A complete type system for functional languages

Manfred Widera, Christoph Beierle
Praktische Informatik VIII
FernUniversität Hagen
58084 Hagen
email: {Manfred.Widera, Christoph.Beierle}@FernUni-Hagen.de

Abstract

This paper presents a powerful type system for functional programming languages consisting of an expressive type language and a type inference algorithm. In contrast to the usual approach it is complete in the sense that every program which is type correct is recognized as such. We get the advantage that every type error reported by our system is a real error. This is useful in debugging programs, since the programmer can rely on the error messages and need not check whether a returned function call really contains a type error and is not just hard to prove type correct. Soundness is a less important property in this context, because there are usually some more stages of debugging that can catch the other bugs.

Contents

1 Introduction

Many errors occurring in software are type errors. In dynamic typed functional languages, 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.

Static type checking of functional programs has been analyzed in detail and 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. Therefore, type systems usually are restricted to soundness, i.e. every program that is type correct according to a sound type system will not produce a type error for any input data. In other words, for any sound type system Milner's slogan "well-typed programs do not go wrong" holds.

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 the context of debugging incorrect programs, sound type systems show a big disadvantage: A sound type system does not detect function calls containing type errors, but function calls whose type correctness cannot be proved. It is the users task to determine whether this results from a type error or just from an inaccuracy of the type system.

In this paper, we present a novel typing approach that avoids this disadvantage of sound type systems. 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 as exactly as possible the origination of each error.
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*. Thus, completeness of a type system guarantees that whenever the system reports an error we are sure that the program is indeed ill-typed. Rephrasing Milner's slogan the motto of complete type systems 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 undecidability of the type checking problem. The missing soundness of a complete system causes some errors not to be caught. But this is not critical because debugging a program consists of further steps that will detect some of the type errors, too. When after debugging type correctness is crucial, a sound type system 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. (See [CF91] for a discussion, why these kinds of type constructors are essential for an expressive type language.) 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 done for the LISP dialect Scheme, but it should be easy to change it to another functional language. The capability to handle side effects is contained in the presented system, as well.
Partial types, as described in [Tha88, Tha94] are no problem at all in the complete type system described here, because union types are usable to express heterogeneous element types of structures, and an exact distinction between the possible element types is not necessary.

The explainability of type errors benefits from the fact that the system presented here does not type functions as a whole, but types function calls where the input types of the main function are provided to the system. Therefore, the user is not forced to handle principle types of functions that might become quite complicated, but the types (s)he has to handle will be close to the types (s)he expects for a function.

The rest of the paper is organized as follows: Section 2 discusses the differences of sound and complete type systems in application and implementation. Section 3 gives a definition of the type language. The type inference algorithm is presented in section 4. Section 5 describes some extensions necessary for handling side effects in functional programs. In section 6 the completeness of the presented type system is proved. Some examples of functional programs and the results of the type system are given in section 7. The summary of the results can be found in section 8.

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 no type language can express every set of values exactly. For example it is often impossible to express the type of all numbers without 0 which would be the input type of the division operator /. Some of those 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 /.
Because of these reasons one must decide for every application whether a sound or a complete type checker is more appropriate.

We can think of the possible functional programs as an axis as shown in figure 1. From left to right the difficulty to prove correctness increases and the difficulty to prove a type error decreases. Then both sound and complete type checkers should try to hit the border between well-typed and ill-typed programs as exactly as possible. But 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.

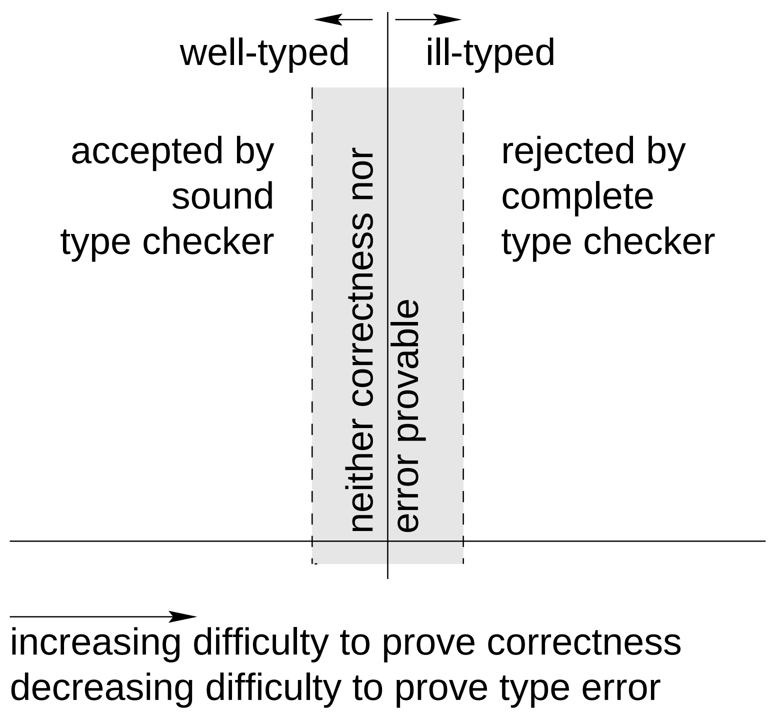

Figure 1: Difference between soundness and completeness

2.1 Differences in Applications

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

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

But sound type systems are only adequate if the correctness of a program is of interest. There usability for detecting errors in programs is restricted, because real type errors may not be distinguished from inaccuracies of the system. This can confuse the users of the system, especially beginners in functional programming. Complete type systems 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. The disadvantage of missing some of the errors does not carry much weight, because these errors normally will be caught by another debugging stage.

2.2 Differences in type inference

Let $A = f(a_1, \ldots, a_n)$ be an expression with some function f and subexpressions a_i at argument position 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. (If 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 in general in a sound type system 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. $\forall i \in \{1, \ldots, n\} \, . \, t_i \subseteq t'_i$, i.e. $\forall i \in \{1, \ldots, n\} \; \forall x \in t_i \, . \, x \in t'_i$ (see Figure 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]).
- A is well-typed in our complete type system 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, i.e. $\forall i \in \{1, \ldots, n\} \; \exists x \in t_i \, . \, x \in t'_i$, i.e. $\forall i \in \{1, \ldots, n\} \, . \, t_i \cap t'_i \neq \emptyset$ (see Figure 2). Unfortunately there seems to be no way of checking constraints of this kind efficiently, e.g. by unification.

Neither sound nor complete type systems can compute the exact set of possible values of any lambda expression: Some sets of possible values cannot be described exactly in the given type

element oriented representation	$\forall x \in t_i : x \in f_i$	$\exists x \in t_i : x \in f_i$
set oriented representation	$t_i \subseteq f_i$	$t_i \cap f_i \neq \emptyset$
	sound type system	complete type system

Figure 2: Comparison of sound and complete type systems

language. Even without this restriction, in special cases, computing the exact set of possible values means evaluating the expression for every mapping of input data to the free variables of the expression which may be impossible because of e.g. non terminating recursion.

Sound and complete type systems usually 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. They differ in the effects of extra values in the computed type:

- In a sound type system 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.
- The complete type system described in this paper cannot detect 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 system detects is a real error, i.e. the program *must* go wrong at this point.

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

3 The type language

In the type language defined in this section types are informally seen as sets of their values. A value a is said to be of type t for every type t that on this informal view fulfills $a \in t$. Ω

denotes the set of all values. The set of all values of type t is denoted by $V(t)$ and fulfills $V(t) \subseteq \Omega$.
Note that formally a type t and the set of its values $V(t)$ are different. The difference in defining t and $V(t)$ avoids the definition of type constructors over subsets of Ω that are not defined over arbitrary sets. We use a structure t instead of $V(t)$ which differs from $V(t)$, but basically contains the same information.

The set of all types consists of base types defined in subsection 3.1 and complex types that are formed using type constructors from subsection 3.2. A base type describes a set of simple values with common properties. (e.g. natural numbers or boolean values). Type constructors are used to form composed types from base types or simpler composed types (e.g. lists with elements of a given type). Some more type constructors are needed to handle sets (e.g. union and intersection of types).

In order to handle programs of a real programming language, we define our type language in more detail than it is done e.g. in [Mil78], [AM91] or [Smi94].

3.1 The base types

In contrast to [Mil78] where the type language is defined over some not further specified set of disjoint base types, we give a definition of our base types, some of which are hierarchically ordered.

- $\perp$ with $V(\perp) = \{\perp\}$ denotes the least type which is a subtype of any other type and stands for a type with no elements except of the non terminating computation. Note that $\perp$ denotes both the least type and the non terminating computation.

- $\top$ denotes the overall type which is supertype of any type and contains every possible element (therefore $V(\top) = \Omega$). $\top$ represents no type restrictions except that it excludes a type error.

- $\mathcal{O}$ denotes the type which is known to contain no value. It is used for input and output types of functions if no argument is expected or no result is given. $\mathcal{O}$ is disjoint from any other type. Formally, besides $\perp$ being an element of every type, $\mathcal{O}$ contains exactly one dummy value `noval` which indicates that no value is expected or returned.
Example: $f : \mathcal{O} \rightarrow \texttt{nat}, g : \perp \rightarrow \texttt{nat}$. With these type definitions we know that f and g produce results of type `nat`. In the case of f we also know that no input is expected, i.e. f is a nullary function representing a constant, but we do not know, whether g expects any input and of what type it must be.

- $\texttt{bool} = \{\texttt{\#t}, \texttt{\#f}\}$ denotes the type of boolean values.

- Numbers are members of type `num`, with subtypes `rat` for rational numbers, `int` for integers, `nat` for natural numbers and `real` for real numbers in floating point representation:
  - $V(\texttt{nat}) = \mathbb{N}$
  - $V(\texttt{int}) = \mathbb{Z}$
  - $V(\texttt{rat}) = \mathbb{Q}$
  - $V(\texttt{num}) = \{(a, b) \in \mathbb{C} \mid a, b \in V(\texttt{rat}) \cup V(\texttt{real})\}$. Furthermore we do the following identifications:
    * $\{(r, 0) \mid r \in \texttt{rat}\}$ is identified with `rat`.
    * $\{(r, 0) \mid r \in \texttt{real}\}$ is identified with `real`.

  $V(\texttt{real}) = \mathbb{R}$ does not hold, since `real` denotes the set of floating point numbers with a representation of fixed size.
  These types fulfill the following conditions:
  - $V(\texttt{real}) \cap V(\texttt{rat}) = \{\perp\}$.[1]
  - $V(\texttt{nat}) \subset V(\texttt{int}) \subset V(\texttt{rat})$
  - $V(\texttt{rat}), V(\texttt{real}) \subset V(\texttt{num})$
- For any symbol a, S_a is a type that contains exactly the symbol a: $V(S_a) = \{\perp, a\}$. The type of all symbols is denoted by `sym`.
- `char` denotes the type of all (printable) characters.
- `string` contains all character strings.
- $\texttt{env}_0$ denotes the empty environment type whose only element besides $\perp$ is the empty environment `emptyenv` containing no frame or symbol binding at all.
  The empty environment is only used to recursively define environment types. It should not occur as a type itself during type inference.

3.2 Definition of type constructors

Type constructors are used to build up complex combined types from simpler combined or base types. We give a definition of a set of type constructors which correspond to the functions that are applied to the resulting types. Every type constructor defines a type scheme that has to be specialized to a type by assigning a type to every free type variable. The type constructors are defined as follows:

[1] Elements of `rat` and `real` are different in every case due to the different internal representations, though a conversion between members of `rat` and `real` may be possible.

- The union type constructor $A \cup B$ generates a type that contains all elements that are in one of the input types: $V(A \cup B) = V(A) \cup V(B)$.[2]

- The intersection type constructor $A \cap B$ contains the elements that are elements of every of the input types: $V(A \cap B) = V(A) \cap V(B)$.

- The complement type constructor $\mathcal{C}A$ generates a type which besides $\perp$ contains exactly those elements of Ω that are not of type A: $V(\mathcal{C}A) = (\Omega \setminus V(A)) \cup \{\perp\} =: \mathcal{C}V(A)$ (where $\mathcal{C}$ on sets of values denotes the complement with regard to the set Ω of all values).
  The use of the complement type constructor is restricted in the following way: It must not be applied to an argument containing any variables or the function type constructor $\rightarrow$.

- The difference type constructor $A \setminus B$ generates a type that besides $\perp$ contains all values of the first type that are not in the second one: $V(A \setminus B) = (V(A) \setminus V(B)) \cup \{\perp\}$.
  The difference type constructor is only used for the external representation of types. Internally $A \setminus B$ is transformed to $A \cap \mathcal{C}B$ where the constraints on the use of the complement type constructor have to be fulfilled.

- The product type constructor $A \times B$ generates the Cartesian product of the element types. Thus we have: $V(A \times B) = V(A) \times V(B)$. The product type constructor $\times$ is associative because obviously $V((A \times B) \times C) = V(A \times (B \times C))$ holds. A^k denotes $\underbrace{A \times \ldots \times A}_{k}$.

- The pair type constructor $(A \;.\; B)$ generates the type of pairs $(a \;.\; b)$, with $a \in A$ and $b \in B$. Note that in contrast to the product type constructor, the pair type constructor is not associative.

- The stream-pair type constructor $[A \;.\; B]$ is defined analogously to the pair type constructor, but denotes different objects.
  Operationally elements of stream pair types will be handled differently because lazy evaluation is used for the second element.

- The function type constructor $A \rightarrow B$ generates the type of functions from source type A to destination type B:
  $V(A \rightarrow B) = \{f : X \rightarrow Y \mid V(A) \subseteq X, Y \subseteq V(B)\}$.
  Informally $A \rightarrow B$ is the set of all functions that can handle at least all elements of A as parameters and need at most B as the set of values when applied to an element of A.

[2] We overload the set operations, but this should cause no confusion because they essentially operate on types as on the corresponding sets of values.

- The fixed point type constructor $\mu X.d(X)$ binds the actually defined type to a type variable X that can be used in the recursive type definition $d(X)$ itself. The resulting type is defined only if the fixed point of the recursive definition exists. This is the case if the function $d(X)$ is contractive in X (cf. [MPS86], section 3.3).

- The list type constructor is defined as follows: $(list\ A) = \mu L.(S_{()} \cup (A\ .\ L))$. It is explicitly defined as an extra type because it is used very often and the abbreviation helps to make more complex type definitions simpler.

- For reasons analogous to the list type constructor, the stream type constructor can be defined as $(stream\ A) = \mu X.(S_{\textit{the-empty-stream}} \cup [A\ .\ X])$.

- $(frame\ s_1 : t_1 \ldots s_k : t_k)$ denotes the type of all frames with symbol s_i bound to a value of type t_i for $i \in \{1, \ldots, k\}$.
  Note that $(frame\ s_1 : t_1 \ldots s_k : t_k)$ contains frames with additional symbol bindings that are not restricted by the type.
  The type of all possible frames is denoted by $\texttt{frame} := (frame)$.

- Environment types are recursively defined from the empty environment type $\texttt{env}_0$ and frame types as follows: If e is an environment type and f is a frame type, then $(env\ e\ f)$ is the environment type containing all environments defined as a frame in f with a parent environment from e. If a symbol s is bound in all elements of f, then we can conclude that the binding given by f overrides the possible binding in e.
  By this definition every environment type is defined as

  $$(env\ (env\ \ldots(env\ \texttt{env}_0\ f_n)\ldots\ f_2)\ f_1)$$

  for frames $f_1, \ldots, f_n$

  Notation: We write $\texttt{env}_{[s_1:t_1,\ldots,s_k:t_k]}$ for an environment if it is just of interest that the binding of symbol s_i is of type t_i for $i \in \{1, \ldots, k\}$.
  $\texttt{env}$ denotes the type of all environments.

The set $\mathcal{T}$ of all types is defined as follows:

Definition 3.1 (Set of all types) *The set $\mathcal{T}$ of all types is the smallest set that fulfills the following conditions:*

1. *$t \in \mathcal{T}$ for every base type t.*

2. *Let $t_1, \ldots, t_k \in \mathcal{T}$ and σ be a k-ary type constructor that is applicable to $t_1, \ldots, t_k$. Then $\sigma(t_1, \ldots, t_k) \in \mathcal{T}$*

3.3 The semantics of types

The semantics of the type language defined above essentially follows [MPS86]. In our notation V has the role of the semantic function and type environments are not necessary, because every free type variable is implicitly seen as universally quantified.

Since the type language in this paper in some parts is an extension to the type language defined in [MPS86], the rest of this subsection is used to justify that this extension does not violate the main properties of the type semantics in the cited article.

The enlarged set of base types does not cause any problems because every base type contains countable many values and all of them can be regarded as flat cpos. We only have to refine the hierarchy of numbers to handle the complement type constructor correctly. Therefore we define the following types:

- `int-no-nat` contains all members of type `int` that are not members of `nat`.
- `rat-no-int` contains all rational numbers that are not members of type `int`.
- `num-no-ratreal` contains all numbers that are neither contained in type `rat` nor in type `real`.

These definitions yield base types expressing e.g. $\texttt{int} \setminus \texttt{nat}$.

From that we can define:

- $\texttt{int} := \texttt{nat} \cup \texttt{int-no-nat}$
- $\texttt{rat} := \texttt{int} \cup \texttt{rat-no-int}$
- $\texttt{num} := \texttt{rat} \cup \texttt{real} \cup \texttt{num-no-ratreal}$

Most of the new type constructors do not cause any trouble either: (.) and [.] are handled like $\times$ in the cited paper and ($list$ $\bullet$) as well as ($stream$ $\bullet$) just give abbreviations to recursive type definitions using (.) or [.].
The countable many frames and environments should be seen as base types in this context because in the context of recursive types (which is our main goal) the use of frames or environments does not seem to be useful at all.

By the given restrictions an argument of the complement type constructor $\mathcal{C}$ does not contain any variables or function type constructors. Therefore recursive types of the kind $\mu X.f(\mathcal{C}X)$ are not possible and thus $\mathcal{C}$ will not cause any recursive types to be undefined.

From a theoretical point of view the complement type constructor can be eliminated by the following observation:

$$\top = \bot \cup \mathcal{O} \cup \texttt{bool} \cup \texttt{nat} \cup \texttt{int-no-nat} \cup \texttt{rat-no-int} \cup \texttt{num-no-ratreal} \cup \texttt{sym} \cup \texttt{char} \cup \texttt{string} \cup \texttt{frame} \cup \texttt{env} \cup \top \times \top \cup (\top \;.\; \top) \cup [\top \;.\; \top] \cup \bot \to \top \quad (1)$$

Note that all the types in the union above are pairwise disjoint. By this observation we can express the complement operator by a restricted union. So consider a type $\mathcal{C}t$.[3] Then t is either a subtype of exactly one of the elements of the union in equation (1), or t consists of a union of types each of which is a subtype of exactly one element of the union in equation (1). So we build a union representing $\mathcal{C}t$ as follows:

- If for an element b_s of the union in equation (1) we have $b_s \cap t = \emptyset$ then b_s is incorporated in the new union. This is always the case for $b_s = \bot \to \top$ since t must not contain any function type constructor by the restrictions on the argument of $\mathcal{C}$.
- If some element b_s of the union in equation (1) is contained in t (either $b_s = t$ or $t = t_1 \cup \ldots \cup t_n$ with $b_s = t_i$ for some i) then b_s is not contained in the new union.
- The last case of some element b_s partly overlapping t is only possible for

  $$b_s \in \{\texttt{sym}, \texttt{frame}, \texttt{env}, \top \times \top, (\top \;.\; \top), [\top \;.\; \top]\}$$

  Then some b'_s containing the elements of $b_s \setminus t$ has to be incorporated in the new union, where b'_s is defined as follows:

  First we consider those t that just contain elements of one kind according to equation (1) and whose top level type constructor is neither $\cup$ nor $\cap$:

  1. For $b_s = \texttt{sym}$ type t could consist of a union of some single symbols. Then b'_s is defined as the infinite union of all symbols except those being element of t.
  2. If $b_s = \texttt{frame}$ and $t = (frame\ x_1 : t_1 \ldots x_k : t_k)$ then $b'_s = \bigcup_{i=1}^{n} (frame\ x_i : \mathcal{C}t_i)$
  3. $b_s = \texttt{env}$ and $t = (env\ (env\ \ldots (env\ \texttt{env}_0\ f_n) \ldots\ f_2)\ f_1) =: (env\ e\ f_1)$ with frames f_i. Then $b'_s = (env\ \mathcal{C}e\ \texttt{frame}) \cup (env\ e\ \mathcal{C}f_1)$ where $\mathcal{C}\texttt{env}_0 := \bot$.
  4. If $b_s = \top \times \top$ and $t = A \times B$ we have $b'_s := (\mathcal{C}A \times \top) \cup (\top \times \mathcal{C}B)$.
  5. $b_s = (\top \;.\; \top)$ is handled analogously to 4.
  6. $b_s = [\top \;.\; \top]$ is handled analogously to 4.

[3]Note that t must not be a variable but is just used for a case distinction here.

The complement type constructors applied to some argument types have to be transformed recursively in the same way.

If $t = t_1 \cup t_2$ where t_1 and t_2 belong to the same type according to equation (1) then we define some $b'_{s,1}$ and $b'_{s,2}$ restricted by t_1 resp. t_2 and set $b'_s := b'_{s,1} \cap b'_{s,2}$.
If $t = t_1 \cap t_2$ we analogously define $b'_s := b'_{s,1} \cup b'_{s,2}$.

As a last step we consider $t = t_1 \cup \ldots \cup t_n$ where the t_i pairwise belong to different types according to equation (1). (This can always be achieved by the associativity and commutativity of $\cup$). In this case every t_i is used to restrict the corresponding b_{s_i} in equation (1) to b'_{s_i}.

3.4 The hierarchy of types

We define a partial ordering $\sqsubseteq$ over the set $\mathcal{T}$ which describes a part of the inclusion ordering over types as sets of their elements. Because of this ordering we can type expressions more precisely (e.g. `nat` instead of `num` for $(*\ 2\ 2)$), but we are still able to generalize types to supertypes with less information to get a common type of two expressions:

Definition 3.2 *The type hierarchy over the set $\mathcal{T}$ is defined as follows: A is a subtype of B ($A \sqsubseteq B$) if this can be derived by one of the following rules:*

1. *Base relations:*
   (a) $\forall A \in \mathcal{T} \,.\, A \sqsubseteq A$
   (b) $\forall A \in \mathcal{T} \,.\, \bot \sqsubseteq A \sqsubseteq \top$
   (c) $\mathtt{nat} \sqsubseteq \mathtt{int} \sqsubseteq \mathtt{rat} \sqsubseteq \mathtt{num}$
   (d) $\mathtt{real} \sqsubseteq \mathtt{num}$
   (e) $S_{()} \sqsubseteq (list\ \bot)$
   (f) $S_{the\_empty\_set} \sqsubseteq (stream\ \bot)$
2. *The set operations:*
   (a) $\forall A, B \in \mathcal{T} \,.\, A, B \sqsubseteq A \cup B$
   (b) $\forall A, B \in \mathcal{T} \,.\, A \cap B \sqsubseteq A, B$
   (c) $\forall A, B \in \mathcal{T} \,.\, A \setminus B \sqsubseteq A$
   (d) $\forall A, B, C \in \mathcal{T} \,.\, A \sqsubseteq C \land B \sqsubseteq C \Rightarrow A \cup B \sqsubseteq C$
   (e) $\forall A, B, C \in \mathcal{T} \,.\, C \sqsubseteq A \land C \sqsubseteq B \Rightarrow C \sqsubseteq A \cap B$

(f) $\forall A, B \in \mathcal{T} \,.\, A \cap B = \bot$ *is provable* $\Rightarrow A \sqsubseteq \mathcal{C}B, B \sqsubseteq \mathcal{C}A$

3. *Relations implied by monotony:*

   (a) $\forall A, B, C, D \in \mathcal{T} \,.\, A \sqsubseteq C \wedge B \sqsubseteq D \Rightarrow A \times B \sqsubseteq C \times D$

   (b) $\forall A, B, C, D \in \mathcal{T} \,.\, A \sqsubseteq C \wedge B \sqsubseteq D \Rightarrow (A \ . \ B) \sqsubseteq (C \ . \ D)$

   (c) $\forall A, B, C, D \in \mathcal{T} \,.\, A \sqsubseteq C \wedge B \sqsubseteq D \Rightarrow [A \ . \ B] \sqsubseteq [C \ . \ D]$

   (d) $\forall A, B, C, D \in \mathcal{T} \,.\, A \sqsupseteq C \wedge B \sqsubseteq D \Rightarrow (A \rightarrow B) \sqsubseteq (C \rightarrow D)$

   (e) $\forall A, B \in \mathcal{T} \,.\, A \sqsubseteq B \Rightarrow (list\ A) \sqsubseteq (list\ B)$

   (f) $\forall A, B \in \mathcal{T} \,.\, A \sqsubseteq B \Rightarrow (stream\ A) \sqsubseteq (stream\ B)$

4. *Hierarchical relations on lists:*

   (a) $\forall A, B \in \mathcal{T} \,.\, (A \ . \ (list\ B)) \sqsubseteq (list\ A \cup B)$

   (b) $\forall A, B \in \mathcal{T} \,.\, (list\ A) \cup (list\ B) \sqsubseteq (list\ A \cup B)$

   (c) $\forall A, B \in \mathcal{T} \,.\, (list\ A) \cap (list\ B) \sqsubseteq (list\ A \cap B)$

5. *Hierarchical relations on streams:*

   (a) $\forall A, B \in \mathcal{T} \,.\, [A \ . \ (stream\ B)] \sqsubseteq (stream\ A \cup B)$

   (b) $\forall A, B \in \mathcal{T} \,.\, (stream\ A) \cup (stream\ B) \sqsubseteq (stream\ A \cup B)$

   (c) $\forall A, B \in \mathcal{T} \,.\, (stream\ A) \cap (stream\ B) \sqsubseteq (stream\ A \cap B)$

6. *Relations on frames and environments:*

   (a) $(frame\ s_1 : t_1 \ldots s_k : t_k) \sqsubseteq (frame\ s_1 : t_1 \ldots s_{i-1} : t_{i-1}\ s_{i+1} : t_{i+1} \ldots s_k : t_k)$

   (b) $\boldsymbol{env}_{[s_1:t_1,\ldots,s_k:t_k]} \sqsubseteq \boldsymbol{env}$

   (c) $\boldsymbol{env}_{[s_1:t_1,\ldots,s_k:t_k]} \sqsubseteq \boldsymbol{env}_{[s_1:t_1,\ldots,s_{i-1}:t_{i-1},s_{i+1}:t_{i+1},\ldots,s_k:t_k]}$

7. *Special relations found useful in practical tests:*

   (a) $\forall A_i, B_i, C_i, D_i \,.\, (A_1 \cap A_2) \sqsubseteq (B_1 \cap B_2) \wedge (C_1 \cap C_2) \sqsubseteq (D_1 \cap D_2) \Rightarrow (A_1 \times C_1) \cap (A_2 \times C_2) \sqsubseteq (B_1 \times D_1) \cap (B_2 \times D_2)$

   (b) $\forall A_i, B_i, C_i, D_i \,.\, (A_1 \cap A_2) \sqsubseteq (B_1 \cap B_2) \wedge (C_1 \cap C_2) \sqsubseteq (D_1 \cap D_2) \Rightarrow (A_1 \ . \ C_1) \cap (A_2 \ . \ C_2) \sqsubseteq (B_1 \ . \ D_1) \cap (B_2 \ . \ D_2)$

   (c) $\forall A_i, B_i, C_i, D_i \,.\, (A_1 \cap A_2) \sqsubseteq (B_1 \cap B_2) \wedge (C_1 \cap C_2) \sqsubseteq (D_1 \cap D_2) \Rightarrow [A_1 \ . \ C_1] \cap [A_2 \ . \ C_2] \sqsubseteq [B_1 \ . \ D_1] \cap [B_2 \ . \ D_2]$

8. *Properties of* $\sqsubseteq$

(a) $\forall A, B, C \,.\, A \sqsubseteq B \wedge B \sqsubseteq C \Rightarrow A \sqsubseteq C$ (transitivity)

(b) $\forall A, B \,.\, A \sqsubseteq B \Rightarrow \mathcal{C}B \sqsubseteq \mathcal{C}A$

Lemma 3.3 *Let A and B be two types. Then*

$$A \sqsubseteq B \Rightarrow V(A) \subseteq V(B)$$

Proof: See appendix A. $\square$

3.5 Simplification of types

3.5.1 Extensions to term rewriting

The steps of type simplification are expressible quite well by term rewriting systems, especially by those being able to handle operators of variable arity and some other extensions as described in [WB98]. The extensions used here are described in short in this subsection:

- For $i \in \{1, 2, \ldots\}$ the pattern $< i \ldots >$ represents an argument list of variable arity. Note that the whole expression 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:

  $$(\cup \;\; <1\ldots> \;\; (\cup \;\; <2\ldots>) \;\; <3\ldots>) \rightarrow (\cup <1\ldots> <2\ldots> <3\ldots>)$$

  transforms nested unions (e.g. $(\cup \; a \; b \; c \; (\cup \; d \; e) \; f)$) to a single union containing all elements (e.g. $(\cup \; a \; b \; c \; d \; e \; f)$)

- The pattern $<a_i \ldots e_i>$ is used instead of $<i \ldots>$ if the single 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. 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 $t(x)$. The pattern $<t(a_i) \ldots t(e_i)>$ matches every list $(v_1 \; v_2 \ldots v_k)$ where each v_j is of the form $t(u_j)$. I.e.:

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

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

Example: The rule

$$((\cup\ <a_1 \dots e_1>) \times A) \rightarrow (\cup\ <(a_1 \times A) \dots (e_1 \times A)>)$$

eliminates unions inside of Cartesian products by distributivity: E.g. it transforms the input $((\cup\ t_1\ t_2\ t_3) \times t')$ to $(\cup\ (t_1 \times t')\ (t_2 \times t')\ (t_3 \times t'))$, where $\cup$ is extended to any number of arguments and used in prefix notation.
As in the example, a rule usually contains some $<a_i \dots e_i>$ together with the corresponding $<t(a_i)\dots t(e_i)>$, one at each side of the rule.

- The constraint $<<\ [\mathit{notarity}\ (i\ n)]\ >>$ blocks the application of the rule for any substitution assigning a list of length n to the argument list identified by the list counter i, i.e. $<a_i \dots e_i>$ resp. $<t(a_i)\dots t(e_i)>$.
Constraints of this type are usually used to exclude special cases of empty lists or lists with one element. Often a rule is applicable for such a special case without performing a real change on the input term. This would violate termination of the rewriting system and therefore constraints of the presented kind have to be used.

- Commutativity is handled by the construct $<\mathit{comm}\ f\ \ <\mathit{choose}\ \ <xxx>>\ \ y>$. Here f is a commutative operator, $<xxx>$ is either $<a_i \dots e_i>$ or $<t(a_i)\dots t(e_i)>$, and y may be a variable, an argument list of variable arity or it may not occur.
When unified with a term with operator f, the elements of the argument list are ordered with respect to a given term ordering. Afterwards $<xxx>$ is unified with the sublist constructed by choosing all elements with special properties from the original list. Possible conditions are:

  - The elements must occur in several lists (denoted by several occurrences of the sublist $<xxx>$ in different *choose*-constructs.
  - Only elements of a given pattern $t(x)$ are chosen (denoted by using $<xxx>= <t(a_i)\dots t(e_i)>$.
  - Combinations of the conditions above.

The remaining elements must be unifyable with y.

Example:

$$((<\mathit{comm}\cup> <\mathit{choose}\ <(a_1 \times B)\ldots(e_1 \times B)>>) \setminus$$
$$(<\mathit{comm}\cup> <\mathit{choose}\ <(a_1 \times C)\ldots(e_1 \times C)>>))\ <<[\mathit{notarity}\ (1\ 0)]>>$$
$$<<[\mathit{notarity}\ (1\ 1)]>> \rightarrow (\cup\ <((a_1 \times B) \setminus (a_1 \times C))\ldots((e_1 \times B) \setminus (e_1 \times C)))$$

This rule can be read as follows: A difference of unions of pairs with identical second arguments for every union and identical sets of first arguments in both unions can be transformed in order to get the set difference inside of the union.

The rule is applicable e.g. on the term

$$(\cup\ (a_1 \times b)\ (a_2 \times b)\ (a_3 \times b)) \setminus (\cup\ (a_2 \times c)\ (a_3 \times c)\ (a_1 \times c))$$

The application results in

$$\begin{aligned}(\cup\ & (a_1 \times B) \setminus (a_1 \times C)\\ & (a_2 \times B) \setminus (a_2 \times C)\\ & (a_3 \times B) \setminus (a_3 \times C))\end{aligned}$$

- Another kind of constraint that is only important in the context of type simplification is the test for some hierarchical ordering:
  $l\ [A \sqsubseteq B] \rightarrow r$ denotes a rule with left hand side l and right hand side r that is only applicable for substitutions fulfilling $A \sqsubseteq B$. The test has to be done according to the definition of $\sqsubseteq$ in section 3.4.

3.5.2 Simplification by equivalence transformation

Equivalent types can be expressed in different ways in the type language defined before. This causes the following problems:

1. Further type inference can be much more expensive because of a too complex representation of types.
2. The comparison is more difficult because of possible differences in the type representations.

To reduce the effort of further computation as much as possible, every newly computed type should be simplified immediately. The simplification improves the comparability of types, as well, though there are representations of the same type which will not be simplified to the same form.

We are using term rewriting as the formalism for simplifying types. To handle commutativity and associativity of $\cup$ and $\cap$ more easily, we define these two constructors as constructors with variable arity in prefix notation:

$$
\begin{aligned}
(\cup) &:= \bot \\
(\cup\ A) &:= A \\
(\cup\ A\ B) &:= A \cup B \\
(\cup\ A_1 \ldots\ A_n\ A_{n+1}) &:= (\cup\ A_1 \ldots A_n) \cup A_{n+1} \text{ für } n \geq 2
\end{aligned}
$$

$$
\begin{aligned}
(\cap) &:= \top \\
(\cap\ A) &:= A \\
(\cap\ A\ B) &:= A \cap B \\
(\cap\ A_1 \ldots\ A_n\ A_{n+1}) &:= (\cap\ A_1 \ldots A_n) \cap A_{n+1} \text{ für } n \geq 2
\end{aligned}
$$

Now we can use the notations of term rewriting with operators of variable arity to rewrite types. We use several rewriting systems, which are applied sequentially.

The first stage ES_1 is used to eliminate $\cup$, $\cap$ and $\setminus$ inside of Cartesian products, pairs and stream-pairs:

ES_1:

$$((\cup\ <a_1 \ldots e_1>) \times A) \rightarrow (\cup\ <(a_1 \times A) \ldots (e_1 \times A)>) \tag{1}$$
$$(A \times (\cup\ <a_1 \ldots e_1>)) \rightarrow (\cup\ <(A \times a_1) \ldots (A \times e_1)>) \tag{2}$$
$$((\cap\ <a_1 \ldots e_1>) \times A) \rightarrow (\cap\ <(a_1 \times A) \ldots (e_1 \times A)>) \tag{3}$$
$$(A \times (\cap\ <a_1 \ldots e_1>)) \rightarrow (\cap\ <(A \times a_1) \ldots (A \times e_1)>) \tag{4}$$
$$((A \setminus C) \times B) \rightarrow (A \times B) \setminus (C \times B) \tag{5}$$
$$(A \times (B \setminus C)) \rightarrow (A \times B) \setminus (A \times C) \tag{6}$$
$$((\cup\ <a_1 \ldots e_1>)\ .\ A) \rightarrow (\cup\ <(a_1\ .\ A) \ldots (e_1\ .\ A)>) \tag{7}$$
$$(A\ .\ (\cup\ <a_1 \ldots e_1>)) \rightarrow (\cup\ <(A\ .\ a_1) \ldots (A\ .\ e_1)>) \tag{8}$$
$$((\cap\ <a_1 \ldots e_1>)\ .\ A) \rightarrow (\cap\ <(a_1\ .\ A) \ldots (e_1\ .\ A)>) \tag{9}$$
$$(A\ .\ (\cap\ <a_1 \ldots e_1>)) \rightarrow (\cap\ <(A\ .\ a_1) \ldots (A\ .\ e_1)>) \tag{10}$$
$$((A \setminus C)\ .\ B) \rightarrow (A\ .\ B) \setminus (C\ .\ B) \tag{11}$$
$$(A\ .\ (B \setminus C)) \rightarrow (A\ .\ B) \setminus (A\ .\ C) \tag{12}$$
$$[(\cup\ <a_1 \ldots e_1>)\ .\ A] \rightarrow (\cup\ <[a_1\ .\ A] \ldots [e_1\ .\ A]>) \tag{13}$$
$$[A\ .\ (\cup\ <a_1 \ldots e_1>)] \rightarrow (\cup\ <[A\ .\ a_1] \ldots [A\ .\ e_1]>) \tag{14}$$
$$[(\cap\ <a_1 \ldots e_1>)\ .\ A] \rightarrow (\cap\ <[a_1\ .\ A] \ldots [e_1\ .\ A]>) \tag{15}$$
$$[A\ .\ (\cap\ <a_1 \ldots e_1>)] \rightarrow (\cap\ <[A\ .\ a_1] \ldots [A\ .\ e_1]>) \tag{16}$$
$$[(A \setminus C)\ .\ B] \rightarrow [A\ .\ B] \setminus [C\ .\ B] \tag{17}$$
$$[A\ .\ (B \setminus C)] \rightarrow [A\ .\ B] \setminus [A\ .\ C] \tag{18}$$

Lemma 3.4 *The term rewriting system ES_1 is terminating.*

Proof: Consider a recursive path ordering over the following partial ordering on the set of function symbols:

$$\left.\begin{array}{c} \times \\ (\ .\ ) \\ [\ .\ ] \end{array}\right\} > \left\{\begin{array}{l} \cup \\ \cap \\ \setminus \end{array}\right.$$

This recursive path ordering is a termination ordering for ES_1. □

Local confluence does not hold for the given term rewriting system, but every critical pair is reduced to a common normal form by the following stages.

Lemma 3.5 *Processing a type t with ES_1 yields an equivalent type t', i.e. $V(t) = V(t')$.*

Proof: See appendix A. □

The second stage ES_2 is used to put type difference inside of unions, intersections and products (i.e. Cartesian products, pairs and stream-pairs) wherever it is possible:

ES_2:

$$(A \times B) \setminus (A \times C) \rightarrow A \times (B \setminus C) \tag{1}$$

$$(A \times C) \setminus (B \times C) \rightarrow (A \setminus B) \times C \tag{2}$$

$$\begin{aligned}&((<\mathit{comm} \cup> <\mathit{choose}\ <(a_1 \times B) \ldots (e_1 \times B)>>) \setminus \\ &\quad (<\mathit{comm} \cup> <\mathit{choose}\ <(a_1 \times C) \ldots (e_1 \times C)>>))\ <<[\mathit{notarity}\ (1\ 0)]>> \\ &\quad <<[\mathit{notarity}\ (1\ 1)]>> \rightarrow (\cup\ <((a_1 \times B) \setminus (a_1 \times C)) \ldots ((e_1 \times B) \setminus (e_1 \times C)))\end{aligned} \tag{3}$$

$$\begin{aligned}&((<\mathit{comm} \cup> <\mathit{choose}\ <(B \times a_1) \ldots (B \times e_1)>>) \setminus \\ &\quad (<\mathit{comm} \cup> <\mathit{choose}\ <(C \times a_1) \ldots (C \times e_1)>>))\ <<[\mathit{notarity}\ (1\ 0)]>> \\ &\quad <<[\mathit{notarity}\ (1\ 1)]>> \rightarrow (\cup\ <((B \times a_1) \setminus (C \times a_1)) \ldots ((B \times e_1) \setminus (C \times e_1)))\end{aligned} \tag{4}$$

$$\begin{aligned}&((<\mathit{comm} \cap> <\mathit{choose}\ <(a_1 \times B) \ldots (e_1 \times B)>>) \setminus \\ &\quad (<\mathit{comm} \cap> <\mathit{choose}\ <(a_1 \times C) \ldots (e_1 \times C)>>))\ <<[\mathit{notarity}\ (1\ 0)]>> \\ &\quad <<[\mathit{notarity}\ (1\ 1)]>> \rightarrow (\cap\ <((a_1 \times B) \setminus (a_1 \times C)) \ldots ((e_1 \times B) \setminus (e_1 \times C)))\end{aligned} \tag{5}$$

$$\begin{aligned}&((<\mathit{comm} \cap> <\mathit{choose}\ <(B \times a_1) \ldots (B \times e_1)>>) \setminus \\ &\quad (<\mathit{comm} \cap> <\mathit{choose}\ <(C \times a_1) \ldots (C \times e_1)>>))\ <<[\mathit{notarity}\ (1\ 0)]>> \\ &\quad <<[\mathit{notarity}\ (1\ 1)]>> \rightarrow (\cap\ <((B \times a_1) \setminus (C \times a_1)) \ldots ((B \times e_1) \setminus (C \times e_1)))\end{aligned} \tag{6}$$

$$(A\ .\ B) \setminus (A\ .\ C) \rightarrow (A\ .\ (B \setminus C)) \tag{7}$$

$$(A\ .\ C) \setminus (B\ .\ C) \rightarrow ((A \setminus B)\ .\ C) \tag{8}$$

$$\begin{aligned}&((<\mathit{comm} \cup> <\mathit{choose}\ <(a_1\ .\ B) \ldots (e_1\ .\ B)>>) \setminus \\ &\quad (<\mathit{comm} \cup> <\mathit{choose}\ <(a_1\ .\ C) \ldots (e_1\ .\ C)>>))\ <<[\mathit{notarity}\ (1\ 0)]>> \\ &\quad <<[\mathit{notarity}\ (1\ 1)]>> \rightarrow (\cup\ <((a_1\ .\ B) \setminus (a_1\ .\ C)) \ldots ((e_1\ .\ B) \setminus (e_1\ .\ C)))\end{aligned} \tag{9}$$

$$
\begin{aligned}
&((<\mathit{comm}\ \cup>\ <\mathit{choose}\ \ <(B\ .\ a_1)\ldots(B\ .\ e_1)>>)\setminus\\
&\qquad(<\mathit{comm}\ \cup>\ <\mathit{choose}\ \ <(C\ .\ a_1)\ldots(C\ .\ e_1)>>))\ \ <<[\mathit{notarity}\ (1\ 0)]>>\\
&\qquad\quad <<[\mathit{notarity}\ (1\ 1)]>>\ \rightarrow (\cup\ \ <((B\ .\ a_1)\setminus(C\ .\ a_1))\ldots((B\ .\ e_1)\setminus(C\ .\ e_1)))
\end{aligned}
\tag{10}
$$

$$
\begin{aligned}
&((<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <(a_1\ .\ B)\ldots(e_1\ .\ B)>>)\setminus\\
&\qquad(<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <(a_1\ .\ C)\ldots(e_1\ .\ C)>>))\ \ <<[\mathit{notarity}\ (1\ 0)]>>\\
&\qquad\quad <<[\mathit{notarity}\ (1\ 1)]>>\ \rightarrow (\cap\ \ <((a_1\ .\ B)\setminus(a_1\ .\ C))\ldots((e_1\ .\ B)\setminus(e_1\ .\ C)))
\end{aligned}
\tag{11}
$$

$$
\begin{aligned}
&((<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <(B\ .\ a_1)\ldots(B\ .\ e_1)>>)\setminus\\
&\qquad(<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <(C\ .\ a_1)\ldots(C\ .\ e_1)>>))\ \ <<[\mathit{notarity}\ (1\ 0)]>>\\
&\qquad\quad <<[\mathit{notarity}\ (1\ 1)]>>\ \rightarrow (\cap\ \ <((B\ .\ a_1)\setminus(C\ .\ a_1))\ldots((B\ .\ e_1)\setminus(C\ .\ e_1)))
\end{aligned}
\tag{12}
$$

$$[A\ .\ B]\setminus[A\ .\ C]\rightarrow[A\ .\ (B\setminus C)] \tag{13}$$

$$[A\ .\ C]\setminus[B\ .\ C]\rightarrow[(A\setminus B)\ .\ C] \tag{14}$$

$$
\begin{aligned}
&((<\mathit{comm}\ \cup>\ <\mathit{choose}\ \ <[a_1\ .\ B]\ldots[e_1\ .\ B]>>\ )\setminus\\
&\qquad(<\mathit{comm}\ \cup>\ <\mathit{choose}\ \ <[a_1\ .\ C]\ldots[e_1\ .\ C]>>\ ))\ \ <<[\mathit{notarity}\ (1\ 0)]>>\\
&\qquad\qquad <<[\mathit{notarity}\ (1\ 1)]>>\ \rightarrow (\cup\ \ <([a_1\ .\ B]\setminus[a_1\ .\ C])\ldots([e_1\ .\ B]\setminus[e_1\ .\ C]))
\end{aligned}
\tag{15}
$$

$$
\begin{aligned}
&((<\mathit{comm}\ \cup>\ <\mathit{choose}\ \ <[B\ .\ a_1]\ldots[B\ .\ e_1]>>\ )\setminus\\
&\qquad(<\mathit{comm}\ \cup>\ <\mathit{choose}\ \ <[C\ .\ a_1]\ldots[C\ .\ e_1]>>\ ))\ \ <<[\mathit{notarity}\ (1\ 0)]>>\\
&\qquad\qquad <<[\mathit{notarity}\ (1\ 1)]>>\ \rightarrow (\cup\ \ <([B\ .\ a_1]\setminus[C\ .\ a_1])\ldots([B\ .\ e_1]\setminus[C\ .\ e_1]))
\end{aligned}
\tag{16}
$$

$$
\begin{aligned}
&((<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <[a_1\ .\ B]\ldots[e_1\ .\ B]>>\ )\setminus\\
&\qquad(<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <[a_1\ .\ C]\ldots[e_1\ .\ C]>>\ ))\ \ <<[\mathit{notarity}\ (1\ 0)]>>\\
&\qquad\qquad <<[\mathit{notarity}\ (1\ 1)]>>\ \rightarrow (\cap\ \ <([a_1\ .\ B]\setminus[a_1\ .\ C])\ldots([e_1\ .\ B]\setminus[e_1\ .\ C]))
\end{aligned}
\tag{17}
$$

$$
\begin{aligned}
&((<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <[B\ .\ a_1]\ldots[B\ .\ e_1]>>\ )\setminus\\
&\qquad(<\mathit{comm}\ \cap>\ <\mathit{choose}\ \ <[C\ .\ a_1]\ldots[C\ .\ e_1]>>\ ))\ \ <<[\mathit{notarity}\ (1\ 0)]>>\\
&\qquad\qquad <<[\mathit{notarity}\ (1\ 1)]>>\ \rightarrow (\cap\ \ <([B\ .\ a_1]\setminus[C\ .\ a_1])\ldots([B\ .\ e_1]\setminus[C\ .\ e_1]))
\end{aligned}
\tag{18}
$$

Lemma 3.6 *The term rewriting system ES_2 is terminating.*

Proof: Consider a recursive path ordering over the following partial ordering on the set of function symbols:

$$\setminus > \left\{ \begin{array}{c} \times \\ (\,.\,) \\ [\,.\,] \end{array} \right\} > \begin{cases} \cup \\ \cap \end{cases}$$

This is a termination ordering for ES_2 (when the *notarity*-constraints block every non-productive rule execution) and thus ES_2 is terminating. □

Remark 3.7 *One should note that the termination proof above implicitly relies on the existence of the* notarity-*constraints of the rules. These constraints exclude those rule executions that do not change the term at all. Thus, for every termination proof using termination orderings one must make sure that there are no unifiers of the left and the right part of the rule that are not blocked.*

ES_2 is not locally confluent. E.g. for arbitrary types A and B the term

$$(A \times B) \setminus (A \times B)$$

could be normalized to:

- $A \times (B \setminus B)$ (by rule (1))
- $(A \setminus A) \times B$ (by rule (2))

Further normalizing these expressions in order to yield local confluence would be possible in the following stage since $V(A \times (B \setminus B)) = V((A \setminus A) \times B) = \{\perp\}$, but this simplification is not always wanted because during type propagation, arguments $\perp$ to type constructors occur as dummy parameters just expressing the structure given by the constructor. These structure information should not be normalized away.

Lemma 3.8 *Type simplification of a type t by ES_2 does not change the represented set of values, i.e. the resulting type t' fulfills $V(t) = V(t')$.*

Proof: See appendix A. □

In the third stage ES_3, type difference is replaced using the type complement and type intersection. Furthermore nested union resp. intersection is eliminated, and precalculations are done, if the result of some union or intersection is $\top$ or $\perp$ respectively. When unions and intersections are nested the result is a union eventually containing intersections.

ES_3:

$$A \setminus B \to A \cap \mathcal{C}B \tag{1}$$

$$\mathcal{C}\mathcal{C}A \to A \tag{2}$$

$$(\cup\ <1\ldots>\ (\cup\ <2\ldots>)\ <3\ldots>) \to (\cup\ <1\ldots><2\ldots><3\ldots>) \tag{3}$$

$$(\cap\ <1\ldots>\ (\cap\ <2\ldots>)\ <3\ldots>) \to (\cap\ <1\ldots><2\ldots><3\ldots>) \tag{4}$$

$$(\mathcal{C}(\cup <a_1\ldots e_1>)) \to (\cap <(\mathcal{C}a_1)\ldots(\mathcal{C}e_1)>) \tag{5}$$

$$(\mathcal{C}(\cap <a_1\ldots e_1>)) \to (\cup <(\mathcal{C}a_1)\ldots(\mathcal{C}e_1)>) \tag{6}$$

$$(\cup\ A) \to A \tag{7}$$

$$(\cap\ A) \to A \tag{8}$$

$$(\cap\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)[A \sqsubseteq \mathcal{C}B] \to \bot \tag{9}$$

$$(\cap\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)[B \sqsubseteq \mathcal{C}A] \to \bot \tag{10}$$

$$(\cup\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)[\mathcal{C}A \sqsubseteq B] \to \top \tag{11}$$

$$(\cup\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)[\mathcal{C}B \sqsubseteq A] \to \top \tag{12}$$

$$\begin{aligned}&(\cap\ <1\ldots>\ (\cup\ <a_2\ldots e_2>)\ <3\ldots>) \to \\ &\qquad (\cup\ <(\cap\ <1\ldots>\ a_2\ <3\ldots>)\ldots(\cap\ <1\ldots>\ e_2\ <3\ldots>)>)\end{aligned} \tag{13}$$

$$\begin{aligned}&(\cup\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)\ <<[A \sqsubseteq B]>>\ \to \\ &\qquad (\cup\ <1\ldots><2\ldots>\ B\ <3\ldots>)\end{aligned} \tag{14}$$

$$\begin{aligned}&(\cup\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)\ <<[B \sqsubseteq A]>>\ \to \\ &\qquad (\cup\ <1\ldots>\ A\ <2\ldots><3\ldots>)\end{aligned} \tag{15}$$

$$\begin{aligned}&(\cap\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)\ <<[B \sqsubseteq A]>>\ \to \\ &\qquad (\cap\ <1\ldots><2\ldots>\ B\ <3\ldots>)\end{aligned} \tag{16}$$

$$\begin{aligned}&(\cap\ <1\ldots>\ A\ <2\ldots>\ B\ <3\ldots>)\ <<[A \sqsubseteq B]>>\ \to \\ &\qquad (\cap\ <1\ldots>\ A\ <2\ldots><3\ldots>)\end{aligned} \tag{17}$$

Lemma 3.9 *The term rewriting system ES_3 is terminating.*

Proof: Consider the recursive path ordering over the following ordering on the set of function symbols:

$$\setminus\ >\ \mathcal{C}\ >\ \cap\ >\ \cup > \begin{cases}\top \\ \bot\end{cases}$$

This is a termination ordering for ES_3, and thus ES_3 is terminating. □

Lemma 3.10 *Every type t simplified by ES_3 to type t' denotes the same set of values before and after the simplification, i.e. $V(t) = V(t')$.*

Proof: See appendix A. $\square$

Local confluence does not hold for this rewriting system. E.g. let A, B, C and D be types with $\mathcal{C}B \sqsubseteq C$. Then the term

$$(\cap\ A\ (\cup\ B\ C)\ D)$$

has the following different normal forms:

1. $(\cap\ A\ D)$ (by applying rule (11) and then rule (16) or (17))
2. $(\cup\ (\cap\ A\ B\ D)\ (A\ C\ D))$ (by applying rule (13))

The rules causing the problem here are (11) (analogously (12)) and (13).

Correcting this would cause great costs. Since it is impossible to normalize types completely, as stated below, we will not worry about this problem. The given ordering of rules defines priorities that provide useful results.

The missing possibility of normalizing types completely can be motivated by the type $\mu X.((list\ X) \cup (\top \setminus (list\ \bot)))$ which is identical to the simple type $\top$. A complete normalization of this type would include the elimination of a fixed point constructor which seems to be quite difficult. Therefore we use an approximating normalization that is easier to compute.

The following theorem shows that the type simplification defined by ES_1, ES_2 and ES_3, can be performed without causing any problems:

Theorem 3.11 (Correctness of equivalent type simplification) *To every given type* t *the type simplification algorithm defined by successively applying* ES_1*,* ES_2 *and* ES_3 *provides a simplified type* t' *such that* $\mathrm{V(t)} = \mathrm{V(t')}$*.*

Proof: Termination follows from lemmas 3.4, 3.6 and 3.9, type equivalence of the resulting term from lemmas 3.5, 3.8 and 3.10. $\square$

3.5.3 Simplification by generalization

The algorithm for type propagation in section 4 needs to make assumptions about input types of functions that we cannot obtain using equivalence transformations. In particular this is the case for operations on lists (e.g. *append*) that cannot cope with pairs even if these pairs describe correct lists. In this case we have to generalize the given input type to meet the conditions on the expected input types for this function.

Generalizing nested pairs to lists: Because of the definition of an extra list type constructor (and a stream type constructor) a function may expect a list as input parameter, but receive a pair denoting a correct list, instead. The following term rewriting system GS_l normalizes nested pairs to the corresponding list expression if it exists. Furthermore GS_l normalizes unions and intersections of lists with a single list as result.

GS_l:

$$S_{()} \rightarrow (list\ \bot) \tag{1}$$
$$(A\ .\ (list\ B)) \rightarrow (list\ (A \cup B)) \tag{2}$$
$$(\cup\ <(list\ a_1) \ldots (list\ e_1)>) \rightarrow (list\ (\cup\ <a_1 \ldots e_1>)) \tag{3}$$
$$(\cap\ <(list\ a_1) \ldots (list\ e_1)>)) \rightarrow (list\ (\cap\ <a_1 \ldots e_1>)) \tag{4}$$
$$(list\ B) \setminus (list\ C) \rightarrow (list\ B)^4 \tag{5}$$

The original type has to be replaced by the result of rewriting only if this result does not contain a pair constructor any more.
An analogous TRS GS_s has to be used to transform nested stream-pairs to stream types.

Theorem 3.12 *The term rewriting system GS_l (and analogously GS_s) is terminating and simplifies every type t to a type t' which fulfills $t \sqsubseteq t'$.*

Proof: To prove termination of GS_l we consider the recursive path ordering over the following ordering on the set of function symbols:

$$(\ .\ ) > \setminus > \cap > \cup > S_{()} > list > \bot$$

This is a termination ordering for GS_l and thus GS_l is terminating.

$t \sqsubseteq t'$ holds, because for every rule $l \rightarrow r$ of GS_l $l \sqsubseteq r$ is true. For rules (1), (2), (5) this follows directly from the definition of $\sqsubseteq$. For rules (3) and (4), induction on the number of arguments to $\cup$ resp. $\cap$ is needed. □

Theorem 3.13 *The term rewriting system GS_l (and also GS_s) is confluent.*

Proof: Local confluence holds because there are no critical pairs between the rules of GS_l (or GS_s). With the proof of termination in theorem 3.12 confluence is implied. □

[4] If B and C are not disjunct, lists of type $(list\ B) \setminus (list\ C)$ may contain elements of type C. Since we want to omit set differences for whole lists, we must add $V((list\ C))$ to the denoted set of values.

3.6 Testing types for common elements

As section 2.2 states, the question whether two types are disjoint or have common elements is fundamental for our type system. More exactly we want to know whether two types have common elements except of the non-terminating computation $\perp$. In the following we do not regard the element $\perp$ of a type any further and call an intersection just containing $\perp$ empty.

In this section we present an algorithm CE which approximates the following problem: Given to types A and B eventually containing variables and two flags $a, b \in \{\texttt{any}, \texttt{all}\}$. Does for any/all instances A' of A resp. B' of B hold that $V(A')$ and $V(B')$ are disjoint?
The approximation is done in such a way that statements of disjointness by the algorithm are reliable, but statements of non-disjointness are not.

If two types are disjoint only because one of them contains $\perp$, a special flag $\perp_A$ resp. $\perp_B$ is set. This is necessary because $\perp$ can denote a dummy value if only the structure of a type is known. We do not want to get disjointness results for that.

The check of two given types A and B for common elements is done similar to the subtyping algorithm for recursive types presented in [AC93]. According to the top-level type constructors of A and B we have a case distinction in the algorithm:

CE:
Input: Two types A and B and corresponding flags $a, b \in \{\texttt{any}, \texttt{all}\}$
Output:

- *disjoint*, if for any/all (according to a) instances A' of A and for any/all (according to b) instances B' of B $V(A')$ and $V(B')$ are disjoint. The additional flags $\perp_A$ and $\perp_B$ show whether the only reason for A and B being disjoint is an occurrence of $\perp$ in A resp. B.
- *non-disjoint*, if no reason for disjointness could be found. If $a = \texttt{all}$ (resp. $b = \texttt{all}$) then a substitution σ_A (resp. σ_B) is returned that specializes A (resp. B) as much as possible without loosing common elements.

1. $A = (\cup\ A_1 \dots A_n)$: If $\exists i \in \{1, \dots, n\} \,.\, A_i \cap B \neq \perp$ then return *non-disjoint*.
2. $B = (\cup\ B_1 \dots B_n)$: If $\exists i \in \{1, \dots, n\} \,.\, B_i \cap A \neq \perp$ then return *non-disjoint*.
3. $A = (\cap\ A_1 \dots A_n)$: If $\forall i \in \{1, \dots, n\} \,.\, A_i \cap B \neq \perp$ then return *non-disjoint*.
   This is a case of approximating the real question of common elements: Since we cannot check whether all A_i have *the same* element in common with B, we just check whether all A_i have *any* element in common with B.
4. $B = (\cap\ B_1 \dots B_n)$: If $\forall i \in \{1, \dots, n\} \,.\, B_i \cap A \neq \perp$ then return *non-disjoint*.

5. $A = \mathcal{C}A'$: If $B \not\sqsubseteq A'$ then return *non-disjoint*.

6. $B = \mathcal{C}B'$: If $A \not\sqsubseteq B'$ then return *non-disjoint*.

7. $A = \mu X.A'(X)$ and B does not contain a fixed point type constructor: If B and $A'(\mu X.A'(X))$ have common elements then return *non-disjoint*. (Maybe A has to be unfolded more than once.)

8. $B = \mu X.B'(X)$ and A does not contain a fixed point type constructor: If A and $B'(\mu X.B'(X))$ have common elements then return *non-disjoint* else return *disjoint*.

9. $A = \mu X.A'(X)$ and B contains a fixed point type constructor: The procedure of unfolding in principle is the same as in (7). We just have to mark every step of unfolding, to prevent an infinite loop. If e.g. A was unfolded once when checking A and some $\tilde{B}$, we will not unfold A for comparison with $\tilde{B}$, again, because this would lead to an infinite loop. If such a test finds common elements then return *non-disjoint*. If no common elements were found until some unfolding must be repeated then return *disjoint*.

10. $A = \mu X.A'(X)$ and B contains a fixed point type constructor: As in (9) when replacing A and B.

11. A is a type variable V: If $a = \texttt{any}$ and $B = \top$ or $B = A$ or if $a = \texttt{all}$ and $B \neq \bot$ then return *non-disjoint*.
    If $a = \texttt{all}$ then add $V := B$ to σ_A.

12. B is a type variable W: If $b = \texttt{any}$ and $A = \top$ or $A = B$ or if $b = \texttt{all}$ and $A \neq \bot$ then return *non-disjoint*.
    If $b = \texttt{all}$ then add $W := A$ to σ_B.

13. A is a base type: If B is a base type and

$$(A \neq \bot) \wedge (B \neq \bot) \wedge ((A \sqsubseteq B) \vee (B \sqsubseteq A))$$

    holds, then return *non-disjoint*.

14. $A = (\times\ A_1 \ldots A_n)$: If $B = (\times\ B_1 \ldots B_n)$ and $\forall i \in \{1, \ldots, n\} \,.\, A_i \cap B_i \neq \bot$ then return *non-disjoint*.

15. $A = (A_1\ .\ A_2)$: If $B = (B_1\ .\ B_2)$ and A_1 and B_1 as well as A_2 and B_2 have a nonempty intersection then return *non-disjoint*.

16. $A = [A_1\ .\ A_2]$: If $B = [B_1\ .\ B_2]$ and A_1 and B_1 as well as A_2 and B_2 have a nonempty intersection then return *non-disjoint*.

17. $A = A_1 \to A_2$: If $B = B_1 \to B_2$ and A_1 and B_1 as well as A_2 and B_2 have a nonempty intersection then return *non-disjoint*.

18. A is a frame type: If B is a frame type, too, then return *non-disjoint*.

19. A is an environment type: If B is a environment type consisting of the same number of frames then return *non-disjoint*. (Note that `env` contains environments consisting of any number of frames.)

20. In every other case return *disjoint*.

Theorem 3.14 *The algorithm CE only states two simplified types A and B as disjoint if for any/all assignments of the type variables in A resp. B the resulting instances A' and B' of A and B are disjoint, i.e. $V(A') \cap V(B') = \emptyset$.*

Proof: See appendix A. □

Remark 3.15 *Following a hint by Christoph Wimmer we want to point out that the implementation of CE is sufficient to check the subtype relation $\sqsubseteq$, as well, because:*

$$A \sqsubseteq B \Leftrightarrow A \cap \mathcal{C}B = \bot$$

4 The type propagation algorithm

In many efficient sound type systems beginning with [Mil78] unification is the basis of type inference. Because of the undirected effect of unification, these systems are able to compute the input types of parameters as well as the overall type of an expression without further information. As stated in section 2.2 our complete system cannot use unification as base algorithm for type inference and thus cannot take the benefit of its undirected effects.

We therefore present an algorithm for solving the following subgoal: Given an expression A and types for all parameters and free variables of A, we want to approximate the type t of A by a supertype t' and try to get a quite small error $t' \setminus t$. The type t' can be used to check consistency of an expression with an expected output type for a given set of input types which is helpful whenever I/O-behavior of the analyzed program can be specified. If our system answers that no type of A exists, we want the algorithm to report a function call in the program that cannot be evaluated without type error for any set of input parameters given by an instance of the input types.
We will just need a specification of the main function of the analyzed program even if the program is defined by several more functions.

4.1 Types of evaluations

In contrast to conventional sound type systems, our type system will not compute principal types of expressions. We are rather interested in typing evaluations of expressions with known types of free variables and parameters. We will give some examples first:

- (+ 1 2 3) has type `nat`
- (< 4 3) has type `bool`
- ($*$ 2 x) can only be typed if a type of x is given. E.g. with $x \in V(\texttt{nat})$ we get the type `nat` for ($*$ 2 x); with $x \in V((\textit{list}\ \texttt{bool}))$ we get a type error.
- (*lambda* (x) ($*$ 2 x)) needs a given input type for the parameter x. With input type $x \in V(\texttt{rat})$ we get the overall type $\texttt{rat} \rightarrow \texttt{rat}$.
- (*let* ((x -4))($*$ x x)) has type `nat`. We do not need a type for x in this case, because x is not free in the given expression, but bound to -4.[5]
- (*lambda* (x) (*let* ((y ($*$ x x))) (+ y 3))) needs an input type for parameter x, but no type for variable y, which is bound in a let-expression. E.g. $x \in V(\texttt{int})$ implies the overall type $\texttt{int} \rightarrow \texttt{nat}$.

We will now give the formal definition of types of evaluations:

Definition 4.1 (Input typing) *Let* a *be a functional expression with the set* $\mathrm{FV(a)}$ *of free variables (including parameters). Every function* $\mathrm{ET(a)} : \mathrm{FV(a)} \rightarrow \mathcal{T}$ *is an input typing of* a.
Notations: *Let* ET *be an input typing.* $\mathrm{ET}_{[\mathrm{s_1:t_1,\ldots,s_k:t_k}]}$ *denotes the input typing which arises from* ET *by assigning the type* $\mathrm{t_i}$ *to symbol* $\mathrm{s_i}$ *for every* $\mathrm{i} \in \{1,\ldots,\mathrm{k}\}$*. Let* e *be an environment.* $\mathrm{ET_e}$ *denotes the input typing we obtain by updating the type bindings of* ET *according to the type bindings given by* e*.*

Definition 4.2 (Types of an expression) *Let* a *be an expression and* $\mathrm{ET(a)}$ *an input typing of* a*. Let* $\mathrm{W(a, ET(a))}$ *denote the set of all values* a *can adopt under* $\mathrm{ET(a)}$*. Every type* t *fulfilling* $\mathrm{W(a, ET(a))} \subseteq \mathrm{V(t)}$ *is called a type of* a *under the input typing* $\mathrm{ET(a)}$*.*

Definition 4.3 (Exact type) *Let* a *be an expression,* $\mathrm{ET(a)}$ *an input typing of* a *and* t *a type of* a *under* $\mathrm{ET(a)}$*.* t *is called exact type of* a *under* $\mathrm{ET(a)}$ *if* $\mathrm{W(a, ET(a))} = \mathrm{V(t)}$*.*

Whenever we talk of completeness of our system in the rest of the paper, we refer to the following definition:

[5] The output type `nat` results because a squared number is always non-negative. This is an example of loss of type information, because the type system can type the product of integer numbers only as `int` in general.

Definition 4.4 (completeness) *The type propagation system is complete, if the following holds: Given a program* P *and an input typing* ET *the system accepts* P *under* ET *if one of the following cases holds:*

1. ET *does not contain a free type variable and there is at least one set of input parameters that is consistent to* ET *and causes a type error free run of* P.
2. ET *contains free type variables and for* P *and for every input typing* ET′ *derived from* ET *by instantiating the type variables with types (1) holds.*

Note that we refine the property of completeness as given in section 1 by (2): Since our type propagation algorithm does not introduce any new free type variables, any free type variable occurring in some type must occur in the user-given input typing, too. By using free type variables the user states that the type at the corresponding argument position is polymorphic and the program works fine when we use any type at this position. Thus we report a type error whenever we can find an instantiation of a free type variable that forces a type error no matter if there are other instantiations that work fine.

4.2 The principle of type propagation

The disadvantage of not using unification for type inference can be compensated by using input typings. Thus, we can use type inference in a directed manner that will be called type propagation: Instead of collecting constraints on the types of different calls and solving the resulting system, we propagate the known input typing along the execution paths through the program and calculate the types of all subexpressions directly. The rules used to propagate the types depend on the functions used in the program. Rules for type propagation are given in section 4.3.

The flow of information during the propagation is determined by the call-graph $G_A = (V_A, R_A)$ of the program. The node set V_A is the set of all functions called by the program, including the main function. R_A contains an arrow (v_1, v_2) if function v_1 calls function v_2.

If G_A consists of strongly connected components with one node only and contains no loops we propagate the input types along the partial ordering given by the call-graph. First we propagate the input types along the partial order to nodes with no successor which denote predefined functions. We use the known rules for the predefined functions to transform the input types to output types which are propagated back to the main function. Different calls of the same function hereby have to be typed independently if their input typings differ. Calls with identical input typing are of the same type which can be cached.

Applicability and efficiency of the propagation rules for some predefined functions depend on the correct order in which the arguments are typed: To type a call of *map* we have to type the second argument (the list of input arguments) first, to know the input typing to check the given function in the first argument. In general every predefined function which type definition contains the same type variable more then once should be checked for applicability and efficiency of the different typing orders.
Predefined functions with such order constraints are mentioned in section 4.3.3.

If G_A contains a loop or a strongly connected component with more then one node, we have a recursively defined function. We describe the treatment of recursion in section 4.4.

Type propagation for a main function f and an input typing ET results in a supertype of the exact type of f under ET. Let A_p be the propagated output type for f under ET and A_g the *exact* expected output type of f under ET. Then we can conclude a type error only if $A_g \not\subseteq A_p$.
Normally we only know a supertype A'_g of A_g instead of A_g itself. Therefore, we must not test for inclusion but for the existence of common elements of A_p and A'_g. We conclude a type error if $A'_g \cap A_p = \emptyset$.

4.3 Propagation rules for types

4.3.1 The generic propagation rule

We propagate types by specifying the types of all arguments of a function and then deriving the type of the function evaluation. Thus, type propagation can be seen as executing the function on types as sets of values instead of single values. In general this derivation is done by the following rule:

$$\frac{ET \vdash a_i : t_i \ (i \in \{1, \dots, k\});\ ET \vdash f : t_1 \times \ldots \times t_k \rightarrow t}{ET \vdash (f\ a_1\ \ldots\ a_k) : t}$$

Hereby $ET \vdash e : t$ means that from the input typing ET the type propagation algorithm infers the type t for the expression e.
By $ET \vdash f : t_1 \times \ldots \times t_k \rightarrow t$ we express that under the input typing ET the function f can handle the input type $t_1 \times \ldots \times t_k$ and returns the output type t. This holds in two cases:
In the case of a predefined function f the set $\mathcal{R}$ of transformation rules in appendix C contains an entry $f : t'_1 \times \ldots \times t'_k \rightarrow t'$ with $t_i \cap t'_i \neq \emptyset$. The set $\mathcal{R}$ as presented in appendix C describes a useful subset of scheme that is sufficient for explaining the idea of type propagation and for giving some examples. E.g. the rule

$$\textit{eq?},\ \textit{eqv?},\ \textit{equal?} : \top \times \top \rightarrow \texttt{bool}$$

(which in fact represents separate rules for *eq?*, *eqv?* and *equal?*) states that *eq?*, *eqv?* and *equal?* expect two arbitrary values and return a boolean. The rule

$$car : (A\ .\ \top) \rightarrow A$$

expresses that *car* expects a pair of any kind. For a pair with the type of the first element known to be of type A the result must be of type A, too.
Note that $\mathcal{R}$ must be adapted if one wants to use other predefined functions.
The test for existence of an appropriate rule in $\mathcal{R}$ and the choice of one (maybe several) such rule is done as follows:

1. Extract the typing rules belonging to the actually checked function from the set $\mathcal{R}$ (cf. appendix C)
2. For every rule do:
   (a) For every argument position do:
      i. Call CE with the following parameters: A is the type from the input typing or the previous type propagation at the actual position, B is the type from the actual typing rule at the actual position, $a =$ `any`, $b =$ `all`.
      ii. If *non-disjoint* is returned and σ_B is valid, i.e. does not contain several assignments of disjoint types to the same variable then instantiate the variables of the actual typing rule according to σ_B. (If there are several assignments of different types to the same variable we use the intersection of all occurring types.)
3. From those rules not yielding *disjoint* for any argument position, choose those restricting the input typing in the least possible way.
4. From the rules from (3) choose the one (sometimes we must choose several) with the most special input types.
5. If no transformation rule could be chosen a type error is found.

If in (3) or (4) several rules are not comparable then we process them independently and use the union of their output types as result.

In the second case of $ET \vdash f : t_1 \times \ldots \times t_k \rightarrow t$ the function f is user-defined and applying the *lambda* rule of subsection 4.3.3 to the body of f yields $ET \vdash body(f) : t_1 \times \ldots \times t_k \rightarrow t$.

If deriving the input types of the function call leads to a type that is defined using type variables, we use $\top$ instead of a type variable A if this variable gives the input type of some argument.
In typing an unknown function we assume the general function type $\bot \rightarrow \top$ for this function.

4.3.2 Input types defined by set operations

The type definitions of some predefined functions are restricted in regard to the valid input types for which the corresponding output types are given. The list of valid input types does not pay attention to the fact that set operations can be used to build up further valid input types. So we have to treat those input types in a special way.

Let f be a function of arity n, with an input typing ET that assigns the type E_l to argument number $l \in \{1, \ldots, l\}$.
Let $E_i = (\cap\ E_{i_1} \ldots E_{i_{m_i}})$ for one $i \in \{1, \ldots, k\}$, and let A_j for $j \in \{1, \ldots, m_i\}$ be the propagated output types of f under the input typings ET_j that arises from ET by assigning type E_{i_j} to argument number i.
The propagated output type A of f under ET is defined as $A = (\cap\ A_1 \ldots A_{m_i})$.
Analogously we get the propagated output type $A = (\cup\ A_1 \ldots A_{m_i})$ if $E_i = (\cup\ E_{i_1} \ldots E_{i_{m_i}})$.

If $E_i = \mathcal{C}E'_i$ we have to construct a new input type that contains all elements of E_i which are relevant in its function as input type for f: Every restricted valid input type (i.e. not a single type variable) I_j of f at position i has to be checked for common elements with $\mathcal{C}E'_i$. If $\mathcal{C}E'_i \cap I_j \neq \emptyset$, I_j is added to some set $\mathcal{I}$. $\mathcal{C}E'_i$ is member of $\mathcal{I}$, as well if there is one type definition of f with a single type variable at input position i. This occurrence of $\mathcal{C}E'_i$ is marked with a flag which prevents it from being decomposed again and is only a valid input type to a free type variable.

If more than one argument is defined using set operations or if one argument is defined by nesting set operations, this procedure must be called recursively.

4.3.3 Special propagation rules for certain functions

In this section we describe some predefined functions that must be treated in a special way during type propagation:

- *if* is typed according to the following rule:

$$\frac{ET \vdash p : \texttt{bool},\ ET \vdash b : t_b,\ ET \vdash c : t_c}{ET \vdash (if\ p\ b\ c) : t_b \cup t_c}$$

  Special treatment is necessary if the predicate p distinguishes between different types of a free variable or parameter. In this case, b and c can be typed using a restricted input typing with a more special overall type as a result. (The idea of special treatment of *if* and *cond* is similar to conditional types in [AWL94], but without the formal framework given in that paper.)

Furthermore one should inspect only those execution paths that may be evaluated, because of the condition in the current input typing. We assign type $\perp$ to every subexpression that cannot be evaluated. Maybe a warning about potentially dead code should be generated.
If the subtask of typing the subexpressions yields a type error for b or c, the program is not rejected as long as the other execution path is evaluable and does not contain any type errors. If the erroneous execution path is evaluable, a warning should be generated.

- Also in typing *cond*, there are cases where the input typing can be restricted for typing certain subexpressions. In general we use the following propagation rule:

$$\frac{ET \vdash p_i : \texttt{bool}\ (i \in \{1,\dots,k\}),\ ET \vdash a_i : t_i\ (i \in \{1,\dots,k\})}{ET \vdash (\mathit{cond}\ (p_1\ a_1)\ \dots\ (p_k\ a_k)) : \bigcup_{i=1}^{k} t_i}$$

If one of the predicates p_i consists of a type test of a free variable or parameter, we can restrict the input typing for a_i accordingly and for $a_{i+1}, \dots, a_k$ in the inverse manner. Those subexpressions that cannot be evaluated in the actual input typing, should get type $\perp$ for typing *cond*, analogously to *if*.
If a type error is detected in a condition or conditional expression that need not be evaluated, only a type warning is generated.

- *let* is typed with the following rule:

$$\frac{ET \vdash l_1 : t_1, \dots, l_k : t_k,\ ET_{[v_1:t_1,\dots,v_k:t_k]} \vdash a : t}{ET \vdash (\mathit{let}\ ((v_1\ l_1)\ \dots\ (v_k\ l_k))\ a) : t}$$

- For typing the simplified version of *apply* that expects exactly 2 arguments, a function symbol and a list of arguments, we use the following rule:

$$\frac{ET \vdash l : (t_1\ .\ (t_2\ .\ (\dots\ .\ (t_k\ .\ S_{()})))),\ ET \vdash f : t_1 \times t_2 \times \dots \times t_k \to t}{ET \vdash (\mathit{apply}\ f\ l) \in_{ET} t}$$

- *eval* is typed by the following rule:

$$\frac{ET \vdash e : t' \sqsubseteq \texttt{env},\ ET_{t'} \vdash a : t}{ET \vdash (\mathit{eval}\ a\ e) : t}$$

- *lambda* will be typed as follows:

$$\frac{ET_{[x_1:t_1,\dots x_k:t_k]} \vdash e : t}{ET \vdash (\mathit{lambda}\ (x_1\ \dots\ x_k)\ e) : t_1 \times \dots \times t_k \to t}$$

Every lambda expression has to be typed under some input typing that assigns types t_i to all formal parameters x_i. Without knowing types for these parameters, only the generic function type $\perp \to \top$ can be derived.

- We type *map*, using the following rule:

$$\frac{ET \vdash l : (list\ t), ET \vdash f : t \to t'}{ET \vdash (map\ f\ l) : (list\ t')}$$

  In using this rule, one must compute the type of l first, in order to get a input typing for typing f. Without that, the only known type of f is $\perp \to \top$ which gives the overall type $(list\ \top)$.

- For *make-environment* we have the typing rule:

$$\frac{ET \vdash e_1 : t_1, \dots, e_n : t_n}{ET \vdash (make\text{-}environment\ e_1 \dots e_n) : \mathtt{env}_{ET_{[e_1:t_1,\dots,e_n:t_n]}}}$$

- Typing *begin* can be done as follows:

$$\frac{ET_{SE:i-1} \vdash e_i : t_i\ (i \in \{1, \dots, n\})}{ET \vdash (begin\ e_1 \dots e_n) : t_n}$$

  Thereby $ET_{SE:i-1}$ denotes the input typing the arises from ET by evaluating $e_1 \dots e_{i-1}$.

  Some special forms in scheme implicitly include the functionality of begin. To pay attention to that the propagation rules of these special forms could be extended, or one could insert an explicit *begin* before starting the type propagation.

4.4 Type propagation and recursive functions

Whenever recursive function definitions are involved, a completely directed type propagation is no longer possible, because the output type of some function is necessary to compute this type itself. Therefore we derive the desired type in several steps:

1. We compute the possible output type, only inspecting the non-recursive parts of a function.
2. We use an iterated algorithm that converges to the wanted type in many cases.

We do a type propagation for only the non-recursive parts first, to get some output type t_0 that can be used in the first iteration step. This is done by normal non-recursive type propagation after replacing every recursive function call (i.e. all calls that are connected to the actually analyzed function by a cycle in the call graph.) by $\perp$. We should always find such a non-recursive part, because recursive calculations can only stop if for some input the appropriate output can be computed without recursion. If the actually analyzed function

does not contain some non-recursive part, and there is some cycle in the call graph with length > 1, we might start the recursive analysis with a different function that contains a non-recursive part. The distinction between the non-recursive and the recursive function part is normally done by some special path in an *if* or *cond* expression. If no non-recursive branch can be found in any function contained in the cycle, the user should be informed about the non-termination of the program.
If no non-recursive subexpression of a function cannot be evaluated under the given input typing, we assume $\perp$ as output type of the first step. In that case we have to check, whether some recursive call with different input typing may stop the recursive computation.

In every further step number i, the intermediate output type t_{i-1} of the previous step is assigned to the recursive calls. Thus we have typed all function calls and can generate a type t'_i by normal type propagation for non-recursive programs. The new output type is defined as $t_i := t_{i-1} \cup t'_i$ to guarantee an ascending sequence of intermediate output types. The iteration ends as soon as we have no change in the generated output type for the first time. If no fixed point is reached, the resulting infinite loop has to be detected and solved, as described in section 4.5.

Until now, we have simplified the description of our algorithm by assuming that every recursive call is done using the same input typing. If there are more than one input typing, each of them is treated as an independent function.
If the sequence of input typing has descending complexity, its length must be finite. Otherwise an infinite loop can occur that one can handle as described in section 4.5, as well.

4.4.1 An example of iterated type propagation

To give an example we want to type the function *faculty* under the input typing $\{n \leftarrow \texttt{nat}\}$. The definition of *faculty* is done as usual:

```
(define (faculty n)
   (if (= n 0)
       1
      (* n (faculty (-1+ n)))))
```

The first step consists of typing the non-recursive part of the function. This function definition is divided into a non-recursive and a recursive part by the *if*-expression. The "then"-case gives the non-recursive part just consisting of the constant 1 which yields the intermediate output type $t_0 := \texttt{nat}$ as the most special type that contains 1.

Every further step of iterated type propagation generates an overall type as union of the "then"-case and the "else"-case where the "then"-case yields `nat` as before. Type propagation

on the "else"-case is done innermost first after propagating the input typing of n to the arguments:
From the transformation rule of $-1+$ we know that $(-1+\ n)$ yields an integer. Now we have to type *faculty* under the input typing $\{n \leftarrow \texttt{int}\}$. This is done in the same way as the typing described here and yields `int` as output type. From the transformation rule of $*$ on type `int` together with the subtyping rule $\texttt{nat} \sqsubseteq \texttt{int}$ the result type of the "else"-case is `int` and stays the same in the following typing steps, because the output type of *faculty* under input typing $\{n \leftarrow \texttt{int}\}$ does not change.

Thus we get the following intermediate output types:

- $t_0 := \texttt{nat}$
- $t'_1 := t_{then} \cup t_{else} = \texttt{nat} \cup \texttt{int} = \texttt{int}$
  $t_1 := t'_1 \cup t_0 = \texttt{int} \cup \texttt{nat} = \texttt{int}$
- $t'_2 := t_{then} \cup t_{else} = \texttt{nat} \cup \texttt{int} = \texttt{int}$
  $t_2 := t'_2 \cup t_1 = \texttt{int} \cup \texttt{int} = \texttt{int}$
- A further step for calculating t_3 is not done, because the fixed point $t := t_2 = t_1$ is reached and can be returned.

Therefore iterated type propagation on *faculty* with input typing $\{n \leftarrow \texttt{nat}\}$ yields the output type *int* and the rule for typing *lambda*-expressions yields

$$\mathit{faculty} : \texttt{nat} \rightarrow \texttt{int}$$

4.5 Handling infinite loops

Infinite loops in iterated type propagation can occur because of two reasons:

1. The input typing of the recursive call changes from step to step. For a sequence of ascending complexity an infinite loop is possible.
2. The iterated type propagation does not stop, because the intermediate output types generated in every step continue to change.

We have to expect these cases especially if we should use a recursive type that

- should be given as input type in case (1), but was replaced by a special case.

- should be derived as an output type in case (2), but cannot be generated by adopting an input type defined by the fixed point type constructor.

In both cases we must first detect the infinite loop before we can try to resolve it by using a recursive type.

To detect infinite loops, we simply define thresholds for the maximal number of iteration steps for every pair of function/input typing, as well as for the number of different input typings for a function. Whenever one of these thresholds is exceeded, we assume the existence of an infinite loop.

We interpret every infinite loop as indicator that a recursive type is required. In both cases (1) and (2) a sequence of subtypes of the intended type is generated, before the infinite loop is recognized. In section 4.5.1 we present an algorithm that tries to generalize a sequence of types to a common supertype defined by the fixed point type constructor.
If the infinite loop was caused by the not converging output type of a recursive function, there is an alternative approach of propagating the expected output type of the main function backwards to get the expected output type of the actually analyzed function. The result of backwards propagation must be checked for consistency with the analyzed function under the actual input typing. An algorithm for backwards propagation of output types is presented in section 4.6.

It is important to note that generation of recursive types as well as backwards propagation only provide a guess t_g of the needed type and that t_g must still be checked for consistency. This check is done as follows: One further step (resp. two further steps in case of an infinite loop because of changing input typings) of iterated propagation is done, using t_g instead of the type that has caused the infinite loop. The test succeeds if neither the intermediate output type nor the input typing of a recursive call have changed during the last step. If the check fails or no guess was computed, $\top$ is used as output type of the top level function call.

4.5.1 Generating recursive types

This algorithm for generating recursive types has to be called when the standard type propagation runs into an infinite loop because of any of the cases described at the beginning of section 4.5. The algorithm tries to generalize a sequence of types to a recursive type that can be tested for consistency with the inspected function afterwards.

We first describe the method under the assumption that no set operations are used as top level type constructors for one of the types in the given sequence. This is not true in most cases, because the types of non-recursive and recursive parts of the program are usually connected by a union. Hence we will generalize the system later.

The algorithm works as follows: recursive types describe the application of a certain (maybe compound) type constructor to a base type for an arbitrary number of times. Every evaluation step of the recursive subexpression adds one application of this type constructor.
We expect that in many cases there is a uniform change in the type sequence from step to step after a small number of initial steps. This change should be extracted and formulated in a use of the fixed point constructor for iterated application:

1. Generate a sequence of types that should be generalized, up to a fixed number n of iterations. (When detecting infinite loops by a threshold we can use the sequence that was generated up to this threshold before detecting the infinite loop.) Consider a subsequence $i \geq i_0$ for which the sequence of top level constructors changes in a cycle.

2. Find pairs (t_1, t_2) of types from the sequence generated in (1) that fulfill:

   (a) t_1 and t_2 have the same top level type constructor.

   (b) t_1 is used for defining t_2, but t_1 and t_2 are not equivalent. (Notation: $t_1 < t_2$)[6]

   (c) There is no such t' that $t_1 < t' < t_2$.

3. For each pair (t_1, t_2) from (2) construct a type t_2' from (t_1, t_2) by replacing every occurrence of t_1 in t_2 by a new variable X.

4. If all t_2' from (3) are equal[7] we can construct a new recursive type:

$$t_2' = d(X) \rightarrow \mu X.d(X)$$

If this algorithm can be executed successfully, it returns a new type defined by the fixed point type constructor. This type is a candidate to be tested for consistency with the actual function.

The algorithm is extended to handle types with union or intersection as top level constructors as follows: We try to find sequences of elements of these set operations in which we can build pairs according to step (2) as described above. A fixed point type is generated to any of these element sequences, and the results are combined with union or intersection.
This decomposition of set operations has to be done at most two times, since nested set operations are eliminated by the type simplification from section 3.5.2.

[6]We should use a test for semantic equivalence, but because there is no such test available, we have to test for syntactic equivalence after simplifying the types as described in section 3.5.

[7]There may be several different types one of which is supertype of all others, or there are several types that denote the same transformation, but start at different points. In the first case, the most general type is used; in the latter case all types are tested.

As an example we type the function *1+-rev-deep* defined below, that reverses a list containing numbers and sublists of the same kind thereby increasing every number by 1. The typing is done under the input typing $\{l \leftarrow \mu X.(list\ (\mathtt{num} \cup X))\}$.

```
(define (1+-rev-deep l)
  (cond ((null? l) '())
         ; return () for empty input list
        ((list? (car l))
         (append (1+-rev-deep (cdr l))
                  ; reverse sublists
                 (list (1+-rev-deep
                        (car l)))))
        (else
         (append (1+-rev-deep (cdr l))
                 (list (1+ (car l)))))))
```

By iterated type propagation the following sequence of intermediate output types is generated:

- $t_0 := (list\ \bot)$
- $t_1 := (list\ \mathtt{num}) \cup (list\ (list\ \bot))$

  Type simplification as described in section 3.5.3 yields the more general type $\bar{t_1}$ that is used instead of t_1 in the further computation:

  $\bar{t_1} := (list\ (\mathtt{num} \cup (list\ \bot)))$
- $t_2 := (list\ (\mathtt{num} \cup (list\ (\mathtt{num} \cup (list\ \bot)))))$
- $t_3 := (list\ (\mathtt{num} \cup (list\ (\mathtt{num} \cup (list\ (\mathtt{num} \cup (list\ \bot)))))))$
- ...

From t_3 on we get a uniform pattern of the type change of every single step. This gives us the recursive type $t_{guess} := \mu X.(list\ (\mathtt{num} \cup X))$ which passes the test for consistency.

If we directly handle the union of list types instead of generalizing it to a single list type we get the following intermediate output types:

- $t_0 := (list\ \bot)$
- $t_1 := (list\ \mathtt{num}) \cup (list\ (list\ \bot))$

- $t_2 := (list\ (\mathtt{num} \cup (list\ \mathtt{num}))) \cup (list\ (list\ (list\ \bot)))$
- $t_3 := (list\ (\mathtt{num} \cup (list\ (\mathtt{num} \cup (list\ \mathtt{num})))) \cup (list\ (list\ (list\ (list\ \bot))))$
- $\ldots$

The recursive type $t_{guess} := \mu X.(list\ (\mathtt{num} \cup X))$ generated from this list is the same as before.

One should note that the presented algorithm can generate only those recursive types that can be defined with one fixed point constructor. If no recursive type can be generated, or if the generated type fails the consistency check, we have to use the most general type $\top$ instead.

4.6 Backwards propagation of output types

Backwards propagation of output types is very similar to the normal type propagation procedure, except of taking place in reverse order through the call graph G_A defined in section 4.2: output types are propagated along the partial ordering defined by G_A. Backwards propagation through recursive definitions takes place in an iterated manner that is similar to treating recursion in normal type propagation, as presented in section 4.4.
Because of the similarity to normal type propagation, we will only show the differences in this section.

The goal of backwards propagation is the determination of the expected output type of some function. It can be used if this output type cannot be computed by normal type propagation. Knowledge about the expected output type can be useful because of two reasons:

1. Assuming the correctness of the program, the expected output type of a function may be a good guess for its real output type. Thus backwards propagation can be used instead of the generation of recursive types described in 4.5.1 if the propagation runs into an infinite loop. The backwards propagated expected output type of a function can be tested for consistency with that function.

2. Whenever a higher order function gives another function as output, we can only determine the generic function type $\bot \to \top$ without additional information. From the expected output type we can get an input typing for typing the output function.

Backwards propagation does not want to detect type errors itself. Therefore, backwards propagation is canceled if the propagated output type of some predefined function cannot be generated by this function, but the program is not rejected. Practical tests must show, whether this situation arises very often and can help to detect type errors.

Additional problems occur in backwards propagation, because some function normally generates an unique output type for any given input typing, but the inverse may not hold.
For the given expression (*map* f l) with expected output type (*list* A) we can only determine the input types $f \in B \to A$, $l \in$ (*list* B), but the new type variable B can mean anything until we can restrict it by backwards propagation through f. Otherwise we have to assume $B = \top$.

The result A_r of backwards propagation is a supertype of the expected output type A_g. When we test some propagated output type A_p against A_r instead of A_g, we get $A_g \cap A_p \neq \emptyset \Rightarrow A_r \cap A_p \neq \emptyset$. The test can only fail if the test of A_p against A_g would fail, too.
To cope with the fact that we cannot determine a single input type for any output type, we use an alternative set $\mathcal{R}_B$ of type definitions of the predefined functions. These definitions are given in appendix D. For functions with variable arity, we get the actual arity from the expression, we want to type. (Since there can be only one type of parameters that vary in arity, this is always possible.)

Using the type definitions in appendix D we can propagate types backwards with the following rule:

$$\frac{AT \vdash (f\ a_1\ \ldots\ a_n) : t,\ AT \vdash f : t \leftarrow t_1 \times \ldots \times t_n}{AT \vdash a_1 : t_1, \ldots, a_n : t_n}$$

AT denotes an output typing which can be defined analogously to the definition of input typings.
$AT \vdash f : t \leftarrow t_1 \times \ldots \times t_n$ may hold because one of the following reasons:

1. There is an entry $f : t \leftarrow t_1 \times \ldots \times t_n$ in $\mathcal{R}_B$ and t is the most special type of all types t' with an entry $f : t' \leftarrow t'_1 \times \ldots \times t'_n$ in $\mathcal{R}_B$.
2. f is a lambda expression or a symbol bound to a lambda expression and the typing rule for *lambda* infers $f : t \leftarrow t_1 \times \ldots \times t_n$.

Like normal propagation as described in section 4.3, backwards propagation has some functions and special forms that must be treated in a special way (where some of these specialties just take effect in a context of side effects, as described in section 5):

- *begin* (even if it is implicitly used in the definition of other special forms) is normally used to code sequences of expressions including side effects. Especially side effects generating new symbol bindings must be processed back to front by carrying over information about a symbol s to the expression that is bound to s.
  Processing the arguments of *begin* back to front may lead to processing the use of symbols before processing their definition, especially if side effects are involved. In every case in which the actual binding of a symbol is not known, $\top$ is assumed.

- The strategy for processing *let* is similar to *begin*: Constraints on a symbol s are propagated to the expression defining s.

- For back-propagation on *if*, we normally use the rule

$$\frac{AT \vdash (if\ p\ a\ b) : t}{AT \vdash p : \texttt{bool},\ AT \vdash a, b\ : t}$$

  This rule however only applies if p can be both true and false. If this condition does not hold, then $\top$ instead of t has to be back-propagated through the subexpression that cannot be evaluated. Doubts in the opportunity of evaluating an an expression already causes the propagation of $\top$ instead of t.

- Backwards propagation through *cond* is done as follows:

$$\frac{AT \vdash (cond\ (p_1\ a_1) \ldots (p_k\ a_k)) : t}{AT \vdash p_i : \texttt{bool},\ AT \vdash a_i : t\ (i \in \{1, \ldots, k\})}$$

  For the same reason as for *if*, $\top$ must be back-propagated instead of t for any expression a_i that may not be evaluable.

- Backwards propagation on *lambda* takes place as follows:

$$\frac{AT_{[e:t]} \vdash x_1 : t_1, \ldots, x_k : t_k}{AT \vdash (lambda\ (x_1\ \ldots\ x_k)\ e) : t \leftarrow t_1 \times \ldots \times t_k}$$

- *set!*, *set-car!* and *set-cdr!* are backwards propagated according to the standard backwards propagation rule. Since the former binding of the changed or defined symbol is not known, $\top$ is assumed.

- Backwards propagation of *define* is done as follows:

$$\frac{AT \vdash (define\ s\ e) : \texttt{sym}}{AT \vdash s : \texttt{sym}, AT \vdash e : \top}$$

  Since usually the former definition of s is unknown, the type binding of s is set to $\top$.

5 Treating side effects

The type system described so far can only be applied if the given input typing is fixed during the evaluation of the function. Changes are only possible for local parameters masking a global definition. In this section we will spot on destructive updates of symbol bindings as a special case of side effects.

Typing functions containing destructive updates involves the following problems:

1. We must know the scope of the side effects. Problems may occur if there are global and local definitions of symbols sharing the same name.

2. In case of side effects in one execution path of *if* or *cond*, it is not known afterwards, whether the side effect was performed.

3. Different symbols may be bound to the same structure, or even worse, to different structures that contain the same value (e.g. a common rest of two lists). A side effect on one of these structures may also affect the other.

Since side effects depend on the semantics of the used programming language very much, we describe the handling of destructive updates exemplarily for the LISP dialect Scheme.

5.1 General extensions to process side effects

Because of (1) we must model the different frames and environments of Scheme during type propagation. Every frame contains the following data:

- A pointer to its parent frame, thereby defining an environment. The main environment consists of a frame without parent, i.e the pointer to parent frame is "()" represented by the empty environment type $\texttt{env}_0$.

- The actual type of every symbol bound in this frame. This type is determined by an input typing, a sequence of side effects or some combination of these. Because of further problems described in subsection 5.3 there are more then one representation of the type to be stored for some of the values. The principal structure of the environments is not affected by this.

- During type propagation, every frame contains information about the subset of its symbols that were possibly changed by some side effect.

The actual environment contains information about all symbols bound in the present context. Therefore we can extract all information given by an input typing from the corresponding environment, too. We can replace the use of input typings by environments. If we want to type a function call $c := (f\ a_1\ a_2\ \ldots\ a_k)$, we just create a new frame with the definition environment of function f as parent. The new frame contains the input typing for every formal parameter of f and all local definitions given in the definition of f.
The definition environments of all top-level self defined functions must be calculated in a preprocessing step before the task of type propagation starts.

To mark self defined functions with side effects, we introduce a function type constructor $\to_{SE}$ that generates subtypes of the function types generated by $\to$. For $\to_{SE}$ the following holds:

$$A \sqsupseteq C \wedge B \sqsubseteq D \Rightarrow (A \to_{SE} B) \sqsubseteq (C \to D) \wedge (A \to_{SE} B) \sqsubseteq (C \to_{SE} D)$$

The test for common elements of two types treats $\to_{SE}$ as $\to$.

With this background, the type system processes the predefined functions, performing side effects, as follows:

- Type propagation for *define* is done by the rule:

  $$\frac{}{(define\ s\ e) \in_{ET} \texttt{sym}}$$

  As a side effect, *define* defines a binding of the result of expression e to the symbol s in the actual frame. A previous binding of s in the same frame is lost, a binding of s in a higher order frame is hidden until the actual frame is left.

  There is a second syntax of *define*:

  $$(define\ (fkt\ par_1 \ldots par_k)\ e_1 \ldots e_k)$$

  Any expression meeting the above syntax should be typed like the expression

  $$(define\ fkt\ (lambda\ (par_1 \ldots par_k)\ e_1 \ldots e_k))$$

- *set!*, *set-car!* and *set-cdr!* do not generate any output:

  $$\frac{v \in \texttt{sym},\ a \in \top}{(set!\ v\ a) \in \mathcal{O}}$$

  $$\frac{l \in (\top\ .\ \top),\ a \in \top}{(\textit{set-car!}\ l\ a) \in \mathcal{O}}$$

  $$\frac{l \in (\top\ .\ \top),\ a \in \top}{(\textit{set-cdr!}\ l\ a) \in \mathcal{O}}$$

  The types of v or l are changed, and a flag signaling a change is set for these symbols. In case of *set-car!* and *set-cdr!* further symbols might be affected by the side effect. The propagation of these effects is described in section 5.3.
  Every function changing a symbol in a higher order frame is marked as function containing side effects ($\to_{SE}$).

5.2 Side effects in conditional expressions

When side effects occur inside a conditional expression, they cause uncertainty about the type of the affected symbols for every occurrence of these symbols that is typed after the conditional. Such a symbol might have the type caused by the side effect, it might have the type it had before processing the conditional, and there might even be different side effects in different execution paths.

The first step when typing a conditional is to type the different execution paths independently, everyone in the environment that is the current one when the path is reached. (Basically this is the environment that was actual before processing the conditional, except of the side effects performed by the conditions evaluated so far.)
The type binding of some symbol s for the further type propagation is generated by taking the union of all type bindings, s could get in the different execution paths. So we get a type binding that is a supertype of the exact type of s no matter which path the program execution might take.

5.3 Side effects affecting several symbols

When different symbols are bound to the same structure or to structures containing references to the same value, a destructive update on one of these structures may affect the others, too.

Example 5.1 *Consider the definitions of two lists shown in figure 3:*

According to these definitions, a and b are of type ($list$ $\boldsymbol{nat}$)*. Now consider the following statement performing a destructive update on a:*

- *The statements*

  ```
  (set-car! (cdr a) 'x)
  ```

  changes the types of both a and b to ($list$ ($\boldsymbol{nat} \cup \boldsymbol{sym}$))*. The resulting structure is also shown in figure 3.*

- *Performing the statement*

  ```
  (set-car! b 0.4)
  ```

  to the original definition given in figure 3 changes b to (0.4 2 3) *and does not affect a.*

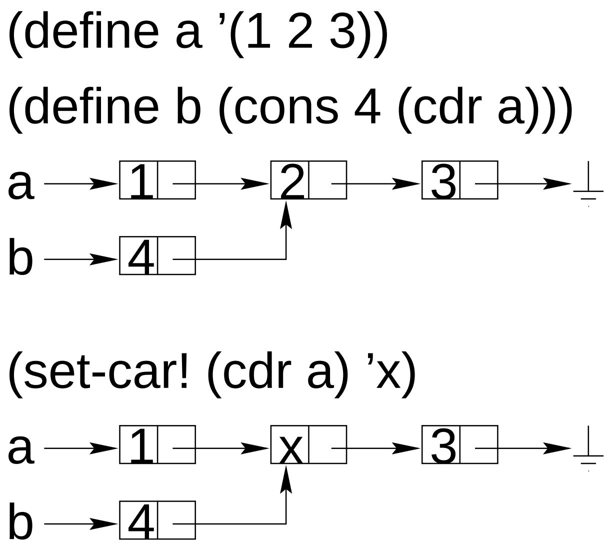

Figure 3: Structures containing common elements

The only commands that can cause destructive updates affecting more than one symbol are *set-car!* and *set-cdr!* which change the contents of an element of some pair. Therefore, only pairs and types derived from pairs (i.e. lists, stream-pairs, streams) are concerned.

To track the influence of some side effect, dependencies of two symbols a and b are stored together with their type information. The type of every symbol s_1, containing another symbol s_2 as part, is decomposed as far as necessary to contain a link to the type of s_2 at the proper position.
The links are established when definitions with dependencies occur: Let a and b be two symbols, where b is defined using a. The following possibilities of defining b exist:

1. b is bound to the same value as a:

   ```
   (define b a)
   ```

   The decomposed type of b is set to a link to a and vice versa.

2. b is defined as a part of a:

   ```
   (define b (car (cdr a)))
   ```

The type of a is decomposed along the definition of b, to determine the proper position for the link to b. If a is of type (*list* A), it gets the decomposed type

$$(A\ .\ (\rightarrow b\ .\ (list\ A)))$$

The type of b is determined by decomposing the type of a as well. In the example b is of type A.

3. b is defined as a structure containing a:

   ```
   (define b (cons a (list 1 2 3)))
   ```

   In this situation, a decomposed type of b has to be constructed along the definition of b. In this example we get $(\rightarrow a\ .\ (list\ \texttt{nat}))$.

4. b is defined as a structure containing a part of a:

   ```
   (define b (cons 4 (cdr a)))
   ```

   A new symbol c that is not used in the typed program is bound to the common part of a and b. In the example, c is bound to (*cdr a*). The decomposed types of a and b are constructed as if c had been defined as a part of a according to 2. and b had been constructed, using c as a part, as described in 3. The resulting structure in the given example is presented in figure 4.

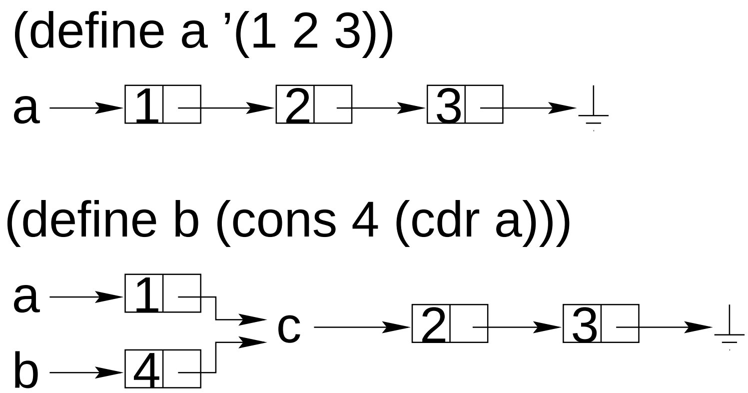

Figure 4: Binding common parts of structures

There might be several symbols, having references to the same symbol s. If s is deleted or redefined, a new internal symbol is defined that catches the references pointing to s before.

Whenever a symbol s is affected by a side effect caused by *set-car!* or *set-cdr!* the following side effect propagation is performed:

- For every decomposed type of s referencing s' do the following:
  1. Identify the position of the change in the decomposed type of s. If the change takes place inside of s', perform a further decomposition of the type of s'.
  2. Check, whether s' is affected by the side effect, change the primary type of s' if necessary and propagate the side effect on s', without changing the type of s again.
- For every symbol s' with some decomposed type referencing s do the following:
  1. Update the primary type of s': replace it by the simplification of the decomposed type using the new type of s.
  2. Propagate the side effect on s', without changing the type of s again.

6 Proof of completeness

In this section we prove our type propagation algorithm to be complete in the following sense: When we type a function f with an input typing ET such that ET consists of ground types only, and there is an execution of f with input parameters compatible with ET that is free of type errors then f under ET is accepted by the type system.
When some types of ET contain type variables then for any instantiation of the type variables in ET there must be a compatible set of input parameters to f not causing a type error to be sure that f is accepted.

Lemma 6.1 *Let f be a predefined function of arity k with type definitions*

$$\begin{gathered} f : t_{1_1}, \ldots, t_{1_k} \to t'_1 \\ \vdots \\ f : t_{m_1}, \ldots, t_{m_k} \to t'_m \end{gathered}$$

Then the following holds:

1. *If $(f\ a_1\ \ldots\ a_k)$ can be evaluated without type error, than there is a $j \in \{1, \ldots, m\}$ with $a_i \in t_{j_i}$, $i = 1, \ldots, k$.*

2. If $(f\ a_1\ \ldots\ a_k) \rightarrow r$ with $a_i \in t_{j_i}$, $i = 1, \ldots, k$, then $r \in t'_j$.

Proof: The lemma follows directly from comparing the defined input/output behaviour of every predefined function with its type definitions in appendix C. □

Lemma 6.2 *If for a function call $c := (f\ a_1\ \ldots\ a_k)$ under an input typing ET there is a type definition in $\mathcal{R}$ appropriate for every instantiation of ET, then the type propagation algorithm will find such one.*

Proof: Let $f : t_1, \ldots, t_k \rightarrow t'$ be an appropriate type definition to the call c under any instantiation $ET' := \sigma(ET)$ of ET where ET assigns the type s_i to argument a_i for $i \in \{1, \ldots, k\}$. I.e. for $i \in \{1, \ldots, k\}$ $\sigma(s_i) \cap t_i \neq \bot$ holds. Therefore when choosing a transformation rule as described in section 4.3.1, *disjoint* will not be returned for any argument position of the transformation rule $f : t_1, \ldots, t_k \rightarrow t'$ (especially because of the call of CE with $b = \texttt{all}$, cf. page 31). Hence the set of applicable transformation rules is not empty and the algorithm will return one or several of them. □

Lemma 6.3 *Let f be a function, ET an input typing assigning the type t_i to argument a_i and let at least one t_j contain a type variable A.*
If we can find an appropriate transformation rule $t'_1 \times \ldots \times t'_k \rightarrow t'$, i.e. $t_i \cap t'_i \neq \bot$ in $\mathcal{R}$ for every possible instantiation of the variables in A then any of these transformation rules is appropriate for every instantiation of the variables in A.

Proof: For every predefined f and every argument position the transformation rules in $\mathcal{R}$ expect $\top$, a restricted set of base types or a type constructor whose arguments consist of restricted sets of base types (or again constructors) of a type variable. The only possibilities to find transformation rules for every instantiation of A is to find a transformation rule that contains $\top$ or a type variable at the position of A. This transformation rule is then appropriate for every instantiation of A. □

Remark 6.4 *As stated in section 4.3.1 one may have to change the set $\mathcal{R}$. Then it is important for the completeness of the system that lemma 6.3 still holds.*
There could be sets of transformation rules where lemma 6.3 does not hold. But functions usually must rely on the types of there input parameters if they transform them or they are polymorphic in certain positions. We do not have any practically relevant example of a function that accepts every input type but are not expressible with a single transformation rule.

Theorem 6.5 *Let P be a non-recursive program that does not cause any side effects, $c = (f\ p_1 \ldots p_k)$ a call in P of function f with parameters $p_1, \ldots p_k$, and ET an input typing for P. Let $s_1, \ldots, s_n$ be a list of input arguments to P which are consistent with some instance ET' of ET and cause an execution of P without type errors.*

If $a_1, \ldots, a_k$ are the arguments which are assigned to $p_1, \ldots, p_k$ during the mentioned type error free execution, then the type t_i, propagated for p_i under ET contains a_i ($i = 1, \ldots, k$).

Proof: See appendix B $\square$

From theorem 6.5 together with lemma 6.3 it follows directly that the type propagation algorithm accepts every non-recursive program P under those input typings ET that for every instantiation ET' of ET a type error free execution of P is possible.

Lemma 6.6 *Let P be a recursive program that does not cause any side effects, and ET an input typing to P such that every instance $\sigma(ET)$ of ET contains an argument set $s_1, \ldots, s_k$ not causing a type error. If the type propagation algorithm does not run into an infinite loop, it accepts P.*

Proof: See appendix B $\square$

Lemma 6.7 *If the iterated type propagation runs into an infinite loop, this is detected and the methods for approximating the needed output type do not violate the completeness of the system.*

Proof: See appendix B $\square$

The following theorem follows directly from lemmas 6.6 and 6.7:

Theorem 6.8 *The type propagation algorithm is complete for every program P that does not cause any side effects and every input typing ET such that for every instance $\sigma(ET)$ of ET a type error free execution is possible.*

To prove the completeness of type propagation with side effects, we need the following lemma:

Lemma 6.9 *Let s and s' be symbols, and consider a side effect changing the type of s which affects s', as well. If the side effect propagation from s' directly back to s is blocked, recursive propagation of the side effect on s' does not affect s again.*

Proof: See appendix B $\square$

Lemma 6.10 *The extension for handling side effects in section 5 does not violate the completeness of the type system.*

Proof: See appendix B $\square$

The main theorem of this section follows directly from theorem 6.8 and lemma 6.10:

Theorem 6.11 (Completeness of the type system) *The type system presented in this paper is complete, i.e. it does not reject any program Pchecked under an input typing ET if for every instance $\sigma(ET)$ of ET a type error free execution is possible.*

7 Examples

In this section we give some example input to show the power and limitations of our type system. The programs are solutions to exercises from a course in functional programming.

7.1 Examples of well-typed programs

The examples given in this subsection are type correct programs that are all accepted by the type system because of its completeness. One should note that the propagated output type is often quite close to the exact type of an expression. Because of this accurate type propagation we can expect a good ratio of recognized to overall type errors since inaccuracies of the type inference do not grow very fast.

- The function *reverse* expects a list as input and returns a list containing the same elements in reverse order:

```
(define (reverse l)
  (if (null? l)
      '() ; reversed empty list is empty
    (append (reverse (cdr l))
              ; reverse rest
            (list (car l)))))
              ; append first element at the end;
              ; (function append expects two (or more) lists.)
```

  We run the type propagation with input typing

$$ET : l \leftarrow (list\ A)$$

  and get the propagated output type ($list\ A$), which is the exact type of *reverse* under ET.

- *flatten* expects any list as input and returns a list which contains any atomic elements of the main list and any atomic elements of pairs of the input list:

```
(define (flatten l)
  (cond ((null? l) '()) ; () results in ()
        ((list? (car l)) ; flatten sublist end append elements
         (append (flatten (car l)) (flatten (cdr l))))
        (else                ; append atomic elements
         (cons (car l) (flatten (cdr l))))))
```

Analyzing this function with the input typing

$$ET : l \leftarrow (list\ \top)$$

results in the exact type $(list\ (\top \setminus (list\ \bot)))$ if the input typing for every conditional subexpression is restricted as much as possible (cf. subsection 4.3.3).

- *reverse-deep* expects a list as input and returns another list resulting from the input list by reversing the order of elements and recursively applying *reverse-deep* to any element that is a list itself:

```
(define (reverse-deep l)
  (cond ((null? l) '()) ; return () for empty input list
        ((list? (car l))
         (append (reverse-deep (cdr l))
                  ; reverse sublists
                 (list (reverse-deep (car l)))))
        (else
         (append (reverse-deep (cdr l))
                 (list (car l))))))
```

We perform a type test on *reverse-deep* for two alternative input typings:

1. Providing the input typing

$$l \leftarrow (list\ \top)$$

we get the output type $(list\ \top)$.

2. If we provide the input typing

$$l \leftarrow \mu X.(list\ (A \cup X))$$

an infinite loop is established:
If we simplify the intermediate output types by generalization, we get the following sequence of intermediate output types:

(a) $(list\ \perp)$
(b) $(list\ A) \cup (list\ (list\ \perp))$
As an argument type of *append* a generalized type is used in further type propagation:
$(list\ (A \cup (list\ \perp)))$
(c) $(list\ (A \cup (list\ (A \cup (list\ \perp)))))$
(d) $(list\ (A \cup (list\ (A \cup (list\ (A \cup (list\ \perp)))))))$
(e) ...

Alternatively we can decompose the union of lists as shown in subsection 4.3.2. This yields the following sequence of intermediate output types:

(a) $(list\ \perp)$
(b) $(list\ A) \cup (list\ (list\ \perp))$
(c) $(list\ (A \cup (list\ A))) \cup (list\ (list\ (list\ \perp)))$
(d) $(list\ (A \cup (list\ (A \cup (list\ A))))) \cup (list\ (list\ (list\ (list\ \perp))))$
(e) ...

In both cases we can generate a recursive type as described in section 4.5.1 which results in the exact output type $\mu X.(list\ (A \cup X))$.

- The following function *tree* generates binary trees in a message oriented implementation:

```
(define (tree root left right)
  (lambda (m)
    (cond ((eq? m 'root) root) ; return the values
          ((eq? m 'left) left)   ; of "root", "left" and
          ((eq? m 'right) right) ; "right" respectively
          ((eq? m 'inorder)   ; compute an inorder traversal
           (append (inorder-subtree left) (list root)
                   (inorder-subtree right)))
          (else (error "Unknown command error " m)))))

(define (inorder-subtree b)
   ;; Inorder traversal of subtrees.
  (if (null? b)
      '()
    (b 'inorder)))
```

With input typing

$$ET : root \leftarrow A,\ left \leftarrow B,\ right \leftarrow B$$

our system returns the generic function type $\bot \to \top$ indicating that the returned object of function *tree* is a function.

Different output is possible if the expected output type of *tree* is known as well. Let

$$AT : (S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder}) \to (A \cup B \cup (list\ A))$$

be this output type. When the system realizes that the output is build by a lambda expression, backwards propagation is performed and states that the given output type is the expected type of the function returned by *tree*. So we type the defining lambda expression with the input typing

$$ET : m \leftarrow (S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder})$$

and get the propagated output type $\top$ which is consistent with the expected output type. Thus *tree* has $((S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder}) \to \top)$ as the propagated output type. This is a strict supertype of the exact type.

The output type $\top$ is derived as a simplification of $A \cup B \cup \top$, where A results from the message *'root*, B results from the messages *'left* and *'right* and $\top$ results from the message *'inorder* because in typing $inorder-subtree$ the result of the unknown function $b : B$ can just be approximated by $\top$.

Now add some further information about the type variables used in the expected type of *tree*:

$$A := \texttt{num}$$
$$B := \mu X.(S_{()} \cup ((S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder}) \to (\texttt{num} \cup X \cup (list\ \texttt{num}))))$$

Typing the lambda-expression defined inside of *tree* the same way as before, we get the output type $A \cup B \cup C$, where C is the result of typing *inorder-subtree* with input typing

$$\{b \leftarrow \mu X.(S_{()} \cup ((S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder}) \to (\texttt{num} \cup X \cup (list\ \texttt{num}))))\}$$

This results in the output type $C := (list\ num)$ and therefore the output type of the lambda-expression inside of *tree* is:

$$(\texttt{num} \cup \mu X.(S_{()} \cup ((S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder}) \to (\texttt{num} \cup X \cup (list\ \texttt{num})))) \cup (list\ \texttt{num}))$$

This implies the following output type of *tree*:

$$(S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder}) \to (\texttt{num} \cup \mu X.(S_{()} \cup ((S_{root} \cup S_{left} \cup S_{right} \cup S_{inorder}) \to (\texttt{num} \cup X \cup (list\ \texttt{num})))) \cup (list\ \texttt{num}))$$

which is the exact type of *tree* under the given input typing.

- *m_m_mult* multiplies two matrices if possible or returns a dimension error. A matrix is a list of rows, every row is a list of elements.

```
(define (v_v_mult row column)
  ;;; Multiplication of two vectors (a row and a column)
  ;;; Both vectors are defined as lists of their elements.
  (cond ((and (eqv? row '()) (eqv? column '())) 0)
          ;; Product of empty vectors is 0
        ((or (eqv? row '()) (eqv? column '()))
         (error "dimension error"))
          ;; Vectors of different dimensions
        (else
         (+ (* (car row) (car column))
                 ;; multiply first elements and process
                 ;; the rests recursively
            (v_v_mult (cdr row) (cdr column))))))

(define (m_v_mult matrix vector)
  ;;; Multiplication of a matrix with a column
  (map (lambda (x) (v_v_mult x vector)) matrix))
    ;; Treat matrix as list of rows and generate the result as
    ;; list of products between these rows and 'column'

(define (empty-matrix? matrix)
  ;;; returns #t if all rows of 'matrix' are empty
  (if (eqv? matrix '())
      #t  ;; No rows left to check
    (and (eqv? (car matrix) '())  ;; test first row
         (empty-matrix? (cdr matrix))))) ;; test rest


(define (empty-row? matrix)
  ;;; returns #t if 'matrix' contains an empty row
  (if (eqv? matrix '())
      #f  ;; false if no rows left
    (or (eqv? (car matrix) '())  ;; test first row
        (empty-row? (cdr matrix))))) ;; test rest


(define (transpose matrix)
  ;;; returns the transposed matrix 'matrix'
```

```
  (cond ((empty-matrix? matrix) '())
        ((empty-row?  matrix) (error "dimension error"))
        (else (append (list (map car matrix))
                ;; first row = list of first elements
                ;; of the rows (= first column)
                      (transpose (map cdr matrix))))))
                          ;; process rest recursively

(define (m_m_mult matrix1 matrix2)
  ;;; Multiplication of two matrices
  (transpose (map (lambda (x) (m_v_mult matrix1 x))
                  (transpose matrix2))))
    ;; matrix-matrix-multiplication is expressed by iterated
    ;; matrix-vector-multiplication taking into account the
    ;; internal matrix representation.
```

We are interested in the output type of the main function *m_m_mult* under some input typing assigning a matrix to *matrix1* and *matrix2*, respectively. The elements are numbers of any kind:

$$ET := \{matrix1 \leftarrow (list\ (list\ num)),\ matrix2 \leftarrow (list\ (list\ num))\}$$

The type check results in the propagated output type (*list* (*list num*)) which is the exact type of *m_m_mult*. We derived this without providing types of any of the sub-functions.

- The following function `traffic` shows how to use the base concept of object oriented programming in functional languages with side effects. *traffic* is called without arguments and returns some object representing a traffic-light which is queried with the message *show* and is switched to the next state with the message *tick*:

```
(define (traffic)
  (define state 1) ; Main road green at startup.

  (define (do-tick)
    ;; process message 'tick'
    (if (eq? state 4)  ; one cycle complete?
        (set! state 1)
        (set! state (1+ state))))

  (define (do-show)
    ;; process message 'show'
```

```
  (cond ((eq? state 1)  ; states in external representation
         (list 'green 'red))
        ((eq? state 2)
         (list 'yellow 'red-yellow))
        ((eq? state 3)
         (list 'red 'green))
        ((eq? state 4)
         (list 'red-yellow 'yellow))
        (else                        ; error in
         (list 'error 'error))))   ; implementation


(define (dispatch message)
   ;; choose operation according to message
  (cond ((eq? message 'tick)
         (do-tick))
        ((eq? message 'show)
         (do-show))
        (else
         (error "Unknown command"))))

dispatch) ; return function 'dispatch'
```

An input typing is irrelevant for typing *traffic* because *traffic* does not expect any input parameters. If we analyze *traffic* without further information, we get the output type $\perp \to \top$ in an environment with the following symbol bindings:

$$\begin{aligned} \mathit{state} &: \mathtt{nat} \\ \mathit{do\text{-}tick} &: \perp \to \top \\ \mathit{do\text{-}show} &: \perp \to \top \\ \mathit{dispatch} &: \perp \to \top \end{aligned}$$

A more accurate analysis of *traffic* is possible if we take into account an expected output type, in this case:

$$\begin{aligned} (S_{\mathtt{tick}} \cup S_{show}) \to ((S_{\mathtt{green}} \,.\, S_{\mathtt{red}}) \cup (S_{\mathtt{yellow}} \,.\, S_{\mathtt{red\text{-}yellow}}) \cup (S_{\mathtt{red}} \,.\, S_{\mathtt{green}}) \\ \cup (S_{\mathtt{red\text{-}yellow}} \,.\, S_{\mathtt{yellow}}) \cup \mathcal{O}) \end{aligned}$$

From backwards propagation we know that this is the expected type of *dispatch*. So we can type *dispatch* by propagating the input type $(S_{\mathtt{tick}} \cup S_{\mathtt{show}})$ and receiving the propagated output type

$$((S_{\texttt{green}} \,.\, S_{\texttt{red}}) \cup (S_{\texttt{yellow}} \,.\, S_{\texttt{red-yellow}}) \cup (S_{\texttt{red}} \,.\, S_{\texttt{green}}) \\ \cup (S_{\texttt{red-yellow}} \,.\, S_{\texttt{yellow}}) \cup (S_{\texttt{error}} \,.\, S_{\texttt{error}}) \cup \mathcal{O})$$

as well as a flag marking *dispatch* as side effecting function. Thus we have the following output type of *traffic*:

$$(S_{\texttt{tick}} \cup S_{\texttt{show}}) \rightarrow_{SE} ((S_{\texttt{green}} \,.\, S_{\texttt{red}}) \cup (S_{\texttt{yellow}} \,.\, S_{\texttt{red-yellow}}) \cup (S_{\texttt{red}} \,.\, S_{\texttt{green}}) \\ \cup (S_{\texttt{red-yellow}} \,.\, S_{\texttt{yellow}}) \cup (S_{\texttt{error}} \,.\, S_{\texttt{error}}) \cup \mathcal{O})$$

7.2 Programs containing type errors

Besides type correct programs, we are especially interested in the reactions of our system on programs containing type errors. In this section we give examples of the different kinds of type errors.

7.2.1 Inconsistency of functions with expected types

We get one kind of type error if some program is executable under some input typing without runtime errors, but the type of its result is disjoint from the expected type.

Consider the exercise of giving a new definition of the function *map*, called *my-map*, with only functions of arity 1 and only one list of arguments. The expected type of this function is

$$(A \rightarrow B) \times (list\ A) \rightarrow (list\ B)$$

Now, let us consider the following erroneous implementation:

```
(define (my-map f l)
   (if (eqv? l ())
       ()
       (cons (list (f (car l)))
             (my-map f (cdr l)))))
```

There is an error in using *list* in line 4 that would be appropriate if we used *append* instead of *cons* to connect head and tail of the output list.

We check this program, providing the following input typing:

$$f \leftarrow (A \rightarrow B),\ l \leftarrow (list\ A)$$

As a result we get the propagated output type (*list* (*list* B)) which is disjoint from the expected output type (*list* B) for some assignments to B (e.g. $B \leftarrow$ `int`). We have to generate an error message and may add $B \leftarrow$ `int` as an example of type assignment causing an error.

7.2.2 Detection of possible runtime errors

The main use of type systems is the detection of runtime errors. The following example illustrates the usability of this type system in the area of detecting runtime errors, as well. Consider the following erroneous implementation to exercise *reverse*, as presented in section 7.1:

```
(define (reverse l)
  (if (null? l)
      '() ; reversed empty list is empty
    (append (reverse (cdr l))
              ; reverse rest
            (car l))))
              ; append first element at the end.
```

When we analyze this program under the input typing

$$ET : l \leftarrow (list\ A)$$

the system reports a type error in the call of *append* in line 4-6, because the second argument is of type A. We can assign types to A that are disjoint from the expected input type (*list* $\top$) of *append* (e.g. $A \leftarrow$ `int`). We have to generate an error message and again we may add the example assignment $A \leftarrow$ `int`.

8 Conclusion

In this paper we have presented an alternative approach to types in functional languages by describing a complete type system. It is *complete* in the sense that it is guaranteed to accept any type-correct program. Our type system may not detect every type error in functional programs, but it detects real errors instead of the unprovability of type correctness. It is therefore better adapted to applications in debugging programs. The need of additional information like an input typing or the expected output type of a function is not crucial, since the intended input/output behavior should be known for any program that one wants to debug. The problem of not detecting some of the errors is not crucial as well when the goal of type checking is not the proof of program correctness but the detection of errors. But our system circumvents the problem of sound type systems that reported errors may be real errors, but may also be correct parts of a program that cannot be proved type correct. Thus, the error messages reported by our system are more reliable and programmers (especially beginners in functional programming) may not waste their motivation on searching for errors that do not exist.

When we consider large programs there are aspects of this work that have to be analyzed in more detail: The larger the analyzed programs grow, the more important an efficient implementation of the type checker becomes. Furthermore, we need to inspect the growth of inaccuracies of types when propagating them through a large program. But we feel that both of these problems can be handled, e.g. by explicit type annotations to sub-functions of a large program. To check for problems in this area we are going to implement a prototype of this type system.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[AM91] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, Florida, January 1991.

[AW92] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

[AW93] Alexander S. Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming and Computer Architecture*, pages 31–41. ACM Press, June 1993.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

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

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, December 1969.

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

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, October/November 1986.

[Smi94] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

[Tha88] Satish R. Thatte. Type inference with partial types. *Springer Verlag (Heidelberg, FRG and NewYork NY, USA) LNCS 317, Automata, Languages and Programming, 15th International Colloquium*, pages 615–629, 1988.

[Tha94] Satish R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124(1):127–148, 14 February 1994.

[WB98] Manfred Widera and Christoph Beierle. Local normalization of functional programs. Technical report, FernUniversität Hagen, 1998.

[WC94] Adrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 250–262, June 1994.

A Proofs for section 3

Lemma 3.3 *Let A and B be two types. Then*

$$A \sqsubseteq B \Rightarrow V(A) \subseteq V(B)$$

Proof of lemma 3.3: In the first part of the proof we do not consider the definitions in (8). Then we can do the proof just along the definition of $\sqsubseteq$. We use induction on the number n of type constructors that are necessary to define the examined types in those cases that have a precondition of the form $X \sqsubseteq Y$.

1. Relations of the base types:
   (a) $V(A) \subseteq V(A)$ for every type A is clear from definition of $\subseteq$.
   (b) For every type A we have $V(\bot) = \{\bot\} \subseteq V(A) \subseteq \Omega = V(\top)$.
   (c) From the definition of `num` and its subtypes we get $V(\texttt{nat}) \subseteq V(\texttt{int}) \subseteq V(\texttt{rat}) \subseteq V(\texttt{num})$
   (d) $V(\texttt{real}) \subseteq V(\texttt{num})$ follows from the definition of `num`.
   (e) $V(S_{()}) = \{\bot, ()\}$ with $() \in V((list\ A))$ for every type A, especially for $A = \bot$. Thus $V(S_{()}) \subseteq V((list\ \bot))$.
   (f) The proof of $V(S_{the\_empty\_stream}) \subseteq V((stream\ \bot))$ is analogous to case (1e).

2. Relations on the set operations:
   (a) $V(A), V(B) \subseteq V(A) \cup V(B) = V(A \cup B)$.
   (b) $V(A \cap B) = V(A) \cap V(B) \subseteq V(A), V(B)$.
   (c) $V(A \setminus B) = (V(A) \setminus V(B)) \cup \{\bot\} \subseteq V(A)$.
   (d) Let $A \cup B \sqsubseteq C$ hold because of condition (2d) in definition 3.2. From the precondition of case (2d) we know $A \sqsubseteq C \wedge B \sqsubseteq C$. Now suppose the overall number of type constructors used to define $A \cup B$ and C is n. Then we need $n-1$ type constructors to define A, B and C. We therefore conclude $V(A) \subseteq V(C) \wedge V(B) \subseteq V(C)$ by induction hypothesis from the precondition. With the definition of $\cup$ we have $V(A \cup B) \subseteq V(C)$.
   (e) If $C \sqsubseteq A \cap B$ holds because of (2e) we have the precondition $C \sqsubseteq A \wedge C \sqsubseteq B$. By induction hypothesis we can conclude $V(C) \subseteq V(A) \wedge V(C) \subseteq V(B)$ and from this and the definition of $\cap$ it follows that $V(C) \subseteq V(A \cap B)$.
   (f) If we have $A \sqsubseteq \mathcal{C}B$ (resp. $B \sqsubseteq \mathcal{C}A$) because of (2f) $A \cap B = \bot$ must hold and therefore $V(A) \cap V(B) = \emptyset$ and $V(A) \subseteq V(\mathcal{C}B)$ (resp. $V(B) \subseteq V(\mathcal{C}A)$)

3. Monotony of type constructors:

   (a) If $A \times B \sqsubseteq C \times D$ holds because of condition (3a), the precondition $A \sqsubseteq C \wedge B \sqsubseteq D$ implies $V(A) \subseteq V(C) \wedge V(B) \subseteq V(D)$ by induction hypothesis. Together with the definition of $\times$ this implies $V(A \times B) \subseteq V(C \times D)$.

   (b) Consider $(A\ .\ B) \sqsubseteq (C\ .\ D)$ because of (3b). This implies the precondition $A \sqsubseteq C \wedge B \sqsubseteq D$ to hold. By induction hypothesis we get $V(A) \subseteq V(C) \wedge V(B) \subseteq V(D)$, and therefore

$$\begin{aligned} V((A\ .\ B)) =& \{(x\ .\ y) \mid x \in V(A), y \in V(B)\} \\ \subseteq& \{(x\ .\ y) \mid x \in V(C), y \in V(B)\} \\ \subseteq& \{(x\ .\ y) \mid x \in V(C), y \in V(D)\} \\ =& V((C\ .\ D)) \end{aligned}$$

   (c) The proof of $[A\ .\ B] \subseteq [C\ .\ D]$ because of (3c) is analogous to (3b).

   (d) For function type constructors $(A \to B) \sqsubseteq (C \to D)$ is defined by (3d) under the precondition $A \sqsupseteq C \wedge B \sqsubseteq D$, differing from the previously given preconditions for (3a-3c) in the first argument. From this precondition we can conclude $V(A) \supseteq V(C) \wedge V(B) \subseteq V(D)$ by the induction hypothesis. We further have:

$$\begin{aligned} V(A \to B) =& \{f : X \to Y \mid V(A) \subseteq X, Y \subseteq V(B)\} \\ \subseteq& \{f : X \to Y \mid V(C) \subseteq X, Y \subseteq V(B)\} \\ \subseteq& \{f : X \to Y \mid V(C) \subseteq X, Y \subseteq V(D)\} \\ =& V(C \to D) \end{aligned}$$

   (e) Let $(list\ A) \sqsubseteq (list\ B)$ hold by condition (3e). This implies $A \sqsubseteq B$ to hold and therefore by induction hypothesis we get $V(A) \subseteq V(B)$. To prove $V((list\ A)) \subseteq V((list\ B))$ under this precondition, consider an arbitrary $l \in V((list\ A))$. We will show $l \in V((list\ B))$ by induction on the length k of list l.
   For $k = 0$ we have nothing to prove because (), which is the only list of length 0, is element of every list type.
   Now let $k > 0$:

$$l = (e\ .\ l')$$

   with $e \in V(A)$ and $l' \in V((list\ A))$. The list l' has length $k - 1$ and thus we know from induction hypothesis $l' \in V((list\ B))$. Therefore $l \in V((A\ .\ (listB)))$ and by (3b) $l \in V((B\ .\ (listB))) = V((list\ B))$.

   (f) $V((stream\ A)) \subseteq V((stream\ B))$ follows from $(stream\ A) \sqsubseteq (stream\ B)$ from (3f) of definition 3.2 analogously to (3e).

4. Relations on lists:

    (a) By (3b) we have

    $$V((A\ .\ (list\ B))) \subseteq V(((A \cup B)\ .\ (list\ B)))$$

    and by (3e)

    $$V(((A \cup B)\ .\ (list\ B))) \subseteq V(((A \cup B)\ .\ (list\ (A \cup B)))) = V((list\ (A \cup B)))$$

    (b) Since by (2a)

    $$V(A), V(B) \subseteq V(A \cup B)$$

    we have by (3e)

    $$V((list\ A)), V((list\ B)) \subseteq V((list\ (A \cup B)))$$

    which implies

    $$V((list\ A)) \cup V((list\ B)) \subseteq V((list\ (A \cup B)))$$

    and thus by (2a)

    $$V((list\ A) \cup (list\ B)) \subseteq V((list\ (A \cup B)))$$

    (c) Consider a list l with $l \in V((list\ A) \cap (list\ B))$ and let e be an arbitrary element of l.

    $$\begin{aligned} & l \in V((list\ A) \cap (list\ B)) = V((list\ A)) \cap V((list\ B)) \\ \Rightarrow & l \in V((list\ A)) \\ \Rightarrow & e \in V(A) \end{aligned}$$

    Analogously we can prove $e \in V(B)$, and from $e \in V(A) \wedge e \in V(B)$ follows $e \in V(A) \cap V(B) = V(A \cap B)$.
    Since this holds for every element e of l, we can conclude $l \in V((list(A \cap B)))$.

5. Relations on Streams:

    (a) $V((A\ .\ (stream\ B))) \subseteq V((stream\ A \cup B))$ follows analogously to (4a).

    (b) $V((stream\ A) \cup (stream\ B)) \subseteq V((stream\ A \cup B))$ follows analogously to (4b).

(c) $V((stream\ A) \cap (stream\ B)) \subseteq V((stream\ (A \cap B)))$ follows analogously to (4c).

6. Relations on frames and environments:

(a) Let $f \in (frame\ s_1 : t_1 \ldots s_k : t_k)$. Then f is a frame in which symbol s_i is bound to some value of type t_i for every $i \in \{1, \ldots, k\}$. Especially this holds for $i \in \{1, \ldots, i-1, i+1, \ldots, k\}$ and therefore $f \in (frame\ s_1 : t_1 \ldots s_{i-1} : t_{i-1}\ s_{i+1} : t_{i+1} \ldots s_k : t_k)$. From that we can conclude

$$V((frame\ s_1 : t_1 \ldots s_k : t_k)) \subseteq \\ V((frame\ s_1 : t_1 \ldots s_{i-1} : t_{i-1}\ s_{i+1} : t_{i+1} \ldots s_k : t_k))$$

(b) $V(\texttt{env}_{[s_1:t_1,\ldots,s_k:t_k]}) \subseteq V(\texttt{env})$ by definition of $\texttt{env}$ and $\texttt{env}_{[s_1:t_1,\ldots,s_k:t_k]}$.

(c) Let $e \in V(\texttt{env}_{[s_1:t_1,\ldots,s_k:t_k]})$. Then e in an environment in which every symbol s_j is bound to some element in $V(t_j)$ for $i \in \{1, \ldots, k\}$.
This holds especially for $j \in \{1, \ldots, i-1, i+1, \ldots, k\}$. Therefore

$$e \in V(\texttt{env}_{[s_1:t_1,\ldots,s_{i-1}:t_{i-1},s_{i+1}:t_{i+1},\ldots,s_k:t_k]})$$

and thus

$$V(\texttt{env}_{[s_1:t_1,\ldots,s_k:t_k]}) \subseteq V(\texttt{env}_{[s_1:t_1,\ldots,s_{i-1}:t_{i-1},s_{i+1}:t_{i+1},\ldots,s_k:t_k]})$$

7. Special hierarchical relations:

(a) Let

$$(A_1 \times C_1) \cap (A_2 \times C_2) \sqsubseteq (B_1 \times D_1) \cap (B_2 \times D_2)$$

hold because of condition (7a). This implies the validity of the precondition

$$(A_1 \cap A_2) \sqsubseteq (B_1 \cap B_2) \wedge (C_1 \cap C_2) \sqsubseteq (D_1 \cap D_2)$$

It follows by induction hypothesis and definition of $\cap$:

$$(V(A_1) \cap V(A_2)) \subseteq (V(B_1) \cap V(B_2)) \wedge (V(C_1) \cap V(C_2)) \subseteq (V(D_1) \cap V(D_2))$$

Now consider some

$$e \in V((A_1 \times C_1) \cap (A_2 \times C_2))$$

It follows $e = (x, y)$ with the following conditions:

$$\begin{aligned}
&((x \in V(A_1)) \wedge (y \in V(C_1))) \wedge ((x \in V(A_2)) \wedge (y \in V(C_2))) \\
\Rightarrow&((x \in V(A_1)) \wedge (x \in V(A_2))) \wedge ((y \in V(C_1)) \wedge (y \in V(C_2))) \\
\Rightarrow&(x \in (V(A_1) \cap V(A_2))) \wedge (y \in (V(C_1) \cap V(C_2))) \\
\Rightarrow&(x \in (V(B_1) \cap V(B_2))) \wedge (y \in (V(D_1) \cap V(D_2))) \\
\Rightarrow&((x \in V(B_1)) \wedge (x \in V(B_2))) \wedge ((y \in V(D_1)) \wedge (y \in V(D_2))) \\
\Rightarrow&((x \in V(B_1)) \wedge (y \in V(D_1))) \wedge ((x \in V(B_2)) \wedge (y \in V(D_2))) \\
\Rightarrow&e = (x, y) \in V((B_1 \times D_1) \cap (B_2 \times D_2))
\end{aligned}$$

It follows

$$V((A_1 \times C_1) \cap (A_2 \times C_2)) \subseteq V((B_1 \times D_1) \cap (B_2 \times D_2))$$

(b) If

$$(A_1 \ . \ C_1) \cap (A_2 \ . \ C_2) \sqsubseteq (B_1 \ . \ D_1) \cap (B_2 \ . \ D_2)$$

holds because of (7b) a proof analogous to (7a) with $e = (x \ . \ y)$ instead of $e = (x, y)$ implies

$$V((A_1 \ . \ C_1) \cap (A_2 \ . \ C_2)) \subseteq V((B_1 \ . \ D_1) \cap (B_2 \ . \ D_2))$$

(c) The validity of

$$[A_1 \ . \ C_1] \cap [A_2 \ . \ C_2] \sqsubseteq [B_1 \ . \ D_1] \cap [B_2 \ . \ D_2]$$

because of (7c) implies

$$V([A_1 \ . \ C_1] \cap [A_2 \ . \ C_2]) \sqsubseteq V([B_1 \ . \ D_1] \cap [B_2 \ . \ D_2])$$

analogously to (7a) with $e = [x \ . \ y]$ instead of $e = (x, y)$.

Thus by complete case analysis we have $V(A) \subseteq V(B)$ whenever $A \sqsubseteq B$ holds because of any of the conditions of definition 3.2 excluding (8).

As a last step we have to consider the properties of $\sqsubseteq$ according to part (8) of the definition, too:

8. Properties of $\sqsubseteq$:

(a) (8a) is directly clear from the transitivity of $\subseteq$.

(b) If $A \sqsubseteq B$ has been proven before, $V(A) \subseteq V(B)$ must hold as well. (Either by the first part of the proof or by induction on the number of complement type constructors.) Therefore:

$$\begin{aligned}
&V(A) \subseteq V(B) \\
\Leftrightarrow &\mathcal{C}V(B) \subseteq \mathcal{C}V(A) \\
\Leftrightarrow &V(\mathcal{C}B) \subseteq V(\mathcal{C}A)
\end{aligned}$$

$\square$

Lemma 3.5 *Processing a type t with ES_1 yields an equivalent type t', i.e. $V(t) = V(t')$.*

Proof of lemma 3.5: Consider the definition of $A \times B$ with:

$$V(A \times B) = \{(a, b) \mid a \in A, b \in B\}$$

Because of the analogous definitions of $(A\ .\ B)$ and $[A\ .\ B]$ we just look at the first 6 rules:

(1)

$$\begin{aligned}
&V(((\cup\ <a_1 \ldots e_1>) \times A)) \\
=&((\cup\ <V(a_1) \ldots V(e_1)>) \times V(A)) \\
=&\{(x, y) \mid x \in (\cup\ <V(a_1) \ldots V(e_1)>), y \in V(A)\} \\
=&\{(x, y) \mid (x \in V(a_1) \vee \ldots \vee x \in V(e_1)) \wedge y \in V(A)\}\text{[8]} \\
=&(\cup\ <\{(x, y) \mid x \in V(a_1), y \in V(A)\} \ldots \{(x, y) \mid x \in V(e_1), y \in V(A)\}>) \\
=&(\cup\ <(V(a_1) \times V(A)) \ldots (V(e_1) \times V(A))>) \\
=&V((\cup\ <(a_1 \times A) \ldots (e_1 \times A)>))
\end{aligned}$$

(2) Analogous to (1) by exchanging first and second argument position of $\times$.

(3) Analogous to (1) by replacing $\cup$ by $\cap$ and $\vee$ by $\wedge$.

(4) Analogous to (3) by exchanging first and second argument position of $\times$.

[8] $(x \in V(a_1) \vee \ldots \vee x \in V(e_1))$ does not match the pattern $<n(a_1) \ldots n(e_1)>$, but since the meaning seems to be clear, we skip the overhead of formally defining this kind of expression.

(5)

$$\begin{aligned}
&V(((A \setminus C) \times B)) \\
=&((V(A) \setminus V(C)) \times V(B)) \\
=&\{(x, y) \mid x \in V(A), x \notin V(C), y \in V(B)\} \\
=&\{(x, y) \mid x \in V(A), y \in V(B)\} \setminus \{(x, y) \mid x \in V(B), y \in V(B)\} \\
=&(V(A) \times V(B)) \setminus (V(C) \times V(B)) \\
=&V((A \times B) \setminus (C \times B))
\end{aligned}$$

(6) Analogous to (5) by exchanging first and second argument position of $\times$.

The left hand side and the right hand side of every rule denote equivalent types and therefore every simplified type is equivalent to the corresponding type before simplification. □

Lemma 3.8 *Type simplification of a type t by ES_2 does not change the represented set of values, i.e. the resulting type t' fulfills $V(t) = V(t')$.*

Proof of lemma 3.8: Like in the proof of lemma 3.5, we prove this lemma for the first six rules which handle $\times$. Then there is an analogous proof for the other rules, by just replacing $\times$ by pairs or stream pairs.

First of all we should note that the commutativity constructs *comm* and *choose* just change the ordering of the elements of unions and intersections. Because of the commutativity of $\cup$ and $\cap$ this does not change the set of values represented by some type. We assume the elements being ordered in a certain way before starting the simplification and do not consider these commutativity constructs any further.
We do not consider the *notarity*-constructs either, because they only restrict the applicability of a rule, and we can prove that each rule represents a type equivalence transformation even if the constraints of the rules are dropped. (Dropping constraints may however cause termination problems.)

The rest of the proof is done for every rule separately:

(1) (In this proof as well as in the following we omit the union with $\{\perp\}$ due to the definition of the type difference.)

$$\begin{aligned}
&V((A \times B) \setminus (A \times C)) \\
=&V(A \times B) \setminus V(A \times C) \\
=&((V(A) \times V(B)) \setminus (V(A) \times V(C))) \\
=&V(A) \times (V(B) \setminus V(C)) \\
=&V(A) \times V(B \setminus C) \\
=&V(A \times (B \setminus C))
\end{aligned}$$

(2) Analogous to (1) by exchanging the first and second argument position of $\times$.

(3)

$$
\begin{aligned}
& V(((\cup\ <(a_1 \times B) \dots (e_1 \times B)>) \setminus (\cup\ <(a_1 \times C) \dots (e_1 \times C)>)))\\
=& ((\cup\ <(V(a_1) \times V(B)) \dots (V(e_1) \times V(B))>) \setminus\\
& (\cup\ <(V(a_1) \times V(C)) \dots (V(e_1) \times V(C))>))\\
=& \{(x,y) \mid (x \in V(a_1) \vee \dots \vee x \in V(e_1)) \wedge y \in V(B)\} \setminus\\
& \{(x,y) \mid (x \in V(a_1) \vee \dots \vee x \in V(e_1)) \wedge y \in V(C)\}\\
=& \{(x,y) \mid (x \in V(a_1) \vee \dots \vee x \in V(e_1)) \wedge (y \in V(B) \wedge y \notin V(C))\}\\
=& \{(x,y) \mid x \in V(a_1) \wedge (y \in V(B) \wedge y \notin V(C))\} \cup \dots \cup\\
& \{(x,y) \mid x \in V(e_1) \wedge (y \in V(B) \wedge y \notin V(C))\}\\
=& (\cup\ <((V(a_1) \times V(B)) \setminus (V(a_1) \times V(C))) \dots ((V(e_1) \times V(B)) \setminus (V(e_1) \times V(C)))>)\\
=& V((\cup\ <((a_1 \times B) \setminus (a_1 \times C)) \dots ((e_1 \times B) \setminus (e_1 \times C))>))
\end{aligned}
$$

(4) Analogous to (3) by exchanging the first and second argument of $\times$.

(5) Analogous to (3) by replacing $\cup$ by $\cap$ and $\vee$ by $\wedge$.

(6) Analogous to (5) by exchanging the first and second argument of $\times$.

$\square$

Lemma 3.10 *Every type t simplified by ES_3 to type t' denotes the same set of values before and after the simplification, i.e. $V(t) = V(t')$.*

Proof of lemma 3.10: We prove this lemma rule by rule:

(1)

$$
\begin{aligned}
& V(A \setminus B)\\
=& V(A) \setminus V(B)\\
=& \{x \mid x \in V(A) \wedge x \notin V(B)\}\\
=& \{x \mid x \in V(A) \wedge x \in V(\mathcal{C}B)\}\\
=& V(A) \cap V(\mathcal{C}B)\\
=& V(A \cap \mathcal{C}B)
\end{aligned}
$$

(2)

$$
\begin{aligned}
&V(\mathcal{C}\mathcal{C}A)\\
=&\mathcal{C}V(\mathcal{C}A)\\
=&\{x \mid x \notin V(\mathcal{C}A)\}\\
=&\{x \mid x \notin \mathcal{C}V(A)\}\\
=&V(A)
\end{aligned}
$$

(3) Rule (3) follows from the associativity of $\cup$

(4) Rule (4) follows from the associativity of $\cap$

(5)

$$
\begin{aligned}
&V(\mathcal{C}(\cup < a_1 \ldots e_1 >))\\
=&\mathcal{C}V((\cup < a_1 \ldots e_1 >))\\
=&\{x \mid x \notin V((\cup < a_1 \ldots e_1 >))\}\\
=&\{x \mid x \notin (\cup < V(a_1) \ldots V(e_1) >)\}\\
=&\{x \mid x \notin V(a_1) \wedge \ldots \wedge x \notin V(e_1)\}\\
=&(\cap\ \{x \mid x \notin V(a_1)\} \ldots \{x \mid x \notin V(e_1)\}\\
=&(\cap < (\mathcal{C}V(a_1)) \ldots (\mathcal{C}V(e_1)) >)\\
=&(\cap < V(\mathcal{C}a_1) \ldots V(\mathcal{C}e_1) >)\\
=&V((\cap < (\mathcal{C}a_1) \ldots (\mathcal{C}e_1) >))
\end{aligned}
$$

(6)

$$
\begin{aligned}
&V(\mathcal{C}(\cap < a_1 \ldots e_1 >))\\
=&\mathcal{C}V((\cap < a_1 \ldots e_1 >))\\
=&\{x \mid x \notin V((\cap < a_1 \ldots e_1 >))\}\\
=&\{x \mid x \notin (\cap < V(a_1) \ldots V(e_1))\}\\
=&\{x \mid x \notin V(a_1) \vee \ldots \vee x \notin V(e_1)\}\\
=&(\cup\ \{x \mid x \notin V(a_1)\} \ldots \{x \mid x \notin V(e_1)\}\\
=&(\cup < (\mathcal{C}V(a_1)) \ldots (\mathcal{C}V(e_1)) >)\\
=&(\cup < V(\mathcal{C}a_1) \ldots V(\mathcal{C}e_1) >)\\
=&V((\cup < (\mathcal{C}a_1) \ldots (\mathcal{C}e_1) >))
\end{aligned}
$$

(9) Let $A \sqsubseteq \mathcal{C}B$ which implies $V(A) \subseteq \mathcal{C}V(B)$ by lemma 3.3. From this follows $V(\perp) = V(A) \cap V(B) \supseteq V((\cap\ < 1 \ldots >\ A\ < 2 \ldots >\ B\ < 3 \ldots >))$. Since $V(\perp) \subseteq V(A)$ for every type A, $V((\cap\ < 1 \ldots >\ A\ < 2 \ldots >\ B\ < 3 \ldots >)) = V(\perp)$.

(10) From $B \sqsubseteq \mathcal{C}A$ we can conclude $V(\bot) = V(A) \cap V(B)$, too. The equivalence of $V((\cap <1\ldots> A <2\ldots> B <3\ldots>))$ and $V(\bot)$ follows analogously to (9).

(11) Let $\mathcal{C}A \sqsubseteq B$. Then it follows $V(\top) = V(A) \cup V(B) \subseteq V((\cup <1\ldots> A <2\ldots> B <3\ldots>))$. Since $V(A) \subseteq V(\top)$ for every type A, we have $V(\top) = V((\cup <1\ldots> A <2\ldots> B <3\ldots>))$.

(12) From $\mathcal{C}B \sqsubseteq A$ we have $V(\top) = V(A) \cup V(B)$, too, and therefore $V((\cup <1\ldots> A <2\ldots> B <3\ldots>)) = V(\top)$ analogously to (11).

(7) Obvious from the definition of $\cup$.

(8) Obvious from the definition of $\cap$.

(14) Let $A \sqsubseteq B$. By lemma 3.3 we get $V(A) \subseteq V(B)$ and it follows $V(A) \cup V(B) = V(B)$. Because of associativity and commutativity of $\cup$ we can conclude:

$$\begin{aligned}
&V((\cup <1\ldots> A <2\ldots> B <3\ldots>))\\
=&(\cup\, V(<1\ldots>)\ V(A)\ V(<2\ldots>)\ V(B)\ V(<3\ldots>))\\
=&(\cup\, (\cup\, V(A)\ V(B))\ (\cup\, V(<1\ldots>)\ V(<2\ldots>)\ V(<3\ldots>)))\\
=&(\cup\, V(B)\ (\cup\, V(<1\ldots>)\ V(<2\ldots>)\ V(<3\ldots>)))\\
=&(\cup\, V(<1\ldots>)\ V(<2\ldots>)\ B\ V(<3\ldots>))\\
=&V((\cup <1\ldots> <2\ldots> B <3\ldots>))
\end{aligned}$$

(15) Let $B \sqsubseteq A$. Analogous to 14, it follows $V((\cup <1\ldots> A <2\ldots> B <3\ldots>)) = V((\cup <1\ldots> A <2\ldots> <3\ldots>))$.

(16) Let $B \sqsubseteq A$ and therefore by lemma 3.3 $V(B) \subseteq V(A)$. It follows $V(A) \cap V(B) = V(B)$ and

$$\begin{aligned}
&V((\cap <1\ldots> A <2\ldots> B <3\ldots>))\\
=&(\cap\, V(<1\ldots>)\ V(A)\ V(<2\ldots>)\ V(B)\ V(<3\ldots>))\\
=&(\cap\, (\cap\, V(A)\ V(B))\ (\cap\, V(<1\ldots>)\ V(<2\ldots>)\ V(<3\ldots>))\\
=&(\cap\, V(B)\ (\cap\, V(<1\ldots>)\ V(<2\ldots>)\ V(<3\ldots>))\\
=&(\cap\, V(<1\ldots>)\ V(<2\ldots>)\ V(B)\ V(<3\ldots>))\\
=&V((\cap <1\ldots> <2\ldots> B <3\ldots>))
\end{aligned}$$

(17) Analogous to (16), from $A \sqsubseteq B$ follows $V((\cap <1\ldots> A <2\ldots> B <3\ldots>)) = V((\cap <1\ldots> A <2\ldots> <3\ldots>))$

(13)

$$
\begin{aligned}
&V((\cap\ {<}1\ldots{>}\ (\cup\ {<}a_2\ldots e_2{>})\ {<}3\ldots{>}))\\
=&(\cap\ V((\cap\ {<}1\ldots{>}))\ (\cup\ {<}V(a_2)\ldots V(e_2){>})\ V((\cap\ {<}3\ldots{>})))\\
=&\{x \mid x \in V((\cap\ {<}1\ldots{>})) \wedge x \in (\cup\ {<}V(a_2)\ldots V(e_2){>}) \wedge x \in V((\cap\ {<}3\ldots{>}))\}\\
=&\{x \mid x \in V((\cap\ {<}1\ldots{>})) \wedge (x \in V(a_2) \vee \ldots \vee x \in V(e_2))\ \wedge x \in V((\cap\ {<}3\ldots{>}))\}\\
=&\{x \mid (x \in V((\cap\ {<}1\ldots{>})) \wedge x \in V(a_2) \wedge x \in V((\cap\ {<}3\ldots{>}))) \vee \ldots\\
&\vee (x \in V((\cap\ {<}1\ldots{>})) \wedge x \in V(e_2) \wedge x \in V((\cap\ {<}3\ldots{>})))\}\\
=&\{x \mid x \in (\cap\ V((\cap\ {<}1\ldots{>}))\ V(a_2)\ V((\cap\ {<}3\ldots{>}))) \vee \ldots\\
&\vee x \in (\cap\ V((\cap\ {<}1\ldots{>}))\ V(e_2)\ V((\cap\ {<}3\ldots{>})))\}\\
=&(\cup\ {<}(\cap\ V((\cap\ {<}1\ldots{>}))\ V(a_2)\ V((\cap\ {<}3\ldots{>})))\ldots\\
&(\cap\ V((\cap\ {<}1\ldots{>}))\ V(e_2)\ V((\cap\ {<}3\ldots{>}))){>})\\
=&V((\cup\ {<}(\cap\ {<}1\ldots{>}\ a_2\ {<}3\ldots{>})\ldots(\cap\ {<}1\ldots{>}\ e_2\ {<}3\ldots{>}){>}))
\end{aligned}
$$

□

Theorem 3.14 *The algorithm CE only states two simplified types A and B as disjoint if for any/all assignments of the type variables in A resp. B the resulting instances A' and B' of A and B are disjoint, i.e. $V(A') \cap V(B') = \emptyset$.*

Proof of theorem 3.14: We should first note that A and B cannot contain the difference type constructor $\setminus$ after type simplification and that the type constructors (*list* T) and (*stream* T) are covered by the fixed point type constructor. For the remaining type constructors and the base types, we will prove the theorem in several steps:

In the first step we will only consider types, not containing free type variable or any of the type constructors $\cup$, $\cap$, $\mathcal{C}$.
Suppose A is a base type and $V(A) \cap V(B) \neq \emptyset$. B must be a base type as well, because all elements of types defined by a type constructor have a special structure that cannot be found in base types. Viewing the definitions of base types shows the following opportunities of two base types $A \neq \perp$ and $B \neq \perp$, having common elements (where $\perp$ cannot have common elements with any other type, because we do not regard the non-terminating computation):

1. $A = B$
2. $A = \top \vee B = \top$
3. A and B are subtypes of type `num` with $V(A) \subseteq V(B) \vee V(B) \subseteq V(A)$.

All of these cases are covered by the definition of $\sqsubseteq$ and therefore the case of base types with nonempty intersection is covered by the algorithm in step (13).

Now suppose A is defined as $A_1 \times A_2$, $(A_1 \; . \; A_2)$, $[A_1 \; . \; A_2]$ or $A_1 \to A_2$. To keep the same structure of the elements, B must contain the same top level type constructor. Under this condition, the question of common elements of A and B reduces to the question of common elements of A_1 and B_1 as well as A_2 and B_2 for the type constructors $\times$, $(\; . \;)$ and $[\; . \;]$. These tests are done in the cases (14), (15) and (16).
If $A = A_1 \to A_2$ and $B = B_1 \to B_2$, we want to know, whether there is a function f that could transform at least one element of $V(A_1 \cap B_1)$ into an element of $V(A_2 \cap B_2)$. Therefore we check for common elements of the input types A_1 and B_1 and the output types A_2 and B_2. This is done by case (17) of the algorithm.

If A is a frame type then there can only be common elements if B is a frame type, too. This is correctly handled by case (18).

If A is an environment type, B must be an environment, too, to have common elements. Different environments can be distinguished by their number of frames and the symbol bindings of the different frames. The number of frames of two environment types must be equal for having common elements while the symbol bindings are not of interest: Two frame types always have common elements. Thus the way environments are handled by case (19) is correct with respect to the theorem.

Now let $A = \mu X.A'(X)$. Because the fixed point type constructor does not determine the top level structure of type A, we have to unfold the recursive definition until we get a constructor which can be used in a direct check. If B does not contain a fixed point constructor, this unfolding can only take place a finite number of times. This situation is covered by case (7). The analogous case of $B = \mu X.B'(X)$ and A not containing a fixed point type constructor is covered by case (8).

If $A = \mu X.A'(X)$ and B also contains a fixed point constructor (analogously $B = \mu X.B'(X)$ and A also contains a fixed point constructor) this can lead to an infinite sequence of unfolding steps. But even if A and B are defined differently (e.g. $A = \mu X.(\mathit{list}\ (A \cup X))$ and $B = \mu Y.(\mathit{list}\ (A \cup (\mathit{list}\ (A \cup Y)))$ we will after some time have to unfold some argument $\tilde{A}$ of A (without loss of generality $\tilde{A} = A$) to compare it with some argument $\tilde{B}$ of B (without loss of generality $\tilde{B} = B$) for a second time. I.e. there already was an unfolding step of A to compare it with B, and this first unfolding was done k steps of unfolding A before the actual one.

Now we will show that if there are common elements of A and B there must be an element of the intersection that can be generated by unfolding A at most k times. Thus we need not unfold A after k steps any further.
For every element e of A and B a finite number of unfoldings of A resp. B suffices to generate e. Assume $V(A) \cap V(B) \neq \emptyset$, and choose one element $e \in V(A) \cap V(B)$. If $e \in V(A'^q(\bot))$, i.e. e is contained in a type, generated by unfolding A q times, with $q \leq k$, A and B are

recognized as not disjoint before checking A and B a second time.
Now assume $e \in V(A'^q(N))$, N resulting from the definition of A by unfolding A k times, replacing every occurrence of X by $\perp$ and omitting the binding of X by μ. (As a result N is a type that is not recursive in X and contains all elements of A that need at most k steps of unfolding to be generated.) Then there is an element $e' \in \Omega$ fulfilling $e = A'^k(e')$,[9] and because of $e \in V(A)$, $e' \in V(N) \subseteq V(A)$ must hold. I.e. if $k < q + k \leq 2k$ unfolding steps generate $e \in V(A)$ then there is some $e' \in V(A)$ that can be generated in $0 < q \leq k$ unfolding steps.

Furthermore we know that after unfolding A k times we have unfolded B lets say m times and returned to the original B. Therefore as above from $e \in B'^r(\perp)$ we can conclude that $e' \in B'^{r-m}(\perp)$ (because A'^k and B'^mdenote an equivalent function) and therefore e' is a common element of A and B.

By induction on i we can show that for every common element of A and B that is generated after $i * k + q$ $(0 < q \leq k)$ steps of unfolding A there is another common element that is generated in q unfolding steps. Therefore it suffices to cut the unfolding, when some situation is reached for the second time. This is done by the algorithm in case (9) (and (10) respectively): When it detects a test that has been done before, k steps of unfolding A have been performed. Instead of unfolding A again this $k + 1^{\text{th}}$ test is replaced by a test of $\perp$ and B, and the test of $\perp$ with any type results in disjointness. By this cut the algorithm does not loose any cases of common elements of A and B.

In the next step, we will consider types that contain $\cup$, $\cap$ or $\mathcal{C}$, too. The only case to prove is if the top level constructor of A of B is one of the constructors $\cup$, $\cap$, $\mathcal{C}$. Then we have the following situation:

- $A = (\cup\ A_1 \ldots A_n)$ has common elements with B if on of the arguments has a common element with B. This is checked by case (1). The inverse situation is covered by (2).
- An intersection $A = (\cap\ A_1 \ldots A_n)$ only has a common element with B if all of the arguments have the *same* common element with B. In case (3) (and analogously (4)) the algorithm states $A = (\cap\ A_1 \ldots A_n)$ and B as not disjoint, if every A_i has *any* common element with B. It may not recognize some disjoint types, but the theorem holds in this case, too.
- If $A = \mathcal{C}A'$, two types A and B are treated as disjoint only if $B \sqsubseteq A'$. This is done in case (5) of the algorithm. As stated in lemma 3.3, this is only the case if $V(B) \subseteq V(A')$ and therefore $V(A) \cap V(B) = \emptyset$. The theorem holds in this case, too, as well as in the analogous situation $B = \mathcal{C}B'$ covered by (6).

[9] We use A' as a function $A' : \Omega \to \Omega$ as well as $A' : \mathcal{T} \to \mathcal{T}$. This should not cause any confusion, because the actual argument type can be determined easily.

In the last step, we consider types containing free type variables, as well. As long as the variables are arguments to type constructors, the structure of these constructors is compared. This leads to the question, whether some type variable Q has common elements with some type T which is answered by the cases (11) and (12) considering the flags a and b. The only case in which every assignment to Q with $Q \neq \bot$ leads to common elements with T is $T = \top$ and $T = Q$. This is recognized by the algorithm for the flag `any`. For the flag `all` disjointness is only possible if the other type is $\bot$. This is handled correctly by the cases (11) and (12), too. Therefore the theorem holds. $\square$

B Proofs for section 6

Theorem 6.5 *Let P be a non-recursive program that does not cause any side effects, $c = (f\ p_1 \dots p_k)$ a call in P of function f with parameters $p_1, \dots p_k$, and ET an input typing for P. Let $s_1, \dots, s_n$ be a list of input arguments to P which are consistent with some instance ET' of ET and cause an execution of P without type errors.*
If $a_1, \dots, a_k$ are the arguments which are assigned to $p_1, \dots, p_k$ during the mentioned type error free execution, then the type t_i, propagated for p_i under ET contains a_i $(i = 1, \dots, k)$.

Proof of theorem 6.5: We prove this theorem by induction on the maximal number n of evaluation steps to evaluate a_i which is finite, because we assume P to be free of recursion:

$\mathbf{n = 0}$: The statement holds, because without any evaluation step the a_i must be some of the arguments $s_1, \dots, s_n$ to P and these are consistent to ET', and thus are contained in ET'.

$\mathbf{n \to n+1}$: Consider $c = (f\ p_1 \dots p_k)$ with the maximal number $n+1$ of steps for evaluating one of the arguments $a_1, \dots, a_k$. Now regard an arbitrary $i \in \{1, \dots, k\}$ with $a_i = (f'\ p'_1 \dots p'_{k'})$.

If a_i contains a type error, a_i must be a subexpression of a conditional that is not evaluated for the given input parameters, because these input parameters do not cause a type error according to the precondition of the theorem. This is handled correctly by the typing rules of *if* and *cond*. Thus in the following, we can assume a_i to be free of type errors.

If at most n evaluation steps are needed to evaluate a_i, we have nothing to show and should choose another i. (At least one a_j needs $n+1$ evaluation steps.)
If $n+1$ steps are needed to evaluate a_i, the arguments a'_j assigned to p'_j can be evaluated in at most n steps and therefore we know from the induction hypothesis that the types t'_j propagated for p'_j contain a'_j for every $j \in \{1, \dots, k'\}$. For f' there is a type definition $f' : t'_1, \dots t'_{k'} \to t''$ with $a'_j \in V(t'_j)$,[10] because the execution of P with arguments $s_1, \dots s_n$ yields no type errors. By lemma 6.2 the search for such a transformation rule by the system is successful (without loss of generality let the system choose $f' : t'_1, \dots t'_{k'} \to t''$), by lemma 6.1 $a_i = (f'\ a'_1 \dots a'_{k'}) \in t''$, and by theorems 3.11 and 3.12 the type t_i which is derived from t'' by type simplification fulfills $a_i \in t'' \subseteq t_i$. $\Box$

Lemma 6.6 *Let P be a recursive program that does not cause any side effects, and ET an input typing to P such that every instance $\sigma(ET)$ of ET contains an argument set $s_1, \dots, s_k$ not causing a type error. If the type propagation algorithm does not run into an infinite loop, it accepts P.*

[10] If there are some p'_j defined as union types, there might be more than one f' applicable for the different elements. In this case, t_i is set to the union of all different types and is therefore a supertype of the exact type of a_i as described in subsection 4.3.2.

Proof of lemma 6.6: Without loss of generality the call graph contains only one cycle which contains the main function f. (Our algorithm starts propagation with the acyclic reduced call graph and processes at most one innermost cycle at a time. Therefore we can assume the existence of only one cycle which may contain f, since the non-recursive part down to the first cycle is processed in the non-recursive manner that was proved to be complete in theorem 6.5.)
Let n be the number of recursive calls to function f (not counting the initial call) when calling P with parameters $s_1, \ldots, s_k$, and let r be the result of the initial call to f. We will show that type t_n which is generated in the n^{th} step of iterated type propagation on f with the input typing given by $s_1, \ldots, s_k$ (resp. the final type when less then n iterations yield a fixed point) contains r. Because of the definition $t_i = t_{i-1} \cup t'_i$ for every $i = 1, 2, \ldots$, this also holds for every $m > n$ and therefore for the final type.

We prove $r \in V(t_n)$ by induction on n under the assumption that at least n iterative type propagation steps are performed, even if the output type does not change any more:
$n = 0$: If no recursive calls are necessary to compute r, this result must be computed in the non-recursive part of f and must be contained in the non-recursive type t_0.
$n \to n+1$: Consider r is computed by f using $n+1$ recursive calls. Consider the calls to f, directly taking place in the top level execution of f. Suppose there are m of those calls, denoted by $c_1, \ldots, c_m$ ($m \geq 1$), where each c_i is a call to f with arguments $s_1^{(i)}, \ldots s_k^{(i)}$ and results in $r^{(i)}$ after at most n recursion steps. Even if the c_i are nested, there is a partial ordering $\prec$ with $c_i \prec c_j$ if c_i can be evaluated without evaluating c_j before.
For one least element c_j of $\prec$ the arguments $s_l^{(j)}$ can be typed with at most n recursion steps on f, and therefore we have an input typing ET' of f which is consistent to $s_1^{(j)}, \ldots s_k^{(j)}$ (theorem 6.5) and generates its result $r^{(j)}$ in at most n recursive calls to f. Induction hypothesis states that iterated type propagation yields a type t_j with $r^{(j)} \in V(t_j)$.
By induction on the length q of the longest list $c_{p_1}, \ldots c_{p_q}$ with $c_{p_l} < c_{p_{l+1}}$ we can prove that every recursive call c_i to f taking place at the top level execution of f on $s_1, \ldots, s_n$ and returning the result $r^{(i)}$, gets a type $t^{(i)}$ with $r^{(i)} \in V(t^{(i)})$.
After all recursive calls in the top level call of f with arguments $s_1, \ldots, s_n$ are typed correctly, type propagation goes on in a non-recursive manner. The rest of the proof of $r \in V(t_n)$ is similar to the proof of theorem 6.5.

Now consider the case $t_i = t_{i-1}$ with $i < n$. This implies $t_{j+1} = t_j$ for every $j \geq i$, because every calculation extending the intermediate output type t_j to t_{j+1} with $V(t_j) \subset V(t_{j+1})$ could have been done in step i already. Because of $r \in V(t_n) = V(t_i)$ it is sufficient to stop the iteration, after the intermediate output type did not change for the first time at step i.

In every case the iterated type propagation provides a type t at termination that fulfills $r \in V(t)$. Therefore the algorithm of iterated type propagation accepts P. $\square$

Lemma 6.7 *If the iterated type propagation runs into an infinite loop, this is detected and* *the methods for approximating the needed output type do not violate the completeness of the system.*

Proof of lemma 6.7: An infinite loop in iterated type propagation can have two reasons:

1. The input typing of the recursive function calls changes from iteration to iteration, and since there is only a finite number of different functions to be called, there is a function which is typed under an infinite number of input typings.
2. The iterated type propagation for a function under one input typing does not stop, because every step generates a more general intermediate output type.

For both cases a maximal number of iterations is defined. If one of these thresholds is exceeded, the existence of an infinite loop is assumed. This is sufficient to detect every infinite loop, because in case (1) there must be at least one function which number of different input typings exceeds every threshold, and in case (2) this is true for the number of different intermediate output types. Therefore the iterated type propagation cannot get stuck in an undetected infinite loop.

Solving an infinite loop cannot violate the completeness of the system, because the types generated by the generation algorithm for recursive types in subsection 4.5.1 or backwards propagation in subsection 4.6 are checked for their usefulness to solve the infinite loop. The type $\top$ that is chosen on failure of this test does not violate the completeness, because the test for common elements of $\top$ and some other type t succeeds for every $t \neq \bot$. □

Lemma 6.9 *Let s and s' be symbols, and consider a side effect changing the type of s which affects s', as well. If the side effect propagation from s' directly back to s is blocked, recursive propagation of the side effect on s' does not affect s again.*

Proof of lemma 6.9: In the first step, we prove the lemma under the following assumption: Every symbol definition of a new symbol s_n involves at most one already defined symbol s_o: Consider the undirected symbol dependency graph $D = (V_D, R_D)$, where V_D is the set of actually defined symbols and R_D contains an edge (s_1, s_2), iff the actual definition of s_2 uses s_1.

This graph does not contain any cycles, because D does not contain any loops and for any two nodes s_1 and s_2 there is at most one path[11] which starts at s_1 and ends at s_2. This can be shown by induction on the number n edges in D:
$n = 0$: Since D does not contain any edges, the proposition is clear.
$n \to n + 1$: Let D be a graph with n edges, fulfilling the proposition. The only way of introducing new edges is defining a symbol s_n using another symbol s_o. If s_n is *re*defined, the

[11]In the following by paths we only mean those paths containing every node at most once.

old definition is lost, and we can therefore assume without loss of generality that s_n is a new symbol. (Otherwise the old node for s_n is deleted with all its edges or renamed, resulting in a graph which clearly fulfills the induction hypothesis.) The new graph is given by adding a new node s_n and connecting this node with s_o. Thereby no loop is introduced and clearly this does not generate any new paths from s_1 to s_2 if $s_n \neq s_1 \wedge s_n \neq s_2$. There also is at most one path from some arbitrary s_1 to s_n for $s_1 \neq s_o$ because:

1. Every path from s_1 to s_n contains s_o as a node.
2. By induction hypothesis there is only one path p_1 from s_1 to s_o.
3. There is only one path p_2 from s_o to s_n, given by the new edge.

Therefore the path $p = p_1 \bullet p_2$ is the only one from s_1 to s_n.

When D does not contain any cycles, every strongly connected component of D is given by a tree. Consider the side effected symbol s as root of its strongly connected component. The propagation of the side effect is done top down in the tree, because:

1. For every son s' of s, the first change of s' is recognized as coming from s and does not directly effect s again.
2. Propagating the side effect on s' in the subtree of s' does not change s' again, as can be shown by induction on the height of the subtree.

Therefore the type of s is not changed again during side effect propagation.

In the next step we extend the prove to symbol definitions binding a symbol s to an expression e involving more than one (without loss of generality we will give a prove for two) already defined symbols. In this case, the symbol dependency graph may contain cycles.
Side effects only have to be propagated through this definition, if the only functions handling the former defined symbols s_1 and s_2 in e are *cons*, *car* and *cdr*. *cons* is the only one of these functions expecting more than one argument, and therefore there must be a call to *cons* separating s_1 and s_2 in its different arguments.
Since one side effect can cause a change only *inside one* of the arguments of *cons* and a side effect replacing the *whole* *cons*-subexpression does not affect s_1 and s_2, at most one of these can be affected by a side effect on s. If we examine the "active part" D_C of D under some side effect C which contains only those edges connecting two symbols that affect each other under C, s gets at most one edge during its definition. We can therefore repeat the induction of the first part to prove D_C to be free of cycles for any C. Therefore, the lemma holds in the general case, too. □

Lemma 6.10 *The extension for handling side effects in section 5 does not violate the completeness of the type system.*

Proof of lemma 6.10: When checking a program P without side effects, the only change is the replacement of input typings by the environment concept. This cannot violate the completeness of the system proved in theorem 6.8, because every information that is provided by an input typing is identically extractable from an equivalent environment.

Now consider a side effect that changes the binding of some symbol s to the result of an expression e with exact type t, where (without loss of generality) the evaluation of e does not cause any further side effects. (The proof for the general case has to be done by induction on the number of side effects.) The type t_d, derived for e by type propagation is a supertype of t, i.e. $V(t) \subseteq V(t_d)$, by theorem 6.8.
A supertype of the actual exact type of s is known at any execution point, because when processing a function with side effect on s, the type of s is changed immediately. Furthermore conditional side effects are handled correctly, because when leaving the conditional construct, the union of the results of all possible side effects is assigned to s immediately. Therefore this union, containing a supertype of the exact type, is used in every situation where it is not known, whether the side effect was performed or not.

The problem of changing several symbols at once only occurs in using *set-car!* and *set-cdr!*, where only a part of an object is changed. Every type change that may affect other symbols is propagated immediately. This is done a finite number of propagation steps according to lemma 6.9.
As shown above, the type generated for the direct target of a side effect is a supertype of its exact type. This follows for every indirectly affected symbol, too, because the pair type constructor and all derived type constructors are monotonic in all their arguments.

Altogether extending the type system to handle side effects as described in section 5 does not violate the completeness. □

C Type definitions of predefined Functions

The following set $\mathcal{R}$ of transformation rules for the predefined functions should be used for type propagation: Whenever there is more than one applicable type definition, the most special definition of input types is used.

Boolean functions:

$$\begin{aligned} \textit{and}, \textit{or} &: \texttt{bool}^k \to \texttt{bool} \\ \textit{not} &: \texttt{bool} \to \texttt{bool} \\ \textit{eq?},\ \textit{eqv?},\ \textit{equal?} &: \top \times \top \to \texttt{bool} \end{aligned}$$

Functions on numbers:

$$\begin{aligned} + &: A^k \to A \text{ für } A \in \{\texttt{nat}, \texttt{int}, \texttt{rat}, \texttt{real}, \texttt{num}\} \\ - &: A^k \to A \text{ für } A \in \{\texttt{int}, \texttt{rat}, \texttt{real}, \texttt{num}\} \\ * &: A^k \to A \text{ für } A \in \{\texttt{nat}, \texttt{int}, \texttt{rat}, \texttt{real}, \texttt{num}\} \\ / &: A^k \to A \text{ für } A \in \{\texttt{rat}, \texttt{real}, \texttt{num}\} \\ 1+ &: A \to A \text{ für } A \in \{\texttt{nat}, \texttt{int}, \texttt{rat}, \texttt{real}, \texttt{num}\} \\ -1+ &: A \to A \text{ für } A \in \{\texttt{int}, \texttt{rat}, \texttt{real}, \texttt{num}\} \\ \textit{odd?},\ \textit{even?} &: \texttt{int} \to \texttt{bool} \\ <,\ >,\ =,\ <=,\ >= &: \texttt{num}^k \to \texttt{bool} \end{aligned}$$

List processing functions:

$$\begin{aligned} \textit{cons} &: A \times B \to (A\ .\ B) \\ \textit{cons} &: A \times (\textit{list}\ A) \to (\textit{list}\ A) \\ \textit{car} &: (A\ .\ \top) \to A \\ \textit{car} &: (\textit{list}\ A) \to A \\ \textit{cdr} &: (\top\ .\ B) \to B \\ \textit{cdr} &: (\textit{list}\ A) \to (\textit{list}\ A) \\ \textit{list} &: A^k \to (\textit{list}\ A) \\ \textit{null?} &: \top \to \texttt{bool} \\ \textit{append} &: (\textit{list}\ A) \times (\textit{list}\ B) \to (\textit{list}\ (A \cup B))^{12} \end{aligned}$$

[12] *append* expects only lists as parameters. Types, defined by the pair type constructor, must be converted to list types, using the generalizing type simplification from section 3.5.3.

Stream processing functions:

$$\begin{aligned}
consstream &: A \times B \to [A\ .\ B] \\
consstream &: A \times (stream\ A) \to (stream\ A) \\
head &: [A\ .\ \top] \to A \\
head &: (stream\ A) \to A \\
tail &: [\top\ .\ B] \to B \\
tail &: (stream\ A) \to (stream\ A) \\
stream &: A^k \to (stream\ A) \\
empty\text{-}stream? &: \top \to \texttt{bool}
\end{aligned}$$

Functions manipulating environments:

$$make\text{-}environment : \top^k \to \texttt{env}$$

Miscellaneous functions:

$$\texttt{error} : \top \to \mathcal{O}$$

D Type definitions for backwards propagation

In this appendix we present the set $\mathcal{R}_B$ of alternative type definitions for backwards propagation of output types. The input types defined by these rules are supertypes of those input types that can generate the given output type. If more than one rule is applicable, the most special output type should be chosen.

Boolean functions:

$$\begin{aligned} \mathit{and}, \mathit{or} &: \texttt{bool} \leftarrow \texttt{bool}^k \\ \mathit{not} &: \texttt{bool} \leftarrow \texttt{bool} \\ \mathit{eq?},\ \mathit{eqv?},\ \mathit{equal?} &: \texttt{bool} \leftarrow \top \times \top \end{aligned}$$

Functions on numbers:

$$\begin{aligned} +, -, *, / &: \texttt{rat} \leftarrow \texttt{rat}^k \\ +, -, *, / &: \texttt{real} \leftarrow \texttt{num}^k \\ +, -, *, / &: \texttt{num} \leftarrow \texttt{num}^k \\ 1+ &: A \leftarrow A \text{ für } A \in \{\texttt{int}, \texttt{rat}, \texttt{real}, \texttt{num}\} \\ -1+ &: A \leftarrow A \text{ für } A \in \{\texttt{nat}, \texttt{int}, \texttt{rat}, \texttt{real}, \texttt{num}\} \\ \mathit{odd?},\ \mathit{even?} &: \texttt{bool} \leftarrow \texttt{int} \\ <,\ >,\ =,\ <=,\ >= &: \texttt{bool} \leftarrow \texttt{num}^k \end{aligned}$$

List processing functions:

$$\begin{aligned} \mathit{cons} &: (A\ .\ B) \leftarrow A \times B \\ \mathit{cons} &: (\mathit{list}\ A) \leftarrow A \times (\mathit{list}\ A) \\ \mathit{car} &: A \leftarrow (A\ .\ \top) \\ \mathit{cdr} &: B \leftarrow (\top\ .\ B) \\ \mathit{list} &: (\mathit{list}\ A) \leftarrow A^k \\ \mathit{null?} &: \texttt{bool} \leftarrow \top \end{aligned}$$

Stream processing functions:

$$\begin{aligned}
\mathit{consstream} &: [A\ .\ B] \leftarrow A \times B \\
\mathit{consstream} &: (\mathit{stream}\ A) \leftarrow A \times (\mathit{stream}\ A) \\
\mathit{head} &: A \leftarrow [A\ .\ \top] \\
\mathit{stream} &: B \leftarrow [\top\ .\ B] \\
\mathit{stream} &: (\mathit{stream}\ A) \leftarrow A^k \\
\textit{empty-stream?} &: \texttt{bool} \leftarrow \top
\end{aligned}$$

Functions manipulating environments:

$$\textit{make-environment} : \texttt{env} \leftarrow \top^k$$

Assignments:

$$\begin{aligned}
\textit{set!} &: \mathcal{O} \leftarrow \texttt{sym} \times \top \\
\textit{set-car!} &: \mathcal{O} \leftarrow (\top\ .\ \top) \times \top \\
\textit{set-cdr!} &: \mathcal{O} \leftarrow (\top\ .\ \top) \times \top
\end{aligned}$$

Miscellaneous:

$$\mathit{map} : A \leftarrow (\perp \rightarrow A) \times (\mathit{list}\ \top)$$