variables in the Output Type

In certain cases the generation of recursive types for the output type of a function can be done more efficiently without exceeding a threshold: When a call to a function \( f \) causes a recursive call to \( f \) with the same arguments, we can introduce new variables \( B \) and \( B' \) for the output type of the initial and the recursive call, respectively. If \( B' \) is not restricted during the calculation of \( B \), we can express \( B \) in terms of \( B' \). Since \( B \) and \( B' \) both represent the output type of \( f \) for the same arguments, we can set \( B' = B \). This yields a definition of \( B \) containing \( B \) itself. The wanted type is then expressible by recursively binding \( B \).

The result type has to be checked in a further step of iterated type inference. If it is not a fixpoint, other termination enforcement tasks for structured types have to be performed.
Example 6.5.14 Consider the function list-inc that expects a list of numbers as argument and returns a list of the numbers incremented by 1. list-inc is defined as follows:\textsuperscript{18}

\[
\text{(define (list-inc l)}
\begin{array}{l}
\text{(if (null? l)} \\
\text{() \\
\text{(cons (+ 1 (car l)) \\
\text{(list-inc (cdr l))))}}
\end{array}
\]

Consider the abstract interpretation of a call to list-inc with the type

\[A := \mu X.(\cup \text{nil} \ (\text{num} \ X))\]

as argument. Iterated type inference without termination enforcement yields the following sequence of intermediate output types:

\[t_1 = (\cup \text{nil} \ (\text{num} \ \bot))\]
\[t_2 = (\cup \text{nil} \ (\text{num} \ (\cup \text{nil} \ (\text{num} \ \bot))))\]
\[t_3 = (\cup \text{nil} \ (\text{num} \ (\cup \text{nil} \ (\text{num} \ (\cup \text{nil} \ (\text{num} \ \bot))))))\]
\[\vdots\]

We can use e.g. the sequence \((t_1,t_2,t_3)\) as input to rec-gen in order to calculate the output type \(t = \mu X. (\cup \text{nil} \ (\text{num} \ X))\).

Alternatively, we can introduce a new variable \(B\) for the output type of list-inc when given an argument of type \(A\). Then we get the definition \(B = (\cup \text{nil} \ (\text{num} \ B'))\) where we can set \(B' = B\) and therefore \(B = \mu X_B. (\cup \text{nil} \ (\text{num} \ X_B))\).

### 6.5.3 Properties of Iterated Type Inference

In this section we state some main properties of the presented type inference system that are important for getting a usable output from the system.

The following theorem states that the introduced termination enforcement methods are powerful enough to guarantee termination for every input.

**Theorem 6.5.15 (termination)** The \(\lambda\)-iterative abstract semantics defined in Sec. 6.5.1 with the termination enforcement methods from Sec. 6.5.2 terminates for every input.

\textsuperscript{18}list-inc can alternatively be defined using map. One would not use the presented definition in practice, but it is useful to illustrate the described situation in a simple framework.

As the following theorem shows, every valid value that can occur as result of an expression in the standard semantics is member of a type inferred by the type inference system for this expression. The following theorem is an extension of Theorem 6.4.16 to the $\lambda$-iterative semantics with termination enforcement: the precondition that $[e_A]^A(E)$ has to terminate can be dropped because of Theorem 6.5.15.

**Theorem 6.5.16 (completeness with termination enforcement)** Let $e \in Expression$ and $e_A \in Expression_A$ such that $e_A$ is compatible with $e$ and let $E = (L, \Sigma) \in Env_A$ such that $[e_A]^A(E)$ is free of eval abstraction.

Then for every $E' \in \{\tau \circ \sigma(L)\}$ for a $\sigma \in \Sigma$ and a closed substitution $\tau$ appropriate for $\sigma(L)$ with $[e](E') \not\in \{\bot, error\}$ there is some $\tilde{\sigma} \in \tilde{\Sigma}$ and some $\tilde{\tau}$ appropriate for $\tilde{\sigma}(\tilde{L})$ such that

$$[e](E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L}(return))\}$$

and

$$EU(e, E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L}[return \leftarrow undef])\}.$$


As Theorem 6.5.16 states the $\lambda$-iterative abstract semantics with termination enforcement fulfills the same conditions on a complete type checker as stated at the end of Sec. 6.4. Furthermore, the current system terminates for every input as stated by Theorem 6.5.15. Thus, the goals formulated for complete type inference in Sec. 6.1 are reached.
Chapter 7

Applying the System to Scheme

7.1 Instantiating $CS$ to Scheme

The type language defined in Chap. 3 and the type inference system defined in Chap. 6 are
defined in a general manner that fits several functional programming languages. In order to
adapt the type inference process to a programming language of practical use, the following
definitions have to be refined:

1. The type language has to be refined by providing precise information about the base
types and free constructors forming the types in the chosen programming language.

2. The set of predefined function definitions in the programming language must be provided
together with an I/O-representation for each of these functions.

In the previous chapters, some design decisions where influenced by the programming language
Scheme as described in [KCE98]. This is the language we want to adapt the type inference
system to as follows:

1. The sets of base types and free type constructors needed to denote the values used by
   Scheme are defined in App. D.

2. The abstract behaviour of the predefined function definitions available in Scheme is
given in App. E.

This covers the complete programming language Scheme defined in the official language report
[KCE98], the only exception being destructive updates and continuations (cf. Chap. 6).
For a restricted version of the type inference system instantiated in this way there exists an implementation described in [Wim99]. The implementation of a term rewriting system with the capability of handling functions with variable arity is described in [Fro98]. [BH00] describes an important step towards a useful output format for error messages.

In the rest of this chapter we present the type inference process instantiated to Scheme and the result for two Scheme examples. Section 7.2 describes type inference of a well-typed Scheme program. A modification of this program containing a type error that is detectable by our system is presented in Sec. 7.3.

7.2 A Well-Typed Example

Consider the following definition of the function $v\cdot v\text{-}mult$. It expects two vectors with the first given as matrix of dimension $1 \times n$ and the second given as matrix with the dimension $n \times 1$ for an arbitrary $n \in \mathbb{N}$.

\[
\text{(define (v-v-mult row column)}
\text{;; Multiplication of two vectors}
\text{;; Both vectors are given as list of their elements.}
\text{(cond ((and (null? row) (null? column)) 0)}
\text{;; non-recursive case}
\text{(else}
\text{(+ (* (car row) (car column))}
\text{;; multiply first elements and process}
\text{;; the vector rests recursively}
\text{(v-v-mult (cdr row) (cdr column))))})
\]

After rewriting the $\text{cond}$-expression using several $\text{if}$-expressions we get the following equivalent function definition:

\[
\text{(define (v-v-mult row column)}
\text{;; Multiplication of two vectors}
\text{;; Both vectors are given as list of their elements.}
\text{(if (and (null? row) (null? column))}
\text{0 ;; non-recursive case}
\text{(+ (* (car row) (car column))}
\text{;; multiply first elements and process}
\text{;; the vector rests recursively}
\text{(v-v-mult (cdr row) (cdr column))))})
\]
In this definition we can rewrite the expressions formed by and or using if-expressions and get

\[
\text{(define (v-v-mult row column)}
\begin{align*}
\text{;;; Multiplication of two vectors} & \\
\text{;;; Both vectors are given as list of their elements.} & \\
\text{(if (if (null? row)} & \\
\text{\quad (null? column)} & \\
\text{\quad #f)} & \\
\text{\quad 0 ;; non-recursive case} & \\
\text{\quad (+ (* (car row) (car column))} & \\
\text{\quad \quad ;; multiply first elements and process} & \\
\text{\quad \quad ;; the vector rests recursively} & \\
\text{\quad \quad (v-v-mult (cdr row) (cdr column)))))} & \\
\end{align*}
\]

Applying this definition in the environment \( E := ((F_0, FS), \emptyset) \) (where \( FS \) is the environment binding the stores and \( F_0 \) contains exactly the bindings of the predefined functions given in App. E) binds the symbol \( v-v-mult \) to the abstract lambda closure

\[
lc := l_{c_A}(e(row, column), (F_0, FS))
\]

with

\[
e := (\text{if (if (null? row)} & \\
\text{\quad (null? column)} & \\
\text{\quad #f)} & \\
\text{\quad 0 ;; non-recursive case} & \\
\text{\quad (+ (* (car row) (car column))} & \\
\text{\quad \quad ;; multiply first elements and process} & \\
\text{\quad \quad ;; the vector rests recursively} & \\
\text{\quad \quad (v-v-mult (cdr row) (cdr column))))})
\]

in \( F_0 \).

We want to check this binding of \( v-v-mult \) without providing an input typing. We therefore choose two new type variables \( R, C \in V_f \) and check the call

\[
(v-v-mult R C)
\]

in the updated environment \( E \).
Evaluating $[[v \cdot v \cdot \text{mult } R \cdot C]]^A(E)$ is processed by rule (App) in Def. 6.3.36. The function apply first of all calculates the following pre-application environment

$$E_p := (\langle \text{func} \mapsto l_c, \text{arg}_1 \mapsto R, \text{arg}_2 \mapsto C \rangle \ F_0 \ \text{FS}, \{\emptyset\}) \ .$$

$E_p$ is just a simple abstract environment with func bound to a lambda closure. It suffices to apply apply-lambda (more precisely apply-lambda$_1$) to $E_p$.

apply-lambda$_1$ adds a new frame $F_1$ to the definition frame list of $l_c$ and applies the substitution $\emptyset$ to the new frame list. The result environment is:

$$E_1 := ((F_1 \ F_0 \ \text{FS}), \{\emptyset\})$$

with

$$F_1 := \langle \text{row} \mapsto R, \text{column} \mapsto C \rangle \ .$$

The body $e$ of $l_c$ is now evaluated in $E_1$.

e is an if-expression, and therefore the rule to choose from Def. 6.3.37 depends on the result of the condition:

$$[[\text{if} \ (\text{null}\ ? \ \text{row}) \ (\text{null}\ ? \ \text{column}) \ #f]]^A(E_1)$$

is again an if-expression. We therefore evaluate the conditional expression, i.e. we calculate

$$[[\text{null}\ ? \ \text{row}]]^A(E') \ .$$

This is an application and therefore the pre-application environment

$$E_p^1 := (\langle \text{func} \mapsto f^A_{\text{null}?}, \text{arg}_1 \mapsto R \rangle \ F_1 \ F_0 \ \text{FS}, \{\emptyset\})$$

is calculated.

Again $E_p'$ is a simple abstract environment. $E_p^1(\text{func})$ is a predefined functions. The only function yielding a result on $E_p'$ is apply-prev. First it calculates

$$PPAF_V(f^A_{\text{null}?}, R, \emptyset) \ .$$

In step (1) of Def. 6.4.9 PPAF$_V$ calculates

$$CEP_V(f^A_{\text{null}?}, R, \emptyset) = \{(\text{nil}, \#t, \{R \leftarrow \text{nil}\}), (\top \ \text{\&} \ \text{nil}, \#f, \{R \leftarrow \top \ \text{\&} \ \text{nil}\})\} \ .$$

The conditions (2a) and (2b) of Def. 6.4.9 do not rule out any of these results and PPAF$_V$ returns

$$\{(#t, \{R \leftarrow \text{nil}\}), (#f, \{R \leftarrow \top \ \text{\&} \ \text{nil}\})\} \ .$$
The result of $apply-pre_V$ is

$$
\{(F_1 F_0 F[S[\text{return} \mapsto \#t]], \{(R \leftarrow \text{n}i{l})\}),
\{(F_1 F_0 FS[\text{return} \mapsto \#f]), \{(R \leftarrow \top \setminus \text{n}i{l})\}\}\}
$$

and combining these two environments yields

$$
E^2 := ((F_1 F_0 F[S[\text{return} \mapsto V_1]], \{(R \leftarrow \text{n}i{l}, V_1 \leftarrow \#t), \{R \leftarrow \top \setminus \text{n}i{l}, V_1 \leftarrow \#f\}\}).
$$

Decomposing $E^2$ into simple abstract environments and evaluating

$$(\text{if} \ (\text{null} \ ? \ \text{row}) \ (\text{null} \ ? \ \text{column}) \ #f)$$

in them yields

- For $E^2_1 := (([\text{row} \mapsto R, \ \text{column} \mapsto C] F_0 FS[\text{return} \mapsto V_1]), \{(R \leftarrow \text{n}i{l}, V_1 \leftarrow \#t)\})$
  the return value fulfills $AT(E^2_1[\text{return}])$, and therefore Rule (If-True) is applied. This rule evaluates the then-expression $(\text{null} \ ? \ \text{column})$ in $E^2_1$ and returns the result. The pre-application environment calculated by $apply$ is

$$
E^2_{1p} := (([\text{func} \mapsto f^A_{\text{null}??}, \ \text{arg}_1 \mapsto C] [\text{row} \mapsto R, \ \text{column} \mapsto C] F_0 FS),
\{(R \leftarrow \text{n}i{l}, V_1 \leftarrow \#t)\})
$$

which is a simple abstract environment with func bound to an abstract predefined function. As before, $apply-pre_V$ returns

$$
E^2_1 := (([\text{row} \mapsto R, \ \text{column} \mapsto C] F_0 FS[\text{return} \mapsto V_2]),
\{(R \leftarrow \text{n}i{l}, V_1 \leftarrow \#t, C \leftarrow \text{n}i{l}, V_2 \leftarrow \#t),
\{R \leftarrow \text{n}i{l}, V_1 \leftarrow \#t, C \leftarrow \top \setminus \text{n}i{l}, V_2 \leftarrow \#f\}\}).
$$

- The return value of $E^2_2 := (([\text{row} \mapsto R, \ \text{column} \mapsto C] F_0 FS[\text{return} \mapsto V_1]), \{(R \leftarrow \top \setminus \text{n}i{l}, V_1 \leftarrow \#f)\})$ fulfills $AF$, and therefore Rule (If-False) applies. The else-expression $\#f$ is evaluated in $E^2_1$ with the result

$$
E^2_2 := (([\text{row} \mapsto R, \ \text{column} \mapsto C] F_0 FS[\text{return} \mapsto \#f]), \{(R \leftarrow \top \setminus \text{n}i{l}, V_1 \leftarrow \#f)\}).
$$

The result of

$$
[(\text{if} \ (\text{null} \ ? \ \text{row}) \ (\text{null} \ ? \ \text{column}) \ #f)^A(E_1)
$$

199
is now given by combining \( E_1^2 \) and \( E_2^2 \) which yields

\[
E^3 := (([\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto V_3]),
\{\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \text{nil}, V_3 \leftarrow \#t\},
\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \top \setminus \text{nil}, V_3 \leftarrow \#f\},
\{R \leftarrow \top \setminus \text{nil}, V_1 \leftarrow \#f, V_3 \leftarrow \#f\})
\].

Recall that the body \( e \) of the lambda closure \( lc \) is an if-expression and that \( E^3 \) represents the result of evaluating its conditional expression. The further evaluation of \( e \) is now done for the individual simple abstract environments of \( E^3 \) independently:

- For \( E^3_1 := (([\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto V_3]),\{\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \text{nil}, V_3 \leftarrow \#t\}) \) the return value fulfills \( AT \), Rule (If-True) applies, and the result is

\[
E^3_1 := (([\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto A(0)]),
\{\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \text{nil}, V_3 \leftarrow \#t\}).
\]

- \( E^3_2 := (([\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto V_3]),\{\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \top \setminus \text{nil}, V_3 \leftarrow \#f\}) \) Because of \( AF(E^3_2(\text{return})) \) Rule (If-False) applies, and the \( else \)-expression

\[
e'_2 := (+ \ ((\ast (\text{car row}) (\text{car column})\) (v-v-mult (\text{cdr row}) (\text{cdr column})))
\]

is evaluated in \( E^3_3 \). In order to evaluate this application expression the pre-application environment with respect to \( E^3_2 \) and \( e'_2 \) is calculated:

- The function expression \( + \) evaluates to \( f^A_+ \).
- To evaluate the first argument the expression \( e''_2 := (* (\text{car row}) (\text{car column})) \) must be evaluated in \( E^3_2 \) (that has not been changed by evaluating \(+\)). This again is an application expression, and the pre-application environment with respect to \( E^3_2 \) and \( e''_2 \) is calculated:
  * The function expression \( * \) evaluates to \( f^A_* \).
  * The first argument expression \( (\text{car row}) \) is an application expression. The pre-application environment is

\[
E^3_{2,1} := (([\text{func} \mapsto f^A_{\text{car}}, \text{arg}_1 \mapsto R] \ [\text{row} \mapsto R, \text{column} \mapsto C] \ F_0
\ \text{FS}[\text{return} \mapsto \#f]),
\{\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \top \setminus \text{nil}, V_3 \leftarrow \#f\})
\].

200
Applying \textit{apply-prev} to $E_{3,1,1}^3$ yields

\[
((\{row \mapsto R, \text{column} \mapsto C\} F_0 \text{FS}[\text{return} \mapsto \text{error}]),
\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \top \setminus \text{nil}, V_3 \leftarrow \#f\})
\]

because the result of $\text{PPAF}_V(f^A_{\text{car}}, R, \{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \top \setminus \text{nil}, V_3 \leftarrow \#f\})$ is empty.

Since the value \textit{error} bound to \textit{arg}_1 causes the application of \textit{apply-error}_V, we do not discuss the evaluation of the second argument. As result we get

\[
E'_{2}^3 := ((\{row \mapsto R, \text{column} \mapsto C\} F_0 \text{FS}[\text{return} \mapsto \text{error}]),
\{R \leftarrow \text{nil}, V_1 \leftarrow \#t, C \leftarrow \top \setminus \text{nil}, V_3 \leftarrow \#f\}).
\]

Note that we do not consider variable restrictions enforced by evaluating (\textit{car column}). This does not make a difference for the result of evaluating the call to \textit{lc} because this environment indicating an error case is discarded by the function \textit{gen-simple-env} that is called by \textit{apply-lambda}.\footnote{\textbf{201}}

$E_{3}^3 := ((\{row \mapsto R, \text{column} \mapsto C\} F_0 \text{FS}[\text{return} \mapsto V_3]), \{\{R \leftarrow \top \setminus \text{nil}, V_1 \leftarrow \#f, V_3 \leftarrow \#f\}\})$ fulfills $\text{AF}(E_{3}^3(\text{return}))$. The further evaluation of $e$ is done by Rule (\textit{If-False}) which evaluates the \textit{else}-expression

\[
e'_{3} := (+ (* (\text{car row}) (\text{car column})) (v-v-mult (\text{cdr row}) (\text{cdr column})))
\]

in $E_{3}^3$. $e'_{3}$ is an application expression and therefore the pre-application environment with respect to $E_{3}^3$ and $e'_{3}$ is calculated:

- The function expression $+$ again evaluates to $f^A_{\#}$.\footnote{\textbf{201}}
- For calculating the first argument the expression $e''_{3} := (* (\text{car row}) (\text{car column}))$ must be evaluated in $E_{3}^3$. This again is an application expression, and the pre-application environment with respect to $E_{3}^3$ and $e''_{3}$ is calculated:
  - The function expression $*$ evaluates to $f^A_{\#}$.
  - The first argument expression (\textit{car row}) is an application expression. The pre-application environment is

\[
E'_{3,1,1}^3 := ((\{\text{func} \mapsto f^A_{\text{car}}, \text{arg}_1 \mapsto R\} [\text{row} \mapsto R, \text{column} \mapsto C] F_0 \text{FS}[\text{return} \mapsto V_3]), \{R \leftarrow \top \setminus \text{nil}, V_1 \leftarrow \#f, V_3 \leftarrow \#f\}).
\]

This is a simple abstract environment with \textit{func} bound to an abstract predefined function and therefore \textit{apply-prev} applies. This function returns

\[
E_{3,1,1}^3 := (([\text{row} \mapsto R, \text{column} \mapsto C] F_0 \text{FS}[\text{return} \mapsto A]),
\{R \leftarrow (\cap (\top \setminus \text{nil}) (A . B)), V_1 \leftarrow \#f, V_3 \leftarrow \#f\}).
\]

For the first argument position we get the value $A$.\footnote{\textbf{201}}
* The second argument again is an application expression and applying it in $E'_{3}$ yields the pre-application environment

$$E'_{3,1,2} := \left( \left( \left[ \text{func} \mapsto f_{\text{car}}, \arg_{1} \mapsto C \right] \left[ \text{row} \mapsto R, \text{column} \mapsto C \right] \right. \right.$$  
$$\left. F_{0} \left[ \text{return} \mapsto A \right] \right),$$  
$$\{ \{ R \leftarrow (\cap (\top \setminus \text{nil}) (A \cdot B)), V_{1} \leftarrow \# f, V_{3} \leftarrow \# f \} \} \}.$$

This is again processed by $\text{apply-pre}_V$ and yields the result

$$E'^{3}_{3,1,2} := \left( \left[ \text{row} \mapsto R, \text{column} \mapsto C \right] \right.$$  
$$\left. F_{0} \left[ \text{return} \mapsto D \right] \right),$$  
$$\{ \{ R \leftarrow (\cap (\top \setminus \text{nil}) (A \cdot B)), V_{1} \leftarrow \# f, V_{3} \leftarrow \# f, C \leftarrow (D \cdot E) \} \} \}.$$

The return value states that the second argument of $e''_{3}$ evaluates to $D$.

Altogether the pre-application environment with respect to $E'_{3}$ and $e''_{3}$ is

$$\left( \left[ \text{func} \mapsto f_{\ast}, \arg_{1} \mapsto A, \arg_{2} \mapsto D \right] \left[ \text{row} \mapsto R, \text{column} \mapsto C \right] \right.$$  
$$\left. F_{0} \left[ \text{return} \mapsto D \right] \right),$$  
$$\{ \{ R \leftarrow (\cap (\top \setminus \text{nil}) (A \cdot B)), V_{1} \leftarrow \# f, V_{3} \leftarrow \# f, C \leftarrow (D \cdot E) \} \} \}.$$

For this pre-application environment just $\text{apply-pre}_V$ yields the following result:

$$E'^{3}_{3,1} := \left( \left[ \text{row} \mapsto R, \text{column} \mapsto C \right] \right.$$  
$$\left. F_{0} \left[ \text{return} \mapsto \text{num} \right] \right),$$  
$$\{ \{ R \leftarrow (\cap (\top \setminus \text{nil}) (A \cdot B)), V_{1} \leftarrow \# f, V_{3} \leftarrow \# f, C \leftarrow (D \cdot E), A \leftarrow \text{num}, D \leftarrow \text{num} \} \} \}.$$

The return-value represents the result of evaluating the first argument position of $e'_{3}$.

− The second argument expression of $e'_{3}$ is $e'''_{3} = (v-v\text{-mult} (\text{cdr} \text{ row}) (\text{cdr} \text{ column}))$. This is an application expression and therefore we calculate the pre-application environment with respect to $E'^{3}_{3,1}$ and $e'''_{3}$:

* The function expression evaluates to $\text{i}c$.

* Evaluating the first argument expression (cdr row) in $E'^{3}_{3,1}$ yields

$$E'^{3}_{3,2,1} := \left( \left[ \text{row} \mapsto R, \text{column} \mapsto C \right] \right.$$  
$$\left. F_{0} \left[ \text{return} \mapsto B \right] \right),$$  
$$\{ \{ R \leftarrow (\cap (\top \setminus \text{nil}) (A \cdot B)), V_{1} \leftarrow \# f, V_{3} \leftarrow \# f, C \leftarrow (D \cdot E), A \leftarrow \text{num}, D \leftarrow \text{num} \} \} \}.$$
Following the definition of \textit{apply-pre}_\mathcal{V} precisely \textbf{return} is bound to a new variable \(B'\) instead of \(B\) and the binding of \(R\) is updated to \((\cap (A \cdot B) (A' \cdot B'))\). The new variables \(A'\) and \(B'\) can be simplified away. We perform this simplification task in the current and the following evaluation steps to simplify the representation.

* The second argument expression (\textit{cdr column}) of \(e''\) analogously yields

\[
E_{3,2,2}^3 := \left(\left\{\left\{\text{\textit{row}} \mapsto R, \text{\textit{column}} \mapsto C\right\}\right.\right)
F_0 \; \text{\textbf{FS}}[\text{\textbf{return}} \mapsto E],
\left\{\left\{R \leftarrow (\cap (\top \setminus \text{\textit{nil}}) (A \cdot B)), V_1 \leftarrow \#f, V_3 \leftarrow \#f, C \leftarrow (D \cdot E), A \leftarrow \text{\textit{num}}, D \leftarrow \text{\textit{num}}\right\}\right.\right)\}.
\]

Altogether we get the following pre-application environment with respect to \(E_{3,1}^3\) and \(e''\):

\[
\left(\left\{\left\{\text{\textit{func}} \mapsto \text{\textit{lc}}, \text{\textit{arg}}_1 \mapsto B, \text{\textit{arg}}_2 \mapsto E\right\}\right.\right) \text{\textit{row}} \mapsto R, \text{\textit{column}} \mapsto C\right\}\right)
F_0 \; \text{\textbf{FS}}[\text{\textbf{return}} \mapsto E],
\left\{\left\{R \leftarrow (\cap (\top \setminus \text{\textit{nil}}) (A \cdot B)), V_1 \leftarrow \#f, V_3 \leftarrow \#f, C \leftarrow (D \cdot E), A \leftarrow \text{\textit{num}}, D \leftarrow \text{\textit{num}}\right\}\right.\right)\}.
\]

Since there already is an open call to \textit{lc} with two variables \((R\) and \(C\)) as arguments termination enforcement stops the current call with result \(\bot\) and identifies \(B\) with \(R\) and \(E\) with \(C\). The result environment is

\[
E_{3,2}^3 := \left(\left\{\left\{\text{\textit{row}} \mapsto R, \text{\textit{column}} \mapsto C\right\}\right.\right)
F_0 \; \text{\textbf{FS}}[\text{\textbf{return}} \mapsto \bot],
\left\{\left\{R \leftarrow (\cap (\top \setminus \text{\textit{nil}}) (A \cdot R)), V_1 \leftarrow \#f, V_3 \leftarrow \#f, C \leftarrow (D \cdot C), A \leftarrow \text{\textit{num}}, D \leftarrow \text{\textit{num}}\right\}\right.\right)\}.
\]

Because of the strictness of the abstract interpretation this is also the result environment of evaluating \(e''\).

Since the second argument position of \(e'\) evaluates to \(\bot\) the result of evaluating \(e'\) in \(E_3^3\) is \(\bot\) and we get the result environment \(E_{3,2}^3 := E_{3,2}^3\).

We get the first intermediate result of evaluating \(e\) by combining \(E_{1,2}^3\), \(E_{2,2}^3\) and \(E_{3,2}^3\) and applying \textit{gen-simple-env} to the result. We get:

\[
\left(\left\{\left\{\text{\textit{row}} \mapsto R, \text{\textit{column}} \mapsto C\right\}\right.\right)
F_0 \; \text{\textbf{FS}}[\text{\textbf{return}} \mapsto I],
\left\{\left\{R \leftarrow \mu V_R. (\cup \text{\textit{nil}} (\cap (\top \setminus \text{\textit{nil}}) (\text{\textit{num}} \cdot V_R))), V_1 \leftarrow (\cup \#t \#f), V_3 \leftarrow (\cup \#t \#f), C \leftarrow \mu V_C. (\cup \text{\textit{nil}} (\text{\textit{num}} \cdot V_C)), A \leftarrow \text{\textit{num}}, D \leftarrow \text{\textit{num}}, I \leftarrow (\cup \mathcal{A}(0) \bot)\right\}\right.\right)\}.
\]
After simplifying some of the types and deleting unused variables from the substitutions we get

\[ E' := (\{\text{first simple abstract environment}\}) \]

\[ F_0 \text{ FS}[\text{return} \mapsto I], \]
\[ \{\{R \leftarrow \mu V_R.(\cup \text{nil} (\text{num} . V_R)), C \leftarrow \mu V_C.((\cup \text{nil} (\text{num} . V_C)), I \leftarrow 0}\} \}. \]

In this environment and with the intermediate result set to \( A(0) \) \( e \) is evaluated again:

Evaluating the conditional expression of \( e \) is done analogously to the step before and yields

\[ \bar{E}^3 := (([\text{row} \mapsto R, \text{column} \mapsto C] F_0 \text{ FS} [\text{return} \mapsto V_3]), \]
\[ \{\{R \leftarrow (\cap \mu V_R.(\cup \text{nil} (\text{num} . V_R)) \text{nil}), \bar{V}_1 \leftarrow \#t, \]
\[ C \leftarrow (\cap \mu V_C.((\cup \text{nil} (\text{num} . V_C)) \text{nil}), \bar{V}_3 \leftarrow \#t\} , \]
\[ \{R \leftarrow (\cap \mu V_R.((\cup \text{nil} (\text{num} . V_R)) \text{nil}), \bar{V}_1 \leftarrow \#t, \]
\[ C \leftarrow (\cap \mu V_C.((\cup \text{nil} (\text{num} . V_C)) \text{nil}), \bar{V}_3 \leftarrow \#t\} , \]
\[ \{R \leftarrow (\cap \mu V_R.((\cup \text{nil} (\text{num} . V_R)) \text{nil}), \bar{V}_1 \leftarrow \#f, \]
\[ C \leftarrow (\cap \mu V_C.((\cup \text{nil} (\text{num} . V_C)) \text{nil}), \bar{V}_3 \leftarrow \#f\} \}. \]

For the first two simple abstract environments the calculation is essentially analogous to the first iteration. For the first simple abstract environment the result is

\[ \bar{E}^3_1 := (\{\text{first simple abstract environment}\} F_0 \text{ FS} [\text{return} \mapsto A(0)], \]
\[ \{\{R \leftarrow (\cap \mu V_R.((\cup \text{nil} (\text{num} . V_R)) \text{nil}), \bar{V}_1 \leftarrow \#t, \]
\[ C \leftarrow (\cap \mu V_C.((\cup \text{nil} (\text{num} . V_C)) \text{nil}), \bar{V}_3 \leftarrow \#t\} \}. \]

and for the second one the result \( E^3_2 \) is not of interest, because its \text{return}-value is \text{error} and it is therefore ruled out by \text{gen-simple-env} later.

The third simple abstract environment is processed as follows:

Since \( \bar{E}^3 \) fulfills \( AF(\bar{E}^3_2[\text{return}]) \) Rule (If-False) evaluates the \text{else}-expression

\[ e' := (+ (* (\text{car row}) (\text{car column})) (v-v-mult (\text{cdr row}) (\text{cdr column}))) \]

This is an application expression and as in the step before we get \( f^A_+ \) for the function expression. Evaluating the first argument expression (*) (\text{car row}) (\text{car column}) in \( \bar{E}^3 \) yields:

\[ \bar{E}^3_{3,1} := (([\text{row} \mapsto R, \text{column} \mapsto C] F_0 \text{ FS} [\text{return} \mapsto \text{num}], \]
\[ \{\{R \leftarrow (\cap \mu V_R.((\cup \text{nil} (\text{num} . V_R)) (\cap \text{nil}) (\bar{A} \cdot \bar{B})), \bar{V}_1 \leftarrow \#f, \bar{V}_3 \leftarrow \#f, \]
\[ C \leftarrow (\cap \mu V_C.((\cup \text{nil} (\text{num} . V_C)) (\cap \text{num} . \bar{D} \cdot \bar{E})), \bar{A} \leftarrow \text{num}, \bar{D} \leftarrow \text{num}\} \}. \]

204
Now we evaluate \((v\cdot v\cdot \text{mult} \ (\text{cdr \ row}) \ (\text{cdr \ column}))\) in \(\tilde{E}_{3,1}^3\):

- The function expression evaluates to \(lc\).
- The first argument expression is evaluated analogously to the first iteration step. The result is:

\[
\tilde{E}_{3,2,1}^3 := (([\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto \tilde{B}]),
\{\{R \leftarrow (\cap \mu V_R.(\cup \text{nil} \ (\text{num} \ . \ V_R)) \ (\top \ \text{\setminus} \ \text{nil}) \ (\tilde{A} \ . \ \tilde{B})), \tilde{V}_1 \leftarrow \#f, \tilde{V}_3 \leftarrow \#f,
C \leftarrow (\cap \mu V_C.(\cup \text{nil} \ (\text{num} \ . \ V_C)) \ (\tilde{D} \ . \ \tilde{E})), \tilde{A} \leftarrow \text{num}, \tilde{D} \leftarrow \text{num}\}\}).
\]

- Evaluating the second argument expression in \(\tilde{E}_{3,2,1}^3\) yields

\[
\tilde{E}_{3,2,2}^3 := (([\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto \tilde{E}]),
\{\{R \leftarrow (\cap \mu V_R.(\cup \text{nil} \ (\text{num} \ . \ V_R)) \ (\top \ \text{\setminus} \ \text{nil}) \ (\tilde{A} \ . \ \tilde{B})), \tilde{V}_1 \leftarrow \#f, \tilde{V}_3 \leftarrow \#f,
C \leftarrow (\cap \mu V_C.(\cup \text{nil} \ (\text{num} \ . \ V_C)) \ (\tilde{D} \ . \ \tilde{E})), \tilde{A} \leftarrow \text{num}, \tilde{D} \leftarrow \text{num}\}\}).
\]

Altogether, we get the pre-application environment

\[
([\text{func} \mapsto lc, \text{arg}_1 \mapsto \tilde{B}, \text{arg}_2 \mapsto \tilde{E}] \ [\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto \tilde{A}(0)],
\{\{R \leftarrow (\cap \mu V_R.(\cup \text{nil} \ (\text{num} \ . \ V_R)) \ (\top \ \text{\setminus} \ \text{nil}) \ (\tilde{A} \ . \ \tilde{B})), \tilde{V}_1 \leftarrow \#f, \tilde{V}_3 \leftarrow \#f,
C \leftarrow (\cap \mu V_C.(\cup \text{nil} \ (\text{num} \ . \ V_C)) \ (\tilde{D} \ . \ \tilde{E})), \tilde{A} \leftarrow \text{num}, \tilde{D} \leftarrow \text{num}\}\}).
\]

This is processed by returning the intermediate result of \(lc\):

\[
\tilde{E}_{3,2}^3 := (([\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto \tilde{A}(0)]),
\{\{R \leftarrow (\cap \mu V_R.(\cup \text{nil} \ (\text{num} \ . \ V_R)) \ (\top \ \text{\setminus} \ \text{nil}) \ (\tilde{A} \ . \ \tilde{B})), \tilde{V}_1 \leftarrow \#f, \tilde{V}_3 \leftarrow \#f,
C \leftarrow (\cap \mu V_C.(\cup \text{nil} \ (\text{num} \ . \ V_C)) \ (\tilde{D} \ . \ \tilde{E})), \tilde{A} \leftarrow \text{num}, \tilde{D} \leftarrow \text{num}\}\}).
\]

After evaluating the subexpressions we get the following pre-application environment with respect to \(\tilde{E}_{3}^3\) and \(c_3^e\):

\[
([\text{func} \mapsto f_{\rightarrow}^A, \text{arg}_1 \mapsto \text{num}, \text{arg}_2 \mapsto \tilde{A}(0)] \ [\text{row} \mapsto R, \text{column} \mapsto C] \ F_0 \ \text{FS}[\text{return} \mapsto \tilde{A}(0)]),
\{\{R \leftarrow (\cap \mu V_R.(\cup \text{nil} \ (\text{num} \ . \ V_R)) \ (\top \ \text{\setminus} \ \text{nil}) \ (\tilde{A} \ . \ \tilde{B})), \tilde{V}_1 \leftarrow \#f, \tilde{V}_3 \leftarrow \#f,
C \leftarrow (\cap \mu V_C.(\cup \text{nil} \ (\text{num} \ . \ V_C)) \ (\tilde{D} \ . \ \tilde{E})), \tilde{A} \leftarrow \text{num}, \tilde{D} \leftarrow \text{num}\}\}).
\]

205
The result of this application is:

\[
\tilde{E}_3 := \left(\left(\begin{array}{r}
\text{\(row\)} & \rightarrow & R,
\text{\(column\)} & \rightarrow & C
\end{array}\right) \mathcal{F}_0 \mathcal{S}[\text{return} \rightarrow \text{\(num\)}],
\{ \{R \leftarrow (\cap \mu V_R. (\cup \text{\(nil\)} (\text{\(num\)} \cdot V_R)) (\top \setminus \text{\(nil\)}) (\tilde{A} \cdot \tilde{B})), \tilde{V}_1 \leftarrow \#f, \tilde{V}_3 \leftarrow \#f,
C \leftarrow (\cap \mu V_C. (\cup \text{\(nil\)} (\text{\(num\)} \cdot V_C)) (\tilde{D} \cdot \tilde{E})), \tilde{A} \leftarrow \text{\(num\)}, \tilde{D} \leftarrow \text{\(num\)}\}\}\right).
\]

By combining \(\tilde{E}_3\), \(\tilde{E}_2\) and \(\tilde{E}_3\) and applying \textit{gen-simple-env} to the result we get (after some simplification):

\[
\tilde{E}' := \left(\left(\begin{array}{r}
\text{\(row\)} & \rightarrow & R,
\text{\(column\)} & \rightarrow & C
\end{array}\right) \mathcal{F}_0 \mathcal{S}[\text{return} \rightarrow \tilde{I}],
\{ \{R \leftarrow \mu \tilde{V}_R. (\cup \text{\(nil\)} (\text{\(num\)} \cdot \tilde{V}_R)), C \leftarrow \mu \tilde{V}_C. (\cup \text{\(nil\)} (\text{\(num\)} \cdot \tilde{V}_C)), \tilde{I} \leftarrow \text{\(num\)}\}\}\right).
\]

The result value \text{\(num\)} is taken as new intermediate value. The third iteration is processed analogously to the second one except of the recursive call to \textit{v-v-mult}. This call returns the new intermediate result \text{\(num\)}. The enclosing call \(\epsilon'_3\) still returns \text{\(num\)} and therefore there is no change in the result of evaluating \(e\). Iterated type inference stops here and returns the following environment:

\[
((\mathcal{F}_0 \mathcal{S}[\text{return} \rightarrow \tilde{I}]),
\{ \{R \leftarrow \mu \tilde{V}_R. (\cup \text{\(nil\)} (\text{\(num\)} \cdot \tilde{V}_R)), C \leftarrow \mu \tilde{V}_C. (\cup \text{\(nil\)} (\text{\(num\)} \cdot \tilde{V}_C)), \tilde{I} \leftarrow \text{\(num\)}\}\}\right).
\]

From getting this result for the call \((\textit{v-v-mult} R C)\) we get the following I/O-representation for the type of \textit{v-v-mult}:

\[
\{ (\times \mu \tilde{V}_R. (\cup \text{\(nil\)} (\text{\(num\)} \cdot \tilde{V}_R)) \mu \tilde{V}_C. (\cup \text{\(nil\)} (\text{\(num\)} \cdot \tilde{V}_C))) \rightarrow \text{\(num\)} \}.
\]

This is exactly the type one can expect for \textit{v-v-mult}.

### 7.3 An ill-typed Example

Consider the following erroneous definition of the function \textit{v-v-mult} from the previous section.

\begin{verbatim}
(define (v-v-mult row column)
  ;;; Multiplication of two vectors
  ;;; Both vectors are given as list of their elements.
  (cond ((and (null? row) (null? column)) 0)
    (else (v-v-mult (cdr row) (cdr column))))
\end{verbatim}

\footnote{For simplification reasons we use the variable names as generated in the second iteration, not in the third one.}
(else
  (+ (* (car row) (car column))
       ;; multiply first elements and process
       ;; the vector rests recursively
       (* row column)))
) ; Ill-typed call to *

Type checking a call $(v\cdot v\cdot \text{mult } R\ C)$ is done as stated in the previous section until the pre-
application environment with respect to $E^3_3 = (((\text{row} \mapsto R, \text{column} \mapsto C) F_0 \ \text{FS[return} \mapsto
V_3]),\{\{R \gets \top \setminus \text{nil}, V_1 \gets \# f, V_3 \gets \# f\}\})$ and

$$e_3^3 := (+ (* (\text{car row}) (\text{car column})) (* \text{row column}))$$

is calculated:

- The function expression + evaluates to $f^A_+$ without changing $E^3_3$.
- The first argument expression evaluates as before. The result environment is

$$E^3_{3,1} = (((\text{row} \mapsto R, \text{column} \mapsto C) F_0 \ \text{FS[return} \mapsto \text{num}])),\{\{R \gets (\cap (\top \setminus \text{nil}) (A \cdot B)), V_1 \gets \# f, V_3 \gets \# f, C \gets (D \cdot E), A \gets \text{num}, D \gets \text{num}\}\}.$$ 

- Evaluating the second argument expression $(\times \text{row column})$ in $E^3_{3,1}$ is done as follows:
  - The function expression $\times$ evaluates to $f^A_\times$ without changing $E^3_{3,1}$.
  - The first argument expression evaluates to $R$.
  - The second argument expression evaluates to $C$.

The call to $\text{apply-pre}_V$ with the resulting pre-application environment yields a call

$$\text{PPAF}_V(f^A_\times, (\times R C), \{R \gets (\cap (\top \setminus \text{nil}) (A \cdot B)), V_1 \gets \# f, V_3 \gets \# f, C \gets (D \cdot E), A \gets \text{num}, D \gets \text{num}\}).$$

Since there are no expected input types of $f^A_\times$ that have common elements with the
bindings of $R$ and $C$ the result of $\text{PPAF}_V$ is empty and $\text{apply-pre}_V$ returns error.

The output type of $v\cdot v\cdot \text{mult}$ is just taken from $E^3_{3,1}$ (i.e. the result type is $A(0)$) and an error
is risen for the call $(\times \text{row column})$ in the last line of the definition of $v\cdot v\cdot \text{mult}$ which cannot
be executed without an error.
Chapter 8

Conclusions and Future Work

In this work we have presented a type inference and type checking system that follows our new approach of completeness of type checking. Our type checker is therefore guaranteed to accept every program that can not cause a runtime type error in a dynamically typed functional language. For each function definition in a program our system calculates a type that denotes every valid pair of input and output for this function.

To get the maximum precision for the type inference process we defined a powerful type language that among others contains a union type constructor, recursive types, and value assignments denoting exactly one value. We furthermore identified a problem caused by the usual function type constructor in the framework of complete type checking and presented a different definition of function types (i.e. I/O-representations as well as PI/PO-representations for higher order functions) to solve this problem.

As one of the main questions arising during complete type checking we identified the question whether two types denote common elements or not. An algorithm $CE$ was presented that generates constraints on variable restrictions and transforms these constraint sets into a set of idempotent substitutions preserving all common elements of two given types. $CE$ was proven to yield an approximation of the above question in the sense that whenever common elements of two types exist they are indeed detected.

For the type inference task abstract interpretation was identified as the appropriate tool. After presenting the standard semantics of our functional language that was heavily influenced by Scheme, a first abstract semantics was defined. This abstract semantics was extended in order to work on type variables and to process recursive function definitions correctly. The resulting abstract interpreter does not need any input additional to the program to be analyzed. It is guaranteed to terminate for every input.
An instantiation of the abstract semantics was provided in order to process example programs expressed in Scheme, and examples of type inference for both well-typed and ill-typed programs were given illustrating the behaviour of the resulting type inference procedure.

There are different applications we have in mind for the system presented here. For a use in software engineering we focus on the type checking capabilities of the system. For dynamically typed functional languages a combination of the complete type checker with a soft typing system can provide output messages in a structured manner:

- Parts of the analyzed program that is not accepted by the complete type checker (and therefore also by the sound soft typing system) are marked with an error. An error always expresses the fact that this program part definitely contains a type error that should be corrected with high priority.

- Program parts that are accepted by the complete type checker, but not by the soft typing system are marked with a warning. As usual in soft typing the programmer has to check whether such a warning indicates a typing error in the program or is just a false alarm.

- Program parts that are accepted by the soft typing system are also accepted by the complete type checker, and therefore no message is generated. These program parts are proven well-typed by the sound type checker.

Another application for our system is a program performing program analysis or program optimization of some kind on a dynamically typed language that needs a type inference system as a subroutine. In contrast to type inference systems with sound type checkers our system is able to return a type for every program part that does not cause runtime type errors. Under certain conditions the inferred type may not be very accurate and in some cases types are inferred for program parts that may cause type errors for certain inputs. If, however, the types of expressions and not the typability itself is of interest our system has the advantage of not erroneously stopping type inference with an error message.

The work presented in this thesis shows that a complete type checker for dynamically typed programming languages can be defined in principle when just purely functional (parts of) languages are considered. In order to provide a complete type checker that is practically usable e.g. for the complete set of language constructs available in Scheme the following additional work has to be done:

- A prototype implementation has to show the runtime behaviour of the system. Techniques for reducing the runtime has to be developed. (E.g. instead of performing complicated typing tasks of identical function calls iteratedly the results could be memorized.)
The system has to be extended in order to capture the whole set of programming constructs of real programming languages. For an application to the different dialects of LISP especially destructive updates must be introduced. Our abstract semantics is already prepared for this extension. Altogether, defining destructive updates in the abstract semantics incorporates solving the following problems:

- The scope of a destructive update on a symbol must be modeled correctly. By defining abstract environments that precisely model the structured environments used in the standard semantics of Scheme much of the necessary work is already done here.

- Destructive updates on a global variable can be used to communicate calculation results instead of return values. The conditions for termination of the \( \lambda \)-iterative semantics therefore have to be refined.

- When structures bound to different symbols share a common substructure destructive updates on substructures can affect several symbols. Sharing of substructures has to be modeled in an appropriate way during structure construction. For destructive updates on such structures the corresponding changes have to be propagated to all affected structures. By considering stored abstract values preparations for detecting the affected symbols have been done already. A further discussion of this problem in a restricted framework can be found in [WB98, Sec. 5.3].
Bibliography


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 $t_{|p} \neq \text{undefined}$ the function $\text{compargs}$ yields the number of proper prefixes $p'$ of $p$ such that $t_{|p'}$ has $C$ as top level constructor. For a type $t$ the set $\text{unint-pos}$ contains all positions $\tilde{p}$ with $t_{|\tilde{p}}$ having the top level constructor $\cup$ or $\cap$. Furthermore, for a type $t$ we define $\text{diffcount}(t)$ as the number of difference type constructors occurring in $t$ and $\text{compcount}(t)$ 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 $\text{diffcount}(t)$ is decremented by one. Since $t$ has a finite representation $\text{diffcount}(t)$ is finite. None of the other rules increases $\text{diffcount}(t)$ and thus the starting value of $\text{diffcount}(t)$ is a bound for the number of applications of the first rule.

- When applying the second or third rule, $\sum_{p \in \text{unint-pos}(t)} \text{compargs}(t, p)$ is decreased by one. The only rule increasing this value is the first one introducing new $\cap$ and $C$ constructors. We can switch to $\sum_{p \in \text{unintdiff-pos}(t)} \text{comp-diff-args}(t, p)$ where $\text{unintdiff-pos}$ also yields all position with $\setminus$ at top level and $\text{comp-diff-args}$ differs from $\text{compargs}$ 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 $\text{compcount}(t)$ by 2 in every application. Since the starting value of $\text{compcount}(t)$ 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 $\text{CC}(\bigcup <a_1 \ldots e_1>)$. This can be normalized to $(\bigcup <a_1 \ldots e_1>)$ by rule 4 directly or we can apply rule 2 followed by rule 3 yielding $(\bigcup <\text{CC}a_1 \ldots \text{CC}a_1>)$. Here we can apply rule 4 to every argument of the union and get the same result $(\bigcup <a_1 \ldots e_1>)$ as before.

- Rule 3 and rule 4 have the critical value $\text{CC}(\bigcup <a_1 \ldots e_1>)$. 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. \hfill $\Box$

**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 $\sigma$.

**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.
   
   \[
   e \in A \setminus B \iff e \in A \land e \not\in B \iff e \in A \land e \in CB \iff e \in A \cap CB.
   \]

2. Second rule: Let $A_1, \ldots , A_k$ be sets and $e$ an element.
   
   \[
   e \in \mathcal{C}(\cup A_1 \ldots A_k)
   \iff e \not\in (\cup A_1 \ldots A_k)
   \iff \forall i . e \not\in A_i
   \iff \forall i . e \in \mathcal{C}A_i
   \iff e \in (\cap \mathcal{C}A_1 \ldots \mathcal{C}A_k).
   \]

3. Third rule: Let $A_1, \ldots , A_k$ be sets and $e$ an element.
   
   \[
   e \in \mathcal{C}(\cap A_1 \ldots A_k)
   \iff e \not\in (\cap A_1 \ldots A_k)
   \iff \exists i . e \not\in A_i
   \iff \exists i . e \in \mathcal{C}A_i
   \iff e \in (\cup \mathcal{C}A_1 \ldots \mathcal{C}A_k).
   \]

4. Fourth rule: Let $A$ be a set and $e$ an element.
   
   \[
   e \in \mathcal{C}CA \iff e \not\in CA \iff e \in A.
   \]
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 = µX.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 ∈ 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) = true ⇒ ⟨σ(t_1)⟩ ⊆ ⟨σ(t_2)⟩

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 ∈ T_{SCS} obviously ⟨σ(t)⟩ ⊆ ⟨σ(t)⟩ holds. Thus, case (1) does not violate the lemma.

- For every closed type t the following properties hold:
  - ⊥ ∈ ⟨t⟩. This implies ⟨⊥⟩ ⊆ ⟨σ(t)⟩ and therefore proves correctness of the first subcase of case (2).
– $\langle \sigma(t) \rangle \subseteq \mathcal{V} = \langle \top \rangle$ proves correctness of the second subcase of (2).

• If $t_1$ and $t_2$ are base types or value assignments the answer given by $ST_{base}$ that is correct due to Assumption 3.4.1. Case (3) is therefore correct.

• Because of $\langle T_{func} \rangle = \langle T_{func_p} \rangle \cup \langle T_{func_t} \rangle$ 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 this number is always finite:

$n = 0$: This case is always processed by the cases already proven above.

$n \to n + 1$: 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 $\sigma$ 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:

$$\langle \sigma((c \ t_{1,1} \ldots t_{1,k})) \rangle \subseteq \langle \sigma((c \ t_{2,1} \ldots t_{2,k})) \rangle \iff \forall_{i=1}^{k} \langle \sigma(t_{1,i}) \rangle \subseteq \langle \sigma(t_{2,i}) \rangle$$

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:

$$ST((c \ t_{1,1} \ldots t_{1,k}), (c \ t_{2,1} \ldots t_{2,k})) = true \Rightarrow \forall_{i=1}^{k} ST(t_{1,i}, t_{2,i}) = true \Rightarrow \forall_{i=1}^{k} \langle \sigma(t_{1,i}) \rangle \subseteq \langle \sigma(t_{2,i}) \rangle \iff \langle \sigma((c \ t_{1,1} \ldots t_{1,k})) \rangle \subseteq \langle \sigma((c \ t_{2,1} \ldots t_{2,k})) \rangle.$$  

• 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:

$$ST(t_i, t'_j) \Rightarrow \langle \sigma(t_i) \rangle \subseteq \langle \sigma(t'_j) \rangle.$$  

By the definition of $\langle \cdot \rangle$ on frames this implies $\langle \sigma(f_1) \rangle \subseteq \langle \sigma(f_2) \rangle$.

• If $ST(e_1, e_2)$ yields true because of case (7) then $e_1$ and $e_2$ must be environments $e_1 = (env \ e'_1 \ f_1)$ and $e_2 = (env \ e'_2 \ f_2)$ with $ST(e'_1, e'_2)$ and $ST(f_1, f_2)$. From the induction hypothesis we can conclude $\langle \sigma(e'_1) \rangle \subseteq \langle \sigma(e'_2) \rangle$ and $\langle \sigma(f_1) \rangle \subseteq \langle \sigma(f_2) \rangle$. Together with the definition of $\langle \cdot \rangle$ on environments this implies $\langle \sigma(e_1) \rangle \subseteq \langle \sigma(e_2) \rangle$.

\[1\text{More exactly because of Cor. 4.3.15 not needing Assumption 3.4.5.}\]
• 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 \( t_1 = (\cup t_{1,1} \ldots t_{1,k}) \) and case (9a) yields true then

\[
\forall i \in \{1, \ldots, k\} \ ST(t_{1,i}, t_2) = \text{true} \Rightarrow \forall i \in \{1, \ldots, k\} \ \{\sigma(t_{1,i})\} \subseteq \{\sigma(t_2)\} \\
\Rightarrow \{\sigma(t_1)\} = (\cup \{\sigma(t_{1,1})\} \ldots \{\sigma(t_{1,k})\}) \subseteq \{\sigma(t_2)\} .
\]

Therefore, case (9a) is correct.

- If \( t_2 = (\cup t_{2,1} \ldots t_{2,k}) \) and case (9b) returns true then

\[
\exists i \in \{1, \ldots, k\} \ ST(t_{1,i}, t_2) = \text{true} \Rightarrow \exists i \in \{1, \ldots, k\} \ \{\sigma(t_{1,i})\} \subseteq \{\sigma(t_2)\} \\
\Rightarrow \{\sigma(t_1)\} = (\cup \{\sigma(t_{1,1})\} \ldots \{\sigma(t_{1,k})\}) = \{\sigma(t_2)\} 
\]

implies the correctness of case (9b).

- If \( t_1 = (\cap t_{1,1} \ldots t_{1,k}) \) and \( ST \) returns true because of case (9c) then

\[
\exists i \in \{1, \ldots, k\} \ ST(t_{1,i}, t_2) = \text{true} \Rightarrow \exists i \in \{1, \ldots, k\} \ \{\sigma(t_{1,i})\} \subseteq \{\sigma(t_2)\} \\
\Rightarrow \{\sigma(t_1)\} = (\cap \{\sigma(t_{1,1})\} \ldots \{\sigma(t_{1,k})\}) \subseteq \{\sigma(t_2)\} .
\]

This implies the correctness of case (9c).

- If \( t_2 = (\cap t_{2,1} \ldots t_{2,k}) \) and case (9d) yields the result true then

\[
\forall i \in \{1, \ldots, k\} \ ST(t_{1,i}, t_2) = \text{true} \Rightarrow \forall i \in \{1, \ldots, k\} \ \{\sigma(t_{1,i})\} \subseteq \{\sigma(t_2)\} \\
\Rightarrow \{\sigma(t_1)\} = (\cap \{\sigma(t_{1,1})\} \ldots \{\sigma(t_{1,k})\}) = \{\sigma(t_2)\} .
\]

Thus, case (9d) is correct.

- If \( \ ST(t_1, t_2) = \text{true} \) because of case (9e) then \( t_1 = Ct_{1}', t_2 = Ct_{2}' \) and

\[
ST(t_{2}', t_{1}') = \text{true} \Rightarrow \{t_2\} \subseteq \{t_1\} \Rightarrow \{t_1\} = \mathcal{V} \setminus \{t_1'\} \subseteq \mathcal{V} \setminus \{t_2'\} = \{t_2\} .
\]

This implies the correctness of case (9e). (Note that the argument of \( C \) must not contain variables and therefore a substitution \( \sigma \) is not necessary in this case.)

- If \( \ ST(t_1, t_2) = \text{true} \) because of case (9f) then \( t_2 = Ct_{2}' \) and \( CE(\tilde{t}_1, t_{2}') = \text{false} \) with \( \tilde{t}_1 \) generated from \( t_1 \) by replacing every quantified variable \( X_{\mathcal{V}} \in V_{\mathcal{Q}} \) by \( \top \). By Assumption 3.4.6 and by the fact that all type constructors allowing variables in their arguments are monotonic and therefore \( \{\sigma(t_1)\} \subseteq \{\tilde{t}_1\} \) this implies:

\[
CE(\tilde{t}_1, t_{2}') = \text{false} \Rightarrow \\
\Rightarrow 0 = \{\tilde{t}_1\} \cap \{t_{2}'\} \supseteq \{\sigma(t_1)\} \cap \{t_2\} \Rightarrow \\
\Rightarrow \{\sigma(t_1)\} \subseteq \mathcal{V} \setminus \{t_{2}'\} = \{t_2\} .
\]

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. $t_1 = (\cup t_{1,1} \ldots t_{1,k})$ and $t_2 = (\cup t_{2,1} \ldots t_{2,k'})$
2. $t_1 = (\cap t_{1,1} \ldots t_{1,k})$ and $t_2 = (\cap t_{2,1} \ldots t_{2,k'})$

because $\lor$ and $\land$ 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:

$$\bigvee_{i=1}^{k} \bigwedge_{j=1}^{k'} ST(t_{1,i}, t_{2,j}) \Rightarrow \forall i \in \{1, \ldots, k\} \exists j \in \{1, \ldots, k'\} \langle t_{1,i} \rangle \subseteq \langle t_{2,j} \rangle \Rightarrow \bigvee_{i=1}^{k} \langle t_{1,i} \rangle \subseteq \bigcup \langle t_{2,k} \rangle \Rightarrow \langle t_{1,i} \rangle \subseteq \bigcup \langle t_{2,k} \rangle \Rightarrow \langle t_{1,i} \rangle \subseteq \langle t_{1,k} \rangle \subseteq \bigcup \langle t_{2,k} \rangle$$

The lemma is not violated by the order in which case 2 is processed.

In case 1 the term representing the performed checks is:

$$\bigwedge_{i=1}^{k} \bigvee_{j=1}^{k'} ST(t_{1,j}, t_{2,j}) \Rightarrow \forall i \in \{1, \ldots, k\} \exists j \in \{1, \ldots, k'\} \langle t_{1,i} \rangle \subseteq \langle t_{2,j} \rangle \Rightarrow \bigvee_{i=1}^{k} \langle t_{1,i} \rangle \subseteq \bigcup \langle t_{2,k} \rangle \Rightarrow \langle t_{1,i} \rangle \subseteq \bigcup \langle t_{2,k} \rangle \Rightarrow \langle t_{1,i} \rangle \subseteq \langle t_{1,k} \rangle \subseteq \bigcup \langle t_{2,k} \rangle$$

• Whenever the top level type constructor of one of the types is the $\mu$ 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 $\mu$. 

225
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 \textit{ST} as follows:

* The initial call to \textit{ST} is represented by the root.
* Every son of a given node represents a recursive subcall to \textit{ST} evaluated by the current call.
* Every leave of the tree represents a boolean value \texttt{true} or \texttt{false}.

This tree can just be infinite when both initial arguments contain the \texttt{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 \texttt{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 \textit{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 \texttt{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 \texttt{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 \texttt{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
Proofs of Chapter 4

B.1 Proofs of Section 4.2

Lemma 4.2.16 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 4.2.16: 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

$$\text{sub-pairs}(t_1, t_2) = \{(t'_1, t'_2) \mid t'_1 \text{ subterm of } t_1 \land t'_2 \text{ subterm of } t_2\}.$$  

$\text{sub-pairs}(t_1, t_2)$ 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 $\text{sub-pairs}(t'_1, t'_2) \subseteq \text{sub-pairs}(t_1, t_2)$.

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:

- $\text{sub-pairs}(t'_1, t'_2) \setminus r' \subset \text{sub-pairs}(t_1, t_2) \setminus r$.
- $\text{sub-pairs}(t'_1, t'_2) \setminus r' = \text{sub-pairs}(t_1, t_2) \setminus r \land (t'_1, t'_2) <_{TP} (t_1, t_2)$.

Since $<_{TP}$ does not contain infinite descending chains we can conclude that in every descending chain $\text{sub-pairs}(t_1, t_2)$ must be smaller or $r$ must be larger after a finite number of steps. Furthermore, $\text{sub-pairs}(t_1, t_2)$ and therefore $\text{sub-pairs}(t_1, t_2) \setminus r$ 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 $\text{sub-pairs}(t_1, t_2)$ 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)) $t'_1 = \text{unfold}(t_1)$, $t'_2 = t_2$ and $r' = ((t_1, t_2) \cdot r)$. This call
fulfills
\[(t_1, t_2) \in \text{sub-pairs}(t_1, t_2) \setminus r \supset \text{sub-pairs}(t'_1, t'_2) \setminus r' \not\in (t_1, t_2)\]

This is the case because \(\text{sub-pairs}(t_1, t_2) \supset \text{sub-pairs}(t'_1, t'_2)\) 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 \(i\)th recursive execution of \((SCE-list-reduce2)\) calculating the intermediate c-collection \(\Sigma_i\) this number of calls is equal to the number of elements in the previous c-collection \(\Sigma_{i-1}\). 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 4.2.19 Let \(\tilde{c}\) be a call to \(S-CE\) and \(c'\) the first call to \(SCE-list-reduce\) performed when processing \(\tilde{c}\). Let \(\Sigma\) be the result of a call
\[c' = SCE-list-reduce((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)\]
and let \(C_j\) denote the set of recursive calls to \(S-CE\) when processing the \(j^{th}\) list element. Let \(\Sigma_j\) be the c-collection used in the recursive call
\[SCE-list-reduce(((t_{j+1}, t'_{j+1}), \ldots, (t_n, t'_n)), \Sigma_j, r)\]
for all \( j \) and \( \Sigma'_j = S-CE(t_j, t'_j, \emptyset, r) \) then:

\[
\forall_{l=0}^n \text{combine-cs}(\Sigma) = \text{combine-cs}(\Sigma_l \otimes \bigcup_{j=l}^k \Sigma'_j) = \text{combine-cs}(\Sigma_l \otimes \text{implicit-cs}_\epsilon(C_l)) .
\]

where \( \Sigma_0 = \Sigma' \).

**Proof of Lemma 4.2.19:** We first prove the following statement (B.1) used in the rest of the proof:

\[
\bigcup_{\sigma \in \Sigma} S-CE(t_1, t_2, \sigma, r) = \Sigma \otimes S-CE(t_1, t_2, \emptyset, r) \quad (B.1)
\]

In proving (B.1) we first rule out several special cases:

- \( \Sigma = \emptyset \). Then both sides trivially yield the empty c-collection \( \emptyset \).
- \( S-CE(t_1, t_2, \emptyset, r) = \emptyset \) (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

\[
S-CE(t_1, t_2, \sigma, r) = \emptyset
\]

for every \( \sigma \in \Sigma \). Therefore,

\[
\bigcup_{\sigma \in \Sigma} S-CE(t_1, t_2, \sigma, r) = \emptyset .
\]

Now we consider the case that \( \Sigma = \{ \sigma \} \) and \( S-CE(t_1, t_2, \emptyset, r) \) returns with a c-collection consisting of exactly one element \( \sigma' \), i.e \( S-CE(t_1, t_2, \emptyset, r) = \{ \sigma' \} \) holds. We have to show that

\[
S-CE(t_1, t_2, \sigma, r) = \{ \sigma \otimes \sigma' \} . \quad (B.2)
\]

Obviously, both sides of (B.2) can just differ for those variables being constrained by an element of \( S-CE(t_1, t_2, \sigma, r) \) or by \( \sigma' \).

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 \((\text{BothVar})\), \((\text{Var1})\), \((\text{Var2})\), \((\text{Comp1})\) and \((\text{Comp2})\). 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).)

- \((\text{BothVar})\) 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 \( \sigma \) 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 $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ and let $\sigma'_k = \sigma_k \otimes \sigma'$. Analogously to (B.2) we get

$$\forall_{i=1}^k S\text{-}CE(t_1, t_2, \sigma_i, r) = \{\sigma_i \otimes \sigma'\} = \sigma'_i.$$ 

Obviously both sides of (B.1) yield exactly the set $\{\sigma'_1, \ldots, \sigma'_k\}$ and are therefore equal.

If $S\text{-}CE(t_1, t_2, \emptyset, r)$ 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 $\sigma_i \in S\text{-}CE(t_1, t_2, \emptyset, r)$ 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 $\sigma_i$. 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

$$S\text{-}CE(t'_1, t'_2, \emptyset, r) = \{\sigma_i\}.$$ 

For these types $t'_1$ and $t'_2$ (B.2) holds. This is the case for all $\sigma_i \in S\text{-}CE(t_1, t_2, \emptyset, r)$. Furthermore, the rules of $S\text{-}CE$ show the same behaviour when a free constraint set $\sigma \neq \emptyset$ is used in calling $S\text{-}CE$, i.e. there is a direct correspondence between the $\sigma_i$ and the results of $\sigma_i \in S\text{-}CE(t_1, t_2, \sigma, r)$ for every $\sigma$ via the common execution path of $S\text{-}CE$. Hence, (B.1) also holds for types $t_1$ and $t_2$ yielding an arbitrary number of free constraint sets as result from $S\text{-}CE$.

We can now prove the lemma. The situation after processing the first $j$ list elements by $SCE\text{-}list\text{-}reduce$ can be described as follows:

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

$$= SCE\text{-}list\text{-}reduce((\bar{t}_1, \bar{t}'_1), \ldots, (\bar{t}_{n-j}, \bar{t}'_{n-j})), \bar{\Sigma}, r)$$

Hereby the remaining calculations after processing $j$ list elements in $SCE\text{-}list\text{-}reduce$ can always be described as a not partially processed call to $SCE\text{-}list\text{-}reduce$. The statement of the
lemma now simplifies to:

\[
\text{combine-cs}(SCE\text{-}\text{list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)) =
\]

\[
= \text{combine-cs}(\Sigma' \otimes \bigotimes_{j=1}^{n} S\text{-}CE(t_j, t'_j, \emptyset, r)).
\]

Because of

\[
\text{combine-cs}(\Sigma' \otimes \Sigma' \otimes \tilde{\Sigma})
\]

\[
= \text{combine-cs}\left(\big\{ \sigma_1 \otimes \sigma_2 \otimes \tilde{\sigma} \mid \sigma_1, \sigma_2 \in \Sigma', \tilde{\sigma} \in \tilde{\Sigma} \right\}\right)
\]

\[
= \big\{ \bigotimes_{\sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2, \tilde{\sigma} \in \tilde{\Sigma}} (\sigma_1 \otimes \sigma_2 \otimes \tilde{\sigma}) \mid \Sigma_1, \Sigma_2 \subseteq \Sigma' \wedge \Sigma \subseteq \Sigma \wedge \Sigma_1, \Sigma_2, \tilde{\Sigma} \neq \emptyset \big\}
\]

\[
= \big\{ \bigotimes_{\sigma \in \Sigma} (\sigma \otimes \tilde{\sigma}), \Sigma \subseteq \Sigma' \wedge \Sigma \subseteq \Sigma \wedge \Sigma, \tilde{\Sigma} \neq \emptyset \big\}
\]

\[
= \text{combine-cs}\left(\big\{ (\sigma \otimes \tilde{\sigma}) \mid \sigma \in \Sigma', \tilde{\sigma} \in \tilde{\Sigma} \right\}\right)
\]

\[
= \text{combine-cs}(\Sigma' \otimes \tilde{\Sigma})
\]

for an arbitrary c-collection \(\tilde{\Sigma}\) the statement above further simplifies to

\[
\text{combine-cs}(SCE\text{-}\text{list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r)) =
\]

\[
= \text{combine-cs}(\Sigma' \otimes \bigotimes_{j=1}^{n} S\text{-}CE(t_j, t'_j, \emptyset, r)).
\]

We prove this by showing

\[
SCE\text{-}\text{list-reduce}((t_1, t'_1), \ldots, (t_n, t'_n)), \Sigma', r) = \Sigma' \otimes \bigotimes_{j=1}^{n} S\text{-}CE(t_j, t'_j, \emptyset, r). \quad (B.3)
\]

by induction on the number \(n\) of list elements:

\(n = 1:\)

\[
SCE\text{-}\text{list-reduce}((t_1, t'_1)), \Sigma', r) = SCE\text{-}\text{list-reduce}((), \bigcup_{\sigma \in \Sigma'} S\text{-}CE(t_1, t'_1, \sigma, r), r) =
\]

\[
= \bigcup_{\sigma \in \Sigma'} S\text{-}CE(t_1, t'_1, \sigma, r) = \Sigma' \otimes S\text{-}CE(t_1, t'_1, \emptyset, r)
\]

234
with the last step because of (B.1) proven above.

**n → n + 1:** Let (B.3) hold for all calls to \(SCE\text{-}\text{list-reduce}\) with a list of length \(n\) as first argument. Consider a call

\[
SCE\text{-}\text{list-reduce}((t_1,t'_1), (t_1,t'_1), \ldots, (t_1,t'_1)), \Sigma, r) = SCE\text{-}\text{list-reduce}(((t_1,t'_1), \ldots, (t_1,t'_1)), \bigcup_{\sigma \in \Sigma} S\text{-}CE(t_1,t'_1, \sigma, r), r)
\]

By applying the induction hypothesis this is equal to

\[
\bigcup_{\sigma \in \Sigma} S\text{-}CE(t_1,t'_1, \sigma, r) \otimes \prod_{j=1}^{n} S\text{-}CE(\bar{t}_j, \bar{t}'_j, \emptyset, r)
\]

Applying (B.1) we get

\[
\Sigma \otimes S\text{-}CE(t_1,t'_1, \emptyset, r) \otimes \prod_{j=1}^{n} S\text{-}CE(\bar{t}_j, \bar{t}'_j, \emptyset, r)
\]

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. \(\square\)

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

\[\forall k \in \mathbb{N}. v \in \text{subst}(\tilde{\sigma})^k(t_1) \sqcup \text{subst}(\tilde{\sigma})^k(t_2). \tag{4.2}\]

Then there exist a free constraint set

\[\sigma \in \text{combine-cs-cond}(S\text{-}CE(t_1, t_2, \tilde{\sigma}, r) \otimes \text{implicit-cs}(c))\]

such that the free type substitution \(\sigma' = \text{subst}(\sigma)\) compatible with \(\sigma\) fulfills

\[\forall k \in \mathbb{N}. v \in \sigma'^k(t_1) \sqcup \sigma'^k(t_2). \tag{4.3}\]

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

\[\langle \bar{t}_X \rangle(\tau) \subseteq \langle t_X \rangle(\tau) \tag{4.4}\]

for every closed type substitution \(\tau\) appropriate for \(\bar{t}_X\) and \(t_X\).
Proof of Lemma 4.2.20: 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

$$\Sigma \subseteq \text{combine-cs-cond}(\Sigma) \subseteq \text{combine-cs}(\Sigma)$$

for every c-collection $\Sigma$. In the following we will use $\Sigma$ instead of $\text{combine-cs-cond}(\Sigma)$ 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 $v \in \langle \rho(t_1) \rangle \cap \langle \rho(t_2) \rangle$ for some $\rho$ such that no $\sigma \in S-CE(t_1, t_2, \tilde{\sigma}, r)$ fulfills

$$\forall k \in \mathbb{N} \exists \tau . v \in \{ \tau \circ \sigma^k(t_1) \} \cap \{ \tau \circ \sigma^k(t_2) \} .$$

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 $\tau$ we have $v_p \not\in \langle \tau \circ \sigma^k(A) \rangle$).

Informally $A$ is instantiated by $\sigma'$ 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 4.2.16 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 $\bar{\sigma}^k(t_1)$ and $\bar{\sigma}^k(t_2)$ 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 $v \in \bar{\sigma}^k(t_1) \square \bar{\sigma}^k(t_2)$ exists there are $\rho_f$ and $\rho_q$ such that

$$v \in \{\rho_q \circ \rho_f \circ \bar{\sigma}^k(t_1)\} \cap \{\rho_q \circ \rho_f \circ \bar{\sigma}^k(t_2)\}.$$ 

For every $\rho'_q$ with $\text{dom}(\rho'_q) = \text{dom}(\rho_q)$ and $\rho'_q(X_\forall) \neq \bot$ for every $X_\forall \in \text{dom}(\rho'_q)$ there exists a $v'$ with

$$v' \in \{\rho'_q \circ \rho_f \circ \bar{\sigma}^k(t_1)\} \cap \{\rho'_q \circ \rho_f \circ \bar{\sigma}^k(t_2)\}.$$ 

We can choose a $v'$ that differs from $v$ at exactly those positions where either $\bar{\sigma}^k(t_1)$ or $\bar{\sigma}^k(t_2)$ 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 $\bar{\sigma}^k(t_1)$ and $\bar{\sigma}^k(t_2)$ 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 $\tau'_q$ such that

$$v' \in \{\tau'_q \circ \tau_f \circ \sigma'^k(t_1)\} \cap \{\tau'_q \circ \tau_f \circ \sigma'^k(t_2)\}$$

whenever

$$v \in \{\tau_q \circ \tau_f \circ \sigma'^k(t_1)\} \cap \{\tau_q \circ \tau_f \circ \sigma'^k(t_2)\}. \quad (B.4)$$

Thus, in the following we will just prove (B.4) for the individual rules.

The substitutions used for $\square$ are denoted $\rho_q$ for the quantified and $\rho_f$ for the free variables in (4.2) and analogously $\tau_q$ and $\tau_f$ in (4.3).

- Let $t_1$, $t_2$ and $\bar{\sigma}$ be processed by $(\text{Top}1)$.\footnote{In order to simplify notations we identify subst($\bar{\sigma}$) and $\bar{\sigma}$ in the following.} Let $v$ be chosen as in (4.2). Because of $t_1 = \top$ from the condition of $(\text{Top}1)$ $v \in \{\tau \circ \sigma'^k(t_1)\}$ for all $\tau$ and $\sigma'$. Since $\sigma' = \bar{\sigma}$ we can choose $\tau = \rho$ according to the choice of $v$ and $k$ and trivially get statement (4.3) of the lemma. (4.4) holds trivially because of $\sigma' = \bar{\sigma}$. For $(\text{Top}2)$ the proof is done analogously.
If \( t_1, t_2 \) and \( \bar{\sigma} \) are processed by rule \((\text{FunQ})\) (i.e. \( t_1 \) and \( t_2 \) are function types or quantified variables) or \((\text{Base})\) (i.e. \( t_1 \) and \( t_2 \) are base types or value assignments) then (4.3) is fulfilled because \( t_1 \) and \( t_2 \) are not affected by \( \sigma' \) and \( \bar{\sigma} \) and we can choose \( \tau = \rho \). (Since \( \sigma' \) and \( \bar{\sigma} \) are generated from free constraint sets they just affect free variables.) (4.4) again holds trivially because of \( \sigma' = \bar{\sigma} \).

Let the current call be processed by rule \((\text{BothVar})\). 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 \( SCE \) and we can choose \( \tau \) to instantiate it according to the value \( v \). This yields (4.3). (4.4) holds because of the definition of \( \text{extend-constraint} \): Whenever one of the variables is already constrained the newly introduced variable and the previous value are united.

Let \((\text{Var1})\) be the rule processing the current call to \( SCE \). Then \( t_1 \) is a free variable and \( t_2 \) is not a free variable. Because of \( v \in \langle \rho \circ \bar{\sigma}^k(t_2) \rangle \) in (4.2) we have \( v \in \langle \bar{\rho} \circ \rho \circ \sigma'(t_2) \rangle \) with \( \bar{\rho} \) instantiating the variables introduced by \( \text{constrain-all-free} \). (All free variables occurring in \( t_2 \) are instantiated with a newly introduced variable (maybe in a union) by \( \sigma' \). These variables are not instantiated by further calls of \( \sigma' \) because they do not occur as subtype of one of the checked types. When assuming that \( \rho \) 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 \( v \in \langle \bar{\rho} \circ \rho \circ \sigma'^{k+1}(t_1) \rangle \) because of the change to the constraint of \( t_1 \) performed by \( \text{extend-constraint} \). We can now choose \( \tau_2 = \bar{\rho} \circ \rho \circ \sigma' \) 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 \( \tau_1 \) and \( \tau_2 \) in both cases and all type constructors that can contain variables in our type language are monotonic we can just choose \( \tau = \{ A \leftarrow T \mid A \in \text{dom}(\tau_1) \cup \text{dom}(\tau_2) \} \).

This yields (4.3). The preservation of previous constraints as stated by (4.4) is obvious here because:

- If there are previous constraints for \( t_1 \) they are preserved because of the union calculated by \( \text{extend-constraint} \) in this case.
- The free variables occurring in \( t_2 \) are constrained by \( \text{constrain-all-free} \). This function preserves previous constraints by calculating unions in \( \text{extend-constraints} \) when necessary.

The correctness of \((\text{Var}2)\) is proven analogously.

If the current call is processed by rule \((\text{Comp1})\) or \((\text{Comp2})\) then the resulting substitution \( \sigma' \) calculated by \( \text{free-to-top} \) differs from \( \bar{\sigma} \) in instantiating some additional variables. All these variables are instantiated to \( \top \). By choosing \( \tau = \rho \) we get \( \langle \rho \circ \bar{\sigma}^k(A) \rangle \subseteq \langle \rho \circ \sigma^k(A) \rangle \) for every variable \( A \) and since all type constructors containing variables in their arguments are monotonic (4.3) of the lemma follows. (4.4) holds because for every type \( t \) and every substitution \( \tau \) we have \( \langle \tau(t) \rangle \subseteq \langle \top \rangle \).
• Let \((U1)\) be the rule processing the current call. Then \(t_1 = (\cup t_{1,1} \ldots t_{i,k})\). Let
\[
v \in \{\rho \circ \text{subst}(\tilde{\sigma})^k(t_1)\} \cap \{\rho \circ \text{subst}(\tilde{\sigma})^k(t_2)\}
\]
for some arbitrary \(k\) and an appropriate \(\rho\). Because of \(v \in \{\rho \circ \text{subst}(\tilde{\sigma})^k(t_1)\}\) there
must be a \(t_{1,i}\) with \(v \in \{\rho \circ \text{subst}(\tilde{\sigma})^k(t_{1,i})\}\) and therefore
\[
v \in \{\rho \circ \text{subst}(\tilde{\sigma})^k(t_{1,i})\} \cap \{\rho \circ \text{subst}(\tilde{\sigma})^k(t_2)\}.
\]
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 \(\rho \in S-CE(t_{1,i}, t_2, \tilde{\sigma}, r)\) with \(\sigma' = \text{subst}(\sigma)\) such that there exists a \(\tau\) with
\[
v \in \{\tau \circ \sigma'^k(t_{1,i})\} \cap \{\tau \circ \sigma'^k(t_2)\}.
\]
By rule \((U1)\) this \(\sigma\) is also a member of the result of \(S-CE(t_{1,i}, t_2, \tilde{\sigma}, r)\). \((4.3)\) holds e.g.
with \(\tau\) chosen as above. \((4.4)\) follows trivially from the induction stating \((4.4)\) for the
subcalls. The correctness of \((U2)\) is proven analogously.

• Let the current call be processed by rule \((II)\), i.e. \(t_1 = (\cap t_{1,1} \ldots t_{1,n})\) and consider a
value \(v\) chosen as in \((4.2)\). \(SCE\text{-}\text{list-reduce}\) called by rule \((II)\) performs a sequence of sub-
calls to \(S-CE\) with the types \((t_{1,1}, t_2), (t_{1,2}, t_2), \ldots, (t_{1,n}, t_2)\) as first two arguments and
free constraint sets that steam from the output of the call before. We will show that the
collection \(\Sigma_i\) that is used in the recursive subcall \(SCE\text{-}\text{list-reduce}(l_{i+1}, \ldots, l_n, \Sigma_i, r)\)
with \(l_j = (t_{1,j}, t_2)\) fulfills:
\[
\exists \sigma \in \Sigma_i \forall k \in \mathbb{N} \exists \tau \cdot v \in \{\tau \circ \sigma^k(t_{1,i})\} \cap \{\tau \circ \sigma^k(t_2)\} \land
\land v \in \{\tau \circ \sigma^k(t_{1,2})\} \cap \{\tau \circ \sigma^k(t_2)\} \land
\vdots
\land\land v \in \{\tau \circ \sigma^k(t_{1,i})\} \cap \{\tau \circ \sigma^k(t_2)\}\]
\[(B.5)\]
For the last conjunction element this follows directly from part \((4.3)\) of the induction
hypothesis applied to the direct \(S-CE\) calls performed by the call to \(SCE\text{-}\text{list-reduce}\)
processing \(l_i\).

For the other conjunction elements (making statements about \(l_j\) with \(j < i\)) we can
apply \((4.3)\) of the induction hypothesis to the calls calculating \(\Sigma_j\) (defined analogously
to \(\Sigma_i\)). All the transformations from \(\Sigma_j\) to \(\Sigma_i\) preserve the needed properties according
to \((4.4)\) of the induction hypothesis.

Since the statement \((B.5)\) especially holds for \(i = n\) rule \((II)\) fulfills \((4.3)\) because from
\(v \in \{\tau \circ \sigma^k(t_{1,i})\}\) for all \(i\) we can conclude \(v \in \{\tau \circ \sigma^k(t_1)\}\). \((4.4)\) directly follows from
the fact that all subcalls to \(S-CE\) performed by \(SCE\text{-}\text{list-reduce}\) fulfill \((4.4)\) because of
the induction hypothesis. The proof of \((II)\) can be done analogously.
• If \((\text{Rec1})\) is the rule processing the current call then \(t_1 = \mu X.t'\) and the recursive call 
\(S-CE(\text{unfold}(t_1), t_2, \hat{\sigma}, ((t_1, t_2) \cdot r))\) fulfills (4.3) and (4.4) by the induction hypothesis. 
Because of \(\langle t_1 \rangle(\sigma) = \langle \text{unfold}(t_1) \rangle(\sigma)\) (see Def. 3.1.49) the lemma also holds for the 
result of \((\text{Rec1})\). For \((\text{Rec2})\) the proof is done analogously. 
If the current call is processed by rule \((\text{RecT})\) t h e n\((t_1, t_2) \in r\), 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 
\(SCE\text{-list-reduce}\) was processed and the necessary constraints will be generated dur-
ing processing the rest of this call to \(SCE\text{-list-reduce}\). In this case the necessary 
constraints are already contained in the implicit constraint set of the current call 
as stated by Lemma 4.2.19. 

There is just one problem left: When calling \((\text{RecT})\) the necessary constraint for a 
variable might be splitted into subconstraints occurring in different free constraint sets 
in the same c-collection. Since \((\text{RecT})\) can only occur as a (maybe indirect) subcall of a 
call to \((\text{Rec1})\) or \((\text{Rec2})\) the complete constraint is generated by \(combine\text{-cs-cond}\) that 
behaves like \(combine\text{-cs}\) in this context. 

• Let the current call be processed by rule \((\text{Constr})\). Then \(t_1 = (c \ t_{1,1} \ldots t_{1,n})\) and 
\(t_2 = (c \ t_{2,1} \ldots t_{2,n})\). Via \(SCE\text{-list-reduce}\) 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 \(\Sigma_i\) used in 
the call \(SCE\text{-list-reduce}(l_{i+1}, \ldots , l_n, \Sigma_i, r)\) contains a \(\sigma\) with: 
\[
\forall k \in \mathbb{N}\exists \tau. \ u_1 \in \{\tau \circ \sigma^k(t_{1,1})\} \cap \{\tau \circ \sigma^k(t_{2,1})\} \wedge 
\wedge u_2 \in \{\tau \circ \sigma^k(t_{1,2})\} \cap \{\tau \circ \sigma^k(t_{2,2})\} \wedge 
\vdots 
\wedge u_i \in \{\tau \circ \sigma^k(t_{1,i})\} \cap \{\tau \circ \sigma^k(t_{2,i})\} 
\]

Since this is especially the case for \(i = n\) rule \((\text{Constr})\) fulfills (4.3). (4.4) is proven 
analogously to rule \((I1)\). The rules \((\text{Frame})\) and \((\text{Env})\) 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 (4.3). We have to show 
\footnote{When finishing a call to Rule \((\text{RecT})\) the recursive parent calls must also apply \(combine\text{-cs}\) to their result 
up to that ancestor processed by Rule \((\text{Rec1})\) or \((\text{Rec2})\) that is finished first. Since all these calls are subcalls 
of \((\text{Rec1})\) or \((\text{Rec2})\) \(combine\text{-cs-cond}\) indeed behaves like \(combine\text{-cs}\). 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 (4.2) 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 \((\text{FunQ})\) 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 \( \rho_q' \) 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, interaction types are covered by either \((I1)\) or \((I2)\).

- If one of the types is a complement type \( t = Ct' \) then the other type \( s \) must not be a subtype of \( t' \) in order to have common elements. Because of Lemma 3.4.8 \( \langle s \rangle \not\subseteq \langle t' \rangle \) implies \( s \not\subseteq t' \). (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 \( \sigma' \). Thus, one of the rules \((\text{Comp1})\) and \((\text{Comp2})\) 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 \((\text{Constr})\) 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 \((\text{Frame})\) and \((\text{Env})\), 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 \textbf{\texttt{Tfunc}}. In all these cases the function \( fq \) returns \texttt{true} and rule \((\text{FunQ})\) 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 \( CE\text{base}(t_1, t_2) = \texttt{true} \). In this case rule \((\text{Base})\) applies.
By this case distinction all cases of two types with common elements are covered by one of the rules of $S$-$CE$.\textsuperscript{3} Therefore, returning the empty $c$-collection in case of no applicable rule is correct.

Part 1 and Part 2 together prove the lemma. \hfill $\square$

\section*{B.2 Proofs of Section 4.3}

\textbf{Lemma 4.3.9} The algorithm SMR terminates for every input substitution $\sigma$ with a finite domain $\text{dom}(\sigma)$.

\textbf{Proof of Lemma 4.3.9:} Let $k := \vert \text{dom}(\sigma) \vert \in \mathbb{N}$. 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. \hfill $\square$

\textbf{Lemma 4.3.10} Let $\sigma'$ be a substitution such that the graph $G = (V, R)$ defined as in GIS with $V = \text{dom}(\sigma')$ 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 $\sigma = \text{SMR}(\sigma')$. Then

$$\langle [\sigma \circ \sigma'(t)] \phi \rangle = \langle [\sigma(t)] \phi \rangle$$

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

\textbf{Proof of Lemma 4.3.10:} For every variable $X \in \text{dom}(\sigma')$ we show $\langle [\sigma \circ \sigma'(X)] \phi \rangle = \langle [\sigma(X)] \phi \rangle$ for every appropriate closed type substitution $\phi$. Because of $\text{dom}(\sigma) = \text{dom}(\sigma')$ this implies $\langle [\sigma \circ \sigma'(t)] \phi \rangle = \langle [\sigma(t)] \phi \rangle$ 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 $\sigma'$.

Let $Y \in \text{dom}(\sigma')$ 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 $\sigma'$ contains exactly one variable $V \in \text{dom}(\sigma')$.

Because of $\sigma \circ \sigma'(X) = \sigma(\sigma'(X)) = \sigma(C[Y]) = C[\sigma(Y)]\textsuperscript{4}$ we have to show $\langle [\sigma(X)] \phi \rangle = \langle [C[\sigma(Y)]] \phi \rangle$. We do this by case distinction on the different possible orderings between $X$ and $Y$:

\textsuperscript{3}The only type that was not inspected in the case distinction is $\bot$ that has no common elements with any type except of the non-termination that is not of interest for the lemma.

\textsuperscript{4}The last step holds because we assume that there are no other variables $V \in \text{dom}(\sigma)$ occurring in $C[Y]$.  

243
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

\[
\begin{align*}
\sigma'(Y) &= C_0[V_1] \\
\sigma'(V_1) &= C_1[V_2] \\
& \vdots \\
\sigma'(V_m) &= C_m[V_{m+1}]
\end{align*}
\]

and let \( V_l <_V X \) for \( l \leq m \) and \( V_{m+1} \geq X \). We prove \( \langle \sigma(X) \rangle(\phi) = \langle C[\sigma(Y)] \rangle(\phi) \) by induction on \( m \).

- \( m = 0 \): When processing \( Y \) in the first loop on \( i \) the assignment to \( X \) in \( \sigma' \) is updated to \( C[C_0[V_1]] \). The processing of \( V_1 \) either yields \( C[\sigma(Y)] = \text{unfold}(\sigma(X)) \) according to (1) for \( V_1 = X \) or \( C[\sigma(Y)] = \sigma(X) \) according to (2) for \( V_1 >_V X \). In both cases \( \langle \sigma(X) \rangle(\phi) = \langle C[\sigma(Y)] \rangle(\phi) \).

- \( m \to m + 1 \): Let the statement hold for \( m \). Assume that

\[
\begin{align*}
\sigma'(Y) &= C_0[V_1] \\
\sigma'(V_1) &= C_1[V_2] \\
& \vdots \\
\sigma'(V_m) &= C_m[V_{m+1}] \\
\sigma'(V_{m+1}) &= C_{m+1}[V_{(m+1)+1}]
\end{align*}
\]

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 \( \sigma' \) 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 \( V_l \) 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 \( \langle \sigma(X) \rangle(\phi) = \langle C[\sigma(Y)] \rangle(\phi) \) for the case that every right hand side of \( \sigma' \) contains exactly one variable \( V \in \text{dom}(\sigma') \).

If \( \sigma'(X) \) contains several \( Y_k \) they can be considered independently in ascending order according to \( <_V \).
If \( \sigma'(Y) \) contains several variables \( Z_k \in \text{dom}(\sigma') \) 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. \( \square \)

**Lemma 4.3.11** The algorithm GIS terminates for every input substitution with a finite domain \( \text{dom}(\sigma) \).

**Proof of Lemma 4.3.11:** By the precondition of the lemma \( |V| = |\text{dom}(\sigma)| \) 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 \( \text{dom}(\sigma') \subseteq \text{dom}(\sigma') \) and therefore \( |\text{dom}(\sigma')| \) is finite. Thus, the call \( \text{SMR}(\sigma') \) terminates by Lemma 4.3.9. \( \square \)

**Lemma 4.3.12** Let \( \sigma' \) be a substitution fulfilling with the following properties:

1. \( \sigma' \) does not contain a variable binding \( A \leftarrow B \) with \( B \in \text{dom}(\sigma') \).

2. If \( \sigma' \) contains a variable binding \( A \leftarrow C[B] \) with a context \( C \) and a variable \( B \in \text{dom}(\sigma') \) then there exists a variable \( B' \notin \text{dom}(\sigma') \) such that \( B \) is bound to \( B' \) or a union containing \( B' \) in \( \sigma' \).

Let \( v \) be a value fulfilling

\[
\forall k \in \mathbb{N} \cdot v \in \sigma'^k(t_1) \square \sigma'^k(t_2)
\]

and let \( \sigma = \text{GIS}(\sigma') \). Then

\[
v \in \sigma(t_1) \square \sigma(t_2).
\]

**Proof of Lemma 4.3.12:** Let \( t_1 \) and \( t_2 \) be types and \( v \) a value fulfilling the precondition of the lemma. We can conclude

\[
\forall k \in \mathbb{N} \exists \tau. v \in (\tau \circ \sigma'^k(t_1)) \cap (\tau \circ \sigma'^k(t_2)) \tag{B.6}
\]

The substitution \( \tau \) transforms the given argument terms to closed terms. Thus, the terms assigned to all variables by \( \tau \) are closed terms. We can divide \( \tau \) into two substitutions \( \tau' \) and
for every closed type substitution $\phi$ and $\text{dom}(\rho) = \text{dom}(\sigma') = \text{dom}(\sigma)$ where $\sigma$ is the substitution generated from $\sigma'$ 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 $\{\tau' \circ \rho \circ \sigma^{k'}(t_1)\}$ and $\{\tau' \circ \rho \circ \sigma^{k'}(t_2)\}$ independently of $\rho$ (with $\tau'$ 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 $\sigma'$ given in the lemma enforce that whenever a variable $A$ is instantiated by $\sigma'$ the instance is either $A'$ or a union containing $A'$ with $A' \notin \text{dom}(\sigma)$. By choosing $\tau'(A')$ appropriately every additional value introduced into such a chain of unions can also be introduced by $\tau'$.

Variables occurring in intersections in $\sigma^{k'}$ are obviously no problem when choosing a $\rho$ that does not introduce any restrictions not introduced by $\sigma'$. This is especially the case for all $\rho$ fulfilling

$$\forall X \in \text{dom}(\langle x \rangle \sigma'). \langle \rho(X) \rangle(\phi) = \langle \rho \circ \sigma'(X) \rangle(\phi) \quad (B.7)$$

for every closed type substitution $\phi$ appropriate for $\rho(X)$ and $\rho \circ \sigma'(X)$.

Altogether we can now formulate the following statement:

$$\forall v \in V \exists k' \in \mathbb{N} \forall \rho \text{ fulfilling (B.7). } v \in \{\tau' \circ \rho \circ \sigma^{k'}(t_1)\} \cap \{\tau' \circ \rho \circ \sigma^{k'}(t_2)\} \quad (B.8)$$

Especially (B.7) is fulfilled for $\rho = \sigma := GIS(\sigma')$. We show this by arbitrarily choosing a fixed $X \in \text{dom}(\sigma')$ and distinguishing several cases on $\sigma'(X)$:

**Case 1:** $\sigma'(X)$ does not contain any variables $Y \in \text{dom}(\sigma')$. 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, $\sigma$ contains the assignment $X \leftarrow t$ and hence $\sigma(X) = \sigma'(X) = \sigma \circ \sigma'(X)$.

**Case 2:** $\sigma'(X)$ contains variables from $\text{dom}(\sigma')$ 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 $Y \in \text{dom}(\sigma')$ 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 $Y \in dom(\sigma'')$.

In the latter case $\sigma''(Y) = \sigma(Y)$ because the binding of $Y$ is not changed after being inserted into $\sigma''$ 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 $\sigma''$ 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 $\sigma''$ affects the value of an already defined variable and therefore

$$\forall X \in dom(\sigma''). \sigma(X) = \sigma''(X).$$

Since $dom(\sigma'') \subseteq dom(\sigma)$ 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 $\sigma_r$ as defined by $GIS$. By definition of $\sigma_r$ the type $\sigma_r(X)$ just contains variables $Y \in dom(\sigma_r)$. Using this information we have

$$\sigma \circ \sigma_r(X) = \sigma|_{dom(\sigma_r)} \circ \sigma_r(X) = \tilde{\sigma} \circ \sigma_r(X)$$

Since $\tilde{\sigma} = SMR(\sigma_r)$, Lemma 4.3.10 implies

$$\tilde{\sigma} \circ \sigma_r(X) = \tilde{\sigma}(X).$$

Since the assignment $X \leftarrow \tilde{\sigma}(X)$ is added to $\sigma''$ 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

$$\forall i \in \{0, 1, \ldots, k_v\}. v \in \{\tau' \circ \sigma \circ \sigma'^{-i}(t_1)\} \cap \{\tau' \circ \sigma \circ \sigma'^{-i}(t_2)\} \tag{B.9}$$

by induction on $i$:

**i = 0:** By choosing $\rho = \sigma$ we get the statement directly from (B.8).

**i → i + 1:** From the induction hypothesis we can conclude

$$v \in \{\tau' \circ \sigma \circ \sigma' \circ \sigma'^{-i+1}(t_1)\} \cap \{\tau' \circ \sigma \circ \sigma' \circ \sigma'^{-i+1}(t_2)\}. \tag{B.9}$$

\(^5\) $X$ 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 $t$.  

247
To get the intended statement
\[ v \in \langle \tau' \circ \sigma \circ \sigma'^{-k_v-(i+1)}(t_1) \rangle \cap \langle \tau' \circ \sigma \circ \sigma'^{-k_v-(i+1)}(t_2) \rangle . \]
we just have to show \( \langle \sigma \circ \sigma'(t) \rangle(\phi) = \langle \sigma(t) \rangle(\phi) \) for every type term \( t \) and every appropriate closed type substitution \( \phi \). In showing this we can restrict ourselves to the statement \( \langle \sigma \circ \sigma'(X) \rangle(\phi) = \langle \sigma(X) \rangle(\phi) \) for \( X \in \text{dom}(\sigma') \). This statement has already been proven above.

In (B.9) we can now choose \( i = k_v \) and can conclude
\[ \exists \tau . v \in \langle \tau \circ \sigma(t_1) \rangle \cap \langle \tau \circ \sigma(t_2) \rangle . \]  
(B.10)

Besides (B.6), the precondition of the lemma implies that when dividing \( \tau \) into \( \tau_f \) for the free and \( \tau_q \) for the quantified variables the following holds: We can replace \( \tau_q \) by every \( \tau_q' \) with the same domain that does not assign \( \bot \) 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 \( \tau \) the proof of (B.10) goes through without a change. This is the case for every \( v' \) and especially every \( \tau_q' \) chosen as above because
\[ \forall k \in \mathbb{N} . v \in \sigma'^k(t_1) \sqcap \sigma'^k(t_2) \]
implies the existence of an appropriate \( v' \) for every \( \tau_q' \).

Altogether we have proven \( v \in \sigma(t_1) \sqcap \sigma(t_2) \).

**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 of Lemma 4.3.13:** 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 \( \sigma' \in \Sigma' \mid \text{dom}(\sigma') \) is finite. Since GIS terminates by Lemma 4.3.11 the termination of CE under the preconditions given in the lemma is proven.

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

**Proof of Theorem 4.3.14:** The proof of lemma 4.3.13 carries over except of executions of the rules (Comp1) and (Comp2) in S-CE. This case might cause a recursive chain of the form
\[ S-CE \rightarrow ST \rightarrow CE \rightarrow S-CE \]
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 $C\tilde{t}_1$. 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 $\mu$ 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 $C$ 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 4.3.15 The algorithm $ST$ terminates for every pair of semi-closed input types in set normalized form.

Proof of Corollary 4.3.15: According to 3.4.7 $ST$ terminates as long as $CE$ is a terminating function. Since $CE$ is terminating due to Theorem 4.3.14 $ST$ terminates for every input.

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

Proof of Corollary 4.3.16: Lemma 4.2.16 gives the statement of the corollary under the condition that $ST$ terminates. Since $ST$ terminates for every input due to Cor. 4.3.15 $S-CE$ terminates for every input.

Theorem 4.3.17 Let $t_1,t_2 \in T$, both in set normalized form. Let there exist a value $v \neq \bot$ such that

$$v \in t_1 \sqcup t_2.$$ 

Then there exist substitutions $\sigma \in CE(t_1,t_2)$ such that

$$v \in \sigma(t_1) \sqcup \sigma(t_2).$$
**Proof of Theorem 4.3.17:** 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 $\Sigma'$ we can apply Lemma 4.2.20. (For applying the lemma we need Assumption 3.4.6. Due to Theorem 4.3.14 $CE$ 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 4.2.20 now gives us:

$$\exists \sigma' \in \Sigma' \forall k \in \mathbb{N}. v \in subst(\sigma')^k(t_1) \sqsupseteq subst(\sigma')^k(t_2) \quad (B.11)$$

(Note that $\tilde{\sigma}$ from Lemma 4.2.20 is the empty substitution in the initial call and is therefore omitted here.) In the following we will identify $\sigma'$ with $subst(\sigma')$.

Because of the definition of $\Sigma$ the substitution $\sigma = GIS(\sigma')$ fulfills $\sigma \in \Sigma$. Due to Lemma 4.3.12 it furthermore fulfills

$$v \in \sigma(t_1) \sqsupseteq \sigma(t_2).$$

This proves the theorem. □
Appendix C

Proofs of Chapter 6

C.1 Proofs of Section 6.2

Lemma 6.2.26 Let $E$ be an environment and $x$ a symbol. Then

$$E(x) = (\cup \{E'(x) \mid E' \in S(E)\}).$$

Proof of lemma 6.2.26: Let $E = (L, \Sigma)$ with $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$. Let $L_i = CAE(L, \sigma_i)$ and $E_i^S = (L_i, \{\sigma_i\})$. We have to distinguish three cases:

1. $x$ is not bound in the frame list $L$. Then $E(x) = \text{undef}$. Since $CAE$ does not introduce new bindings in its argument frame list the symbol $x$ is unbound in $L_i$. We have $L_i(x) = \text{undef}$, therefore $GIS(\sigma_i)(L_i(x)) = \text{undef}$ and $E_i^S(x) = \text{undef}$. Since this holds for all $i$ we have $(\cup \{E'(x) \mid E' \in S(E)\}) = \text{undef}$.

2. $x$ is bound to a non-variable type term $t \neq \text{undef}$ in $L$. Then $E(x) = t$. $t$ is not changed by applying any of the $GIS(\sigma_i)$ to $L$. Since $t \neq \text{undef}$ the binding of $x$ in $L_i$ is not removed by $CAE$ for $\sigma_i$ and further bindings of $x$ in more special frames are not introduced. Therefore, $(\cup \{E'(x) \mid E' \in S(E)\}) = (\cup \{E_i^S(x) \mid i = 1, \ldots, k\}) = (\cup t) = t$.

3. In the most special frame $F_j$ of $L$ with $F_j(x) \neq \text{undef}$ the symbol $x$ is bound to a type variable $A$. There are two subcases on the value of $A$ in $\sigma_i$:
   - If $GIS(\sigma_i(A)) = t'_i \neq \text{undef}$ then applying $E$ to $x$ under the index $i$ yields $E_i(x) = t'_i$. Because of $t'_i \neq \text{undef}$ the binding of $x$ to $A$ is not removed by $CAE(L, \sigma_i)$. 

New bindings for $x$ in frames more special than $F_j$ are not introduced by $CAE$ and therefore $E_i^S(x) = t'_i$.

- If $GIS(\sigma_i(A)) = \text{undef}$ then the value $\tilde{t}_i := E_i(x)$ of $x$ in $E$ under the index $i$ depends on bindings of $x$ in frames more general the $F_j$. Instead of $F_j$ the environment $L_i$ contains a frame $F_{j,i}$ with (among other changes) the binding of $x$ removed. Thus, the value $E_i^S(x)$ in the simple abstract environment $E_i^S$ corresponding to the substitution index $i$ depends on the bindings of $x$ in more general frames of $L_i$. The statement $E_i^S(x) = \tilde{t}_i$ is given by the same case distinction on the binding of $x$. Induction on the maximal number of frames in $E$ yields the statement $E_i(x) = E_i^S(x)$.

Together, these two subcases state $E(x) = (\cup t_1 t_2 \ldots t_k) = (\cup \{E'(x) \mid E' \in S(E)\})$ where $t_i$ either equals $t'_i$ or $\tilde{t}_i$ depending on the case appropriate for $\sigma_i$.

The three cases together prove the lemma. $\square$

**Lemma 6.2.32** For every abstract environment $E \in Env_A$ the result $E' := \text{canonical}(E)$ is a canonical abstract environment.

**Proof of lemma 6.2.32:** The condition for an abstract environment $E = (L, \Sigma)$ to be canonical is a condition on the single variables bound in the substitutions of $\Sigma$. The algorithm checks every variable $X \in Vars$ i.e. every variable that is defined in any of the substitutions whether it violates this condition. This is the case for every variable that is instantiated with the same abstract value in every substitution. If such a variable is found it is removed from all substitutions and replaced by its common instance in the frame list.

There are no new variables introduced into any of the substitutions and no variable instantiation is changed in one of the substitutions. Thus, every variable violating the condition for canonical abstract environments in $E'$ must have violated this condition in $E$ already. Since all these variables have been eliminated the condition for canonical abstract environments is fulfilled by $E'$.

Since the call to $FS$-env-update in the last line does not change the substitutions it cannot violate the property of the result being a canonical environment. $\square$

**Lemma 6.2.34** The function combine is commutative, i.e. for two abstract environments $E_1, E_2 \in Env_A$ we have:

\[ \text{combine}(E_1, E_2) = \text{combine}(E_2, E_1). \]

**Proof of lemma 6.2.34:** If $E_1$ and $E_2$ have a different number of frames there is nothing to show because the result is $\text{undef}$ in both cases.

252
Otherwise, the algorithm processes $E_1$ and $E_2$ in a completely similar manner except of the last line where the transformed frame list of $E_1$ is returned. Thus, we can prove the lemma by showing that after processing the for-loops the frame lists of $E_1$ and $E_2$ are equal. We prove this by induction on the number $k$ of frames in $E_1$ and $E_2$:

$k = 1$: We show that $F_{1,1}$ and $F_{2,1}$ are equal after being processed in the outermost for loop by induction on the number $n$ of symbols $s \in \text{Symbol}$.

$n = 0$: Both $F_{1,1}$ and $F_{2,1}$ are empty and therefore equal.

$n \to n+1$: Let $\text{Symbols} = \{s_1, \ldots, s_n, s_{n+1}\}$ numbered in the order they are processed by the innermost for-loop and let $F_{1,1}' = F_{1,1}|_{\{s_1,\ldots,s_n\}}$, $F_{2,1}' = F_{2,1}|_{\{s_1,\ldots,s_n\}}$. From the induction hypothesis on $n$ we can conclude that $F_{1,1}'$ and $F_{2,1}'$ are equal after being processed by the innermost for-loop. Since processing of one symbol does not change the bindings of any other symbol we just have to show $F_{1,1}(s_{n+1}) = F_{2,1}(s_{n+1})$ after executing the innermost for-loop with $s = s_{n+1}$.

1. Because of the first two if-statement $s_{n+1}$ is bound in both $F_{1,1}$ and $F_{2,1}$ at the end of the loop, probably to $\text{undef}$.

2. Because of the third if-statement $F_{1,1}(s_{n+1}) = F_{2,1}(s_{n+1})$.

Altogether, $F_{1,1} = F_{2,1}$ after processing the outermost for-loop.

$k \to k+1$: Let $E_1 = (L_1, \Sigma_1)$ and $E_2 = (L_2, \Sigma_2)$ with $L_i =: (F_{i,1}, F_{i,2}, \ldots, F_{i,k+1})$. Let $L_i' := (F_{i,2}, \ldots, F_{i,k+1})$. $L_1'$ and $L_2'$ have both length $k$ and thus are equal after the transformation by induction hypothesis. Since $F_{1,1}$ and $F_{2,1}$ are processed first and the processing of one pair of frames does not change any other frames we just have to prove that $F_{1,1}$ and $F_{2,1}$ are equal after the first execution of the first for-loop. This can be done as before and we get $L_1 = L_2$ when returning $L_1$ in the last line. Thus, $\text{combine}$ is commutative. \hfill $\Box$

**Lemma 6.2.49** Let $E$ be an abstract environment. Then

$$\langle E \rangle = \bigcup_{E' \in S(E)} \langle E' \rangle.$$  

\footnote{At every time during program execution there is just a finite number of stores in use. Thus, the induction also works for the frame state.}
Proof of lemma 6.2.49: Let $E = (L, \Sigma)$ be an abstract environment. Then

$$\langle E \rangle = \langle L \rangle = \bigcup_{\sigma \in \Sigma} \langle GIS(\sigma)(L) \rangle = \bigcup_{\sigma \in \Sigma} \langle GIS(\sigma)(CAE(L, \sigma)) \rangle = \bigcup_{\sigma \in \Sigma} \langle (CAE(L, \sigma), \{\sigma\}) \rangle = \bigcup_{E' \in S(E)} \langle E' \rangle.$$ 

\[ \square \]

Lemma 6.2.50 For every abstract environment $E \in Env_A$ the result $\text{canonical}(E)$ expresses the same set of environments in the standard semantics as $E$, i.e.

$$\forall E \in Env_A. \langle E \rangle = \langle \text{canonical}(E) \rangle.$$ 

Proof of lemma 6.2.50: We prove the lemma by induction on the number $n$ of variables eliminated by $\text{canonical}$:

$n = 0$: This is only the case if none of the variables occurring in $E$ violates the condition for canonical abstract environments. In this case $E$ is already canonical and the algorithm $\text{canonical}$ does not change $E$ at all, i.e. $E = \text{canonical}(E)$ and thus $\langle E \rangle = \langle \text{canonical}(E) \rangle$.

$n \rightarrow n+1$: Let $E$ contain $n + 1$ variables violating the condition on canonical abstract environments, let $\tilde{E} = (\tilde{L}, \{\tilde{\sigma}_1, \ldots, \tilde{\sigma}_k\})$ be the intermediate abstract environments after eliminating $n$ of these variables and let $X$ be the remaining variable.

Since all the variables are processed independently from each other by $\text{canonical}$ we can apply the induction hypothesis and get:

$$\langle E \rangle = \langle \tilde{E} \rangle.$$ 

Now let $v$ be the abstract value that is assigned to $X$ by all of the $\sigma_i$ and therefore by all of the $\tilde{\sigma}_i$, let $\bar{\sigma} = GIS(\{X \leftarrow v\})$ and let $\sigma'_i := \bar{\sigma} \circ \tilde{\sigma}_i|_{\text{dom}(\tilde{\sigma}_i) \setminus \{X\}}$.

Since $\text{dom}(\sigma'_i) \cap \text{dom}(\bar{\sigma}) = \emptyset$ we can conclude that

$$GIS(\bar{\sigma} \circ \tilde{\sigma}_i|_{\text{dom}(\tilde{\sigma}_i) \setminus \{X\}}) = GIS(\bar{\sigma}_i)$$

because the two arguments to $GIS$ just differ in some occurrences of $X$ that are already replaced by a recursive type definition on the left hand side. These recursive types are introduced by $GIS$ at the right hand side as well.

Furthermore, $GIS(\bar{\sigma}_i) \circ \bar{\sigma} = GIS(\bar{\sigma}_i)$: Both substitutions behave equally except for applications to $X$. Because of the definition of $\bar{\sigma}$ we have $\bar{\sigma}(X) = \mu V_X. v[X|V_X]$ and $GIS(\bar{\sigma}_i)(X) = GIS(\bar{\sigma}_i)(\mu V_X. v[X|V_X])$.

254
Now let $E'$ be the result returned by canonical with

$$E' = (L', \{\sigma'_1, \ldots, \sigma'_k\}) = (\bar{\sigma}(\bar{L}), \{\sigma'_1, \ldots, \sigma'_k\}).$$

We have

$$\langle E' \rangle = \{ \langle L' \rangle \}_{\{\sigma'_1, \ldots, \sigma'_k\}} = \bigcup_{i=1}^{k} \{ \text{GIS}(\sigma'_i)(L') \} = \bigcup_{i=1}^{k} \text{GIS}(\bar{\sigma} \circ \bar{\sigma}_i \mid \text{dom}(\bar{\sigma}_i) \setminus \{X\})(\bar{\sigma}(\bar{L}))$$

$$= \bigcup_{i=1}^{k} \{ \text{GIS}(\bar{\sigma}_i) \circ \bar{\sigma}(\bar{L}) \} = \bigcup_{i=1}^{k} \{ \text{GIS}(\bar{\sigma}_i)(\bar{L}) \} = \langle \bar{L} \rangle \{ \bar{\sigma}_1, \ldots, \bar{\sigma}_k \} = \langle \bar{E} \rangle.$$

Altogether we get $\langle E \rangle = \langle \text{canonical}(E) \rangle$ for every $E \in Env_A$. \hfill \Box

**Lemma 6.2.51** Let $E_1, E_2 \in Env_A$, $E = \text{combine}(E_1, E_2)$. The function combine leaves the set denoted environments unchanged, i.e. $\langle E \rangle = \langle E_1 \rangle \cup \langle E_2 \rangle$.

**Proof of lemma 6.2.51:** Let $E_1 = (L_1, \Sigma_1)$, $E_2 = (L_2, \Sigma_2)$ and $E = (L, \Sigma)$. We prove the lemma by showing the two set inclusions $\langle E \rangle \subseteq \langle E_1 \rangle \cup \langle E_2 \rangle$ and $\langle E \rangle \supseteq \langle E_1 \rangle \cup \langle E_2 \rangle$:

$\langle E \rangle \subseteq \langle E_1 \rangle \cup \langle E_2 \rangle$: Let $\bar{E} \in \langle \bar{E}_1 \rangle$ with $\bar{E}_A \in S(E)$ be an environment denoted by the result of combine and let $\sigma \in \Sigma$ be the substitution creating $\bar{E}_A = (\text{CAE}(L, \sigma), \{\sigma\})$. Since $E = \text{combine}(E_1, E_2)$ the substitution $\sigma$ resulted either from a transformation of some $\sigma_1 \in \Sigma_1$ or some $\sigma_2 \in \Sigma_2$. Without loss of generality (because of the commutativity of combine, cf. Lemma 6.2.34) let $\sigma$ stem from $\sigma_1 \in \Sigma_1$ and let $\bar{E}_{1,A} = (\text{CAE}(L_1, \sigma_1), \{\sigma_1\})$.

We will now show $\bar{E} \in \langle \bar{E}_{1,A} \rangle$: $\bar{E}_A$ and $\bar{E}_{1,A}$ have the same number of frames because the frame list $L$ was generated from the frame list $L_1$ without introducing or deleting frames. All symbols that are bound in a certain frame of the frame list $L_1$ are also bound in the corresponding frame of $L$.

Now let $s$ be an arbitrarily chosen symbol that is bound in a certain frame of $L$.\footnote{Since all symbols bound in a frame of $L_1$ or $L_2$ are also bound in the corresponding frame of $L$ by definition of $L$ no other symbols have to be analyzed.} Because of $\bar{E} \in \langle \bar{E}_A \rangle$ we know that $\bar{E}(s) \in \langle \bar{E}_A(s) \rangle$. We show that $\bar{E}(s) \in \bar{E}_{1,A}(s)$:

1. Let $s$ be bound to the same value $e$ in $L_1$ and $L_2$. Then $s$ is bound to the same value in $L$ and $\text{GIS}(\sigma)(e) = \text{GIS}(\sigma_1)(e) =: e'$ because the transformation of $\sigma_1$ to $\sigma$ did not change the binding of existing variables. Thus, $\bar{E}_A(s) = \bar{E}_{1,A}(s) = e'$ and $\bar{E}(s) \in \{e'\}$.

2. Let $s$ be bound to $e$ in $L_1$ but bound to another value or unbound in $L_2$. Then $s$ is bound to a variable $X$ in $L$ and $\text{GIS}(\sigma)(X) = \text{GIS}(\sigma_1)(e) =: e'$ by construction of $\sigma$. Thus, $\bar{E}(s) = \bar{E}_1(s) = e'$ and again $\bar{E}(s) \in \{e'\}$. 

255
3. Let $s$ be unbound in $L_1$. Then $s$ is bound to a variable $X$ in $L$ and $\sigma(X) = \text{undef}$. Thus, $s$ is undefined in both $\tilde{E}$ and $\tilde{E}_1$ and $\tilde{E}(s) = \text{undef}$ must hold in order to get $\tilde{E}(s) \in \{ \tilde{E}_A(s) \}$.

Altogether $\tilde{E} \in \{ \tilde{E}_1, A \}$ and since we had chosen $\tilde{E} \in \{ E \}$ the statement is proven.

$\{ E \} \supseteq \{ E_1 \} \cup \{ E_2 \}$: Let $\tilde{E} \in \{ E \}_{1,A}$ with $\tilde{E}_1, A \in S(E_1) \cup S(E_2)$, without loss of generality (because of the commutativity of combine, cf. Lemma 6.2.34) $\tilde{E}_1, A \in S(E_1)$. Then there exists a $\sigma_1 \in \Sigma$ with $\tilde{E}_1, A = (\text{CAE}(L_1, \sigma_1), \{ \sigma_1 \})$ and from the construction of $E$ there is a corresponding $\sigma \in \Sigma$ with $(\text{CAE}(L, \sigma), \{ \sigma \}) =: E_A \in S(E)$. Comparing the symbol bindings as in Part 1 yields $\tilde{E} \in \{ \tilde{E}_A \}$ and since $\tilde{E}$ was chosen arbitrarily we get $\{ E \} \supseteq \{ E_1 \} \cup \{ E_2 \}$.

Altogether we have proven equality: $\{ E \} = \{ E_1 \} \cup \{ E_2 \}$. □

**Lemma 6.2.52** The function combine is associative with respect to the denoted sets of environments in the standard semantics, i.e. for three abstract environments $E_1, E_2, E_3 \in Env_A$ we have:

$$\{ \text{combine}(E_1, \text{combine}(E_2, E_3)) \} = \{ \text{combine}(\text{combine}(E_1, E_2), E_3) \}.$$ 

**Proof of lemma 6.2.52:** The prove is done by comparing the denoted sets of environments in the standard semantics. Using Lemma 6.2.51 we get:

$$\{ \text{combine}(E_1, \text{combine}(E_2, E_3)) \} = \{ E_1 \} \cup \{ \text{combine}(E_2, E_3) \}$$

$$= \{ E_1 \} \cup \{ \{ E_2 \} \cup \{ E_3 \} \}$$

$$= \{ \{ E_1 \} \cup \{ E_2 \} \} \cup \{ E_3 \}$$

$$= \{ \text{combine}(E_1, E_2) \} \cup \{ E_3 \}$$

$$= \{ \text{combine}(\text{combine}(E_1, E_2), E_3) \}.$$ 

□

**Lemma 6.2.53** Let $E \in Env_A$ be an extended abstract environment. Then

$$\{ E \} = \{ \text{compress}(E) \}.$$ 

**Proof of lemma 6.2.53:** Let $E = (L, \Sigma)$, $E' = (L, \Sigma')$. (compress does not change the frame list $L$ when transforming $E$ to $E'$.) We will prove the two set inclusions $\{ E' \} \subseteq \{ E \}$ and $\{ E' \} \supseteq \{ E \}$.

$\{ E' \} \subseteq \{ E \}$: Let $v \in \{ E' \}$ and let $\sigma' \in \Sigma'$ be a substitution with $v \in \{ \text{GJ}(\sigma')(L) \}$. For $\sigma' \in \Sigma$ we have nothing to show. Otherwise $\sigma'$ was generated by compressing $\sigma_1, \ldots, \sigma_l \in \Sigma$. 

256
We show that there exists an $i$ such that $v \in \sigma_i(L)$. More exactly we show that whenever a single execution of the while-loop compresses $\sigma_1$ and $\sigma_2$ to $\sigma'$ and $v \in \sigma'(L)$ then $v \in \sigma_j(L)$ for at least one $j \in \{1, 2\}$. (The statement for more substitutions results from a simple induction.)

The conditions for executing the while-loop we know that $\sigma_1(L)(s) = \sigma_2(L)(s) = \sigma'(L)(s)$ for every symbol $s \neq \text{return}$ because even if $s$ is bound to a variable $Y$ in $L$ then $Y \neq X$ (where $X$ is the variable return is bound to) and therefore $\sigma_1(Y) = \sigma_2(Y) = \sigma'(Y)$. If $s$ is not bound to a variable in $L$ then there is nothing to show.

Now consider the value $w \in \{E'(\text{return})\}$. Since $L(\text{return}) = X$ we know that $w = \{\text{GIS}(\sigma')(X)\}$. By construction of $\sigma'$ we have $\sigma'(X) = (\cup \sigma_1(X) \sigma_2(X))$ and by definition of the union type constructor either $w \in \{\text{GIS}(\sigma_1)(X)\}$ or $w \in \{\text{GIS}(\sigma_2)(X)\}$. (Note that the application of GIS behaves identical on $\sigma'$, $\sigma_1$ and $\sigma_2$ because the only change is the binding to the variable $X$ that does not occur on a right hand side.) The corresponding substitution then fulfills $v \in \{\sigma_i(L)\}$ and therefore $v \in \{E\}$.

$\{E'\} \supset \{E\}$: Now let $v \in \{E\}$, more exactly $v \in \{\text{GIS}(\sigma)(L)\}$ with $\sigma \in \Sigma$. If $\sigma \in \Sigma'$ we have nothing to show. Otherwise there exists a substitution $\sigma' \in \Sigma'$ that was generated by compressing $\sigma$ with one or several other substitutions. We show that the result of a single compression step performed by a single execution of the while loop still contains $v$. The statement can then be obtained by induction on the number of compressed substitutions.

Let $\sigma, \tilde{\sigma} \in \Sigma$ and let $\sigma'$ be the result of compressing $\sigma$ and $\tilde{\sigma}$. Consider the bindings of $v$. For a symbol $s \neq \text{return}$ bound in $v$ we have $\text{GIS}(\sigma)(L)(s) = \text{GIS}(\tilde{\sigma})(L)(s) = \text{GIS}(\sigma')(L)(s)$. For $s = \text{return}$ let $w = v(\text{return})$. Because of $w \in \{\text{GIS}(\sigma)(L)(\text{return})\}$ and $L(\text{return}) = X$ we have $w \in \{\text{GIS}(\sigma)(X)\}$. By the definition of $\sigma'$ we have $\{\text{GIS}(\sigma)(X)\} \subseteq \{\text{GIS}(\sigma')(X)\}$ and therefore $w \in \{\text{GIS}(\sigma')(X)\}$. Therefore, $w \in \{\text{GIS}(\sigma')(L(\text{return}))\}$ and $v \in \{\text{GIS}(\sigma')(L)\} \subseteq \{E'\}$. $\square$

### C.2 Proofs of Section 6.3

**Lemma 6.3.39** Let $v_{\mathcal{A}} \in \text{AssValue}_{\mathcal{A}}$ be generated by abstract evaluation of an expression of the form $e_q = (\text{quote } \ldots) \in \text{Expression}$ and let $v$ be generated by evaluating $e_q$ in the standard semantics. Then

1. $v \in \{v_{\mathcal{A}}\}$.
2. If $e_{\mathcal{A}} = \text{valueexpression}_{\mathcal{A}}(v_{\mathcal{A}})$, $e = \text{valueexpression}(v)$, $E_{\mathcal{A}} \in \text{Env}_{\mathcal{A}}$ and $E \in \{E_{\mathcal{A}}\}$

then

$$[e](E) \in \{[e_{\mathcal{A}}]^A(E_{\mathcal{A}})(\text{return})\}.$$
Proof of lemma 6.3.39: Let \( v \) and \( v_A \) be as in the lemma.

The first statement \( v \in \{v_A\} \) follows from analyzing the functions \( \text{quotesem} \) and \( \text{quotesem}_A \) case by case:

- If \( e \in \text{ConstT} \) or \( e \in \text{Sym} \) then \( \text{quotesem}_A(e) = A(\text{quotesem}(e)) \).
- If \( e \) is a list expression then \( \text{make-list} \) and \( \text{make-list}_A \) generate equivalent list structures.

From these two cases \( v \in \{v_A\} \) follows trivially by induction on the number of nested list expressions.

Because of \( v \in \{v_A\} \) the standard and abstract values \( v \) and \( v_A \) are similar in their structure, i.e.:

- \( v \in \text{Const} \iff \text{either } v_A \text{ is a value assignment or } v_A \in \text{Const}_A \).\(^3\)
- \( v \) is a list \( \iff v_A \) is a list.

This implies the following correspondence of cases in the definitions of \( \text{valuesexpression} \) and \( \text{valuesexpression}_A \).

1. \( v \) is processed by case 1 in Def. 5.2.31 iff \( v_A \) is processed by one of the cases 1 and 2 in Def. 6.3.29.
2. \( v \) is processed by case 3 in Def. 5.2.31 iff \( v_A \) is processed by case 4 in Def. 6.3.29.
3. \( v \) is processed by case 4 in Def. 5.2.31 iff \( v_A \) is processed by case 5 in Def. 6.3.29.

Note that especially case (3) of the definition of \( \text{valuesexpression}_A \) cannot occur because \( v_A \) was generated by evaluating an expression of the form \( e_q(\text{quote} \ldots) \) and therefore \( v_A \) contains value ascriptions for every symbol occurring in \( e_q \).

We now discuss the cases enumerated above individually:

Case (3) cannot violate the lemma: Because of the choice of \( \text{express} \) in Def. 5.2.31 we have \( [e](E) = v \) and because of the choice of \( \text{express}_A \) in Def. 6.3.29 we have \( [e_A]^A(E_A)(\text{return}) = v_A \). The precondition \( v \in \{v_A\} \) implies the statement of the lemma.

\(^3\)From the definition of \( \text{quotesem}_A \) we can conclude that \( v_A \) must be a value assignment. However, the rest of the proof works for a greater class of abstract values \( v_A \) fulfilling \( v \in \{v_A\} \) with \( v_A \) not containing \( \text{sym} \) as subexpression, as well.
In Case (2) the result is a list expression. The statement of the lemma holds for list expressions as long as it holds for the list elements (This is provable by case distinction on the different evaluation rules for list expressions for \[ \cdot \] and \([\cdot]_A \). Note that especially every syntactic keyword is reconstructed in the same way in \( e \) and \( e_A \) since \texttt{sym} does not occur at any position in \( v_A \).) The statement for the elements can be proven by induction on the number of recursive calls to \texttt{valueexpression} or \texttt{valueexpression}_A, respectively.

For Case (1) we make a case distinction on \( v_A \):

- \( v_A = A(v') \) is a value expression. Because of \( v \in \{v_A\} \) we can conclude \( v' = v \). This implies \( e_A := \texttt{valueexpression}_A(v_A) = \texttt{valueexpression}(v') = \texttt{valueexpression}(v) =: e \). For arbitrary \( E \) and \( E_A \) as in the lemma this implies

\[
\llbracket e \rrbracket(E) = v \in \{A(v)\} = \{\llbracket e_A \rrbracket^A(E_A)('\texttt{return}')\}.
\]

- If \( v_A \in \texttt{Const}_A \) then \( \llbracket e_A \rrbracket^A(E_A) = v_A \) and \( \llbracket e \rrbracket(E) = v \) for arbitrary \( E \). With \( v \in \{v_A\} \) this implies the lemma.

\[\square\]

\textbf{Lemma 6.3.41} Let \( e \in \text{Expression} \), \( e_A \in \text{Expression}_A \) such that \( e_A \) is compatible with \( e \) and let \( E \in \text{Env}_A \) such that \( \llbracket e_A \rrbracket^A(E) \) is free of eval abstraction. Suppose that \( \llbracket e_A \rrbracket^A(E) = \tilde{E} = (L, \tilde{\Sigma}) \) can be calculated without an infinite loop.

Then for every \( E' \in \{E\} \) such that \( \llbracket e \rrbracket(E') \notin \{\bot, \texttt{error}\} \) we have

\[
\llbracket e \rrbracket(E') \in \{\llbracket e_A \rrbracket^A(E)('\texttt{return}')\}
\]

and

\[
\text{EU}(e, E') \in \{\llbracket e_A \rrbracket^A(E)['\texttt{return} \leftarrow \texttt{undef}']\}.
\]

\textbf{Proof of lemma 6.3.41:} Let \( E = (L, \Sigma) \). We prove the lemma by induction on the maximal depth \( k \) of recursive calls to \([\cdot]_A \). (This is possible because of the termination precondition for \( \llbracket e_A \rrbracket^A(E) \).

\( k = 1 \): \( e_A \) can be of the following form:

- \( e_A \) is a constant expression: Because of the compatibility of \( e_A \) and \( e \) this implies \( e_A = e \) and we get \( \llbracket e \rrbracket(E) = \texttt{denoteconst}(e) \in \{\texttt{denoteconst}(e)\} = \{\texttt{Adenoteconst}(e)\} = \{\texttt{Adenoteconst}(e)\} = \{\llbracket e \rrbracket^A(E)('\texttt{return}')\} \).

\[
\{\llbracket e \rrbracket^A(E)['\texttt{return} \leftarrow \texttt{undef}']\} = \{E\} = \{E' \in \{E\}\} = \{\text{EU}(e, E') \mid E' \in \{E\}\}.
\]

259
• $e$ is an abstract constant: The compatibility of $e_A$ and $e$ implies $\text{denoteconst}(e) \in \{\text{denoteabsconst}_A(e_A)\}$ and therefore

$$[e](E) = \text{denoteconst}(e) \in \{\text{denoteabsconst}_A(e_A)\} = \{[e]^A(E)(\text{return})\}.$$  

Furthermore, $EU(e, E') = E' \in \{E\} = \{[e]^A(E)(\text{return} \leftarrow \text{undef})\}$.

• $e$ is a symbol: Because of the compatibility of $e_A$ and $e$ this is just possible with $e_A = e$ and therefore $[e](E') = E'(e) \in \{\bigcup_{\sigma \in \tilde{\Sigma}} \{\sigma(L(e))\}\} = \{L(e)\} = \{E(e)\} = \{E(e_A)\} = \{[e]^A(E)(\text{return})\}$.  

Furthermore, $EU(e, E') = E' \in \{E\} = \{[e]^A(E)(\text{return} \leftarrow \text{undef})\}$.

• $e_A = \text{(lambda} (x_1 \ldots x_m) e')$ is a lambda expression: Because of the compatibility of $e$ and $e_A$ this implies $e = e_A$. Hence we get:

$$[e](E') = lc(e'(x_1, \ldots, x_m), E') \in \{lc_A(e'(x_1, \ldots, x_m), L)\} = \{[e]^A(E)(\text{return})\}.$$  

The step from standard to abstract lambda closures makes especially use of $\tilde{\Sigma} = \Sigma$. For the environment update we get

$$EU(e, E') = E' \in \{E\} = \{[e]^A(E)(\text{return} \leftarrow \text{undef})\}.$$  

$k \rightarrow k + 1$: Let $e_A^0, e_A^1, \ldots, e_A^n$ be expressions evaluated in at most $k$ recursive steps of $[\cdot]^A$ in $E_i$ with result $[e_A^i](E_i) = \tilde{E}_i = (\tilde{L}_i, \tilde{\Sigma}_i)$. Furthermore, let $e^i \in \text{Expression}$ such that $e_A^i$ is compatible with $e^i$ and let $E_i' \in \{E_i\}$ for $i \in \{0, \ldots, k\}$. If $[e^i](E_i') \not\in \{\bot, \text{error}\}$ the induction hypothesis yields

$$[e^i](E_i') \in \{[e^i]^A(E_i')(\text{return})\}$$  

and

$$EU(e^i, E_i') \in \{[e^i]^A(E_i')(\text{return} \leftarrow \text{undef})\}.$$  

We distinguish the following cases for the expression $e_A$:

1. $e_A$ is an application expression. Three subcases on the function position in $e$ have to be distinguished:

   • A lambda closure is applied.
   • A predefined function is applied.
   • The function position does not hold an applicable value.
2. $e_A$ is a begin expression. The cases for 1 or $>1$ subexpressions are discussed independently.

3. $e_A$ is a define expression.

4. $e_A$ is an if expression. There is a case distinction on the value of the condition whether it
   - fulfills $AT$
   - fulfills $AF$
   - does not fulfill $AT$ or $AF$

5. $e_A$ is a quote expression.

These cases are now discussed in detail:

**Applications**

$e_A = (e_A^0 e_A^1 \ldots e_A^n)$ is an application expression: The compatibility of $e_A$ and $e$ implies $e = (e^0 e^1 \ldots e^n)$ such that every $e^i_A$ is compatible with $e^i$.

If one of the $e^i$ evaluates to $\perp$ or error the whole expression $e$ also evaluates to $\perp$ or error and we have nothing to show. Otherwise, according to rule (App) of Def. 6.3.36 we get $[e]^A(E) = apply(e, E)$ where apply first evaluates the subexpressions $e^i_A$ by calculating $\hat{E}(L, \Sigma) := pre-app(E, e)$. For every $E' \in \{E\}$ we now show that $v = [e](E') \in [\{e\}]^A(E) (\text{return})$. We do this by case distinction on $v_0 = [e_0](E')$. The following cases are possible:

- $v_0 \in LC$, i.e. $e_0$ denotes a user defined function.
- $v_0 \in PFun$, i.e. $e_0$ denotes a predefined function.
- $v_0 \not\in LC \cup PFun$. This is the error case with $v_0$ denoting a non-applicable value.

Before analyzing these cases in detail we introduce the abbreviations $v_0, \ldots, v_n$ for the evaluation results $[e_i](E'_i)$ of the subexpressions $e_0, \ldots, e_n$ when evaluating $[e](E')$ in the standard semantics.

From the induction hypothesis for the subexpressions we can conclude that $E'_i \in \{E_i\}$ where $E_i$ is the abstract environment $e^i_A$ is evaluated in by $pre-app$. This implies $[e_i](E'_i) \in [\{e_i\}]^A(E_i) (\text{return})$ with $\Sigma_i$ streaming from $\hat{E}_i = (\hat{L}_i, \hat{\Sigma}_i) = [e_i]^A(E_i)$.

Now we analyze the three cases independently:

**Case 1 $v_0 \in LC$:** Therefore, $v_0$ must be of the form $v_0 =: lc(e'(x_1, \ldots, x_m), E_d)$. Because of the induction hypothesis for $e^0_A, \ldots, e^n_A$ the pre-application environment must contain a
Since no side effects are allowed in the analyzed expressions we get\n\[ \{E_s(\text{func})\} \] and\n\[ \{E_s(\text{arg})\} \] for \( i \in \{1, \ldots, n\} \). From \( v_0 \in \{E_s(\text{func})\} \) we know that \( E_s(\text{func}) = lc_A(e'(x_1, \ldots, x_m), L_d^A) \). The definition environment \( E_d \) of \( lc \) fulfills \( E_d \in \{ (L_d^A, \{\sigma_s\}) \} \). Hence\n\[ E_d[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \in \{ (L_d^A[x_1 \mapsto \tilde{E}(\text{arg}_1), \ldots, x_n \mapsto \tilde{E}(\text{arg}_n)], \{\sigma_s\})\} \).

For \( m = n \) (i.e. the number of arguments is correct for the lambda closure) the induction hypothesis is valid for the evaluation of \( e' \) in the appropriate environments (because \( [e']^A \) can be calculated in at most \( k \) steps) and yields
\[ \begin{align*}
v &= [e'](E_d[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]) \\ &\subseteq \{ [e']^A(L_d^A[x_1 \mapsto \tilde{E}(\text{arg}_1), \ldots, x_n \mapsto \tilde{E}(\text{arg}_n)], \{\sigma_s\})(\text{return})\} \subseteq \{ [e']^A(E)(\text{return})\}.
\end{align*} \]

Since \textit{compress} and \textit{combine} used in Def. 6.3.32 are known not to change the semantics of the processed environments by the Lemmas 6.2.53 and 6.2.51 we can analyze the change of environments independently for the simple abstract environments, too.

Since no side effects are allowed in the analyzed expressions we get \( EU(e, E') = E' \) and \( [e_A]^A(E) = \tilde{E} = (L, \tilde{\Sigma}) \) such that for every \( \sigma \in \Sigma \) occurring in \( E \) there is a \( \tilde{\sigma} \in \tilde{\Sigma} \) just binding additional variables that do not occur in \( \tilde{E}[\text{return} \mapsto \text{undef}] \). Together with \( E' \in \{E\} \) this implies
\[\begin{align*}
EU(e, E') &\subseteq \{ [e_A]^A(E)[\text{return} \mapsto \text{undef}]\}.
\end{align*}\]

For \( m \neq n \) (i.e. the lambda closure is called with a wrong number of arguments) we have \( [e](E') = \text{error} \) and therefore we have nothing to show.

\textbf{Case 2} \( v_0 \in \text{PF} \): First of all we analyze the functions \( f_{\text{eval}} \) and \( f_{\text{scheme-report-environment}} \) having a special semantics.

For a call \( f_{\text{eval}} \) the induction hypothesis holds for all arguments. In the error case we have nothing to show.

In the case of success let \( E'' \) be the environment the second argument of \textit{eval} evaluates to in the standard semantics when evaluating \( e \) in some \( E' \in \{E\} \). Let \( E''_A \) be the abstract environment the second argument evaluates to in the abstract semantics. Then the induction hypothesis yields \( E'' \in \{E''_A\} \). The same holds for the results of evaluating the first argument (\( v \) in the standard semantics and \( v_A \) in the abstract semantics), i.e. \( v \in \{v_A\} \). Now let \( \tilde{e} = \text{valueexpression}(v) \) and \( \tilde{e}_A = \text{valueexpression}_A(v_A) \). Then Lemma 6.3.39 (which is applicable because of the prerequisite on the first argument of \( f^A \) given in the lemma) states
\[\begin{align*}
[e''(E'')] &\subseteq \{ [e''_A]^A(E''_A)(\text{return})\} = \{ [e'_A]^A(E''_A)(\text{return})\}.
\end{align*}\]
By the definition of the semantics of $f_{\text{eval}}$ in Def. 5.2.33 and Def. 6.3.28 this gives us the statement of the lemma for the considered case.

The induction hypothesis also implies $EU(\bar{e}, E'') \in \llbracket \bar{e}_A \rrbracket^A (E''_A)$. This implies the second statement of the lemma.

For $f_{\text{interaction-environment}}$ just the error free case with no arguments given is of interest: Rule $(IE)$ of Def. 5.2.33 gives a return value $E_i = \text{envcopy}(E_i)$ and for the abstract semantics rule $(IE)$ in Def. 6.3.28 yields the abstract result $E^A_i = \langle \text{frame-list-copy}_A(AF_i), \Sigma \rangle$ for some environment instantiation $\Sigma$. Because of $E_I \in \llbracket AE_i \rrbracket$ and $AF_i$ does not contain variables we also have $E_i \in \llbracket E^A_i \rrbracket$ because both $\text{envcopy}$ and $\text{frame-list-copy}_A$ return an environment consisting of one frame with all visible bindings of the given environment.

Altogether for every $E' \in \llbracket E \rrbracket$ with $[e^0](E') = f_{\text{interaction-environment}}$ we get:

$$[e](E') \in \llbracket \llbracket e_A \rrbracket^A(E)(\text{return}) \rrbracket.$$  

Environment updates are not performed by $f_{\text{interaction-environment}}$ and therefore we get:

$$EU(e, E') = E' \in \llbracket E \rrbracket = \llbracket \llbracket e_A \rrbracket^A(E)[\text{return} \leftarrow \text{undef}] \rrbracket.$$  

In the case of functions $v_0 = f_{<\text{xyz}>}$ not having a special semantics $f^A_{<\text{xyz}>}$ is the only possible abstract value $e^0_A$ can evaluate to in the abstract semantics. Furthermore,

$$D(f^A_{<\text{xyz}>}) = \{(IN_1, OUT_1), \ldots, (IN_r, OUT_r)\}$$  

is an I/O-representation for $f^A_{<\text{xyz}>}$, i.e.

- $\text{dom}(f_{<\text{xyz}>}) \subseteq \bigcup_{i=1}^r \llbracket IN_i \rrbracket$
- $\forall_i f_{<\text{xyz}>}(\llbracket IN_i \rrbracket) \subseteq \llbracket OUT_i \rrbracket \cup \{\text{error}\}$

Now consider the input value $v_{in} = v_1 \times \ldots \times v_n$ and the corresponding output value $v_{out} = (f_{<\text{xyz}>} v_{in})$ in the standard semantics.

Since $D(f^A_{<\text{xyz}>})$ is an I/O-representation of $f^A_{<\text{xyz}>}$ with $f^A_{<\text{xyz}>} \in \llbracket f^A_{<\text{xyz}>} \rrbracket$ there exists a substitution $\rho_1$ and an I/O-representation pair $(IN, OUT)$ in $D(f^A_{<\text{xyz}>})$ with $v_{in} \in \llbracket \rho_1(IN) \rrbracket$ and $v_{out} \in \llbracket \rho_1(OUT) \rrbracket$. Because of the induction hypothesis holding for the $e^i_A$ we can conclude that the current pre-application environment $\hat{E}$ fulfills $v_i \in \llbracket \rho_2(\hat{E}(\text{arg}_i)) \rrbracket$ for all $i \in \{1, \ldots, n\}$ and some substitution $\rho_2$. Altogether, there exists a substitution $\rho$ fulfilling $^4$

$$v_{in} \in \llbracket \rho(IN) \rrbracket \cap \llbracket \rho(\hat{E}(\text{arg}_1) \times \ldots \hat{E}(\text{arg}_n)) \rrbracket.$$  

^4 We assume $IN$ and $OUT$ not to contain any variables occurring in $\hat{E}$. This is easily achievable by renaming the variables in $IN$ and $OUT$.  

263
By Theorem 4.3.17 there exists a $\sigma \in CE(\hat{E}(\text{arg}_1) \times \ldots \times \hat{E}(\text{arg}_n), IN)$ and a $\tau$ such that $v_{in} \in \{\tau \circ \sigma(\hat{E}(\text{arg}_1) \times \ldots \times \hat{E}(\text{arg}_n))\} \cap \{\tau \circ \sigma(IN)\}$. Especially, $\tau \circ \sigma$ is not more restrictive than $\rho$ and therefore $v_{out} \in \tau \circ \sigma(OUT)$.

By the definition of $PPAF$ the result of $PPAF(f^A_{<xyz>}, \hat{E}(\text{arg}_1) \times \ldots \times \hat{E}(\text{arg}_n))$ contains either $(OUT, \sigma)$ or a pair $(OUT', \sigma')$ such that $\sigma'(OUT')$ is more general than $\sigma(OUT)$. Therefore, $\{\tau' \circ \sigma(OUT)\} \subseteq \{\tau'(PPAF(f^A_{<xyz>}, \hat{E}(\text{arg}_1) \times \ldots \times \hat{E}(\text{arg}_n)))\}$ is fulfilled for every closed type substitution substitution $\tau'$ appropriate for $\sigma(OUT)$ and $PAF(f^A_{<xyz>}, \hat{E}(\text{arg}_1) \times \ldots \times \hat{E}(\text{arg}_n))$. Since the change from $PAF$ to $PAF_c$ does not eliminate values in $\{\tau'(PPAF(\ldots))\}$ for any $\tau'$ and especially for $\tau' = \tau$ we get

$$\{\tau \circ \sigma(OUT)\} \subseteq \{PAF_c(f^A_{<xyz>}, \hat{E}(\text{arg}_1) \times \ldots \times \hat{E}(\text{arg}_n))\}.$$

Altogether, we have proven

$$v_{out} \in \{PAF_c(f^A_{<xyz>}, \hat{E}(\text{arg}_1) \times \ldots \times \hat{E}(\text{arg}_n))\} = \{[[e]^A(E')]\} = \{[[e]^A(E')]\}.$$

The predefined functions considered in this work just return a value, but do not change the environment. Therefore:

$$EU(e, E') = E' \in \{E\} = \{[[e]^A(E)][\text{return} \leftarrow \text{undef}]\}.$$

**Case 3** $v_0 \not\in LC \cup PFunc$: In that case $[[e](E')] = \text{error}$ and we have nothing to show.

Up to now we have shown that for every case on $v_0$ there is a call to a subfunction of apply providing the correct result. The Lemmas 6.2.51 (equivalence of combined abstract environments) and 6.2.53 (equivalence of compressed environments) now imply that the function apply in the abstract semantics does not drop any return values or environments of some of the cases.

**begin expressions**

Let $e = (\text{begin } e^1 \ldots e^n)$ be a begin expression: The compatibility of $e_A$ and $e$ implies $e = e_A$. We distinguish the two cases $n = 1$ and $n > 1$:

**Case 1** ($n = 1$): $e = (\text{begin } e^1)$. From the induction hypothesis we know:

$$[[e^1](E')] \in \{[[e^1]^A(E)(\text{return})]\}$$

and

$$EU(e^1, E') \in \{[[e^1]^A(E)[\text{return} \leftarrow \text{undef}]\}$$

From Rule (Begin-1) of Def. 6.3.37 and Rule (Std) of Def. 6.3.36 we can conclude that $[[e]^A = [[e^1]^A$. For the standard semantics Rule (Begin-1) of Def. 5.2.16 implies $[[e](E') = [[e^1](E')$ and Rule (Begin-1) Def. 5.2.15 implies $EU(e, E') = EU(e^1, E')$. Altogether we get:

$$[[e](E') \in \{[[e]^A(E)(\text{return})\}$$

264
For the return values of the begin expression we get

\[ EU(e, E') \in \{[[c]^A(E)\text{[return } \leftarrow \text{undef}]] \} \]

**Case 2 (n > 1):** This case is processed by rule (Begin-k) of Def. 6.3.37. The result is based on the results for \( e_1 \) and \( e' := (\text{begin } e^2 \ldots e^n) \). For both the induction hypothesis holds.

For the return values of the begin expression we get

\[
[e](E') = [e'](EU(e^1, E')) \in \{[[e'](E'') \mid E'' \in \{[[e]^A(E)]\}] \subseteq \\
\subseteq \{[[e']^A([e]^A(E))](\text{return}) = \{[[e]^A(E)\text{[return } \leftarrow \text{undef}]] \}
\]

(Note that the additional binding of \text{return} in \{[[e]^A(E)]\} is not used during the evaluation of \{[[e']^A([e]^A(E))]\} and can therefore be ignored.)

For the result of the environment update we get

\[
EU(e, E') = EU(e', EU(e^1, E)) \in \{EU(e', E'') \mid E'' \in \{[[e]^A(E)]\}
\subseteq \{[[e']^A([e]^A(E))\text{[return } \leftarrow \text{undef}] = \{[[e]^A(E)\text{[return } \leftarrow \text{undef}]] \}
\]

**define expressions**

Let \( e = (\text{define } s \ e') \) be a define expression with a symbol \( s \) and a subexpression \( e' \): The compatibility of \( e \) and \( e_A \) implies \( e_A = e \). For \( e' \) the induction hypothesis holds, i.e. with \([[[e']^A(E)\text{[return } \leftarrow \text{undef}]] = (\vec{F}_1 \ldots \vec{F}_j), \tilde{\Sigma}) \) the following holds:

\[ [e'](E') \in \{[[e']^A(E)\text{(return)}] \]

and

\[ EU(e', E') \in \{[[e']^A(E)\text{[return } \leftarrow \text{undef}]] \}
\]

This implies

\[ [e](E') = \emptyset \in \{\emptyset\} = \{[[e]^A(E)\text{(return)}] \]

independently of \( \tilde{\Sigma} \) and (with considering the frame state as an ordinary frame)

\[
EU(e)(E') \in \{(f_1[s \mapsto v] f_2 \ldots f_j) \mid (f_1 \ldots f_j) = EU(e', E'), v = [e'](E') \} \subseteq \\
\subseteq \{(f_1[s \mapsto v] f_2 \ldots f_j) \mid f_i \in \{F_i\}, v \in \{[e'](E') \mid E' \in \{E\}\} \} \subseteq \\
\subseteq \{(f_1[s \mapsto v] f_2 \ldots f_j) \mid f_i \in \{\vec{F}_i\}, v \in \{[[e']^A(E)\text{(return)}] \} = \\
= \{((\vec{F}_1[s \mapsto [e']^A(E)\text{(return)}] \vec{F}_2 \ldots \vec{F}_j), \tilde{\Sigma}) \}
= \{[[e]^A(E)\text{[return } \leftarrow \text{undef}]] \}.
\]
**if** expressions

Let \( e = (\text{if } e' \ e_t \ e_f) \) be an *if* expression: Again \( e_A = e \) because of the compatibility of \( e_A \) and \( e \). Since Rule \((\text{Std})\) of Def. 6.3.36 processes every simple abstract environment in \( S(\bar{E}) \) and combines the results correctly (Lemma 6.2.51 states this for *combine* and Lemma 6.2.53 for *compress*) we just need to prove the statement of the lemma for simple abstract environments \( \bar{E} \in S(E) \). The induction hypothesis holds for \( e' \), \( e_t \) and \( e_f \).

According to the rules of Def. 6.3.37 we distinguish the following cases:

- **Case 1** \( AT(\llbracket e' \rrbracket_A^3(\bar{E})(\text{return})) \) holds: Rule \((\text{If-True})\) in Def. 6.3.37 applies for the abstract semantics and because of \( \text{bi}(v) = \text{true} \) for all \( v \in \llbracket e' \rrbracket_A^3(\bar{E})(\text{return}) \) Rule \((\text{If-True})\) in Def. 5.2.16 and Def. 5.2.15 applies for the standard semantics.

  The result value fulfills
  \[
  \llbracket e \rrbracket (E') = \llbracket e_t \rrbracket (EU(e', E')) \in \\
  \in \{ \llbracket e_t \rrbracket (E'') \mid E'' \in \llbracket e' \rrbracket_A^4(\bar{E}) \} \\
  \subseteq \llbracket e_t \rrbracket_A^4(\llbracket e' \rrbracket_A^4(\bar{E}))(\text{return}) = \\
  = \llbracket e \rrbracket_A^4(\bar{E})(\text{return})
  \]

  and for the environment update we get
  \[
  EU(e, E') = EU(e_t, EU(e', E')) \in \\
  \in \{ EU(e_t, E'') \mid E'' \in \llbracket e' \rrbracket_A^4(\bar{E}) \} \subseteq \\
  \subseteq \llbracket e_t \rrbracket_A^4([e']_A^4(\bar{E})[\text{return} \leftarrow \text{undef}]) = \\
  = \llbracket e \rrbracket_A^4(\bar{E})[\text{return} \leftarrow \text{undef}].
  \]

- **Case 2** \( AF(\llbracket e' \rrbracket_A^3(\bar{E})(\text{return})) \): This case is analogous to Case 1 with \( e_t \) replaced by \( e_f \) and \((\text{If-True})\) replaced by \((\text{If-False})\).

- **Case 3** \( \neg AT(\llbracket e' \rrbracket_A^3(\bar{E})(\text{return})) \), \( \neg AF(\llbracket e' \rrbracket_A^3(\bar{E})(\text{return})) \): Consider a value \( v = \llbracket e \rrbracket (E') \) with \( E' \in \{ \bar{E} \} \). We prove \( v \in \llbracket e \rrbracket_A^4(\bar{E}) \) by case distinction on the result of \( \llbracket e' \rrbracket (E') \):
Case 3.1 $bi([e'](E')) = true$:

\[
\begin{align*}
v &= [e_1](EU(e', E')) \\
&\in \{[e_i](EU(e', E'')) | E'' \in \{\hat{E}\}\} \subseteq \\
&\subseteq \{[e_i](E'') | E'' \in \{[e']^A(\hat{E})\}\} \subseteq \\
&\subseteq \{[e_i]^A([e']^A(\hat{E}))\} \subseteq \\
&\subseteq \{[e]^A(\hat{E})\}.
\end{align*}
\]

The last inclusion holds because according to rule (If) of Def. 6.3.37 $\{[e]^A(\hat{E})\}$ contains all results from evaluating $e_t$ and $e_f$.

Now consider the result environment of evaluating the if-condition in $E' \in \{\hat{E}\}$:

\[
\begin{align*}EU(e, E') &= EU(e_t, EU(e', E')) \\
&\in \{EU(e_t, E'') | E'' \in \{[e']^A(\hat{E})\}\} \subseteq \\
&\subseteq \{[e_i]^A([e']^A(\hat{E}))[\text{return} \leftarrow \text{undef}]\} \subseteq \\
&\subseteq \{\text{compress(combine([e_i]^A([e']^A(\hat{E})), [e_j]^A([e']^A(\hat{E})))}[\text{return} \leftarrow \text{undef}]\} = \\
&= \{[e]^A(\hat{E})\}.
\end{align*}
\]

Case 3.2 $bi([e'](E')) = false$: This case is analogous to Case 1 with $e_t$ replaced by $e_f$.

This finishes the proof for if expressions.

quote expression

Let $e = (quote e')$ be a quote expression. Because of the compatibility of $e_A$ and $e$ we have $e = e_A$.

Because of the Rules (Quote) in Def. 5.2.23 and Def. 6.3.36 we have to consider the definitions of $quotesem$ and $quotesem_A$ for the different cases of $e'$:

- $e' \in \text{ConstT}$: $quotesem(e') = \text{denoteconst}(e') \in \{A(\text{denoteconst}(e'))\} = \text{quotesem}_A(e')$.
- $e' \in \text{Sym}$: $quotesem(e') = \text{sym-to-const}(e') \in \{A(\text{sym-to-const}(e'))\} = \text{quotesem}_A(e')$.
- $e' = (e_1 \ldots e_k)$: Since make-list and make-list$_A$ create corresponding list structures in the standard and abstract domain the statement follows from induction on the number $k$ of subexpressions of $e'$.

From these cases we get $[e](E') \in \{[e]^A(\hat{E})\}$.

Environment updates are not performed by quote expressions, i.e.

\[
EU(e, E') = E' \in \{E\} = \{\{e\}^A(\hat{E})[\text{return} \leftarrow \text{undef}]\}.
\]

Altogether, all cases of expressions are considered and the lemma is proven. \qed
C.3 Proofs of Section 6.4

**Theorem 6.4.16** Let \( e \in \text{Expression} \) and \( e_A \in \text{Expression}_A \) such that \( e_A \) is compatible with \( e \) and let \( E = (L, \Sigma) \in \text{Env}_A \) such that \( [e_A]^A(E) \) is free of eval abstraction. Suppose that \( [e_A]^A(E) = \tilde{E} = (\tilde{L}, \tilde{\Sigma}) \) can be calculated without an infinite loop. Then for every \( E' \in \{\sigma \circ \sigma(L)\} \) for a \( \sigma \in \Sigma \) and a closed substitution \( \tau \) appropriate for \( \sigma(L) \) with \( [e](E') \notin \{\bot, \text{error}\} \) there is some \( \tilde{\sigma} \in \Sigma \) and some \( \tilde{\tau} \) appropriate for \( \tilde{\sigma}(\tilde{L}) \) such that

\[ [e](E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L} | \text{return})\} \]

and

\[ EU(e, E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L} | \text{return} \leftarrow \text{undef})\}. \]

**Proof of Theorem 6.4.16:** There is nothing to prove for those rules that did not change from Def. 6.3.36 to Def. 6.4.14. These rules do not change any variable bindings and the proof can be done completely analogous to Lemma 6.3.41 with setting \( \tilde{\tau} = \tau \) and \( \tilde{\sigma} = \sigma \). Thus, the only rule that has to be proven correct is \( (\text{AppV}) \).

As for Lemma 6.3.41 we prove the theorem by induction on the maximal depth \( k \) of recursive calls to \( [\cdot]^A(\cdot) \):

**k = 1:** Rules returning an answer without recursive calls are \( (\text{Const}), (\text{AbsConst}), (\text{Inst}), (\text{InstErr}), (\text{PFunc}) \) and \( (\text{Lambda}) \). As stated before these rules do not restrict any variables and can therefore be proven correct analogously to the proof of Lemma 6.3.41.

**k \to k + 1:** The rules \( (\text{If-True}), (\text{If-False}), (\text{If}), (\text{Begin-1}), (\text{Begin-k}) \) and \( (\text{Def}) \) are proven analogously to the proof of Lemma 6.3.41. The remaining rule is \( (\text{AppV}) \) which is just applied when evaluating applications.

Let \( e_A = (e_A^0, e_A^1, \ldots, e_A^n) \), \( e = (e^0, e^1, \ldots, e^n) \), and \( \tilde{E} = (\tilde{L}, \tilde{\Sigma}) := \text{pre-app}(E, e_A) \). We prove the lemma for Rule \( (\text{AppV}) \) for every simple abstract environment \( E = (L, \sigma) \in S(\tilde{E}) \) (with \( \sigma \in \Sigma \)). Let \( v_i = [e^i](E'_i) \) where \( E'_0 = E' \) and \( E'_{i+1} = EU(e^i, E'_i) \). The induction hypothesis yields the following: There exists a \( \rho \) such that

\[ v_0 \in \{\rho(\tilde{E}(\text{func}))\} \]

\[ v_i \in \{\rho(\tilde{E}(\text{arg}_i))\} \text{ for } i \in \{1, \ldots, n\} \]

\[ E'' := E'_{n+1} \in \{\text{clearapp}(\tilde{E})\}. \]

We give the proof by case distinction on the binding \( \tilde{E}(\text{func}) \):

1. \( X = \tilde{E}(\text{func}) \in V_f \)
2. \( \bar{E}(\text{func}) \in \text{pip}\)
3. \( \bar{E}(\text{func}) \in \text{LC}_A \)
4. \( \bar{E}(\text{func}) \in \text{PFunc}_A \)
5. \( \bar{E}(\text{func}) \) is not an applicable abstract value.

Note that an application of \( \text{apply-strict}_V \) implies the result \( \bot \) in the standard semantics and we therefore have nothing to show for \( \bot \in \{ \bar{E}(\text{func}), \bar{E}(\text{arg}_i) \} \).

Case (5) implies \( v_0 \not\in \text{PFunc} \cup \text{LC} \). Therefore, \( v = \text{error} \) and we have nothing to show.

From now on we can assume \( v_0 \in \text{PFunc} \cup \text{LC} \) in the standard semantics because otherwise \( v = \text{error} \) in contradiction to the prerequisites of the theorem.

An abstract function application does not change the environment it is called in except of binding \text{return} and restricting variables.\(^5\) Thus, for the statement

\[
E'' = EU(e, E') \Rightarrow E'' \in \langle \tilde{\tau} \circ \tilde{\sigma}(\tilde{L})[\text{return} \leftarrow \text{undef}] \rangle
\]

of the theorem just the restriction of variables is of interest. When a variable with the same index occurs more than once in an environment then restricting one of the occurrences affects all other occurrences of the same indexed variable in the identical way. This is correct because of the following fact: When an indexed variable occurs more than once in an environment it always denotes the same value in the standard semantics. Thus, it does not violate the theorem when restricting one occurrence of an indexed variable affects its other occurrences in the same way.

The correctness of the value bound to \text{return} is now proven for the individual cases:

In Case (1) the free variable \( X \) can be restricted to a type just containing functions. This is done by restricting it to a PI/PO-representation. By the prerequisite \( v \neq \text{error} \) of the theorem we can assume that the function \( v_0 \) is applicable to \( v_1 \times \ldots \times v_n \) in the standard semantics. Because of \( v_i \in \langle \rho(E(\text{arg}_i)) \rangle \) (from the induction hypothesis) restricting \( X \) to a PI/PO-representation \( \{ E(\text{arg}_1) \times \ldots \times E(\text{arg}_n) \rightarrow B \} \) does not rule out \( v_0 \). Note that the output type \( B \) does not restrict the output of \( v_0 \) at the moment. A later restriction of \( B \) is valid because:

1. The restriction of \( B \) can just be done when the output value of the application occurs as input to another function.

\(^5\)Note that this is just the case as long as no functions with side effects are considered. We can assume this here because we will not drop the restriction to functions without side effects in this work. The change of frame states is just straightforward and not considered here in detail.
2. The output value calculated in the standard semantics must be a valid input to the function mentioned in (1) because otherwise \( v = \text{error} \) in contradiction to the theorems prerequisite.

In case (2) of the possible bindings of \( \bar{E}(\text{func}) \), i.e. \( \bar{E}(\text{func}) \in \text{pipo} \), the argumentation is analogous to case (1): the PI/PO-representation pair introduced in the current step does not violate the lemma because otherwise the current call yields \( v = \text{error} \). The former representation pairs are also valid because the same argumentation holds for the calls they were introduced for.

Case (3), i.e. \( \bar{E}(\text{func}) \in \text{LC}_{A} \), does not introduce restrictions itself, but just passes through restrictions given by evaluating the lambda closure’s body. This is correct even for additional occurrences of a variable that were just indirectly affected by the restriction of another variable occurrence as states above.

In case (4), i.e. \( \bar{E}(\text{func}) \in \text{PFunc}_{A} \), we first analyze those predefined functions having a special semantics. The functions usually bound to \textit{eval} and \textit{interaction-environment} both behave like in the proof of Lemma 6.3.41 except of the following specialty: Besides setting the \textit{return}-binding the rule \textit{(Eval)} passes through the substitutions containing the restrictions of the performed sub-evaluation. The argument that several occurrences of the same indexed variable must be restricted identically provides the correctness of this change. The behaviour of \textit{valueexpression} \(_{A}\) of returning \textit{error} for abstract values containing variables does not occur because of the prerequisite on the first argument of \( f_{\text{eval}}^{A} \) (which must be the result of evaluating an expression \( \text{(quote \ldots)} \in \text{Expression} \) in the theorem.

For the predefined functions with non-special semantics there exists an \( f \in \text{PFunc}_{A} \) with an I/O-representation \( D(f) = \{(\text{IN}_i, \text{OUT}_i)\} \) such that the only element \( v_0 \in \{f\} \) fulfills

\[
\begin{align*}
\bullet & \quad \text{dom}(v_0) \subseteq \bigcup_i \{\text{IN}_i\} \\
\bullet & \quad \forall_i v_0(\{\text{IN}_i\}) \subseteq \{\text{OUT}_i\} \cup \{\text{error}\}.
\end{align*}
\]

(cf. Assump. 6.2.42)

Furthermore, we have

\[
v_1 \times \ldots \times v_n \in \{\rho(\bar{E}(\text{arg}_1)) \times \ldots \times \rho(\bar{E}(\text{arg}_n))\}
\]

for some ground substitution \( \rho \) appropriate for \( \bar{E}(\text{arg}_1) \times \ldots \times \bar{E}(\text{arg}_n) \) from the induction hypothesis and

\[
v_1 \times \ldots \times v_n \in \{\rho'(\text{IN}_i)\}
\]
for some I/O-representation-pair $IN_i \rightarrow OUT_i \in D(f)$ and some ground substitution $\rho'$ appropriate for $IN_i$.

By renaming the variables occurring in $IN_i \rightarrow OUT_i$ we can make the sets of variables in $ar{E}(\text{arg}_1) \times \ldots \times ar{E}(\text{arg}_n)$ and $IN_i$ disjoint. Then there exists a substitution $\rho_{\text{complete}}$ with:

$$\rho_{\text{complete}}(ar{E}(\text{arg}_1) \times \ldots \times ar{E}(\text{arg}_n)) = \rho(\bar{E}(\text{arg}_1)) \times \ldots \times \rho(\bar{E}(\text{arg}_n))$$

and

$$\rho_{\text{complete}}(IN_i) = \rho'(IN_i).$$

Thus, by Theorem 4.3.17

$$\Sigma = CE(IN_i, \bar{E}(\text{arg}_1) \times \ldots \times \bar{E}(\text{arg}_n))$$

contains a substitution $\sigma \in \Sigma$ such that there is a further substitution $\tau$ with

$$v_1 \times \ldots \times v_n \in \llbracket \tau \circ \sigma((\cap \text{IN}_i (\bar{E}(\text{arg}_1) \times \ldots \times \bar{E}(\text{arg}_n)))) \rrbracket$$

By the definition of $PPAF_V$ the result of $CEP_V$ contains a triple $(t, s, \sigma)$ with $t = IN_i$ and $s = OUT_i$.

Now $PPAF_V(f, \bar{E}(\text{arg}_1) \times \ldots \times \bar{E}(\text{arg}_n), \sigma)$ contains a pair $(s, \sigma)^6$ with $v \in \llbracket \tau \circ \sigma(s) \rrbracket$ for some $\tau$. This output of $PPAF_V$ leads to a $\bar{E}' \in S(\bar{E})$ with $\bar{E}'(\text{return}) = \sigma(s)$ and therefore $v \in \tau(\bar{E}'(\text{return})) = \tau \circ \sigma(s)$.

Altogether, we have proven the correctness of Rule $(AppV)$ which implies the correctness of the theorem. \hfill $\Box$

### C.4 Proofs of Section 6.5

**Lemma 6.5.12** The algorithm $rec-gen$ terminates for every input.

**Proof of lemma 6.5.12:** The termination of the steps (2) and (3) is obvious. For step (1) the algorithm is not given in detail. But since the input list $l$ is of finite length the number of possible partitions of $l$ is also finite. Therefore even a naive algorithm generating all possible sets of lists and checking them for the wanted properties terminates. \hfill $\Box$

---

6If the triple $(s, t, \sigma)$ was deleted by case (2) of the definition of $PPAF_V$ then there is a triple $(t', s', \sigma')$ in the result of $CEP_V$ such that $\sigma'(t')$ meets the abstract input value $(\bar{E}(\text{arg}_1) \times \ldots \times \bar{E}(\text{arg}_n))$ more precisely and we can consider $(s', \sigma')$ instead of $(s, \sigma)$. 
Lemma 6.5.13 Let \( l = (t_1 \ t_2 \ldots \ t_n) \) be the input of rec-gen. There exists a closed substitution \( \sigma \) appropriate for every \( t_i \) and for \( t \) such that \( t = \text{rec-gen}(l) \) fulfills \( \llbracket \sigma(t_i) \rrbracket \subseteq \llbracket \sigma(t) \rrbracket \) for every \( i \in \{1, \ldots, n\} \).

Proof of lemma 6.5.13: Choose an arbitrary \( i \in \{1, \ldots, n\} \) and let \( l_j \) be the list containing \( t_i \). We prove \( \llbracket \sigma(t_i) \rrbracket \subseteq \llbracket \sigma(t_j) \rrbracket \). Because of \( \llbracket \sigma(t_j) \rrbracket \subseteq \llbracket \sigma(t) \rrbracket \) from the definition of \( t \) this directly implies the statement.

If \( l_j \) just contains one element then \( \tilde{t}_j = t_i \) and we have nothing to show.

For \( l_j \) containing more than one element we show \( \llbracket \sigma(t_j) \rrbracket \in \llbracket \sigma(t) \rrbracket \) by induction on \( k \):

\[ k = 1: \text{ Since } \text{unfold}(\tilde{t}_j) = (\cup \ \tilde{t}_j, t') \text{ for some } t' \text{ the statement is obvious. (The variable renaming does not cause problems.) This is obvious e.g. with } \sigma \text{ instantiating every variable with } \top. \]

\[ k \rightarrow k + 1: \text{ Let the statement hold for } k. \text{ We do not consider the variable renaming performed by } \text{rec-gen}, \text{ because it does not cause any problems as stated above. } \text{unfold}(t_j) = (\cup \ t_{j_1} t'_{j_{k+1}}[p_j|\tilde{t}_j]). \text{ Since } t_{j_{k+1}} = t'_{j_{k+1}}[p_i|t_{j_k}], \llbracket t_{j_k} \rrbracket \subseteq \llbracket \tilde{t}_j \rrbracket \text{ and the type language just contains monotonic type constructors we have } \llbracket t_{j_{k+1}} \rrbracket \subseteq \llbracket \tilde{t}_j \rrbracket. \]

Theorem 6.5.15 The \( \lambda \)-iterative abstract semantics defined in Sec. 6.5.1 with the termination enforcement methods from Sec. 6.5.2 terminates for every input.

Proof of theorem 6.5.15: From inspecting the individual rules in defining the abstract semantics we can see that every rule terminates when calculation of the needed subexpressions terminates. Non-termination is therefore just possible when at least one expression in the program is evaluated an infinite number of times. The only way of repeating the evaluation of an expression by evaluating its subexpressions is the call to a user defined recursive function. Non-termination hence means the calling of at least one user defined function an infinite number of times.

By using \( \text{apply-lambda} \) in the \( \lambda \)-iterative abstract semantics processing recursive calls to a function with the same arguments as before is changed into iterated type inference. Furthermore, an infinite number of calls to a function with always changing arguments is possible. These two cases are the only reason for an abstract evaluation not to terminate.

None of the methods for termination enforcement does not terminate itself. (For the function \( \text{rec-gen} \) used in the case of structured input types this is proven in Lemma 6.5.12.) All cases for non-termination are detected by the thresholds introduced for structured types. This enforces termination in every case even if all the other methods of termination enforcement fail. \[ \square \]
Theorem 6.5.16 Let $e \in \text{Expression}$ and $e_A \in \text{Expression}_A$ such that $e_A$ is compatible with $e$ and let $E = (L, \Sigma) \in \text{Env}_A$ such that $[e_A]^A(E)$ is free of eval abstraction.

Then for every $E' \in \{\tau \circ \sigma(L)\}$ for a $\sigma \in \Sigma$ and a closed substitution $\tau$ appropriate for $\sigma(L)$ with $[e](E') \notin \{\bot, \text{error}\}$ there is some $\tilde{\sigma} \in \tilde{\Sigma}$ and some $\tilde{\tau}$ appropriate for $\tilde{\sigma}(\tilde{L})$ such that

$$[e](E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L}(\text{return}))\}$$

and

$$EU(e, E') \in \{\tilde{\tau} \circ \tilde{\sigma}(\tilde{L})[\text{return} \leftarrow \text{undef}]\}.$$ 

Proof of theorem 6.5.16: According to Theorem 6.5.15 the theorem cannot be violated because of non-termination. For those cases where $\tilde{E}$ is calculated without using iterated type inference the statement is already given by Theorem 6.4.16.

Now let $v$ be calculated using iterated type inference. Without loss of generality the only recursive function used in the calculation of $v$ is $f$. (If there are several such functions induction on the number $m$ of them yields the wanted result.)

Furthermore, without loss of generality $f$ is the main function with respect to $e$, i.e. $e$ is a call to the lambda closure $f$. (Otherwise, there is a first call to $f$. The actions taken before and after evaluating this call do not incorporate iterated type inference and therefore do not violate the theorem according to Theorem 6.4.16.)

First we prove the $EU$ statement for $\text{apply-lambda}_I$ extended by the termination enforcement methods by considering the two ways of changing an abstract environment (introduction of new symbol bindings in the frame list and updating substitutions):

The introduction of symbol bindings by evaluating of $\text{define}$-statements in the body of the lambda closure $f$ just effects local frames, but not the frames occurring in $E''$ and $\tilde{L}$.

Instantiation of variables during normal application of the body of a lambda closure is already proven correct in Theorem 6.4.16 and directly carries over to $\text{apply-lambda}_I$. The only remaining possibility for changing the abstract environment is the restriction of variables occurring in an input type by a termination enforcement method:

- The restriction to $\top$ cannot cause any problems.
- When restricting variables occurring in an input type to a recursive type by identifying input types of successive calls to $f$ the restriction is not stronger than necessary and does not violate the theorem. This is easily proven by induction on the number of recursive calls and the corresponding number of unfolding steps of the recursive type. The proof is straightforward and we omit it here.
• processing a list of different output types that were generated successively by rec-gen is correct according to Lemma 6.5.13.

The environment update function is therefore approximated correctly by the abstract semantics.

In the rest of the proof, we just need to show the first statement

\[ v \in \langle \tilde{\tau} \circ \tilde{\sigma}(\tilde{L}(\text{return})) \rangle \text{ for some } \tilde{\sigma} \in \tilde{\Sigma} \text{ and a closed substitution } \tilde{\tau} \text{ appropriate for } \tilde{\sigma}(\tilde{L}). \]

For the iterated type inference without termination enforcement we prove the following statement: If the function \( f \) is called \( n \) times during calculation of \( v \) (including the initial call) then the \( n \text{th} \) intermediate abstract environment \( E_n = (L_n, \Sigma_n) \) (\( E_i \) denotes the environment after finishing the \( i \text{th} \) call) of iterated type inference fulfills

\[ v \in \langle \tilde{\tau} \circ \sigma_n(L_n(\text{return})) \rangle \]

for some \( \sigma_n \in \Sigma_n \) and a closed substitution \( \tilde{\tau} \) appropriate for \( \sigma_n(L_n) \).

(If there are less than \( n \) iteration steps then the last intermediate abstract environment \( E_f \) (that is returned as result environment) is equal to all \( E_{n'} \) with \( n' > f \). Therefore, the proof works even if \( E_n \) is never calculated.)

\( n = 1 \): No recursive call to \( f \) is needed. \( E_1 \) is calculated as if there where no recursive call to \( f \) and therefore Theorem 6.4.16 provides the statement.

\( n \rightarrow n + 1 \): Let the statement hold for \( n \) necessary calls to \( f \) and consider an expression \( e \) that is evaluated to \( v \) incorporating \( n + 1 \) calls to \( f \). The first recursive call to \( f \) is given by an expression \( e' \) in the standard semantics. For \( e' \) and the result value \( v' \) the statement holds, i.e. if \( (L'_n, \Sigma'_n) \) is the \( n \text{th} \) intermediate environment when evaluating \( e' \) in the abstract semantics then

\[ v' \in \langle \tilde{\tau}' \circ \sigma'_n(L'_n(\text{return})) \rangle \]

for some \( \sigma'_n \in \Sigma'_n \) and a closed substitution \( \tilde{\tau}' \) appropriate for \( \sigma'_n(L'_n) \).

Since the recursive call in the \( \lambda \)-iterative abstract semantics returns \( \sigma'_n(L'_n(\text{return})) \) (or a type denoting at least the same set of values) for the recursive call performed during the \( n + 1 \text{th} \) iteration the rest of the calculation of \( L_{n+1} \) in the abstract semantics (that does not incorporate any further iterated type inference) is already proven in Theorem 6.4.16 and thus the statement holds for \( n + 1 \).

As a last step we show that the termination enforcement tasks described in Sec. 6.5.2 always change an (intermediate) output type \( t \) to another type \( t' \) such that there exists a substitution \( \rho \) with \( \llbracket t \rrbracket(\rho) \subseteq \llbracket t' \rrbracket(\rho) \):
• If \( t = \mathcal{A}(v) \) is a value ascription and the threshold for the number of calls with value ascriptions in the arguments is exceeded then \( t' \) is a base type with \( v \in \langle t' \rangle \) and hence \( \langle t \rangle \subseteq \langle t' \rangle \) independently from a substitution \( \rho \).

• For the termination enforcement for structured types the statement is obvious for \( t' = \top \).

• Termination enforcement by \textit{rec-gen} fulfills the statement because of Lemma 6.5.13.

• For introducing variables for output types the following holds: A call to a function \( f \) with a fixed argument tuple always yields the same result independently of the position of the call and the number of identical calls performed before. (This is the case because we do not consider side effects.) Thus, it is correct to denote the results of identical calls by the same variable. The correctness of the recursive binding used for the output type can be proven by induction on the number of recursive calls to calculate a value \( v \).

Altogether, the first statement of the theorem and hence the whole theorem is proven. \( \square \)
Appendix D

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]).

D.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 $STbase : \mathcal{B} \times \mathcal{B} \rightarrow \{\text{true}, \text{false}\}$. For base types $b_1$ and $b_2$ from different partition sets $STbase(b_1, b_2) = \text{false}$ always holds. (We will just introduce the cases that cause the result $\text{true}$ for $STbase(b_1, b_2)$ which can be given for the different partition sets separately.) Furthermore we define the function $CEbase : \mathcal{B} \times \mathcal{B} \rightarrow \{\text{true}, \text{false}\}$ 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 $\bot$. 277
D.1.1 The Boolean Types

There are three boolean types $\text{Ttrue}$, $\text{Tfalse}$ and $\text{bool}$ where $\text{Ttrue}$ just contains the value $\#t$, $\text{Tfalse}$ contains the value $\#f$ and $\text{bool}$ contains both of these values.

Besides the trivial cases of two equal arguments the type hierarchy on these types is defined by:

- $STbase(\text{Ttrue}, \text{bool})$
- $STbase(\text{Tfalse}, \text{bool})$

$CEbase$ on the boolean partition set can be given in terms of $STbase$ as follows:

$$STbase(b_1, \text{bool}), STbase(b_2, \text{bool}) \Rightarrow$$

$$CEbase(b_1, b_2) = \text{true} \iff b_1 = b_2 \lor b_1 = \text{bool} \lor b_2 = \text{bool}.$$

Note that $\text{Ttrue}$ and $\text{Tfalse}$ are just used for external representation of types. Internally the value assignments $A(\#t)$ and $A(\#f)$ are used instead.

D.1.2 The Symbol Types

The type containing all symbols is denoted by $\text{sym}$. It has just the trivial subtype and common element properties $STbase(\text{sym}, \text{sym})$ and $CEbase(\text{sym}, \text{sym}) = \text{true}$.

D.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:

- $\text{posint-e}$ (exactly represented positive integers)
- $\text{posint-i}$ (inexactly represented positive integers)
- $\text{zero-e}$ (0 in exact representation)
- $\text{zero-i}$ (0 in inexact representation)
- $\text{negint-e}$ (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 $\langle xyz \rangle := \langle xyz-e \rangle \cup \langle xyz-i \rangle$ and the type hierarchy function is extended by $STbase(\text{xyz-e}, \text{xyz}) = \text{true}$ and $STbase(\text{xyz-i}, \text{xyz}) = \text{true}$. 

279
• 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 \( \{abc\} = \{posabc\} \cup \{\text{zero}\} \cup \{negabc\} \) and the following extensions to \( STbase \):

- \( STbase(posabc, abc) = true \)
- \( STbase(negabc, abc) = true \)
- \( STbase(zero*, abc) = true \)

\( zero* \) 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

\[
\{sigrat*\} := \{sigin*\} \cup \{sigrat-no-int*\}
\]

and the type hierarchy is extended by

- \( STbase(sigin*, sigrat*) = true \)
- \( STbase(sigrat-no-int*, sigrat*) = true \)

• Types of real numbers are of the form \( sigreal* \) with \( sig \) and \( * \) as above. The semantics is given by

\[
\{sigreal*\} := \{sigrat*\} \cup \{sigreal-no-rat*\}
\]

and the type hierarchy is extended by

- \( STbase(sigrat*, sigreal*) = true \)
- \( STbase(sigreal-no-rat*, sigreal*) = true \)

• Types of complex numbers are of the form \( comp* \) with \( * \) as above. The semantics is given by \( \{comp*\} := \{real*\} \cup \{comp-no-real*\} \) and the type hierarchy is extended by

- \( STbase(real*, comp*) = true \)
- \( STbase(comp-no-real*, comp*) = true \)

• Further types of general numbers have the form \( num* \) with \( * \) as above, the semantics given by \( \{num*\} := \{comp*\} \cup \{num-no-comp*\} \) and the type hierarchy extended by
\(- STbase(comp*, num*) = true\)
\(- STbase(num-no-comp*, num*) = true\)

The function \(CEbase\) on the number types can now be defined in terms of \(STbase\) by:

\[
STbase(b_1, num), STbase(b_2, num) \Rightarrow
CEbase(b_1, b_2) = true \iff \exists b \neq \bot . STbase(b, b_1) \land STbase(b, b_2).
\]

D.1.4 The Character Types

The type of all characters is denoted by \texttt{char}. There are the following subtypes of characters:

- \texttt{num-char} denotes the type of all characters denoting digits.
- \texttt{whitespace-char} is the type of all whitespace characters.
- With \texttt{upper-char} we denote the type of all uppercase alphabetic characters.
- \texttt{lower-char} is the type of all lowercase alphabetic characters.
- \texttt{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 \texttt{char}.

\(CEbase\) is defined by

\[
STbase(b_1, \texttt{char}), STbase(b_2, \texttt{char}) \Rightarrow
CEbase(b_1, b_2) = true \iff b_1 = b_2 \lor b_1 = \texttt{char} \lor b_2 = \texttt{char}.
\]

D.1.5 Input/Output and Ports

There are two types of ports: \texttt{in-port} is the type of all input type and \texttt{out-port} the type of all output ports. The supertype containing all input and output ports is denoted by \texttt{port}. Besides the trivial cases \(STbase\) is defined by:

- \(STbase(\texttt{in-port}, \texttt{port}) = true\).
• \( ST_{base}(\text{out-port, port}) = \text{true} \).

\( CE_{base} \) is defined by:

\[
ST_{base}(p_1, \text{port}), ST_{base}(p_2, \text{port}) \Rightarrow \\
CE_{base}(p_1, p_2) = \text{true} \iff p_1 = p_2 \lor p_1 = \text{port} \lor p_2 = \text{port} .
\]

The type \( \text{eof} \) contains exactly those objects that can occur as end of file object. \( ST_{base} \) and \( CE_{base} \) are defined in the trivial way.

### D.1.6 \( ST_{base} \) and \( CE_{base} \) on value assignments

The functions \( ST_{base} \) and \( CE_{base} \) 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 \( t_1 = A(c_1), t_2 = A(c_2) \).

  \[
  ST_{base}(t_1, t_2) = \text{true} \iff CE_{base}(t_1, t_2) = \text{true} \iff c_1 = c_2 .
  \]

- If one of the arguments is a value assignment \( A(c) \) and the other one is a type \( t \) the result depends on the test \( c \in [t] \):

  \[
  ST_{base}(A(c), t) = \text{true} \iff CE_{base}(A(c), t) = \text{true} \iff CE_{base}(t, A(c)) = \text{true} \iff c \in [t] .
  \]

### D.1.7 Correctness of \( ST_{base} \) and \( CE_{base} \)

**Lemma D.1.1 (termination and correctness of \( ST_{base} \))** Let \( b_1 \) and \( b_2 \) be base types or value assignments. Then

\[
ST_{base}(b_1, b_2) = \text{true} \Rightarrow [b_1] \subseteq [b_2] .
\]

**Proof:** The lemma can be proven along the definition of \( ST_{base} \):

- For boolean types we obviously have:

  \[
  [\text{true}] = \{ \bot, \#t \} \subseteq \{ \bot, \#t, \#f \} = [\text{bool}]
  \]

  and

  \[
  [\text{false}] = \{ \bot, \#f \} \subseteq \{ \bot, \#t, \#f \} = [\text{bool}]
  \]
For the symbol type nothing has to be proven.

The definitions of \textit{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 \textit{STbase} on all number types. E.g. int-e is defined by \{int-e\} = \{posint-e\} \cup \{negint-e\} \cup \{zero-e\}. This definition directly gives a correctness proof for the following corresponding definitions of \textit{STbase}:

- \textit{STbase}(posint-e, int-e) = true.
- \textit{STbase}(negint-e, int-e) = true.
- \textit{STbase}(zero-e, int-e) = true.

The definition of \textit{STbase} for character types is obviously correct because of

\{char\} = \{num-char\} \cup \{whitespace-char\} \cup \{upper-char\} \cup \{lower-char\} \cup \{sym-char\}.

D.2 Type Constructors

This section introduces the tuple like type constructors that can be processed by our type checker.

D.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 \(\mu X. (\cup nil (A . X))\) where \textit{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 \textit{list} defined independently from \textit{pair} the type constructors are no longer free.
D.2.2 Strings

For every type \( A \) with \( STbase(A, \text{char}) \) 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 \( (string\ \text{char}) \).

D.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 \( (vector\ \top) \).
Appendix E

Abstract Semantics of Predefined Scheme Functions

This appendix presents the predefined function definitions of Scheme according to the official language report [KCE98] with their abstract semantics. The semantic notation is based on Def. 6.2.3 in Sec. 6.2.1. Instead of writing pairs of corresponding abstract input and output values as pairs $(IN_i, OUT_i)$ we use the notation $IN_i \rightarrow OUT_i$.

In the following the function definitions are named by the symbols they are initially bound to according to [KCE98]. Formally, in the context of abstract interpretation for every symbol $<xyz>$ bound to a predefined function this function is called $f^{A}_{<xyz>}$. For the list type $\mu X.(\cup \text{nil}(A . X))$ for an arbitrary element type $A$ we use the abbreviation $(\text{list } A)$.

E.1 Boolean Functions

\begin{verbatim}
not : Ttrue \rightarrow Tf\ell se
    Tf\ell se \rightarrow Ttrue

boolean? : bool \rightarrow Ttrue
            \top \setminus \text{bool} \rightarrow Tf\ell se
\end{verbatim}

285
E.2 Equivalence predicates

\[ \text{eqv?}, \text{eq?}, \text{equal?} : \top \times \top \rightarrow \text{bool} \]

E.3 Pairs and Lists

\[ \text{pair?} : (\top \times \top) \rightarrow \text{True} \]
\[ \top \setminus (\top \times \top) \rightarrow \text{False} \]

\[ \text{cons} : A \times B \rightarrow (A . B) \]

\[ \text{car} : (A . \top) \rightarrow A \]

\[ \text{cdr} : (\top . A) \rightarrow A \]

\[ \text{caar, cadr, ...} \text{ need no special typings because these can be generated by typing the corresponding expression using } \text{car} \text{ and } \text{cdr.} \]

\[ \text{null?} : \text{nil} \rightarrow \text{True} \]
\[ \top \setminus \text{nil} \rightarrow \text{False} \]

\[ \text{list?} : (\text{list } \top) \rightarrow \text{True} \]
\[ \top \setminus (\text{list } \top) \rightarrow \text{False} \]

\[ \text{list} : A^{k \geq 0} \rightarrow (\text{list } A) \]

\[ \text{length} : (\text{list } \top) \rightarrow \text{zero-e } \cup \text{posint-e} \]

\[ \text{append} : \text{nil} \rightarrow (\text{list } \bot) \]
\[ (\text{list } A)^{k \geq 1} \rightarrow (\text{list } A) \]

\[ \text{reverse} : (\text{list } A) \rightarrow (\text{list } A) \]

\[ \text{list-tail} : (\text{list } A) \times \text{zero-e } \rightarrow (\text{list } A) \]
\[ (\text{list } A) \times \text{posint-e } \rightarrow (\text{list } A) \]

\[ \text{list-ref} : (\text{list } A) \times \text{zero-e } \rightarrow A \]
\[ (\text{list } A) \times \text{posint-e } \rightarrow A \]

\[ \text{memq}, \text{memv}, \text{member} : \top \times (\text{list } A) \rightarrow (\cup (\text{list } A) \text{ Tfalse}) \]

\[ \text{assq}, \text{assv}, \text{assoc} : \top \times (\text{list } (A . B)) \rightarrow (\cup (\text{list } (A . B)) \text{ Tfalse}) \]
E.4 Symbols

symbol? : sym → Ttrue
   ⊤ \ sym → Tfalse

symbol- > string : sym → string

string- > symbol : string → sym

E.5 Numbers

number? : num → Ttrue
   ⊤ \ num → Tfalse

complex? : comp → Ttrue
   ⊤ \ comp → Tfalse

real? : real → Ttrue
   ⊤ \ real → Tfalse

rational? : rat → Ttrue
   ⊤ \ rat → Tfalse

integer? : int → Ttrue
   ⊤ \ int → Tfalse

exact? : num-e → Ttrue
   num-i → Tfalse

inexact? : num-i → Ttrue
   num-e → Tfalse

=, <, >, <=, >= : O → Ttrue
   num^{k≥1} → bool

zero? : zero → Ttrue
   posreal → Tfalse
   negreal → Tfalse
   num → bool
positive?: posreal → Ttrue
       negreal → Tfalse
       zero → Tfalse

negative?: negreal → Ttrue
             posreal → Tfalse
             zero → Tfalse

odd?, even?: int → bool (One could think of further subtypes for odd and even integers, but there are only few examples where this could be helpful in typing.)

max, min : \( A^k \rightarrow A \) for every number type \( A \).

+ : \( O \rightarrow \text{zero-e} \)
   \( A^k \rightarrow A \) for every number type \( A \).

* : \( A^{i \geq 0} \times \text{zero-e} \times A^{j \geq 0} \rightarrow \text{zero-e} \)
   \( O \rightarrow (\text{value 1}) \)
   posreal × negreal → negreal
   negreal × posreal → negreal
   negreal × negreal → posreal\(^1\)
   \( A^k \rightarrow A \)

for every number type \( A \sqsubseteq \text{posint} \).

−: \( \text{int-e}^k \rightarrow \text{int-e} \)
  \( \text{int}^k \rightarrow \text{int} \)
  \( \text{rat-e}^k \rightarrow \text{rat-e} \)
  \( \text{rat}^k \rightarrow \text{rat} \)
  \( \text{real-e}^k \rightarrow \text{real-e} \)
  \( \text{real}^k \rightarrow \text{real} \)
  \( \text{comp-e}^k \rightarrow \text{comp-e} \)
  \( \text{comp}^k \rightarrow \text{comp} \)
  \( \text{num-e}^k \rightarrow \text{num-e} \)
  \( \text{num}^k \rightarrow \text{num} \)

\(^1\)Typings under the consideration of the signs of the arguments might be useful for arities > 2 as well. They are given straightforward, but since the type language does not provide such considerations for arbitrary arity, there can always just be a finite number of arities with sign consideration.
\[
\begin{align*}
/ : & \quad \text{rat-e} \times \text{posnegrat-e}^{k \geq 1} \rightarrow \text{rat-e} \\
& \quad \text{rat} \times \text{posnegrat}^{k \geq 1} \rightarrow \text{rat} \\
& \quad \text{real-e} \times \text{posnegreal-e}^{k \geq 1} \rightarrow \text{real-e} \\
& \quad \text{real} \times \text{posnegreal}^{k \geq 1} \rightarrow \text{real} \\
& \quad \text{comp-e} \times (\text{comp-e} \setminus \text{zero-e})^{k \geq 1} \rightarrow \text{comp-e} \\
& \quad \text{real} \times (\text{comp} \setminus \text{zero})^{k \geq 1} \rightarrow \text{rat} \\
abs : & \quad \text{zero-e} \rightarrow \text{zero-e} \\
& \quad \text{zero-i} \rightarrow \text{zero-i} \\
& \quad \text{zero} \rightarrow \text{zero} \\
& \quad \text{int-e} \rightarrow \text{posint-e} \\
& \quad \text{int-i} \rightarrow \text{posint-i} \\
& \quad \text{int} \rightarrow \text{posint} \\
& \quad \text{rat-e} \rightarrow \text{posrat-e} \\
& \quad \text{rat-i} \rightarrow \text{posrat-i} \\
& \quad \text{rat} \rightarrow \text{posrat} \\
& \quad \text{real-e} \rightarrow \text{posreal-e} \\
& \quad \text{real-i} \rightarrow \text{posreal-i} \\
& \quad \text{real} \rightarrow \text{posreal} \\
\text{quotient} : & \quad \text{zero-e} \times \text{posnegint-e} \rightarrow \text{zero-e} \\
& \quad \text{zero} \times \text{posnegint} \rightarrow \text{zero} \\
& \quad \text{posint-e} \times \text{posint-e} \rightarrow \text{posint-e} \\
& \quad \text{posint} \times \text{posint} \rightarrow \text{posint} \\
& \quad \text{negint-e} \times \text{negint-e} \rightarrow \text{posint-e} \\
& \quad \text{negint} \times \text{negint} \rightarrow \text{posint} \\
& \quad \text{posint-e} \times \text{negint-e} \rightarrow \text{negint-e} \\
& \quad \text{posint} \times \text{negint} \rightarrow \text{negint} \\
& \quad \text{negint-e} \times \text{posint-e} \rightarrow \text{negint-e} \\
& \quad \text{negint} \times \text{posint} \rightarrow \text{negint} \\
\text{remainder} : & \quad \text{zero-e} \times \text{posnegint-e} \rightarrow \text{zero-e} \\
& \quad \text{zero} \times \text{posnegint} \rightarrow \text{zero} \\
& \quad \text{posint-e} \times \text{posnegint-e} \rightarrow \text{posint-e} \cup \text{zero-e} \\
& \quad \text{posint} \times \text{posnegint} \rightarrow \text{posint} \cup \text{zero} \\
& \quad \text{negint-e} \times \text{posnegint-e} \rightarrow \text{negint-e} \cup \text{zero-e} \\
& \quad \text{negint} \times \text{posnegint} \rightarrow \text{negint} \cup \text{zero}
\end{align*}
\]
modulo:  
\[
\text{zero-e } \times \text{ posnegint-e } \rightarrow \text{ zero-e} \\
\text{zero } \times \text{ posnegint } \rightarrow \text{ zero} \\
\text{posnegint-e } \times \text{ posint-e } \rightarrow \text{ posint-e } \cup \text{ zero-e} \\
\text{posnegint } \times \text{ posint } \rightarrow \text{ posint } \cup \text{ zero} \\
\text{posnegint-e } \times \text{ negint-e } \rightarrow \text{ negint-e } \cup \text{ zero-e} \\
\text{posnegint } \times \text{ negint } \rightarrow \text{ negint } \cup \text{ zero}
\]

gcd:  
\[
\text{zero-e }^k \geq 0 \rightarrow \text{ zero-e} \\
\text{zero}^k \geq 1 \rightarrow \text{ zero} \\
\text{posnegint-e }^k \geq 1 \rightarrow \text{ posint-e} \\
\text{posnegint}^k \geq 1 \rightarrow \text{ posint} \\
\text{int-e }^k \geq 1 \rightarrow \text{ posint-e } \cup \text{ zero-e} \\
\text{int}^k \geq 1 \rightarrow \text{ posint } \cup \text{ zero}
\]

lcm:  
\[
\text{zero-e }^k \geq 1 \rightarrow \text{ zero-e} \\
\text{zero}^k \geq 1 \rightarrow \text{ zero} \\
\text{posnegint-e }^k \geq 1 \rightarrow \text{ posint-e} \\
\text{posnegint}^k \geq 1 \rightarrow \text{ posint} \\
\text{int-e }^k \geq 1 \rightarrow \text{ posint-e } \cup \text{ zero-e} \\
\text{int}^k \geq 1 \rightarrow \text{ posint } \cup \text{ zero}
\]

numerator:  
\[
\text{real-e } \rightarrow \text{ int-e} \\
\text{real-i } \rightarrow \text{ int-i}
\]

denominator:  
\[
\text{real-e } \rightarrow \text{ posint-e} \\
\text{real-i } \rightarrow \text{ posint-i}
\]

floor, ceiling, truncate, round:  
\[
\text{real-e } \rightarrow \text{ int-e} \\
\text{real-i } \rightarrow \text{ int-i}
\]

rationalize:  
\[
\text{posreal-e } \times \text{ posreal-e } \rightarrow \text{ posrat-e} \\
\text{negreal-e } \times \text{ negreal-e } \rightarrow \text{ negrat-e} \\
\text{real-e } \times \text{ real-e } \rightarrow \text{ rat-e} \\
\text{posreal-i } \times \text{ posreal-i } \rightarrow \text{ posrat-i} \\
\text{negreal-i } \times \text{ negreal-i } \rightarrow \text{ negrat-i} \\
\text{real-i } \times \text{ real-i } \rightarrow \text{ rat-i}
\]

exp:  
\[
\text{real-e } \rightarrow \text{ real-e} \\
\text{real-i } \rightarrow \text{ real-i} \\
\text{comp-e } \rightarrow \text{ comp-e} \\
\text{comp-i } \rightarrow \text{ comp-i}
\]
log: \[ \text{real-e \{zero-e \rightarrow real-e} \]
\[ \text{real-i \{zero-i \rightarrow real-i} \]
\[ \text{comp-e \{zero-e \rightarrow comp-e} \]
\[ \text{comp-i \{zero-i \rightarrow comp-i} \]

\[ \text{sin, cos, tan, asin, acos, atan: real-e \rightarrow real-e} \]
\[ \text{real-i \rightarrow real-i} \]
\[ \text{comp-e \rightarrow comp-e} \]
\[ \text{comp-i \rightarrow comp-i} \]

\[ \text{sqrt: zero-e \rightarrow zero-e} \]
\[ \text{zero-i \rightarrow zero-i} \]
\[ \text{posreal-e \rightarrow posreal-e} \]
\[ \text{posreal-i \rightarrow posreal-i} \]
\[ \text{negreal-e \rightarrow comp-no-real-e} \]
\[ \text{negreal-i \rightarrow comp-no-real-i} \]
\[ \text{comp-e \rightarrow comp-e} \]
\[ \text{comp-i \rightarrow comp-i} \]

\[ \text{expt: posint-e × posint-e \rightarrow posint-e} \]
\[ \text{posint × posint \rightarrow posint} \]
\[ \text{comp-e × zero-e \rightarrow posint-e} \]
\[ \text{comp × zero \rightarrow posint} \]
\[ \text{int-e × posint-e \rightarrow int-e} \]
\[ \text{int × posint \rightarrow int} \]
\[ \text{rat-e × int-e \rightarrow rat-e} \]
\[ \text{rat × int \rightarrow rat} \]
\[ \text{real-e × real-e \rightarrow real-e} \]
\[ \text{real × real \rightarrow real} \]
\[ \text{comp-e × comp-e \rightarrow comp-e} \]
\[ \text{comp × comp \rightarrow comp} \]

\[ \text{make-rectangular, make-polar: A × zero-e \rightarrow A} \]
\[ \text{A × zero-i \rightarrow A - i} \]
\[ \text{real-e × real-e \rightarrow comp-e} \]
\[ \text{real × real \rightarrow comp} \]

for every number type \( A \subseteq \text{real} \) where \( A - i \) denotes the corresponding type of inexact numbers.

\[ \text{real-part: A \rightarrow A} \]
\[ \text{comp-e \rightarrow real-e} \]
\[ \text{comp-i \rightarrow real-i} \]
for every number type $A \subseteq \text{real}.$

**imag-part:**
- real-e $\rightarrow$ zero-e
- real-i $\rightarrow$ zero-i
- comp-e $\rightarrow$ real-e
- comp-i $\rightarrow$ real-i

**magnitude:**
- $A \rightarrow \text{pos} - A \cup \text{zero} \cup e$
- $B \rightarrow \text{pos} - B \cup \text{zero} \cup i$
- comp-e $\rightarrow$ posreal-e $\cup$ zero-e
- comp-i $\rightarrow$ posreal-i $\cup$ zero-i

for every number type $A \subseteq \text{real-e}$ or $B \subseteq \text{real-i}.$

**angle:**
- posreal-e $\rightarrow$ zero-e
- posreal-i $\rightarrow$ zero-i
- comp-e $\rightarrow$ real-e
- comp-i $\rightarrow$ real-i

**exact $\rightarrow$ inexact:**
- zero $\rightarrow$ zero-i
- posint $\rightarrow$ posint-i
- negint $\rightarrow$ negint-i
- posrat-no-int $\rightarrow$ posrat-no-int-i
- negrat-no-int $\rightarrow$ negrat-no-int-i
- posreal-no-rat $\rightarrow$ posreal-no-rat-i
- negreal-no-rat $\rightarrow$ negreal-no-rat-i

**inexact $\rightarrow$ exact:**
- zero $\rightarrow$ zero-e
- posint $\rightarrow$ posint-e
- negint $\rightarrow$ negint-e
- posrat-no-int $\rightarrow$ posrat-no-int-e
- negrat-no-int $\rightarrow$ negrat-no-int-e
- posreal-no-rat $\rightarrow$ posreal-no-rat-e
- negreal-no-rat $\rightarrow$ negreal-no-rat-e

**number $\rightarrow$ string:**
- num $\rightarrow$ string

**string $\rightarrow$ number:**
- string $\rightarrow$ num

### E.6 Characters

\[\text{char-ci=?}, \text{char-ci<?}, \text{char-ci?>}, \text{char-ci<=?}, \text{char-ci>=?} : \text{char} \times \text{char} \rightarrow \text{bool}\]

\[
\text{char-alphabetic?} : \text{upper-char} \rightarrow \text{true}
\quad \text{lower-char} \rightarrow \text{true}
\quad \text{num-char} \rightarrow \text{false}
\quad \text{whitespace-char} \rightarrow \text{false}
\quad \text{sym-char} \rightarrow \text{false}
\]

\[
\text{char-numeric?} : \text{num-char} \rightarrow \text{true}
\quad \text{upper-char} \rightarrow \text{false}
\quad \text{lower-char} \rightarrow \text{false}
\quad \text{whitespace-char} \rightarrow \text{false}
\quad \text{sym-char} \rightarrow \text{false}
\]

\[
\text{char-whitespace?} : \text{whitespace-char} \rightarrow \text{true}
\quad \text{upper-char} \rightarrow \text{false}
\quad \text{lower-char} \rightarrow \text{false}
\quad \text{num-char} \rightarrow \text{false}
\quad \text{sym-char} \rightarrow \text{false}
\]

\[
\text{char-upper-case?} : \text{upper-char} \rightarrow \text{true}
\quad \text{lower-char} \rightarrow \text{false}
\quad \text{num-char} \rightarrow \text{false}
\quad \text{whitespace-char} \rightarrow \text{false}
\quad \text{sym-char} \rightarrow \text{false}
\]

\[
\text{char-lower-case?} : \text{lower-char} \rightarrow \text{true}
\quad \text{upper-char} \rightarrow \text{false}
\quad \text{num-char} \rightarrow \text{false}
\quad \text{whitespace-char} \rightarrow \text{false}
\quad \text{sym-char} \rightarrow \text{false}
\]

\[
\text{char-\text{\textgreater}integer} : \text{char} \rightarrow \text{posint-e}
\]

\[
\text{integer-\text{\textless}char} : \text{posint-e} \rightarrow \text{char}
\]

\[
\text{char-upcase} : \text{upper-char} \rightarrow \text{upper-char}
\quad \text{lower-char} \rightarrow \text{upper-char}
\quad \text{num-char} \rightarrow \text{num-char}
\quad \text{whitespace-char} \rightarrow \text{whitespace-char}
\quad \text{sym-char} \rightarrow \text{sym-char}
\]

293
E.7 Strings

\text{char-downcase} : \text{upper-char} \rightarrow \text{lower-char} \\
\text{lower-char} \rightarrow \text{lower-char} \\
\text{num-char} \rightarrow \text{num-char} \\
\text{whitespace-char} \rightarrow \text{whitespace-char} \\
\text{sym-char} \rightarrow \text{sym-char}

\begin{align*}
\text{string? : string} & \rightarrow \text{Ttrue} \\
\text{T} \setminus \text{string} & \rightarrow \text{Tfalse}
\end{align*}

\text{make-string} : \begin{aligned}
\text{zero} - e & \rightarrow (\text{string } \bot) \\
\text{posint} - e & \rightarrow \text{string} \\
\text{zero} - e \times A & \rightarrow (\text{string } \bot) \\
\text{posint} - e \times A & \rightarrow (\text{string } A)
\end{aligned}

\text{for every character type } A \subseteq \text{char}.

\begin{align*}
\text{string : } & \mathcal{O} \rightarrow (\text{string } \bot) \\
A^{k \geq 1} & \rightarrow (\text{string } A)
\end{align*}

\text{for every character type } A \subseteq \text{char}.

\text{string-length} : \text{string} \rightarrow \text{posint-e} \cup \text{zero-e}

\text{string-ref} : \begin{aligned}
(\text{string } A) \times \text{zero-e} & \rightarrow A \\
(\text{string } A) \times \text{posint-e} & \rightarrow A
\end{aligned}

\text{string=?}, \text{string-ci=?} : \text{string} \times \text{string} \rightarrow \text{bool}

\text{string<, string>, string<=, string>=} : \text{string} \times \text{string} \rightarrow \text{bool}

\text{string-ci<, string-ci>, string-ci<=, string-ci>=} : \text{string} \times \text{string} \rightarrow \text{bool}

\text{substring} : \begin{aligned}
(\text{string } A) \times \text{zero-e} \times \text{zero-e} & \rightarrow (\text{string } \bot) \\
(\text{string } A) \times \text{zero-e} \times \text{posint-e} & \rightarrow (\text{string } \bot) \\
(\text{string } A) \times \text{posint-e} \times \text{posint-e} & \rightarrow (\text{string } A)
\end{aligned}

\text{for every character type } A \subseteq \text{char}.

\text{string-append} : \begin{aligned}
\mathcal{O} & \rightarrow (\text{string } \bot) \\
(\text{string } A)^{k \geq 1} & \rightarrow (\text{string } A)
\end{aligned}

\text{for every character type } A \subseteq \text{char}.

\text{string->list} : (\text{string } A) \rightarrow (\text{list } A)
for every character type $A \subseteq \text{char}$.

$\text{list} \rightarrow \text{string} : (\text{list } A) \rightarrow (\text{string } A)$

for every character type $A \subseteq \text{char}$.

$\text{string-copy} : (\text{string } A) \rightarrow (\text{string } A)$

$\text{string-fill!} : \text{string} \times \text{char} \rightarrow_{SE} \emptyset$

### E.8 Vectors

$\text{vector?} : \text{vector} \rightarrow \text{Ttrue}$

$\text{Tfalse} \setminus \text{vector} \rightarrow \text{Tfalse}$

$\text{make-vector} : \text{zero-e} \rightarrow (\text{vector } \bot)$

$\text{posint-e} \rightarrow \text{vector}$

$\text{zero-e} \times A \rightarrow (\text{vector } \bot)$

$\text{posint-e} \times A \rightarrow (\text{vector } A)$

$\text{vector} : \emptyset \rightarrow (\text{vector } \bot)$

$A^{k \geq 1} \rightarrow (\text{vector } A)$

$\text{vector-length} : \text{vector} \rightarrow \text{zero-e} \cup \text{posint-e}$

$\text{vector-ref} : (\text{vector } A) \times \text{zero-e} \rightarrow A$

$(\text{vector } A) \times \text{posint-e} \rightarrow A$

$\text{vector} \rightarrow \text{list} : (\text{vector } A) \rightarrow (\text{list } A)$

$\text{list} \rightarrow \text{vector} : (\text{list } A) \rightarrow (\text{vector } A)$

$\text{vector-fill!} : \text{vector} \times \text{T} \rightarrow_{SE} \emptyset$

### E.9 Control Features

Note that there are no function types $A \rightarrow B$ available directly. Closures have to be used instead.

$\text{procedure?} : (\text{T} \rightarrow_{\mathcal{Z}} \text{T}) \rightarrow \text{Ttrue}$

$\text{Tfalse} \setminus (\text{T} \rightarrow_{\mathcal{Z}} \text{T}) \rightarrow \text{Tfalse}$
apply: \((A_1 \times \ldots \times A_k \to B) \times (A_1 \cdot (A_2 \cdot \ldots \cdot (A_k \cdot \text{nil}))) \to B\)

\((A_1 \times \ldots \times A_k \to B) \times A_1 \times \ldots \times A_i \times (A_{i+1} \cdot (A_{i+2} \cdot \ldots \cdot (A_k \cdot \text{nil}))) \to B^2\)

map: \{((A_1,1 \times \ldots \times A_{k,1} \to B_1), \ldots, (A_{1,n} \times \ldots \times A_{k,n} \to B_n)) \times

\((A_{1,1} \cdot (A_{1,2} \cdot \ldots \cdot (A_{1,n} \cdot \text{nil})))) \times \ldots \times

\((A_{k,1} \cdot (A_{k,2} \cdot \ldots \cdot (A_{k,n} \cdot \text{nil}))) \to (B_{1} \cdot (B_{2} \cdot \ldots \cdot (B_n \cdot \text{nil})))\}^3\)

for-each: \{((A_{1,1} \times \ldots \times A_{k,1} \to \text{true}), \ldots, (A_{1,n} \times \ldots \times A_{k,n} \to \text{true})) \times

\((A_{1,1} \cdot (A_{1,2} \cdot \ldots \cdot (A_{1,n} \cdot \text{nil})))) \times \ldots \times

\((A_{k,1} \cdot (A_{k,2} \cdot \ldots \cdot (A_{k,n} \cdot \text{nil}))) \to SE O\}

scheme-report-environment : A(5) \to AE_{SR}\)

null-environment : A(5) \to AE_0\)

interaction-environment : special\)

eval : special\)

E.10 Input and Output

E.10.1 Ports

call-with-input-file : string \times (in-port \to A) \to SE A\)

call-with-output-file : string \times (out-port \to A) \to SE A\)

\(^2\)The PI/PO-representations occurring as arguments to apply (or to map or for-each have to be extended in a manner similar to Sec. 6.4. Specifying this task is straightforward and is omitted in this work.

\(^3\)Some of the \((A_{1,i} \times \ldots \times A_{k,i} \to B_i)\) might be equal.
input-port? : in-port \rightarrow \text{T}\text{true}  \quad \text{T} \setminus \text{in-port} \rightarrow \text{T}\text{false}

output-port? : out-port \rightarrow \text{T}\text{true}  \quad \text{T} \setminus \text{out-port} \rightarrow \text{T}\text{false}

port? : in-port \rightarrow \text{T}\text{true}  
\quad \text{out-port} \rightarrow \text{T}\text{true}  
\quad \text{T} \setminus (\text{in-port} \cup \text{out-port}) \rightarrow \text{T}\text{false}

current-input-port : \mathcal{O} \rightarrow \text{in-port}

current-output-port : \mathcal{O} \rightarrow \text{out-port}

with-input-from-file : \text{string} \times (\mathcal{O} \rightarrow \mathcal{A}) \rightarrow_{\text{SE}} \mathcal{A}

with-output-from-file : \text{string} \times (\mathcal{O} \rightarrow \mathcal{A}) \rightarrow_{\text{SE}} \mathcal{A}

open-input-file : \text{string} \rightarrow \text{in-port}

open-output-file : \text{string} \rightarrow \text{out-port}

close-input-port : \text{in-port} \rightarrow \mathcal{O}

close-output-port : \text{out-port} \rightarrow \mathcal{O}

\textbf{E.10.2 \ Input}

read : \mathcal{O} \rightarrow_{\text{SE}} \text{T}  
\quad \text{in-port} \rightarrow_{\text{SE}} \text{T}

read-char : \mathcal{O} \rightarrow_{\text{SE}} \text{char}  
\quad \text{in-port} \rightarrow_{\text{SE}} \text{char}

peek-char : \mathcal{O} \rightarrow_{\text{SE}} \text{char}  
\quad \text{in-port} \rightarrow_{\text{SE}} \text{char}

eof-object? : \text{eof} \rightarrow \text{T}\text{true}  \quad \text{T} \setminus \text{eof} \rightarrow \text{T}\text{false}

char-ready? : \mathcal{O} \rightarrow \text{bool}  
\quad \text{in-port} \rightarrow \text{bool}
E.10.3 Output

\[\text{write} : \ T \to_{SE} \mathcal{O} \]
\[\ T \times \text{out-port} \to_{SE} \mathcal{O}\]

\[\text{display} : \ T \to_{SE} \mathcal{O} \]
\[\ T \times \text{out-port} \to_{SE} \mathcal{O}\]

\[\text{newline} : \ \mathcal{O} \to_{SE} \mathcal{O} \]
\[\text{out-port} \to_{SE} \mathcal{O} \]

\[\text{write-char} : \ \text{char} \to_{SE} \mathcal{O} \]
\[\text{char} \times \text{out-port} \to_{SE} \mathcal{O}\]

E.10.4 System Interface

\[\text{load} : \ \text{string} \to_{SE} T\]

\[\text{transcript-on} : \ \text{string} \to_{SE} \mathcal{O}\]

\[\text{transcript-off} : \ \mathcal{O} \to_{SE} \mathcal{O}\]
Index

(∩ ...), 12
(∪ ...), 12
B, 30
K, 30
O, 90
T, 44, 47
T_{CS}(V), 27
T_{C}(V), 44, 47
T_{GS}(V), 27
T_{G}(V), 44, 47
T_{SCS}(V), 27
T_{SC}(V), 44, 47
T_{S}(V), 26
⊥, 30
∩, 12
t_1 □ t_2, 53
∪, 12
\llbracket . \rrbracket_0, 93
\llbracket . \rrbracket^A, 165
\llbracket . \rrbracket^S, 105
\llbracket . \rrbracket, 40
⊗, 56
Γ, 34
t_{p\mid t'}, 27
t_{[p\mid t'], 30
\llbracket . \rrbracket_A, 20
\llbracket . \rrbracket, 20
< a_i \ldots e_i >, 17
<i \ldots>, 17

abstract application
  variable, 174
abstract application, 162
  error
  with variables, 176
PI/PO-representation, 174
predefined, 161
predefined function
  with variables, 178, 179
  with variables, 180
abstract application error, 154
abstract boolean interpretation, 148
abstract domain
  primitive, 125
abstract domain constructor, 127
abstract environment, 130
  canonical, 137
  extended, 136
  master copy, 148
  mutability, 149
  simple, 134
  semantics, 143
abstract environment application, 131
abstract environment update, 133
abstract error value, 125
abstract expression, 147
abstract frame, 129
  partial, 129
abstract frame list, 129
abstract frame state
  joinable, 134
abstract frame update, 133
abstract interpretation, 20
abstract lambda closure, 126
abstract lambda-application, 152
abstract predefined application, 161
abstract predefined function, 125
semantics, 142
abstract semantic function, 165
  simple, 165
abstract semantics
  abstract constant term, 147
  constant term, 147
  define, 163
environment, 144
frame, 143
I/O-representation, 142
lambda closure, 144
\(\lambda\)-iterative, 186
predefined function
  special, 159
recursion interrupting, 185
simple abstract environment, 143
  with \(\lambda\)-history, 185
  with variables, 180
abstract state condition, 132
  enforcement, 133
abstract value
  error, 125
  partial, 129
  stored, 128
abstraction function, 20
application
  abstract environment, 131
application error
  abstract, 154
application function
  closed predefined, 157
  partial predefined, 155
  predefined, 156
application test function, 158
apply\((e, E)\), 162
apply-error\(v\)(\(E\)), 176
apply-pipo\((E)\), 174
apply-pre\((E)\), 161
apply-pre\(v\)(\(E\)), 178, 179
apply-strict, 154
apply-strict\(v\), 176
apply-var\((E)\), 174
apply\(v\)(\(E)\), 180
appropriate closed type substitution, 40
ascription constructed types, 158
assignable abstract value, 127
assignable value, 91, 117
  abstract, 127
  stored, 96
base type, 21, 30, 277
boolean interpretation, 92
  abstract, 148
collection, 55
  union, 56
canonical abstract environment, 137
canonical\((E)\), 137
\(CE\), 79
circuit, 19
clean environment, 92
closed predefined application function, 157
combine\((E_1, E_2)\), 138
combine-cs, 61
common element, 53
completeness
  in abstract interpretation, 21
  of a type checker, 2
component graph, 19
compress\((E)\), 140
concatenation of positions, 28
concretization function, 20
conditional types, 23
confluence, 16
connected nodes, 19
constrain-all-free, 59
constraint set
  implicit, 70
critical pair, 16
\(CS^0\)
  standard semantics, 93
  syntax, 88
CSL
syntax, 108

CSS
standard semantics, 105
syntax, 95
cyclic value
finite approximation, 41
denoteconst, 90
denoteconst_A, 147
dependent types, 23
domain
abstract
primitive, 125
of booleans, 90
primitive, 90
domain constructor, 91
abstract, 127
pair, 108
enforcement of the abstract state condition,
133
envcopy(E), 116
environment, 92
abstract, 130
application, 131
canonical, 137
extended, 136
master copy, 148
mutability, 149
simple, 134
abstract semantics, 144
clean, 92
master copy, 115
mutability, 115
standard semantics, 118
state condition, 100
stored, 98
structured, 97
unchangeable
standard semantics, 119
environment type, 32
environment update, 101, 102
abstract, 133
error, 1, 5, 24
error value, 144
EU^S(e, E), 102
expression
abstract, 147
expression compatibility, 168
with variables, 180
extend-constraint, 59
extended abstract environment, 136
finite approximation
type, 40
value, 41
type, 40
value, 41
fq, 66
frame, 97
abstract, 129
partial, 129
abstract semantics, 143
stored, 97
frame list
abstract, 129
frame state
abstract
joinable, 134
frame type, 31
shuffle, see shuffle of frame types
frame update, 101
abstract, 133
frame-list-copy_A(E), 149
free constraint set, 55
union, 56
free type constructor, 30
free type substitution, 54
natural, 55
free-to-top, 59
FS-env-update, 133
function
polymorphic, 22
predefined, 90
function definition
predefined, 285
function symbol, 13
with variable arity, 16
function type, 31
functional language
dynamically typed, 22
statically typed, 22
strictly typed, 22

get-stored-value, 101
getframestate\textsubscript{A}, 130
GIS, 79
graph, 19
directed, 19

implicit-cs\textsubscript{C}(c), 70
interpretation
abstract, see abstract interpretation
intersection, 12
intersection existence property, 37
I/O-representation, 126

joinable abstract frame state, 134

label
of an expression, 89
lambda closure
abstract semantics, 144
lambda closure, 91
abstract, 126
lambda-application
abstract, 152
\lambda-iterative abstract semantics, 186
local confluence, 16

make-list, 109
make-list\textsubscript{A}, 164
master copy abstract environment, 148
master copy of an environment, 115
mutability of environments, 115

normal form, 15
open S-CE calculation, 70
PAF(f, t), 156
PAF\textsubscript{c}(f, t), 157
pair domain constructor, 108
partial abstract frame, 129
semantics, 143
partial abstract frame list
semantics, 143
partial abstract value, 129
partial predefined application function, 155
partial types, 23
path, 19
PI/PO-representation, 46
polymorphism
parametric, 22
position, 27
concatenation, 28
PPAF(f, t), 155
PPAF\textsubscript{V}, 177
pre-application environment, 150
predefined application function, 156
predefined function, 90
abstract, 125
special
abstract semantics, 159
primitive abstract domain, 125
primitive domain, 90
proper subterm, 29

quote-sem\textsubscript{A}, 164
quotesem, 109

rec-gen, 191
recursion interrupting abstract semantics, 185
recursive type, 33
finite approximation, 40
replace-constraint, 59
rewrite relation, 15
S-CE, 67, 57–77
s-collection, 55
SCE-list-reduce, 58
Scheme, 277, 285
  base type, 277
  function definition, 285
  type constructor, 283
semantic domain, 20
  abstract, 20
  concrete, 20
semantic domain of types, 35
semantic function
  abstract, 165
  simple, 165
semantics
  abstract, 20, 285
    abstract constant term, 147
    constant term, 147
define, 163
environment, 144
frame, 143
I/O-representation, 142
lambda closure, 144
\lambda-iterative, 186
recursion interrupting, 185
simple abstract environment, 143
  with \lambda-history, 185
  with variables, 180
abstract predefined function, 142
base type, 37
of type languages, 23
partial abstract frame, 143
partial abstract frame list, 143
recursive type, 41
simple abstract environment, 135
standard, 20
  \textit{CS^0}, 93
  \textit{CS^S}, 105
environment, 118
side effect, 110
unchangeable environment, 119
type
  base, 37
  environment, 38
  frame, 38
  free constructor, 37
  function, 38
set, 11
set based analysis, 23
set normalized form, 42
set oriented type constructor, 30
shuffle of frame types, 39
simple abstract environment, 134
  semantics, 135, 143
simple abstract semantic function, 165
\textit{SMR}, 80
soft-typing, 22
soundness
  in abstract interpretation, 21
  of a type checker, 1
\textit{ST(t_1, t_2)}, 50
standard semantics
  \textit{CS^0}, 93
  \textit{CS^S}, 105
environment, 118
side effect, 110
unchangeable environment, 119
standard type, 26
  closed, 27
  ground, 27
  semi-closed, 27
state, 98
state compatibility, 101
state condition
  abstract, 132
  enforcement, 133
  structured, 109
state condition of environments, 100
store
  look up, 98
stored abstract value, 128
stored assignable value, 96
stored environment, 98
stored frame, 97
stored value
  look up, 101
strongly connected component, 19
structured environment, 97
structured state condition, 109
substitution, 14
subterm, 14, 27
  position of, 14
  proper, see proper subterm
subterms(t), 28
subtype relation, 50
subtyping, 6, 23
symbol binding, 31
syntax
  $CS^0$, 88
  $CS^S$, 108
  $CS^S$, 95
term, 13
  instance of, 14
  update at a position, 14
term rewriting, 13–18
  extended syntax of, 16–18
term rewriting rule, 15
term rewriting system, 15
  confluent, 16
  locally confluent, 16
  terminating, 15
term update, 30
termination, 15
TRS, see term rewriting system
type, 21
  base, see bse type
  closed, 44, 47
  complex, 22
  equivalence, 43
  ground, 44, 47
  semi-closed, 44, 47
set normalized form, see set normalized form
  standard, 26
type variable, 22
type constant, 30
type constructor, 22, 283
  free, see free type constructor
  set oriented, see set oriented type constructor
type error, see error
type inference
  static, 22
type substitution, 39
  application, 39
  appropriate closed, 40
  closed, 39
  free, see free type substitution
  GIS appropriate, 130
type variable, 26
type warning, see warning
types
  conditional, see conditional types
  dependent, see dependent types
  partial, see partial types
unchangeable environment
  standard semantics, 119
unfolding
  recursive type, 33
  value, 36
union, 12
  c-collections, 56
  free constraint sets, 56
update
  environment, 101
  frame, 101
update, 101
update_A, 133
value
  abstract, 20
error, 125
assignable, 91, 117
simple, 21
Value, 92
value assignment, 44
valueexpression(v), 117
valueexpression_A(v), 161
Var_A, 128
warning, 5, 23, 24
zero value, 90