Reconstruction of a logic for inductive proofs of properties of functional programs

David Sabel and Manfred Schmidt-Schauß

Institut für Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
{sabel,schauss}@ki.informatik.uni-frankfurt.de

Technical Report Frank-39

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

January 21, 2010

Abstract. The interactive verification system VeriFun is based on a polymorphic call-by-value functional language and on a first-order logic with initial model semantics w.r.t. constructors. This paper provides a reconstruction of the corresponding logic when partial functions are permitted. Typing is polymorphic for the definition of functions but monomorphic for terms in formulas. Equality of terms is defined as contextual equivalence based on observing termination in all contexts. The reconstruction also allows several generalizations of the functional language like mutual recursive functions and abstractions in the data values. The main results are: Correctness of several program transformations for all extensions of a program, which have a potential usage in a deduction system. We also proved that universally quantified equations are conservative, i.e. if a universally quantified equation is valid w.r.t. a program P, then it remains valid if the program is extended by new functions and/or new data types.

1 Introduction

Proving properties of recursively defined functions by induction is a clean method for validating properties of programs and functions that operate on finite data structures. There are a couple of tools that are designed to perform this task automatically, or give support in constructing a proof. Such a system is VeriFun (see [WS05,SWGA07,Wal94,Ade09] and [Wal09]).

The logical framework consists of a pure and strict functional programming language and a logical component that allows to formulate and prove lemmas about properties and the behavior of functions. Since one is usually interested in the behavior of functions on data objects like numbers, lists or trees, the focus of the logic and the system is to prove properties of the functions on the data, like commutativity of addition (on Peano-numbers), or associativity of append on lists.

Proofs of the system VeriFun require as a "must" that functions are proved to terminate before any other lemma about the function can be proved. However, this does not force all functions to be total; there are also partial functions permitted, like the selector-functions *head* and *tail* for lists, which are undefined for the empty list. The method to deal with this undefinedness is to add conditions to the lemmas that ensure that all function applications in the to-be-proved theorem have a defined value.

The semantics and justification for the logics in the papers [WS05,SWGA07,Ade09] are constructed by defining term-algebras over the data constructors, and functions are viewed as mapping data-arguments to a result that is a data object. The approach in [WS05] is slightly different from ours: In our logic all undefined objects (of a certain type) are contextually equal, whereas in the approach of [WS05] there may be different undefined objects. For example $minus(0,1)$ and $minus(0,2)$ are not equal using the approach [WS05], whereas these terms are equal w.r.t. our contextual equivalence. Also, $minus(2,4) = 25$ is false in our logic, whereas it is neither true nor false in [WS05].

The main goal of this paper is to provide a reconstruction of the semantics of the logical system with the intention to be as general as possible. Informally, programs comprise polymorphic function definitions and data types, and the logical formulas are monomorphic where the quantification is over closed data values. The following generalizations have been integrated: higher-order objects are not prevented to be also data, in the logical level functions that may not terminate on certain arguments are permitted, and mutual recursive function definitions are permitted on the top level. The use of contextual semantics gives a clear understanding of equality of any kinds of objects: it allows to prove correctness of several program transformations: call-by-value (beta) and (case) reductions are correct (see Theorem 6.12), Bot-reductions, adapted call-by-name reductions and some other transformations are correct (see Theorem 6.12). For call-by-name reductions, an appropriate `seq` is introduced and the reductions are adapted as follows: for example (VNbeta): $((\lambda x.s)\ t) \rightarrow (\texttt{seq}\ t\ s[t/x])$ is correct. Also program transformations for undefined expressions like $(s\ \texttt{Bot}) \rightarrow \texttt{Bot}$ are shown to be correct. What can also be formally treated and proved is conservativity of certain forms of theorems under signature extensions. Universally quantified equations that are valid with respect to program P are also valid in all extensions, in particular if data constructors are added. The proof required techniques for contextual equality in combination with polymorphic types, (see [SSSH09]),

techniques to prove a CIU-Theorem from context lemmas (see [SSS09], and an adaptation of the subterm property of simply-typed lambda-calculi.
An interesting generalization are polymorphic formulas, which are expressible and in the scope of the prover VeriFun, provided the quantifier prefix is $\forall^*$. Polymorphic formulas allow type variables in the type of quantified variables. Conservativity of polymorphic theorems is false in general. It appears too hard to prove for even very restricted classes. However, with a little bit of care, the usual inductive proof techniques automatically ensure that a proved theorem holds for all program extensions.

Structure of the Paper In Sections 2 and 3 we define the syntax and semantics of the polymorphic call-by-value functional language, its operational semantics and the equality relation. Then we show the CIU-Lemma via a context lemma (Section 4).
We show in Section 5 that equality is conservative if programs are extended by new function definitions and new data types, provided some preconditions hold. In Section 7 bisimulation is defined and show to be a characterization of equality. Finally, in Section 8 we explain the logic and its semantics.

2 The Functional Language

There are two levels of the syntax: (i) terms and defined functions, and (ii) the logical level. We focus now on (i), whereas (ii) is postponed to Section 8. Terms (or expressions) as well as types are built over a signature $(\mathcal{F}, \mathcal{K}, \mathcal{D})$ where $\mathcal{F}$ is a finite set of *function symbols*, $\mathcal{K}$ is a finite set of *type constructors*, and $\mathcal{D}$ is a finite set of *data constructors.* Type constructors $K \in \mathcal{K}$ have a fixed arity $ar(K)$ and for every $K \in \mathcal{K}$ there is a finite set $\emptyset \neq D_K \subseteq \mathcal{D}$ of data constructors $c_{K,i}$ where $c_{K,i} \in D_K$ comes with a fixed arity $ar(c_{K,i})$. For different $K_1, K_2 \in \mathcal{K}$ it holds $D_{K_1} \cap D_{K_2} = \emptyset$ and $\mathcal{D} = \bigcup\limits_{K \in \mathcal{K}} D_k$.

2.1 Syntax of Types

Since terms are constructed under polymorphic typing restrictions, we first define types, data and type constructors and then the expression level.
Types T are defined by: $T ::= X \mid (T_1 \to T_2) \mid (K\ T_1 \dots T_{ar(K)})$, where the symbols X, X_i are type variables, T, T_i stand for types, and $K \in \mathcal{K}$ is a type constructor. As usual we assume function types to be right-associative, i.e. $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$. Types of the form $\tau_1 \to \tau_2$ are called *arrow types*, and types $(K\ T_1 \dots T_{ar(K)})$ are called *constructed types.* We also allow *quantified types* $\forall \mathcal{X}.T$, where T is a type, and where $\mathcal{X}$ is the set of all free type variables in T.
Types T are defined by: $T ::= X \mid (T_1 \to T_2) \mid (K\ T_1 \dots T_{ar(K)})$, where the symbols X, X_i are type variables, T, T_i stand for types, and $K \in \mathcal{K}$ is a type constructor. As usual we assume function types to be right-associative, i.e. $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$. Types of the form $\tau_1 \to \tau_2$ are called *arrow types*, and types $(K\ T_1 \dots T_{ar(K)})$ are called *constructed types.* We also allow

quantified types $\forall\mathcal{X}.T$, where T is a type, and where $\mathcal{X}$ is the set of all free type variables in T. Let K be a type constructor with data constructors D_K. Then the (universally quantified) type $\mathit{typeOf}(c_{K,i})$ of every constructor $c_{K,i} \in D_K$ must be of the form $\forall X_1,\ldots,X_{ar(K)}.T_{K,i,1} \to \ldots \to T_{K,i,m_i} \to K\ X_1 \ldots X_{ar(K)}$, where $m_i = ar(c_{K,i})$, $X_1,\ldots,X_{ar(K)}$ are distinct type variables, and only X_i occur as free type variables in $T_{K,i,1},\ldots,T_{K,i,m_i}$.

2.2 Syntax of Expressions of $\mathcal{P}$

The (type-free) syntax of expressions over a signature $(\mathcal{F},\mathcal{K},\mathcal{D})$ is as follows, where E means expressions, $K \in \mathcal{K}$ is a type constructor, c, c_i are data constructors (i.e. elements of some set D_K where $K \in \mathcal{K}$, V generates a variable of some infinite set of variables, and Alt is a `case`-alternative:

$$\begin{array}{ll} E & ::= V \mid F \mid (E\ E) \mid \lambda V.E \mid (c_i\ E_1 \ldots E_{ar(c_i)}) \\ & \quad \mid (\texttt{case}_K\ E\ Alt_1 \ldots Alt_n) \qquad\qquad \text{where } n = |D_K| \\ Alt_i & ::= ((c_i\ V_1 \ldots V_{ar(c_i)})\ \texttt{->}\ E) \end{array}$$

Note that data constructors can only be used with all their arguments present. We assume that there is a $\texttt{case}_K$ for every type constructor K. The $\texttt{case}_K$-construct is assumed to have a case-alternative $((c_i\ x_1 \ldots x_{ar(c_i)})\ \texttt{->}\ e)$ for every constructor $c_i \in D_K$, where the variables in a pattern have to be distinct. The scoping rules in expressions are as usual. We assume that expressions satisfy the distinct variable convention before reduction is applied, which can be achieved by a renaming of bound variables. We assume that the 0-ary constructors `True`, `False` for type constructor `Bool`, and the 0-ary constructor `Nil` and the infix binary constructor ":" for lists with unary type constructor `List` are among the constructors.

Additionally we require the notion of *contexts* C, which are like expressions with the difference that the hole $[\cdot]$ may occur at a subexpression position, and where the hole occurs exactly once in C. The notation $C[s]$ means the expression that results from replacing the hole in C by s, where perhaps variables are captured. A *value* v is defined as $v ::= x \mid \lambda x.s \mid (c\ v_1 \ldots v_n)$, i.e. a variable, an abstraction, or a constructor-expression $(c\ v_1 \ldots v_n)$, where the immediate subexpressions are also values.

For an expression t the set of free variables of t is denoted as $FV(t)$ and the set of function symbols occurring in t is denoted as $FS(t)$. An expression t is called *closed* iff $FV(t) = \emptyset$, and otherwise called *open*. For a (perhaps universally quantified) type T the set of free type variables is denoted with $FTV(T)$.

Definition 2.1. *A program $\mathcal{P}$ consists of*

1. *a signature $(\mathcal{F},\mathcal{K},\mathcal{D})$.*
2. *a set of pairs $\{(f,d_f) \mid f \in \mathcal{F}\}$, where d_f is a closed value called the* definitional expression *of f, and $FS(d_f) \subseteq \mathcal{F}$. Usually, the pairs (f,d_f) are written $f = d_f$.*

*With $L_{\mathcal{P}}$ we denote the language for the expressions built over the signature corresponding to $\mathcal{P}$. Accordingly for a given program we call the expressions $\mathcal{P}$-*expressions*, the values $\mathcal{P}$-*values*, the contexts $\mathcal{P}$-*contexts*, and the types $\mathcal{P}$-*types*.*

Note that it is allowed that functions are defined mutually recursive.

Example 2.2. The identity function can be defined as $id = \lambda x.x$ where $id \in \mathcal{F}$, and the *map*-function as $map = \lambda f, xs.$`case` xs $((y : ys$ `->` $(f\ y : map\ f\ xs)$ (`Nil` `->` `Nil`)), provided $map \in \mathcal{F}$.

2.3 Typing of Expressions

We extend expressions now with type labels and distinguish between usual expressions and definitional expressions that are used to built the definition of the functions:

Definition 2.3. *Every expression and subexpressions of $\mathcal{P}$ is labeled with a closed (unquantified) type, and every pair $(f, d_f) \mid f \in \mathcal{F}$ is labelled with a perhaps quantified type. This type is also called the* type of d_f *for short.*
There are two kinds of expressions:

- Program expressions*, which are the expressions and subexpressions that appear in the definitions of the function symbols. These may be labelled with types that may contain free variables.*
- (usual) expressions*: These may be labeled only with monomorphic types (i.e. closed types) that do not contain free type variables.*

We also assume that contexts are type-labelled like expressions, where the hole is labeled with a closed type T, written $C[\cdot :: T]$.

Assumption 2.4. *We assume that the polymorphic types of the function definitions can be verified by a polymorphic type system using a type derivation system as given in the appendix.*

Below in Subsection 2.5 we will define consistency rules for the type labels.

2.4 Type-Substitutions

Given a quantified type $\forall\mathcal{X}.T$, a *(type-)substitution* ρ for $\forall\mathcal{X}.T$ substitutes types for type variables X, such that $\rho(T)$ is an (unquantified) type.

Example 2.5. Let T be the type $\forall a, b.a \rightarrow b$. Then `Int` $\rightarrow$ `Int` is an instance of T, as well as $a \rightarrow$ `Int`, where the latter has a variable name in common with T.

Example 2.6. The polymorphic type of the identity $\lambda x.x$ is $\forall a.a \rightarrow a$. The type of the function composition $\lambda f, g, x.f\ (g\ x)$ is $\forall a, b, c.(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$.

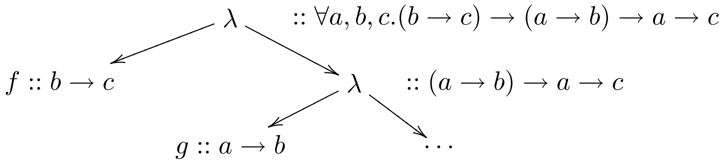
2.5 Type Consistency Rules

In this section we will detail the assumptions on the Church-style polymorphic type system that fixes the type also of subexpressions using labels at every subexpression. We will define consistency rules that ensure that the labeling of the subexpressions is not contradictory.
We assume that for every quantifier-free type T, there is an infinite set V_T of variables of this type. If $x \in V_T$, then T is called the *built-in* type of the variable x. This means that renamings of bound variables now have to keep exactly the type.

Example 2.7. This example shows a type-labeled expression that may appear in the definition of a function symbol. The type of the composition is `(.)` $::$ $\forall a, b, c.(b \to c) \to (a \to b) \to a \to c$. A type labeling (the types of some variables are not repeated) for the composition may be:

$$\begin{array}{l}(\lambda f :: (b \to c).(\lambda g :: (a \to b). \\ \qquad (\lambda x :: a.(f\ (g\ x) :: b) :: c) :: (a \to c)) :: ((a \to b) \to a \to c)) \\ \qquad\qquad :: \forall a, b, c.(b \to c) \to (a \to b) \to a \to c\end{array}$$

An illustration is as follows:


λ
$:: \forall a, b, c.(b \to c) \to (a \to b) \to a \to c$
$f :: b \to c$
λ
$:: (a \to b) \to a \to c$
$g :: a \to b$
$\cdots$

Type-Constraints:

1. The type-label of a variable $x \in V_T$ is its built-in type T.
2. Function symbols f are labeled with a type that is an instance of the polymorphic type of the equation $f = d_f$.
3. The label S of a constructor c must be an instance of the predefined type of c.
4. In the definition $f = d_f$, where $\forall \mathcal{X}.T$ is the type of the definition $f = d_f$, the type label of d_f is T and any symbol g in d_f can only have type variables that also occur in $\mathcal{X}$.
5. The type-label of every compound expression must be derivable using the rules of *MonoTp* defined in figure 1 based on the type labels of the subexpressions.

Definition 2.8. *If an expression $t :: T$ satisfies all the type constraints above, then we call the type labeling* admissible*, and the expression $t :: T$* well-typed.

Definition 2.9. *We say a program $\mathcal{P}'$* extends *the program $\mathcal{P}$, if $\mathcal{P}'$ is a program that may add type constructors, together with their data constructors, and function symbols together with their definitions, and where the type labels of the definitions of $\mathcal{P}$ are the same in $\mathcal{P}'$.*

Application

$$(s :: S_1 \to S_2 \ t :: S_1) \qquad\qquad \mapsto S_2$$

Constructor expressions

$$(c :: (S_1 \to \ldots \to S_n \to S) \ s_1 :: S_1 \ldots s_n :: S_n) \qquad\qquad \mapsto S$$

Abstractions

$$(\lambda x :: S_1.s :: S_2) \qquad\qquad \mapsto S_1 \to S_2$$

Case-expression

$$\left.\begin{aligned} &(\mathtt{case}_K \ s :: S \ ((c_{K,1} \ x_{1,1} \ldots x_{1,n_1}) :: S \ \texttt{->} \ t_i :: T) \\ &\ldots \\ &\qquad\qquad\qquad ((c_{K,m} \ x_{m,1} \ldots x_{m,n_m}) :: S \ \texttt{->} \ t_m :: T)) \end{aligned}\right\} \mapsto T$$

Fig. 1. Computation of *MonoTp*

(beta) $R[((\lambda x.s) \ v)] \to R[s[v/x]]$

(delta) $R[f :: T] \to R[d_f]$ if $f = d_f :: T'$ for the function symbol f
The reduction is accompanied by a type instantiation
$\rho(d_f)$, where $\rho(T') = T$

(case) $R[(\mathtt{case} \ (c \ v_1 \ldots v_n) \ \ldots ((c \ y_1 \ldots y_n) \ \texttt{->} \ s) \ldots)]$
$\to R[s[v_1/y_1, \ldots, v_n/y_n]]$

Fig. 2. Standard Reduction rules

3 Operational Semantics

For the definition of the standard reduction $\to$ we require the notion of reduction contexts. For a fixed program $\mathcal{P}$ the *$\mathcal{P}$-Reduction contexts* R are defined by the following grammar:

$$R ::= [\cdot] \mid (R \ s) \mid (v \ R) \mid \mathtt{case} \ R \ \mathtt{of} \ \mathit{alts} \mid (c \ v_1 \ \ldots \ v_i \ R \ s_{i+2} \ldots \ s_n)$$

where s, s_i are $\mathcal{P}$-expressions and v, v_i are $\mathcal{P}$-values. Standard reduction rules are defined in figure 2, without mentioning types.

Definition 3.1. *An* answer *is a value, but not a variable. The* evaluation *of an expression t is a maximal reduction sequence consisting of standard-reductions. We say that an expression s* terminates *(or* converges*) iff s reduces to an answer by its evaluation, denoted by $s\downarrow$. Otherwise, we say s* diverges*, denoted by $s\Uparrow$.*

Note that for every expression, there is at most one standard reduction possible. It is easy to see that reduction of expressions keeps the type of the expressions. Hence reduction will not lead to dynamic type errors:

Lemma 3.2 (Type Safety). *Reducing $t :: T$ by standard reduction leaves the term well-typed and does not change the type. I.e. $t \to t'$ implies that t' is well-typed and $t' :: T$.*

Lemma 3.3 (Progress Lemma). *A closed and well-typed expression without reduction is an answer.*

3.1 Assumptions on Valid Programs

Assumption 3.4. *We assume that for every type T of the program $\mathcal{P}$ there is at least one closed value of type T.*

Remark 3.5. This excludes types like the following: Let `Foo` by a type with one constructor `foo` : `Foo` → `Foo`. The only potentially closed expressions would be an infinitely nested expression `foo(foo(...))`, which of course does not exist. Hence there is no closed expression of type `Foo`.

Assumption 3.6. *We assume that every program $\mathcal{P}$ contains for every type τ a closed diverging expression, denoted as $\bot^\tau$.*

This could be achieved by defining f as $f = (\lambda x.f\ x) : \forall a, b.a \to b$, then the expression $(f\ v)^\tau$ does not converge, where v is any closed value.
The latter assumption e.g. allows to construct the value $\lambda x.\bot$, hence for every ground function type, there is a closed value. This assumption also would allow us to weaken Assumption 3.4.

3.2 Equivalence of Expressions

The conversion relation defined by the reductions (beta), (case) and (delta) in every context is too weak to justify sufficiently many equations. So we will observe termination in all contexts. For the definition of contextual equivalence, we will also need to take all program extensions into account.

Definition 3.7. *Given a program $\mathcal{P}$. Let s, t be two $\mathcal{P}$-expressions of (ground) type T. Then*
$s \leq_{\mathcal{P}\forall,T} t$ iff for all programs $\mathcal{P}'$ that extend $\mathcal{P}$, and all $\mathcal{P}'$-contexts $C[\cdot :: T]$: if $C[s], C[t]$ are closed, then $C[s]\downarrow \implies C[t]\downarrow$, and
$s \sim_{\mathcal{P}\forall,T} t$ iff $s \leq_{\mathcal{P}\forall,T} t$ and $t \leq_{\mathcal{P}\forall,T} s$.
If contexts $C[\cdot]$ are restricted to be $\mathcal{P}$-contexts, then we denote the relations as $\leq_{\mathcal{P},T}$ and $\sim_{\mathcal{P},T}$.

Lemma 3.8. *$\leq_{\mathcal{P},T}$ and $\leq_{\mathcal{P}\forall,T}$ are precongruences, and $\sim_{\mathcal{P},T}$ and $\sim_{\mathcal{P}\forall,T}$ are congruences.*

Proof. It is sufficient to show this for a fixed $\mathcal{P}$, and to prove the first two claims on precongruences. First we prove that $\leq_T$ is transitive. Let s, r, t be expressions of type T with $s \leq_T r \leq_T t$. Let $C[\cdot :: T]$ be a context such that $C[s], C[t]$ are closed and such that $C[s]\downarrow$. We have to show that $C[t]\downarrow$. Let $\{x_1, \ldots, x_n\} = FV(r) \setminus (FV(s) \cup FV(t))$. Let $D := (\lambda x_1, \ldots, x_n.C[\cdot])\ v_1 \ldots v_n$ be a context, where v_i are closed values of the same type as x_i for $i = 1, \ldots, n$. By our assumption on programs, for every type T, there exists at least one value. Obviously, $D[s]\downarrow$, since $D[s] \xrightarrow{*} C[s]$. Since $D[r]$ is closed, we also have $D[r]\downarrow$ and also $D[t]\downarrow$. Since reduction is deterministic and $D[t] \xrightarrow{*} C[t]$, we also obtain $C[t]\downarrow$.
Now we show that $\leq_T$ is compatible with contexts. Let s, t be expressions of

type T, and let $C[\cdot :: T] :: t'$ be a context. Now let D be any context, such that $D[C[s]]$ and $D[C[t]]$ are closed and $D[C[s]]\downarrow$. This implies $D[C[t]]\downarrow$, hence $C[s] \leq_{T'} C[t]$.
By standard arguments, this also holds for $\leq_{\mathcal{P}\forall,T}$.

Example 3.9. Note that in call-by-value calculi there is a difference between looking for termination in all contexts vs. termination in closing contexts.
The $\leq_{\mathcal{P},T}$-relation defined for closing contexts is different from the relation $\leq'_{\mathcal{P},T}$ defined for all contexts: Assume the usual definition of lists, and let $s = \texttt{Nil}, t = (\texttt{case}\ x\ \texttt{of Cons}\ y\ z\ \texttt{->}\ \texttt{Nil};\texttt{Nil}\ \texttt{->}\ \texttt{Nil})$. Then $s \not\leq'_{\mathcal{P},T} t$, since t does not converge: it is irreducible and not a value. However, it is not hard to verify, using induction on the number of reductions, that $s \sim_{\mathcal{P},T} t$ for our definition using closing contexts.

A program transformation $\mathcal{T}$ is a binary relation on $\mathcal{P}$-expressions, where $(s,t) \in \mathcal{T}$ always implies that s and t are of the same type. A program transformation $\mathcal{T}$ is *correct* iff for all $(s,t) \in \mathcal{T}$ of type T the relation $s \sim_{\mathcal{P},T} t$ holds. A program transformation $\mathcal{T}$ is *globally correct* iff for all $(s,t) \in \mathcal{T}$ of type T the relation $s \sim_{\mathcal{P}\forall,T} t$ holds.

4 Context Lemma for Programs

In order to prove a CIU-Lemma, we first have to prove a context lemma for L. extended with a `let`. In the following we assume that a fixed program $\mathcal{P}$ is given. We are interested in the contextual semantics of $\mathcal{P}$-expressions. However, we will also look for extensions $\mathcal{P}'$ of $\mathcal{P}$ and for the relation $\leq_{\mathcal{P}\forall,T}$.

4.1 Context Lemma for a Sharing Extension

We consider the let-language L_{let} that is an extension of our language that shares values using the expression syntax:

$$\begin{array}{lll} E & ::= V \mid F \mid (E\ E) \mid \lambda V.E \mid (c_i\ E_1 \ldots E_{ar(c_i)}) & \\ & \mid\ (\texttt{case}_K\ E\ Alt_1 \ldots Alt_n) & \text{where } n = |D_K| \\ & \mid\ (\texttt{let}\ V = W\ \texttt{in}\ E) & \end{array}$$

$$Alt_i ::= ((c_i\ V_1 \ldots V_{ar(c_i)})\ \texttt{->}\ E)$$

where W stands for values, i.e. $W ::= V \mid (c\ W_1\ \ldots W_N) \mid \lambda V.E$. The `let`-construct is non-recursive, i.e. the scope of x in $(\texttt{let}\ x = v\ \texttt{in}\ s)$ is only s.
Now we use a label-shift to determine the reduction contexts: With (lll) we denote the union of the rules (lapp), (lrapp), (lcapp), and (lcase).
The *type-constraints* for the `let`-construct are as follows: in $(\texttt{let}\ x = v\ \texttt{in}\ s)$, the type labels of x, v must be identical, and the type label of s is the same as for the let-expression, i.e. only $(\texttt{let}\ x :: T_1 = v :: T_1\ \texttt{in}\ s :: T_2) :: T_2$ is a correct typing.

$$\begin{array}{ll}
(s\ t)^{\mathsf{sub}\vee\mathsf{lr}} & \to (s^{\mathsf{sub}}\ t) \quad \text{if } s \text{ is not a value} \\
(v^{\mathsf{sub}}\ s) & \to (v\ s^{\mathsf{sub}}) \\
(c\ s_1 \dots s_n)^{\mathsf{sub}\vee\mathsf{lr}} & \to (c\ s_1^{\mathsf{sub}} \dots s_n) \\
(c\ v_1 \dots v_i^{\mathsf{sub}}\ s_{i+1} \dots s_n) & \to (c\ v_1 \dots v_i\ s_{i+1}^{\mathsf{sub}} \dots s_n) \\
(\mathtt{case}\ s\ alts)^{\mathsf{sub}\vee\mathsf{lr}} & \to (\mathtt{case}\ s^{\mathsf{sub}}\ alts) \\
(\mathtt{let}\ x = v\ \mathtt{in}\ s)^{\mathsf{lr}} & \to (\mathtt{let}\ x = v\ \mathtt{in}\ s^{\mathsf{lr}})
\end{array}$$

Shifting starts with t^{lr}, where t has no other occurrences of labels $\mathsf{sub}, \mathsf{lr}$.
We assume that the label is not removed during the label shift; In the rules above, only the new label is shown.

Fig. 3. Searching the redex in the let-language L_{let}

(beta$_{\mathtt{let}}$) $C[((\lambda x.s)^{\mathsf{sub}}\ v)] \to C[\mathtt{let}\ x = v\ \mathtt{in}\ s]$

(delta$_{\mathtt{let}}$) $C[f^{\mathsf{sub}} :: T] \to C[d_f]$ if $f = d_f :: T'$ for the function symbol f.
The reduction is accompanied by a type instantiation
$\rho(d_f)$, where $\rho(T') = T$

(case$_{\mathtt{let}}$) $C[(\mathtt{case}\ (c\ v_1 \dots v_n)^{\mathsf{sub}}\ \dots ((c\ y_1 \dots y_n)\ \texttt{->}\ s) \dots)]$
$\to C[\mathtt{let}\ y_1 = v_1\ \mathtt{in}\ \dots \mathtt{let}\ y_n = v_n\ \mathtt{in}\ s]$

(cp) $C[\mathtt{let}\ x = v\ \mathtt{in}\ C'[x^{\mathsf{sub}}]] \to C[\mathtt{let}\ x = v\ \mathtt{in}\ C'[v]]$

(lapp) $C[((\mathtt{let}\ x = v\ \mathtt{in}\ s)^{\mathsf{sub}}\ t)] \to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (s\ t))]$

(lrapp) $C[(v_1\ (\mathtt{let}\ x = v\ \mathtt{in}\ t)^{\mathsf{sub}})] \to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (v_1\ t))]$

(lcapp) $C[(c\ v_1 \dots v_{i-1}\ (\mathtt{let}\ x = v\ \mathtt{in}\ s_i)^{\mathsf{sub}}\ s_{i+1} \dots s_n)]$
$\to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (c\ v_1 \dots v_{i-1}\ s_i \dots s_n))]$

(lcase) $C[(\mathtt{case}\ (\mathtt{let}\ x = v\ \mathtt{in}\ s)^{\mathsf{sub}}\ alts)]$
$\to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (\mathtt{case}\ s\ alts))]$

Fig. 4. Standard Reduction rules in the let-language L_{let}

We denote a reduction as $t \xrightarrow{ls} t'$ (standard-let-reduction), and write $t \xrightarrow{ls,a} t'$ if we want to indicate the kind a of the reduction.

Values are expressions x, $\lambda x.s$, or $(c\ s_1 \dots\ s_n)$, where s_i are variables or values. The *answers* of reductions are values but not variables that may be embedded in lets. I.e., expressions of the form $(\mathtt{let}\ x_1 = v_1\ \mathtt{in}\ (\mathtt{let}\ x_2 = v_2\ \mathtt{in}\ \dots (\mathtt{let}\ x_n = v_n\ \mathtt{in}\ v) \dots))$ where v is a value, but not a variable. We say an expression t *converges*, denoted as $t\downarrow$ iff there is a reduction $t \xrightarrow{ls,*} t'$, where t' is an answer. The contexts C that we allow in the language may have their holes at the usual positions where an expression is permitted; if it is in v of $(\mathtt{let}\ x = v\ \mathtt{in}\ t)$, then the hole must be within an abstraction of v. Contextual approximation and contextual equivalence for $L_{\mathtt{let}}$ are defined accordingly, where we use the symbols $\leq_{\mathtt{let},T}$ and $\sim_{\mathtt{let},T}$ for the corresponding relations. Now we can show the context lemma for L_{let} :

A *reduction context* $R[\cdot]$ for L_{let} is a context, where the sub-shifting will end successfully at the hole. Note that the hole cannot occur as $(\mathtt{let}\ x = [\cdot]\ \mathtt{in}\ t)$.

4.2 Context Lemmas in the let-Language

For a reduction sequence RED the function $\mathrm{rl}(RED)$ computes the length of the reduction sequence RED.

Definition 4.1. *For well-typed expressions $s, t :: T$, the relation $s \leq_{\mathtt{let},R,T} t$ holds iff for all ρ where ρ is a variable-permutation such that variables are renamed, the following holds: $\forall R[\cdot :: \tau]$: if $R[\rho(s)], R[\rho(t)]$ are closed, then $(R[\rho(s)]\downarrow \Longrightarrow R[\rho(t)]\downarrow))$*

We require the notion of *multicontexts*, i.e. expressions with several (or no) typed holes $\cdot_i :: T_i$, where every hole occurs exactly once in the expression. We write a multicontext as $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$, and if the expressions $s_i :: T_i$ for $i = 1, \ldots, n$ are placed into the holes $\cdot_i$, then we denote the resulting expression as $C[s_1, \ldots, s_n]$.

Lemma 4.2. *Let C be a multicontext with n holes. Then the following holds: If there are expressions $s_i :: T_i$ with $i \in \{1, \ldots, n\}$ such that $C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ is a reduction context, then there exists a hole $\cdot_j$, such that for all expressions $t_1 :: T_1, \ldots, t_n :: T_n$ $C[t_1, \ldots, t_{j-1}, \cdot_j :: T_j, t_{j+1}, \ldots, t_n]$ is a reduction context.*

Proof. Let us assume there is a multicontext C with n holes and there are expressions $s_1, \ldots, s_n$ such that $C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ is a reduction context. Applying the labeling algorithm to the multi-context C alone will hit hole number j, perhaps with $i \neq j$. Then $C[t_1, \ldots, t_{j-1}, \cdot_j :: T_j, t_{j+1}, \ldots, t_n]$ is a reduction context for any expressions t_i.

Lemma 4.3 (Context Lemma). *The following holds:*
$\leq_{\mathtt{let},R,T} \subseteq \leq_{\mathtt{let},T}$

Proof. We prove a more general claim:
For all $n \geq 0$ and for all multicontexts $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$ and for all well-typed expressions $s_1 :: T_1, ..., s_n :: T_n$ and $t_1 :: T_1, ..., t_n :: T_n$:
If for all $i = 1, \ldots, n$: $s_i \leq_{\mathtt{let},R,T} t_i$, and if $C[s_1, \ldots, s_n]$ and $C[t_1, \ldots, t_n]$ are closed, then $C[s_1, \ldots, s_n]\downarrow \quad \Longrightarrow \quad C[t_1, \ldots, t_n]\downarrow$.
The proof is by induction, where n, $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$, $s_i :: T_i, t_i :: T_i$ for $i = 1, \ldots, n$ are given. The induction is on the measure (l, n), where

- l is the length of the evaluation of $C[s_1, \ldots, s_n]$.
- n is the number of holes in C.

We assume that the pairs are ordered lexicographically, thus this measure is well-founded. The claim holds for $n = 0$, i.e., all pairs $(l, 0)$, since if C has no holes there is nothing to show.
Now let $(l, n) > (0, 0)$. For the induction step we assume that the claim holds for all n', C', s'_i, t'_i, $i = 1, \ldots, n'$ with $(l', n') < (l, n)$. Let us assume that the precondition holds, i.e., that $\forall i : s_i \leq_{\mathtt{let},R,T} t_i$. Let C be a multicontext and RED be the evaluation of $C[s_1, \ldots, s_n]$ with $\mathrm{rl}(RED) = l$. For proving $C[t_1, \ldots, t_n]\downarrow$, we distinguish two cases:

- There is some index j, such that $C[s_1, \ldots, s_{j-1}, \cdot_j :: T_j, s_{j+1}, \ldots, s_n]$ is a reduction context. Lemma 4.2 implies that there is a hole $\cdot_i$ such that $R_1 \equiv C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ and $R_2 \equiv C[t_1, \ldots, t_{i-1}, \cdot_i :: T_i, t_{i+1}, \ldots, t_n]$ are both reduction contexts. Let $C_1 \equiv C[\cdot_1 :: T_1, \ldots, \cdot_{i-1} :: T_{i-1}, s_i, \cdot_{i+1} :: T_{i+1}, \ldots, \cdot_n :: T_n]$. From $C[s_1, \ldots, s_n] \equiv C_1[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n]$ we derive that RED is the evaluation of $C_1[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n]$. Since C_1 has $n-1$ holes, we can use the induction hypothesis and derive $C_1[t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n] \downarrow$, i.e. $C[t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots, t_n] \downarrow$. This implies $R_2[s_i] \downarrow$. Using the precondition we derive $R_2[t_i] \downarrow$, i.e. $C[t_1, \ldots, t_n] \downarrow$.
- There is no index j, such that $C[s_1, \ldots, s_{j-1}, \cdot_j :: T_j, s_{j+1}, \ldots, s_n]$ is a reduction context. If $l = 0$, then $C[s_1, \ldots, s_n]$ is an answer and since no hole is in a reduction context, $C[t_1, \ldots, t_n]$ is also an answer, hence $C[t_1, \ldots, t_n] \downarrow$. If $l > 0$, then the first normal order reduction of RED can also be used for $C[t_1, \ldots, t_n]$. This normal order reduction can modify the context C, the number of occurrences of the expressions s_i, the positions of the expressions s_i, and s_i may be renamed by a (cp) reduction.

  We now argue that the elimination, duplication or variable permutation for every s_i can also be applied to t_i. More formally, we will show if $C[s_1, \ldots, s_n] \xrightarrow{ls,a} C'[s'_1, \ldots, s'_m]$, then $C[t_1, \ldots, t_n] \xrightarrow{ls,a} C'[t'_1, \ldots, t'_m]$, such that $s'_i \leq_{T',\downarrow,R} t'_i$. We go through the cases of which reduction step is applied to $C[s_1, \ldots, s_n]$ to figure out how the expressions s_i (and t_i) are modified by the reduction step, where we only mention the interesting cases.

  - For a (lapp), (lrapp), (lcapp), (lcase), and (beta$_{\texttt{let}}$) reduction, the holes $\cdot_i$ may change their position.
  - For a (case$_{\texttt{let}}$) reduction, the position of $\cdot_i$ may be changed as in the previous item, or if the position of $\cdot_i$ is in an alternative of `case`, which is discarded by a (case)-reduction, then s_i and t_i are both eliminated.
  - If the reduction is a (cp) reduction and there are some holes $\cdot_i$ inside the copied value, then there are variable permutations $\rho_{i,1}, \rho_{i,2}$ with $s'_i = \rho_{i,1}(s_i)$ and $t'_i = \rho_{i,2}(t_i)$. One can verify that we may assume that $\rho_{i,1} = \rho_{i,2}$ for all i. Now the precondition implies $s'_i \leq_{\texttt{let},R,T} t'_i$.
  - If the standard reduction is a (delta$_{\texttt{let}}$)-reduction, then s_i, t_i cannot be influenced, since within d_f, there are no holes.

  Now we can use the induction hypothesis: Since $C'[s'_1, \ldots, s'_m]$ has a terminating sequence of standard reductions of length $l-1$ we also have $C'[t'_1, \ldots, t'_m] \downarrow$. With $C[t_1, \ldots, t_n] \xrightarrow{ls,a} C'[t'_1, \ldots, t'_m]$ we have $C[t_1, \ldots, t_n] \downarrow$.

4.3 The CIU-Theorem

Now we use the context lemma for the let-language L_{let} and transfer the results to our language L using the method in [SSNSS08]. Let Φ be the translation from L to $L_{\texttt{let}}$ defined as the identity, that translates expressions, contexts and types. This translation is obviously compositional, i.e. $\Phi(C[s]) = \Phi(C)[\Phi(s)]$. We also define a backtranslation $\overline{\Phi}$ from $L_{\texttt{let}}$ into L. The translation is defined as

$\overline{\Phi}(\texttt{let}\ x = v\ \texttt{in}\ s) := \overline{\Phi}(s)[\overline{\Phi}(v)/x]$ for `let`-expressions and homomorphic for all other language constructs. The types are translated in the obvious manner. For extending $\overline{\Phi}$ to contexts, the range of $\overline{\Phi}$ does not consist only of contexts, but of contexts plus a substitution which "affects" the hole, i.e. for a context C, $\overline{\Phi}(C)$ is $C'[\sigma[]]$ where $C' = \overline{\Phi}'(C)$ where $\overline{\Phi}'$ treats contexts like expressions (and the context hole is treated like a constant).
With this definition $\overline{\Phi}$ satisfies compositionality, i.e. $\overline{\Phi}(C)[\overline{\Phi}(s)] = \overline{\Phi}(C[s])$ holds. The difference to the usual notion is that $\overline{\Phi}(C)$ is not a context, but a function mapping expressions to expressions.
The important property to be proved for the translations is convergence equivalence, i.e. $t\downarrow \iff \Phi(t)\downarrow$, and $t\downarrow \iff \overline{\Phi}(t)\downarrow$, resp.
By inspecting the (ls,lll)- and (ls,cp)-reductions and the Definition of $\overline{\Phi}$ the following properties are easy to verify:

Lemma 4.4. *Let $t \in L_{\texttt{let}}$ and $t \xrightarrow{ls,lll} t'$ or $t \xrightarrow{ls,cp} t'$. Then $\overline{\Phi}(t') = \overline{\Phi}(t)$. Furthermore, all reduction sequences consisting only of of $\xrightarrow{ls,lll}$ and $\xrightarrow{ls,cp}$ are finite.*

Lemma 4.5. *Let t be a expression of $L_{\texttt{let}}$ such that $\overline{\Phi}(t) = R[s]$, where (ls,cp)- and (ls,lll)-reductions are not applicable to t, and R is a reduction context. Let t be represented as $t = \texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t_1$ where t_1 is not a `let`-expression. Then there is some reduction context R' and a expression s', such that $t_1 = R'[s']$, $R = \overline{\Phi}(\sigma(R'))$, $s = \overline{\Phi}(\sigma(s'))$ and $R[s] = \overline{\Phi}(\sigma(R'[s']))$, where $\sigma = \{x_1 \mapsto s_1\} \circ \ldots \circ \{x_n \mapsto s_n\}$. Furthermore, $\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'$ is a reduction context in $L_{\texttt{let}}$.*

Proof. It is easy to see that there exists a context R' and an expression s', such that $R = \overline{\Phi}(\sigma(R'))$ and $s = \overline{\Phi}(\sigma(s'))$. We have to show that R' is a reduction context of $L_{\texttt{let}}$. Let M be a multicontext such that $R' = M[r_1, \ldots, \cdot, \ldots, r_k]$ such that r_i are all the maximal subexpressions in non-reduction position of R'. Since neither let-shifting nor copy reductions are applicable to t, we have that $\overline{\Phi}(\sigma(R')) = R = M[\overline{\Phi}(\sigma(r_1), \ldots, \cdot, \ldots, \overline{\Phi}(\sigma(r_k)]$. Since the hole in R is in reduction position, this also holds for R', i.e. R' is a reduction context. By the construction of reduction contexts in $L_{\texttt{let}}$ it is easy to verify that $\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[]$ is also a reduction context.

Lemma 4.6. *Let t be a $L_{\texttt{let}}$ expression such that no (ls,lll)-, or (ls,cp)-reductions are applicable to t. If $\overline{\Phi}(t) \to s$ then there exists some t' such that $t \to t'$ and $\overline{\Phi}(t') = s$.*

Proof. Since neither (ls,lll)- nor (ls,cp)-reductions are applicable to t, the expression t is either a non-let expression t_1 or of the form $\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t_1$ where t_1 is a non-let expression. Let $\sigma = \{x_1 \mapsto s_1\} \circ \ldots \circ \{x_n \mapsto s_n\}$ in the following.
We treat the (beta)-reduction in detail, and omit the details for (case)- and (delta)-reductions, since the proofs are completely analogous. Hence, let $\overline{\Phi}(t) \to s$ by a (beta)-reduction. I.e., $\overline{\Phi}(t) = R[(\lambda x.r)\ v] \to R[r[v/x]] = s$. Then there exists

a context R' and expressions r_0, v_0, such that $R = \overline{\Phi}(\sigma(R'))$, $r = \overline{\Phi}(\sigma(r_0))$, $v = \overline{\Phi}(\sigma(v_0))$. Since no (ls,cp)- and (ls,lll)- reductions are applicable to t we also have that $t = \texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[(\lambda x.r_0)\ v_0]$. Lemma 4.5 shows that $\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[]$ is a reduction context of $L_{\texttt{let}}$. The expression v_0 must be a value, since v is a value and no (ls,lll)- and no (ls,cp)-reductions are applicable to t.
Hence, we can apply a (beta$_{\texttt{let}}$)-reduction to t: $\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[(\lambda x.r_0)\ v_0] \xrightarrow{ls,beta_{\texttt{let}}} \texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[\texttt{let}\ x = v_0\ \texttt{in}\ r_0]$. Now it is easy to verify that $\overline{\Phi}(t') = s$ holds.

Lemma 4.7. *The following properties hold:*

1. *For all $t \in L_{\texttt{let}}$: if t is an answer, then $\overline{\Phi}(t)$ is an answer for L, and if $\overline{\Phi}(t)$ is an answer, then $t \xrightarrow{ls,*} t'$ where t' is an answer for $L_{\texttt{let}}$*
2. *For all $t \in L$: t is an answer iff $\Phi(t)$ is an answer for $L_{\texttt{let}}$.*
3. *Let $t_1, t_2 \in L_{\texttt{let}}$ with $t_1 \xrightarrow{ls} t_2$. Then either $\overline{\Phi}(t_1) = \overline{\Phi}(t_2)$ or $\overline{\Phi}(t_1) \rightarrow \overline{\Phi}(t_2)$*
4. *Let $t_1 \in L_{\texttt{let}}$ with $\overline{\Phi}(t_1) \rightarrow t_2'$. Then $t_1 \xrightarrow{ls,+} t_2$ with $\overline{\Phi}(t_2) = t_2'$.*

Proof. Part 1 and 2 follow by definition of answers in L and $L_{\texttt{let}}$ and the definitions of Φ, $\overline{\Phi}$. Note that it may be possible that $\overline{\Phi}(t)$ is an answer, but for t some (ls,lll)- or (ls,cp)- reductions are necessary to obtain an answer in $L_{\texttt{let}}$.
3: If the reduction is a (ls,lll) or (ls,cp), then $\overline{\Phi}(t_1) = \overline{\Phi}(t_2)$. If the reduction is a (beta$_{\texttt{let}}$), (delta$_{\texttt{let}}$), or (case$_{\texttt{let}}$), then $\overline{\Phi}(t_1) \rightarrow \overline{\Phi}(t_2)$ by the reduction with the same name. Part 4 follows from Lemma 4.4 and 4.6.

Lemma 4.8. *Φ and $\overline{\Phi}$ are convergence equivalent.*

Proof. We have to show four parts:

- $t\downarrow \implies \overline{\Phi}(t)\downarrow$: This follows by induction on the length of the evaluation of t. The base case is shown in Lemma 4.7, part 1. The induction step follows by Lemma 4.7, part 3.
- $\overline{\Phi}(t)\downarrow \implies t\downarrow$: We use induction on the length of the evaluation of $\overline{\Phi}(t)$. For the base case Lemma 4.7, part 1 shows that if $\overline{\Phi}(t)$ is an answer, then $t\downarrow$. For the induction step let $\overline{\Phi}(t) \rightarrow t'$ such that $t'\downarrow$. Lemma 4.7, part 4 shows that $t \xrightarrow{ls,+} t''$, such that $\overline{\Phi}(t'') = t'$. The induction hypothesis implies that $t''\downarrow$ and thus $t\downarrow$.
- $t\downarrow \implies \Phi(t)\downarrow$: This follows by induction on the length of the evaluation of t. The base case follows from Lemma 4.7, part 2. For the induction step let $t \xrightarrow{a} t'$, where $t'\downarrow$ and $a \in \{(\text{beta}), (\text{delta}), (\text{case})\}$. If $a = (\text{delta})$ then $\Phi(t) \xrightarrow{ls,delta_{\texttt{let}}} \Phi(t')$, and hence the induction hypothesis shows $\Phi(t')\downarrow$ and thus $\Phi(t)\downarrow$. For the other two cases we have $\Phi(t) \xrightarrow{ls,a} t''$, with $\overline{\Phi}(t'') = t'$. The second part of this proof shows that $t'\downarrow$ implies $t''\downarrow$. Hence, $\Phi(t)\downarrow$.
- $\Phi(t)\downarrow \implies t\downarrow$: This follows, since the first part of this proof shows $\Phi(t)\downarrow$ implies $\overline{\Phi}(\Phi(t))\downarrow$, and since $\overline{\Phi}(\Phi(t)) = t$.

The framework in [SSNSS08] shows that convergence equivalence and compositionality of Φ imply adequacy, i.e.:

Corollary 4.9 (Adequacy of Φ). $\Phi(s) \leq_{\mathtt{let},T} \Phi(t) \implies s \leq_T t$.

Lemma 4.10 (CIU-Lemma). *Let $s, t :: T$ be two expressions of L such that for all value substitutions σ and for all reduction contexts R, such that $R[\sigma(s)], R[\sigma(t)]$ are closed, the implication $R[\sigma(s)] \downarrow \implies R[\sigma(t)] \downarrow$ is valid. Then $s \leq_T t$ holds.*

Proof. Let $R[\sigma(s)] \downarrow \implies R[\sigma(t)] \downarrow$ hold for all value substitutions σ and reduction contexts R, such that $R[\sigma(s)], R[\sigma(t)]$ are closed. We show that $\Phi(s) \leq_{\mathtt{let},R,T} \Phi(t)$ holds. Then the context lemma 4.3 shows that $\Phi(s) \leq_{\mathtt{let},T} \Phi(t)$ and the previous corollary implies $s \leq_T t$.
Let R_{let} be a reduction context in L_{let} such that $R_{let}[\Phi(s)]$ and $R_{let}[\Phi(t)]$ are closed and $R_{let}[\Phi(s)] \downarrow$. We extend the translation $\overline{\Phi}$ to reduction contexts: For reduction contexts R_{let} that are not a `let`-expression, $\overline{\Phi}(R_{let})$ is defined analogous to the translation of expressions. For $R_{let} = \mathtt{let}\ x_1 = v_1\ \mathtt{in}\ (\mathtt{let}\ x_2 = v_2\ \mathtt{in}\ (\ldots(\mathtt{let}\ x_n = v_n\ \mathtt{in}\ R'_{let}))))$ where R'_{let} is not a `let`-expression we define $\overline{\Phi}(R_{let}) = \overline{\Phi}(R'_{let})[\sigma(\cdot)]$, where $\sigma := \sigma_n$ is the substitution defined inductively by $\sigma_1 = \{x_1 \mapsto v_1\}, \sigma_i = \sigma_{i-1} \circ \{x_i \mapsto v_i\}$.
Since $R_{let}[\Phi(s)] \downarrow$ and $\overline{\Phi}(R_{let}[\Phi(s)]) = R'[\sigma(\overline{\Phi}(\Phi(s)))] = R'[\sigma(s)]$ where R' is a reduction context for L and σ is a value substitution, convergence equivalence of $\overline{\Phi}$ shows $R'[\sigma(s)] \downarrow$. Since $R'[\sigma(s)]$ and $R'[\sigma(t)]$ are closed, the precondition of the lemma now implies $R'[\sigma(t)] \downarrow$. Since $R'[\sigma(t)] = R'[\sigma(\overline{\Phi}(\Phi(t)))] = \overline{\Phi}(R_{let}[\Phi(t)])$ and since $\overline{\Phi}$ is convergence equivalent, we have $R[\Phi(t)] \downarrow$.

Proposition 4.11. *The transformation (beta), (delta), and (case) are correct program transformations in L.*

Proof. We use the CIU-Lemma 4.10: Let $a \in \{(\text{beta}), (\text{delta}), (\text{case})\}$. Let $s \xrightarrow{a} t$, R be a reduction context, and σ be a value substitution, such that $R[\sigma(s)]$ is closed. If $R[\sigma(t)] \downarrow$, then $R[\sigma(s)] \xrightarrow{a} R[\sigma(t)]$ by a standard reduction, and thus $R[\sigma(s)] \downarrow$.
For the other direction let $R[\sigma(s)] \downarrow$, i.e. $R[\sigma(s)] \rightarrow t_1 \xrightarrow{*} t_n$ where t_n is an answer. Since standard reduction is unique one can verify that then $R[\sigma(s)] \xrightarrow{a} R[\sigma(t)] = t_1$ must hold, i.e. $R[\sigma(t)] \downarrow$.

Note that ordinary (i.e. call-by-name) beta-reduction may be incorrect, for example $(\lambda x.\mathtt{True})$ `Bot` is equivalent to `Bot` :: `Bool`, however, using a call-by-name beta-reduction results in `True`, which is obviously not equivalent to `Bot`.

Theorem 4.12 (CIU-Theorem). *For $\mathcal{P}$-expressions $s, t :: \tau$: $R[\sigma(s)] \downarrow \implies R[\sigma(t)] \downarrow$ for all $\mathcal{P}$-value substitutions σ and $\mathcal{P}$-reduction contexts R where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P},T} t$ holds.*

Proof. One direction is the CIU-Lemma 4.10. For the other direction, let $s \leq_T t$ hold and $R[\sigma(s)] \downarrow$ for a value substitution $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, where $\sigma(s), \sigma(t)$ are closed, and let R be a reduction context. Since (beta) is a correct program transformation, we have $R[(\lambda x_1.\ldots.x_n.s)\ v_1\ \ldots\ v_n] \sim_T R[\sigma(s)]$. Thus, $R[(\lambda x_1.\ldots.x_n.s)\ v_1\ \ldots\ v_n]\downarrow$ and applying $s \leq_T t$ we derive $R[(\lambda x_1.\ldots.x_n.t)\ v_1\ \ldots\ v_n]\downarrow$. Using correctness of (beta) once more shows $R[\sigma(t)]\downarrow$.

Applied to extensions $\mathcal{P}'$ of $\mathcal{P}$, we obtain the following corollary:

Corollary 4.13. *Let $\mathcal{P}$ be a program. For $\mathcal{P}$-expressions $s, t :: \tau$: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ for all extensions $\mathcal{P}'$ of $\mathcal{P}$ and all $\mathcal{P}'$-value substitutions σ and $\mathcal{P}'$-reduction contexts R where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P}\forall,T} t$ holds.*

4.4 Local CIU-Theorems

In this subsection the CIU-theorem can be made stronger by restricting R and σ to be free of function symbols from $\mathcal{F}$.

Let an *F-free expression*, value, or context be an expression, value, or context that is built over the language without function-symbols, but where $\bot$-symbols of every type are allowed according to Assumption 3.6.

We will use the lambda-depth-measure for subexpression-occurrences s of some expression t: it is the number of lambda's and pattern-alternatives that are crossed by the position of the subexpression.

Lemma 4.14 (CIU-Lemma F-free). *Let $s, t :: T$ be two expressions of L such that for all F-free value substitutions σ and all F-free reductions contexts R such that $R[\sigma(s)], R[\sigma(t)]$ are closed: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$. Then $s \leq_T t$ holds.*

Proof. We show that the condition of this lemma implies the precondition of the CIU-lemma.

Let $s, t :: T$ be two expressions of L such that for all F-free value substitutions σ and all F-free reductions contexts R where $R[\sigma(s)], R[\sigma(t)]$ are closed: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$. Let R be any reduction context and σ be any value substitution such that $R[\sigma(s)], R[\sigma(t)]$ are closed, and assume $R[\sigma(s)]\downarrow$. Let n be the number of reductions of $R[\sigma(s)]$ to an answer. We construct F-free reduction contexts R' and F-free value substitutions σ' as follows: apply $n+1$ times a delta-step for every occurrence of function symbol in R and σ. As a last step, replace every remaining function symbol by $\bot$ of the appropriate type. Note that a single reduction step can shift the bot-symbols at most one lambda-level higher. By standard reasoning and induction, we obtain that $R'[\sigma'(s)]\downarrow$, by using the reduction sequence of $R[\sigma(s)]$ also for $R'[\sigma'(s)]$, where the induction is by the number of reduction steps. The assumption now implies that $R'[\sigma'(t)]\downarrow$, We have $R'[\sigma'(t)] \leq_T R[\sigma(t)]$, since delta-reduction is correct and the insertion of $\bot$ makes the expression smaller w.r.t. $\leq_c$. Hence $R[\sigma(t)]\downarrow$. Then we can use the CIU-Theorem 4.12.

Theorem 4.15 (CIU-Theorem F-free). *For $s, t :: \tau \in L$: $R[\sigma(s)] \downarrow \implies R[\sigma(t)] \downarrow$ for all F-free value substitutions σ and F-free reduction contexts R, where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_\tau t$ holds.*

Proof. One direction is the F-free CIU-Lemma 4.14. The other direction is the same as in the proof of the CIU-theorem.

Corollary 4.16. *Let $s, t :: \tau \in L$. If for all closing F-free value substitutions σ, we have $\sigma(s) \leq_\tau \sigma(t)$, then $s \leq_\tau t$.*

Proof. Follows from the F-free CIU-theorem 4.15.

Corollary 4.17. *Let $s, t :: \tau \in L$. If for all closing F-free value substitutions σ, $\sigma(s)$ and $\sigma(t)$ reduce to the same value using standard-reduction, then $s \sim_\tau t$.*

Proof. Follows from the F-free CIU-theorem 4.15, since reduction of $R[\sigma(s)]$ (respectively $R[\sigma(t)]$ first evaluates the expressions $\sigma(s)$ (respectively $\sigma(t)$).

Note that adequacy of the translation $\overline{\Phi}$ could not be derived as in Corollary 4.9), since $\overline{\Phi}$ is not compositional in the usual sense: the image of a context may be a context together with a substitution for the hole. In the proof below we will use a custom-tailored variant of compositionality.

Theorem 4.18 (Adequacy of $\overline{\Phi}$). $\overline{\Phi}(s) \leq_T \overline{\Phi}(t) \implies s \leq_{\texttt{let},T} t$.

Proof. We use the framework in [SSNSS08,SSNSS09] that shows that convergence equivalence and compositionality of $\overline{\Phi}$ imply adequacy. It is easy to see that $\overline{\Phi}(C[s]) \sim_T \overline{\Phi}(C)[\overline{\Phi}(s)])$, if we admit that $\overline{\Phi}(C) = C'[\sigma(\cdot)]$, where $C' = \overline{\Phi}(C)$ and the hole is considered a constant, and σ is the substitution that is derived from all the let-bindings that have the hole in their scope. Thus adequacy implies that $\overline{\Phi}(s) \leq'_T \overline{\Phi}(t) \implies s \leq_{\texttt{let},T} t$, where $\leq'_T$ is defined using all observers $D[\sigma(\cdot)]$ and using also the closedness condition. However, since (beta) is correct in L by Proposition 4.11, the relation $\leq'_T$ is the same as $\leq_T$, since $D[\sigma(r)] \sim_T D[\lambda x_1, \ldots x_n.[r]) \ v_1 \ldots v_n]$ for all expressions r of the appropriate type.

4.5 Properties of Ω-Expressions

Definition 4.19. *We say an expression s is an* Ω-expression *iff for all value substitutions σ where $\sigma(s)$ is closed, $\sigma(s)\Uparrow$ holds. The symbol* `Bot`*, labeled with a type, is used as a representative (i.e. a meta-symbol) for any Ω-expression of the corresponding type.*

We can show that the property of being an Ω-expression inherits to reduction contexts:

Proposition 4.20. *Let $s :: \tau$ be an Ω-expression. Then for every reduction context $R[\cdot :: \tau]$, the expression $R[s]$ is an Ω-expression.*

Proof. This follows by structural induction of R. If R is the empty context then the claim obviously holds. For the induction step there exists a context R_1 with $R = R_1[([\cdot]\ t)]$, $R = R_1[(v\ [\cdot])]$, $R = R_1[(\texttt{case}\ [\cdot]\ \mathit{alts})]$, or $R = R_1[(c\ v_1\ \dots\ v_i\ [\cdot]\ s_{i+1}\ \dots s_n)]$.
It is easy to verify that for any closing value substitution σ the expression $\sigma(s\ t)$, $\sigma(v\ s)$, $\sigma(\texttt{case}\ s\ \mathit{alts})$, or $\sigma(c\ v_1\ \dots\ v_i\ s\ s_{i+1}\ \dots\ s_n)$, respectively, cannot be evaluated to an answer, since $\sigma(s){\Uparrow}$. Hence, $(s\ t)$, $(v\ s)$, $(\texttt{case}\ s\ \mathit{alts})$, or $(c\ v_1\ \dots\ v_i\ s\ s_{i+1}\ \dots\ s_n)$, respectively, is an Ω-expression. Thus, the induction hypothesis can be applied to R_1 which shows that $R[s]$ is an Ω-expression.

Corollary 4.21. *Let $s, t :: \tau$ and let s be an Ω-expression. Then $s \leq_\tau t$. If also t is an Ω-expressions, then $s \sim_\tau t$.*

Proof. We only prove $s \leq_\tau t$, since the other direction is symmetric. We use the CIU-Theorem 4.12: Let R be a reduction context, σ be a value substitution such that $\sigma(s), \sigma(t)$ are closed. Then $\sigma(s)$ must be an Ω-expression, and by Proposition 4.20 $R[\sigma(s)]$ is an Ω-expression, too. Thus $R[\sigma(s)]{\Uparrow}$, and $s \leq_\tau t$ holds. The second claim follows by symmetry.

5 Recognizing Equality of Expressions

This section proves criteria for equality of expressions that are easier to use than the definition of contextual equality. In particular, it is shown that equality is conservative w.r.t. extending programs. Later we will also show that applicative simulation methods can be applied using Howe's proof technique ([How89]). We say an expression or a context is *F-free* if only the symbol $\bot$, but no further function symbols from $\mathcal{F}$ occur.

Definition 5.1. *Let T be a constructed type. We say T is a* singleton type*, iff for all closed values v_1, v_2 of type T, the relation $v_1 \sim v_2$ holds.*

Lemma 5.2. *Let x, y be different variables of type T. Then $x \sim y$ iff T is a singleton type.*

Proof. If T is a function type $T_1 \to T_2$, then there is an abstraction $\lambda x.\bot_{T_2}$, as well as an abstraction $\lambda x.v_{T_2}$, where v_{T_2} is a value of type T_2, and we can distinguish the variables x, y using the CIU-theorem. If T is a constructed type, and there are two closed values $v_1 \not\sim v_2$ of this type, then we can distinguish the variables using $\sigma_1 = \{x \mapsto v_1, y \mapsto v_2\}$ and an appropriate context R_1, and $\sigma_1 = \{x \mapsto v_2, y \mapsto v_1\}$ and an appropriate context R_2.

Lemma 5.3. *Let s, t be (open) expressions of type T. Then $s \leq_T t$ iff for all closing F-free value-substitutions σ: $\sigma(s) \leq_T \sigma(t)$.*

Proof. If $s \leq_T t$, then $\sigma(s) \leq_T \sigma(t)$ for closing value substitutions follows, since beta-reduction is correct. The converse follows from the assumption and the CIU-Theorem.

Thus, it is sufficient to check equality of closed expressions.

Lemma 5.4. *Let s, t be closed expressions of constructed type T. Then $s \leq_T t$ iff $s\Uparrow$ or $s \xrightarrow{*} c\, v_1 \dots v_n$ and $t \xrightarrow{*} c\, w_1 \dots w_n$ for some constructor c, and $v_i \leq_{\mathcal{P},T_i} w_i$ for $i = 1, \dots, n$.*

Proof. If $s \leq_{\mathcal{P},T} t$, then either $s\Uparrow$, or $s\downarrow, t\downarrow$. Since T is a constructed type, the result is an answer with constructor of type T. Using case-expressions and the correctness of (case)-reductions, the claim follows. The other direction holds, since $\leq_{\mathcal{P},T}$ is a pre-congruence and due to Corollary 4.21.

Proposition 5.5. *Let s, t be closed expressions of function type T. Then $s \leq_T t$ iff $s\Uparrow$ or $s \xrightarrow{*} \lambda x.s'$ and $t \xrightarrow{*} \lambda x.t'$ and $s'[v/x] \leq_T t'[v/x]$ for all closed F-free values v.*

Proof. If $s \leq_T t$, then either $s\Uparrow$, or $s\downarrow, t\downarrow$. Since T is a function type, the results must be abstractions. The conclusion follows since $\leq_T$ is a congruence.
The other direction also holds, using Corollary 4.16 which implies $s' \leq_T t'$. Then we can use the pre-congruence property.

Now we show that function symbols from $\mathcal{F}$ are not necessary in the contexts to define the contextual ordering:

Definition 5.6. *Let $\leq_T^{\neg\mathcal{F}}$ be defined as follows for expressions s, t of equal type T: $s \leq_T^{\neg\mathcal{F}} t$ iff for all contexts $C[\cdot :: T]$ that do not contain function symbols, but may contain $\bot$-expressions: if $C[s], C[t]$ are closed, then $C[s]\downarrow \implies C[t]\downarrow$.*

Note that s, t may contain function symbols.

Proposition 5.7. $\leq_T \;=\; \leq_T^{\neg\mathcal{F}}$.

Proof. It is sufficient to show that $\leq_T^{\neg\mathcal{F}} \;\subseteq\; \leq_T$. Therefore, let s, t be expressions of type T, let C be a context, such that $C[s], C[t]$ are closed and $C[s]\downarrow$. We have to show that $C[t]\downarrow$. Let n be the length of the reduction of $C[t]$. Let $C^{\neg\mathcal{F}}$ be the context constructed from C as follows: apply $n+1$ times the following step: a delta-reduction for every occurrence of a function symbol. As a last step, replace every remaining function symbol by $\bot$ of the appropriate type. Note that a single reduction step can shift the bot-symbols at most one lambda-level higher. Thus, by standard reasoning and induction, we obtain that $C^{\neg\mathcal{F}}[s]\downarrow$, by using the reduction sequence of $C[s]$ also for $C^{\neg\mathcal{F}}[s]$, where the induction is by the number of reduction steps. The assumption now implies that $C^{\neg\mathcal{F}}[t]\downarrow$. We have $C^{\neg\mathcal{F}}[t] \leq_T C[t]$, since delta-reduction is correct and the insertion of $\bot$ makes the expression smaller w.r.t $\leq_T$. Hence $C[t]\downarrow$. □

Corollary 5.8 (F-extensions). *Let $\mathcal{P}$ be a program and $\mathcal{P}'$ be an extension of $\mathcal{P}$ where only the set $\mathcal{F}$ is extended to $\mathcal{F}'$. Then for all $\mathcal{P}$-expressions $s, t :: T$:*

$$\begin{array}{lcll} s \leq_{\mathcal{P},T} t & \iff & s \leq_{\mathcal{P}',T} t & \text{and} \\ s \sim_{\mathcal{P}} t & \iff & s \sim_{\mathcal{P}'} t & \end{array}$$

Proof. This follows from Proposition 5.7.

$$
\begin{array}{ll|ll}
\texttt{Bot}\ s & \to \texttt{Bot} & & \\
s\ \texttt{Bot} & \to \texttt{Bot} & (c\ \ldots\texttt{Bot}\ldots) & \to \texttt{Bot} \\
\texttt{case}_K\ \texttt{Bot}\ \ldots & \to \texttt{Bot} & (\texttt{seq}\ t\ \texttt{Bot}) & \to \texttt{Bot} \\
\texttt{case}_K\ s\ \ (p_1 \to \texttt{Bot})\ldots & & (\texttt{seq}\ \texttt{Bot}\ t) & \to \texttt{Bot} \\
\qquad\quad (p_n \to \texttt{Bot}) & \to \texttt{Bot} & &
\end{array}
$$

Fig. 5. Bot-reduction rules

$$
\begin{array}{lll}
\text{seqc} & (\texttt{seq}\ (c\ s_1 \ldots s_n)\ s) & \to \texttt{seq}\ s_1\ (\ldots(\texttt{seq}\ s_n\ s)\ldots) \\
\text{seqlam} & (\texttt{seq}\ (\lambda x.s)\ t) & \to t \\
\text{seqx} & (\texttt{seq}\ x\ s) & \to s \\
\text{seqapp} & ((\texttt{seq}\ s_1\ s_2)\ s_3) & \to (\texttt{seq}\ s_1\ (s_2\ s_3)) \\
\text{seqseq} & ((\texttt{seq}\ (\texttt{seq}\ s_1\ s_2)\ s_3) & \to (\texttt{seq}\ s_1\ (\texttt{seq}\ s_2\ s_3)) \\
\text{caseseq} & (\texttt{case}_K\ (\texttt{seq}\ r\ s)\ \mathit{alts}) & \to (\texttt{seq}\ r\ (\texttt{case}_K\ s\ \mathit{alts})) \\
\text{VNbeta} & ((\lambda x.s)\ t) & \to \texttt{seq}\ t\ s[t/x] \\
\text{VNcase} & \left\{\begin{array}{l} (\texttt{case}_K\ (c\ s_1 \ldots s_n) \\ \qquad (c\ x_1 \ldots x_n)\ \texttt{->}\ t \\ \qquad \ldots \end{array}\right\} & \to \texttt{seq}\ s_1\ (\ldots(\texttt{seq}\ s_n\ t[s_1/x_1,\ldots,s_n/x_n]))
\end{array}
$$

Fig. 6. Adapted call-by-name-reduction rules

$$
\begin{array}{ll}
\text{caseapp} & ((\texttt{case}_K\ t_0\ (p_1\,\texttt{->}\,t_1)\ldots(p_n\,\texttt{->}\,t_n))\ r) \\
& \qquad\qquad \to (\texttt{case}_K\ t_0\ (p_1\,\texttt{->}\,(t_1\ r))\ldots(p_n\,\texttt{->}\,(t_n\ r))) \\
\text{casecase} & (\texttt{case}_K\ (\texttt{case}_{K'}\ t_0\ (p_1\,\texttt{->}\,t_1)\ldots(p_n\,\texttt{->}\,t_n))\ \ (q_1\,\texttt{->}\,r_1)\ldots(q_m\,\texttt{->}\,r_m)) \\
& \qquad\qquad \to (\texttt{case}_{K'}\ t_0\ (p_1\,\texttt{->}\,(\texttt{case}_K\ t_1\ (q_1\,\texttt{->}\,r_1)\ldots(q_m\,\texttt{->}\,r_m))) \\
& \qquad\qquad\qquad\qquad \ldots \\
& \qquad\qquad\qquad\qquad (p_n\,\texttt{->}\,(\texttt{case}_K\ t_n(q_1\,\texttt{->}\,r_1)\ldots(q_m\,\texttt{->}\,r_m)))) \\
\text{seqcase} & (\texttt{seq}\ (\texttt{case}_K\ t\ (q_1\,\texttt{->}\,r_1)\ldots(q_m\,\texttt{->}\,r_m))\ r) \\
& \qquad\qquad \to (\texttt{case}_K\ t\ (q_1\,\texttt{->}\,(\texttt{seq}\ r_1\ r))\ldots(q_m\,\texttt{->}\,(\texttt{seq}\ r_m\ r)))
\end{array}
$$

Fig. 7. Case-Shifting Transformations

6 Localizing Values

In the following we intend to show that $\leq_T$ and $\sim_T$ do not change, when $\mathcal{P}$ is extended to $\mathcal{P}'$. The technique is to show a CIU-Theorem for $\mathcal{P}$ that only uses $\mathcal{P}$-reduction contexts and $\mathcal{P}$-value substitutions.

We want to show an analogue to the subexpression property of simply-typed lambda-calculus: That irreducible expressions t only have subexpressions, whose type can be composed of subtypes of the expression t. However, the usual notion of call-by-value reduction is not sufficient: we need an extended set of reductions in order to standardize the values of $\mathcal{P}$-type, such that there are no further subexpressions that mention types of an extension. There are the following patterns, where a type may be eliminated:

- In $(s\ t) :: b$ with $s :: a \rightarrow b$ and $t :: a$, the type a is eliminated.
- In $(\texttt{case}_T\ s\ \mathit{alts})$, the type of s is eliminated.
- In $(\texttt{seq}\ s\ t)$, the type of s is eliminated

New types may be generated in the following constructions:

- In $(c\ s_1 \ldots s_n)$ where c introduces a $\mathcal{P}'$-type.
- In $\lambda x.s$, the variable x may have a $\mathcal{P}'$-type.
- In $(\texttt{case}_T\ s\ ((c\ x_1 \ldots x_n) \rightarrow r_1) \ldots$, the pattern variables x_i may introduce a $\mathcal{P}'$-type, if T is a $\mathcal{P}'$-type.

In the following, we will add a seq-construct that always comes with two arguments. The expression seq can be seen as an abbreviation for $(\lambda x, y.y)$, where we assume that the seq is labelled such that it can be distinguished from other lambda-abstractions. Note that the seq-construct will be used, since we deal with subexpressions that contain free variables, and so the progress-Lemma is not applicable. E.g. $((\lambda x. \ldots)\ (\texttt{case}\ y\ \ldots))$ may be irreducible, but not a value. However, for the lemma below it is necessary to be able to apply a general kind of beta-reduction to this expression. We also permit the symbol Bot, labeled with a type, for Ω-expressions. The extended set of VN-reductions is in figures 5, 6 and 7.

Lemma 6.1. *Let $\mathcal{P}$ be a program and $\mathcal{P}'$ be an extension of $\mathcal{P}$. Let v be a closed F-free $\mathcal{P}'$-value of closed $\mathcal{P}$-type T, and assume that $v \xrightarrow{VN,*} v'$, where v' is VN-irreducible. Then v' is a closed F-free value such that every subexpression of v' has a $\mathcal{P}$-type. In particular, v' is a $\mathcal{P}$-value.*

Proof. We have assumed that there is a closed and VN-irreducible value v' with $v \xrightarrow{VN,*} v'$. It is obvious that v' is a value, since on the top level of v, there are no potential reductions or transformations.
Assume for contradiction that there is a subexpression s_1 of v' of non-$\mathcal{P}$-type. We choose s_1 as follows: It is not in the scope of a binder that binds a variable of non-$\mathcal{P}$-type. This is possible, since if s_1 is within such a scope, then we can choose another s_1' as follows: if it is a lambda-binder, then we choose the corresponding abstraction. If the binding comes from a pattern in a case-expression, then the case-expression is of the form $\texttt{case}_T\ s_1'\ (c\,x_1 \ldots x_n) \rightarrow r \ldots$, where T is a $\mathcal{P}'$-type and s_1 is contained in r. In this case we choose s_1' as the next one. This selection process terminates, since the binding-depth is strictly decreased. We arrive at an expression that is not within the scope of a non-$\mathcal{P}$-binder. Among these expressions we choose an s_1 that has maximal size. Note that s_1 is irreducible. We check all cases for the location of s_1:

- s_1 cannot be an argument of a constructor due to maximality.
- s_1 cannot be the body of an abstraction due to maximality.
- s_1 cannot be the second argument in seq due to maximality, but may be the first argument in the seq-expression.
- s_1 cannot be an argument in an application due to maximality, but may be in function position.

- s_1 may be the first argument of a `case`, but not the result expression of an alternative due to maximality.

Now we analyze the remaining cases:

- s_1 is an application. Then $s_1 = s'_1\ s'_2\ \dots s'_n$ with $n \geq 2$, such that s'_1 is not an application. Obviously, s'_1 is also of non-$\mathcal{P}$-type. Now s'_1 cannot be a variable, since all bound variables above s_1 have $\mathcal{P}$-type. The expression s'_1 can also not be an abstraction, `Bot`, a `seq`-expression, or a case-expression, since v' is irreducible. It cannot be a constructor application due to typing. Hence this case is impossible.
- s_1 is in function position in an application. Then there is an expression $s_1\ s_2$. By the previous item, s_1 is not an application, and by assumption, $s_1\ s_2$ has $\mathcal{P}$-type. Now s_1 cannot be a variable, since all variables above s_1 have $\mathcal{P}$-type. The expression s_1 can also not be an abstraction, `Bot`, a `seq`-expression, or a case-expression, since v' is irreducible. It cannot be a constructor application, due to typing. Hence this case is impossible.
- s_1 is the first argument of a `case` for a non-$\mathcal{P}$-type. Then s_1 cannot be a variable, since variables bound above have $\mathcal{P}$-type. Also, s_1 is neither of the following: `case`-expression, `Bot`, constructor-expression, and `seq`-expression, since v' is irreducible. It cannot be an abstraction due to typing. Hence this case is impossible.
- s_1 is the first element of a `seq`. Irreducibility shows that it is neither an abstraction nor a constructor application, a `seq`-expression, `Bot`, a `case`-expression nor a variable. The expression s_1 is not an application due to previous items, hence this case is also impossible.

□

6.1 VN-reductions: Approximating the Values

The goal of this subsection is to show that $\mathcal{P}$-values and $\mathcal{P}$-reduction contexts are sufficient to check global contextual equality of $\mathcal{P}$-expressions, i.e., The arguments require several steps. Unfortunately, is not clear, whether VN-reduction is (strongly) terminating (i.e. every reduction terminates): we could not find a proof. Hence we have to use other methods to show that alien symbols are not required in values or contexts.

Partial Termination of VN-Reduction We show that VN-reduction without VN-beta- and VN-case -reductions terminates:
Therefore we use the following measure *css* of expressions:

$$
\begin{array}{ll}
css(\texttt{case}\ s\ (p_1 \to r_1) \dots (p_n \to r_n)) & = 1 + 2css(s) + \max_{i=1,..,n}(css(r_i)) \\
css(s\ t) & = 1 + 2css(s) + 2css(t) \\
css(\texttt{seq}\ s\ t) & = 2css(s) + css(t) \\
css(\texttt{Bot}) & = 1 \\
css(x) & = 1 \\
css(c\ s_1 \dots s_n) & = 1 + css(s_1) + \dots + css(s_n) \\
css(\lambda x.s) & = 1 + css(s)
\end{array}
$$

$$
\begin{array}{ll}
(s\ t)^{VNS} & \to (s^{VNS}\ t) \\
(s\ t)^{VNS} & \to (s\ t^{VNS}) \\
(\texttt{seq}\ s\ t)^{VNS} & \to (\texttt{seq}\ s^{VNS}\ t) \\
(\texttt{seq}\ s\ t)^{VNS} & \to (\texttt{seq}\ s\ t^{VNS}) \\
(\texttt{case}\ s\ \ldots)^{VNS} & \to (\texttt{case}\ s^{VNS}\ \ldots) \\
(c\ s_1\ \ldots\ s_n)^{VNS} & \to (c\ s_1\ \ldots\ s_i^{VNS} \ldots\ s_n)
\end{array}
$$

Fig. 8. The *VNS*-label-shifting rules

Lemma 6.2. *Every VN-reduction sequence without the (VNcase)- and (VNbeta)-reduction steps is finite.*

Proof. We check that for every possible reduction rule, the measure is strictly decreased:

The reduction rules that reduce to `Bot` strictly reduce the measure.
seqc : reduces the size by 2.
seqlam,seqx : strictly reduce the size.
seqapp : $4css(s_1) + 2css(s_2) + 1 + 2css(s_3) > 2css(s_1) + 2css(s_2) + 1 + 2css(s_3)$.
seqseq : $4css(s_1) + 2css(s_2) + css(s_3) > 2css(s_1) + 2css(s_2) + css(s_3)$.
caseseq : $4css(r) + 2css(s) + a > 2css(r) + 2css(s) + a$.
caseapp : $4css(t_0) + 2\max(t_i) + 2css(r) > 2css(t_0) + \max(2css(t_i) + 2css(r))$.
casecase : $4css(t_0) + 2\max(css(t_i)) + \max(css(r_i)) > 2css(t_0) + \max(2css(t_i) + \max(css(r_i)))$.
seqcase : $4css(t) + 2\max(r_i) + css(r) > 2css(t) + \max(2css(r_i) + css(r))$.

Lemma 6.3. *All VN-reduction rules are correct.*

Proof. The bot-reduction rules are correct, which follows from the CIU-Theorem. The other rules are also correct using Corollary 4.17, since the left and right hand side will in any case reduce to the same value after applying a closing value-substitution.

Now we want to show that infinite VN-reduction sequences for a expression indicate that this expression can only be equal to `Bot`. For enable a proof, we define a standard reduction that is usually applied to subexpressions of v.

Definition 6.4. *A* VN-standard-reduction *of a perhaps open expression t is defined as follows: Apply the VNS-label-shift in Figure 8 to t, starting with t^{VNS} and where no other subexpression is labelled VNS, and perform it exhaustively and also in all non-deterministic executions. If at least one* `Bot`*-redex according to Figure Fig. 5, 6 and 7 is labeled, then the corresponding leftmost-outermost* `Bot`*-reduction is applied. If there is no such* `Bot`*-reduction, then the innermost-leftmost VN-reduction according to Fig. 5, 6 and 7 is applied to a labelled redex. The reduction is denoted as $\xrightarrow{\text{VNsr}}$.*

Note that there may be multiple redexes with *VNS*-labels, but due to the above priority rules, the VN-standard-reduction is uniquely defined.

In the following, if t is an expression, and σ is a value-substitution such that $\sigma(t)$ is closed, then $val_\sigma(t)$ denotes the value defined by $\sigma(t) \xrightarrow{sr,*} val_\sigma(t)$. The standard-reduction treats the `seq`-constant as the lambda-expressions $\lambda x, y.y$. For counting, we assume that this lambda-expression is labelled to distinguish it from other abstractions. The `seq`-reduction $(\texttt{seq}\ v\ s) \to s$, where v is a closed value (which corresponds to 2 beta-reductions) is not counted in the length of standard-reductions.

Lemma 6.5. *Let t be an expression. If for some closing value-substitution σ the reduction $\sigma(t) \xrightarrow{sr,n} v$ holds for some value v, then $t \xrightarrow{\text{VNsr},*} t'$, where t' is* VNsr*-irreducible, and $\sigma(t') \xrightarrow{\leq n,sr} v$.*

Proof. Note that if the VNsr-reduction sequence includes a Bot-reduction, then the final result will be `Bot`, and hence not a value. Hence no `Bot`-reduction could be used in the reduction sequence $t \xrightarrow{\text{VNsr},*} t'$. We show by induction first on the number of (VNbeta), (VNcase)-VNsr-reductions and then on the total number of VN-standard-reductions that $t \xrightarrow{\text{VNsr}} t'$ and $t \xrightarrow{sr,n} v$ implies that $t' \xrightarrow{sr,\leq n} v$ if the VN-reduction is not a (VNcase) nor a (VNbeta)-reduction and that $t \xrightarrow{\text{VNsr},(beta)\vee(case)} t'$ and $t \xrightarrow{sr,n} v$ implies that $t' \xrightarrow{sr,\leq n-1} v$.

First we assume that $t \xrightarrow{\text{VNsr}} t'$ for a VN-reduction not in $\{\texttt{Bot}, (\mathit{VNcase}), (\mathit{VNbeta})\}$. For the reductions (seqc), (seqlam), (seqx), (seqapp), (seqseq) and (caseseq), it is easy to see that the sr-reduction sequence (not counting the `seq`-reductions) is the same. The same holds for the (caseapp), (casecase) and (seqcase)-reductions.

Now we look at the (VNbeta)-reduction. The sr-reduction of $\sigma(R[((\lambda x.s)\ r)]$ compared with $\sigma(R[\texttt{seq}\ r\ s[r/x]])$ first sr-reduces $\sigma(r)$ to a value, and then makes a (beta)-reduction and proceeds with $\sigma(s[r/x])$. On the right hand side, this is the same reduction sequence, if the `seq`-reduction is not counted. Thus the number of reductions of t' to a value is the same as for t, with one (beta)-reduction less.

The same reasoning holds also for the (VNcase)-reduction.

Lemma 6.2 shows that there are no infinite $\xrightarrow{\text{VNsr}}$-reductions without (VNcase), (VNbeta)-reductions. Hence the ordering on VNsr-reductions is well-founded, which consisting of the lexicographically ordered pairs (l_1, l_2) where l_1 is the number of standard-(VNcase), (VNbeta)-reductions, and l_2 is the number of other non-`Bot`-VNsr-reduction.

The base case is that there are no sr-reductions necessary, i.e. $\sigma(t)$ is a value. Then t is either an abstraction, and there are no VNsr-reductions, or it is a variable, or of the form $(c\ t_1 \dots t_n)$, where t_i is constructed from constructors, variables and abstractions. In this case also no VNsr-reduction is possible.

Finally, we conclude that the claim of the lemma holds.

Corollary 6.6. *If t has an infinite* VNsr*-reduction, then for every closing value-substitution σ: $\sigma(t)\Uparrow$, i.e. t is an Ω-expression.*

Proof. Assume that for some σ: $\sigma(t)\downarrow$. Then Lemma 6.5 shows that t has a finite VNsr-reduction, which contradicts the assumption.

Now we can justify the following mathematical (non-effective) construction $\mathit{ValueConstr}_n$ of a $\mathcal{P}$-value for a $\mathcal{P}'$-value v of $\mathcal{P}$-type, given a depth n:

- $\mathit{ValueConstr}_n(t)$: Apply the VN-standard-reduction to t: if it does not terminate, then the result is `Bot`. Otherwise, let t' be the irreducible result of the VN-standard-reduction sequence starting from t.
- Apply the same construction to the immediate subexpressions of t' and replace these subexpressions with the results.
- If the abstraction-depth of the subexpression exceeds $n+1$, then replace the subexpression by `Bot`.
- Apply the same construction to the bodies of the maximal abstractions of t' using parameter $n-1$ and replace these subterms with the results.
- If the abstraction-depth of the subexpression exceeds $n+1$, then replace the subexpression by `Bot` not changing its type.

Lemma 6.7. *Let $\mathcal{P}'$ be an extension of the program $\mathcal{P}$. Given a $\mathcal{P}'$-value v of $\mathcal{P}$-type, the construction $\mathit{ValueConstr}_n(v)$ results in a $\mathcal{P}$-value v' with $v' \leq_T v$.*

Proof. The (mathematical) construction terminates and results in a value. The reason is that after one step Lemma 6.1 shows that the result is a $\mathcal{P}$-value.

Lemma 6.8. *Let t be an expression. If for some closing value-substitution σ the reduction $\sigma(t) \xrightarrow{sr,n} v$ holds for some value v, and t' is constructed from t using $\mathit{ValueConstr}$ for binder-depth $n+1$, then $\sigma(t') \xrightarrow{\leq n,sr} v$.*

Proof. This follows from Lemma 6.5, and since the `Bot`-insertions are below binder-depth n, and since Ω-expressions are smaller than other expressions w.r.t. $\leq_T$.

Lemma 6.9. *Let s,t be expressions, such that for all closing $\mathcal{P}$-value substitution and for all closed $\mathcal{P}$-reduction contexts R the implication $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ holds. Then for all closing $\mathcal{P}'$-value substitution σ' and all closed $\mathcal{P}'$-reduction contexts R', also the implication $R'[\sigma'(s)]\downarrow \implies R'[\sigma'(t)]\downarrow$ holds.*

Proof. Let σ' be a $\mathcal{P}'$-closing value substitution and R' be a closed $\mathcal{P}'$-reduction context, such that $R'[\sigma'(s)]\downarrow$ holds. If the type of R' is a $\mathcal{P}'$-type, then we use $R'' = \texttt{seq}\ R'\ \texttt{True}$, where we w.l.o.g. assume that the type `Bool` with constructors `True`, `False` is a $\mathcal{P}$-type. Let n be the length of the reduction of $R''[\sigma'(s)]$, let $\sigma' = \{x_1 \mapsto v'_1, \ldots, x_m \mapsto v'_m\}$, and let $r' := \lambda x.R''[x]$. Then for every v'_i construct $v_i := \mathit{ValueConstr}_n(v'_i)$, i.e. for depth n, and also construct r from r' for depth n. Then with $R[\cdot] := r\ [\cdot]$, we have $R[\sigma(s)]\downarrow$, since every standard reduction step reduces the lambda-depth of the approximating `Bot`s at most by one, and by Lemma 6.7. By the assumption, we also have $R[\sigma(t)]\downarrow$, and since $r \leq_T r'$ and $\sigma(t) \leq_T \sigma'(t)$, we also obtain $R''[\sigma'(t)]\downarrow$.

Theorem 6.10 (CIU-Theorem F-free and global). *Let $\mathcal{P}'$ be an extension of $\mathcal{P}$. For $s, t :: T \in L$, where T is a $\mathcal{P}$-type, the implication $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ holds for all F-free $\mathcal{P}$-value substitutions σ and F-free $\mathcal{P}$-reduction contexts R, where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P}',T} t$ holds.*

Proof. This follows from the F-free CIU-theorem 4.15 and from Lemma 6.9.

Corollary 6.11 (CIU-Theorem F-free and local). *Let $\mathcal{P}$ be a program and $s, t :: T \in L$ be $\mathcal{P}$-expressions.*
Then $s \leq_{\mathcal{P},T} t$ iff $s \leq_{\mathcal{P}\forall,T} t$.

Proof. This follows from the F-free CIU-theorem 6.10, since the condition holds for all extensions $\mathcal{P}'$ of $\mathcal{P}$.

6.2 Global Correctness of Several Reductions and Transformations

The VN-reductions in figures 5, 6 and 7 are not only interesting as normalization rules for values. They are also globally correct reductions, as we will see. The same holds for the call-by-value reduction rules.
6.10, the following is obtained:

Theorem 6.12. *The transformations (beta), (delta), and (case), i.e. the call-by-value reduction rules, are globally correct program transformations in L.*

Proof. This follows from Proposition 4.11 and Corollary 6.11.

Theorem 6.13. *The transformations in figures 5, 6 and 7, i.e. the Bot-reductions, the adapted call-by-name reduction rules and the case-shifting transformations, are globally correct program transformations in L.*

Proof. Lemma 6.3 shows that the transformations in figures 5, 6 and 7 are correct. if only $\mathcal{P}$-reduction contexts and $\mathcal{P}$-value-substitutions are used. Then Corollary 6.11 shows that the transformations are also globally correct.

7 Bisimulation

We show that equality of expressions can be determined by bisimulation. For simplicity, we only prove the properties of a simulation. We assume that a program $\mathcal{P}$ is fixed. The proof method is basically from Howe [How89], but since it is used here for a typed language, the adaptation of Gordon [Gor99] for PCF is closer. A difference is that we have recursive polymorphic types and data constructors. The approach was also worked out for a call-by-need non-deterministic calculi in a similar way in [Man05,MSS07].
A substitution σ that replaces variables by closed values (of equal type) and that closes the argument expressions is called a *closing value substitution.* In this section we assume that binary relations ν only relate expressions of equal monomorphic type, i.e. $s\ \nu\ t$ only if s, t have the same monomorphic type. The

restriction of the relation μ to the type T is usually indicated by an extra suffix T: i.e. μ_T. Typing is usually omitted, if it is clear from the context. We mention typing only if it is necessary. This is justified, since types appear as labels, and thus we can argue as in a simply typed system. Substitutions are also typed and can only replace variables by expression of the same type.
Let ν be a binary relation on closed expressions. Then $s\ \nu^o\ t$ for any expressions s, t iff for all closing value substitutions σ: $\sigma(s)\ \nu\ \sigma(t)$. Conversely, for binary relations μ on open expressions, μ^c is the restriction to closed expressions.

Lemma 7.1. *For a relation ν on closed expressions, the equality $((\nu)^o)^c = \nu$ holds. For a relation μ on open expressions: $s\ \mu\ t \implies \sigma(s)\ (\mu)^c\ \sigma(t)$ for all closing value substitutions σ is equivalent to $\mu\ \subseteq ((\mu)^c)^o$.*

For simplicity, we sometimes use as e.g. in [How89] the higher-order abstract syntax and write $\tau(..)$ for an expression with top operator τ, which may be `case`, application, a constructor or λ, and θ for an operator that may be the head of a value i.e. a constructor or λ. Note that θ may represent also the binding λ using $\theta(x.s)$ as representing $\lambda x.s$. Abstract syntax expressions $x.s$ only occur in relational formulas, where we permit α-renaming and follow the convention that $x.s\ \mu\ x.t$ means $s\ \mu\ t$ for open expressions s, t.
A relation μ is *operator-respecting*, iff $s_i\ \mu\ t_i$ for $i = 1, \ldots, n$ implies $\tau(s_1, \ldots, s_n)\ \mu\ \tau(t_1, \ldots, t_n)$.

Definition 7.2. *Let $\leq_b$ be the greatest fixpoint (on the set of binary relations over closed expressions) of the following operator $[\cdot]$ on binary relations ν over closed expressions: $s\ [\nu]\ t$ if $s\Uparrow$ or $s \downarrow (c\ s_1 \ldots s_n)$ and $t \downarrow (c\ t_1 \ldots t_n)$ and $s_i\ \nu\ t_i$ for all i or $s \downarrow \lambda x.s'$ and $t \downarrow \lambda x.t'$ and $s'\ \nu^o\ t'$*

The principle of co-induction for the greatest fixpoint of $[\cdot]$ shows that for every relation ν on closed expressions with $\nu \subseteq [\nu]$, we derive $\nu\ \subseteq \leq_b$. This obviously also implies $\nu^o\ \subseteq \leq_b^o$.

Lemma 7.3. $\leq_{\mathcal{P}}\ \subseteq\ \leq_b^o$

Proof. Since reduction is deterministic, we have $(\leq_{\mathcal{P}})^c \subseteq [(\leq_{\mathcal{P}})^c]$ and hence $(\leq_{\mathcal{P}})^c\ \subseteq \leq_b$. This implies $\leq_{\mathcal{P}}\ \subseteq\ \leq_b^o$.

Lemma 7.4. *For closed values $(c\ s_1 \ldots s_n), (c\ t_1 \ldots t_n)$ of equal type, we have $(c\ s_1 \ldots s_n) \leq_b (c\ t_1 \ldots t_n)$ iff $s_i \leq_b t_i$. For abstractions $\lambda x.s, \lambda x.t$ of equal type, we have $\lambda x.s \leq_b \lambda x.t$ iff $s \leq_b^o\ t$.*

Proof. These properties follow from the fixpoint property of $\leq_b$.

Lemma 7.5. *The relations $\leq_b$ and $\leq_b^o$ are reflexive and transitive*

Proof. Transitivity follows by showing that $\nu :=\ \ \leq_b \cup (\leq_b \circ \leq_b)$ satisfies $\nu \subseteq\ [\nu]$ and then using co-induction.

The goal in the following is to show that $\leq_b$ is a precongruence. We will show that this implies that $\leq_b^o = \leq_c$.

Definition 7.6. *The congruence candidate* $\widehat{\leq_b^o}$ *is a binary relation on open expressions (ala Howe) and is defined inductively on the structure of expressions:*

1. $x \widehat{\leq_b^o} s$ *if* $x \leq_b^o s$.
2. $\tau(s_1, \dots, s_n) \widehat{\leq_b^o} s$ *if there is some expression* $\tau(s'_1, \dots, s'_n) \leq_b^o s$ *with* $s_i \widehat{\leq_b^o} s'_i$.

The following is easily proved by standard arguments (for Howe's technique).

Lemma 7.7.

1. $\widehat{\leq_b^o}$ *is reflexive*
2. $\widehat{\leq_b^o}$ *and* $(\widehat{\leq_b^o})^c$ *are operator-respecting*
3. $\leq_b^o \subseteq \widehat{\leq_b^o}$.
4. $\widehat{\leq_b^o} \circ \leq_b^o \ \subseteq \ \widehat{\leq_b^o}$.
5. $(s \ \widehat{\leq_b^o} \ s' \wedge t \ \widehat{\leq_b^o} \ t') \implies t[s/x] \ \widehat{\leq_b^o} \ t'[s'/x]$
   if s, s' *are closed values, i.e. the substitutions* $[s/x]$, $[s'/x]$ *replace variables by closed values.*
6. $\widehat{\leq_b^o} \ \subseteq ((\widehat{\leq_b^o})^c)^o$

Proof. The proofs of the first claims are by structural induction. The last claim (6) follows from part (5) using Lemma 7.1.

Lemma 7.8. *The middle expression in the definition of* $\widehat{\leq_b^o}$ *can be chosen as closed, if* s, t *are closed: Let* $s = \tau(s_1, \dots, s_{ar(\tau)})$, *such that* $s \ \widehat{\leq_b^o} \ t$ *holds. Then there are operands* s'_i, *such that* $\tau(s'_1, \dots, s'_{ar(\tau)})$ *is closed,* $\forall i : s_i \ \widehat{\leq_b^o} \ s'_i$ *and* $\tau(s'_1, \dots, s'_{ar(\tau)}) \leq_b^o s$.

Proof. The definition of $\widehat{\leq_b^o}$ implies that there is a expression $\tau(s''_1, \dots, s''_{ar(\tau)})$ such that $s_i \widehat{\leq_b^o} s''_i$ for all i and $\tau(s''_1, \dots, s''_{ar(\tau)}) \ \leq_b^o t$. Let σ be the substitution with $\sigma(x) := v_x$ for all $x \in FV(\tau(s''_1, \dots, s''_{ar(\tau)}))$, where v_x is the closed value for the type of x that exists by Assumption 3.4.
Lemma 7.7 now shows that $s_i = \sigma(s_i) \widehat{\leq_b^o} \sigma(s''_i)$ holds for all i. The relation $\sigma(\tau(a''_1, \dots, a''_{ar(\tau)})) \ \leq_b^o t$ holds, since t is closed and due to the definition of an open extension. The requested expression is $\tau(\sigma(a''_1), \dots, \sigma(a''_{ar(\tau)}))$.

The proof of the following theorem is an adaptation of [How96, Theorem 3.1] to closing value substitutions.

Theorem 7.9. *The following claims are equivalent.*

1. $\leq_b^o$ *is a precongruence*
2. $\widehat{\leq_b^o} \ \subseteq \ \leq_b^o$
3. $(\widehat{\leq_b^o})^c \ \subseteq \leq_b$

Proof. The claim is shown by a chain of implications.

"1 $\implies$ 2": Let $\leq_b^o$ be a precongruence. Then we show that $s\ \widehat{\leq_b^o}\ t$ implies $s\ \leq_b^o\ t$ by induction on the definition of $\widehat{\leq_b^o}$.

- If s is a variable, then $s\ \leq_b^o\ t$.
- Let $s = \tau(s_1, \ldots, s_{ar(\tau)})$. Then there is some $\tau(s'_1, \ldots, s'_{ar(\tau)})\ \leq_b^o\ t$ with $s_i\ \widehat{\leq_b^o}\ s'_i$ for every i. By induction on the expression structure: $\forall i : s_i\ \leq_b^o\ s'_i$. Since $\leq_b^o$ is a precongruence by assumption, we derive $\tau(s_1, \ldots, s_{ar(\tau)})\ \leq_b^o\ \tau(s'_1, \ldots, s'_{ar(\tau)})$ and furthermore $\tau(s_1, \ldots, s_{ar(\tau)})\ \leq_b^o\ s$ by transitivity of $\leq_b^o$.

"2 $\implies$ 3": From $\widehat{\leq_b^o}\ \subseteq\ \leq_b^o$ we have $(\widehat{\leq_b^o})^c\ \subseteq\ (\leq_b^o)^c = \leq_b$.

"3 $\implies$ 2": From $(\widehat{\leq_b^o})^c\ \subseteq \leq_b$ we have $((\widehat{\leq_b^o})^c)^o\ \subseteq\ \leq_b^o$ by monotonicity. Lemma 7.7 (6) implies $\widehat{\leq_b^o}\ \subseteq\ ((\widehat{\leq_b^o})^c)^o\ \subseteq\ \leq_b^o$.

"2 $\implies$ 1": Lemma 7.7 and $\widehat{\leq_b^o}\ \subseteq\ \leq_b^o$ together imply $\widehat{\leq_b^o}\ =\ \leq_b^o$, thus $\leq_b^o$ is operator-respecting by Lemma 7.7 and a precongruence. □

7.1 Determining the Congruence Candidate

Lemma 7.10. *If $s \to s'$, then $s \leq_b^o s'$*

Proof. This holds, since standard reduction is deterministic and by the definition of $\leq_b^o$.

Lemma 7.11. *If $s\ \widehat{\leq_b^o}\ t$ and $t \to t'$, then $s\ \widehat{\leq_b^o}\ t'$*

Proof. Follows from Lemma 7.10.

Definition 7.12. *We call* $\widehat{\leq_b^o}$ stable*, iff for all closed s, s', t: $s\ (\widehat{\leq_b^o})^c\ t$ and $s \to s'$ implies $s'\ (\widehat{\leq_b^o})^c\ t$.*

Proposition 7.13. *If $\leq_b$ is a precongruence, then $\leq_b = \leq_{\mathcal{P}}$.*

Proof. Let $s\ \leq_b^o\ t$. Then for all closing value substitutions σ: $\sigma(s)\ \leq_b\ \sigma(t)$ by definition of open extensions. This implies that for all closed contexts C and all closing value substitutions σ: $\forall C : C[\sigma(s)]\ \leq_b\ C[\sigma(t)]$, since $\leq_b^o$ is a precongruence. Hence $s\ \leq_{\mathcal{P}}\ t$. The other direction follows from Lemma 7.3.

Lemma 7.14. *Let s, t be closed expressions such that $s = \theta(s_1, \ldots, s_n)$ is a value and $s\ \widehat{\leq_b^o}\ t$. Then there is some closed value $t' = \theta(t_1, \ldots, t_n)$ with $t \xrightarrow{*} t'$ and for all $i : s_i\ \widehat{\leq_b^o}\ t_i$.*

Proof. The definition of $\widehat{\leq_b^o}$ implies that there is a closed expression $\theta(t'_1, \ldots, t'_n)$ with $s_i\ \widehat{\leq_b^o}\ t'_i$ for all i and $\theta(t'_1, \ldots, t'_n) \leq_b t$. We use induction on the structure of s:
If $s = \lambda x.s'$, then there is some closed $\lambda x.t'\ \leq_b^o\ t$ with $s'\ \widehat{\leq_b^o}\ t'$. The relation $\lambda x.t'\ \leq_b^o\ t$ implies that $t \xrightarrow{*} \lambda x.t''$. Lemma 7.10 now implies $\lambda x.s'\ \widehat{\leq_b^o}\ \lambda x.t''$. Definition of $\widehat{\leq_b^o}$ now shows that there is some closed $\lambda x.t^{(3)}$ with $s'\ \widehat{\leq_b^o}\ t^{(3)}$ and

$\lambda x.t^{(3)} \leq_b \lambda x.t''$. The latter relation implies $t^{(3)} \leq_b^o t''$, which also shows $s' \mathrel{\widehat{\leq_b^o}} t''$.

If θ is a constructor, then there is a closed expression $c(t'_1,\ldots,t'_n)$ with $s_i \mathrel{\widehat{\leq_b^o}} t'_i$ for all i and $c(t'_1,\ldots,t'_n) \leq_b t$. By applying the induction hypothesis to $s_i \mathrel{\widehat{\leq_b^o}} t'_i$ we obtain that $t'_i \xrightarrow{*} t''_i$, where t''_i are values, and hence $c(t''_1,\ldots,t''_n)$ is a value. It follows that $s_i \mathrel{\widehat{\leq_b^o}} t''_i$ by Lemma 7.11 and $c(t''_1,\ldots,t''_n) \leq_b t$, by arranging the reduction $c(t'_1,\ldots,t'_n) \xrightarrow{*} c(t''_1,\ldots,t''_n)$ from left to right to obtain a standard reduction. The definition of $\leq_b$ implies that $t \xrightarrow{*} \theta(t_1^{(3)},\ldots,t_n^{(3)})$ with $t''_i \leq_b t_i^{(3)}$ for all i. By definition of $\widehat{\leq_b^o}$, we obtain $s_i \mathrel{\widehat{\leq_b^o}} t_i^{(3)}$ for all i.

Proposition 7.15. *If* $\widehat{\leq_b^o}$ *is stable, then* $(\widehat{\leq_b^o})^c \subseteq [(\widehat{\leq_b^o})^c]$. *Hence* $(\widehat{\leq_b^o})^c \subseteq \leq_b$ *and* $\leq_b^o$ *is a precongruence.*

Proof. Let s,t be closed, such that $s \mathrel{\widehat{\leq_b^o}} t$. Let $s \downarrow \theta(s_1,\ldots,s_n)$. Then $\theta(s_1,\ldots,s_n)\ (\widehat{\leq_b^o})^c\ t$ by stability. There is some $\theta(t_1,\ldots,t_n)$, such that $t \downarrow \theta(t_1,\ldots,t_n)$ and $\forall i : s_i\ ((\widehat{\leq_b^o})^c)^o\ t_i$. This means that $(\widehat{\leq_b^o})^c \subseteq [(\widehat{\leq_b^o})^c]$. By co-induction and Lemma 7.11, the relation $(\widehat{\leq_b^o})^c \subseteq \leq_b$, and hence also $\widehat{\leq_b^o} \subseteq ((\widehat{\leq_b^o})^c)^o \subseteq \leq_b^o$ hold.

Theorem 7.16. *If* $\widehat{\leq_b^o}$ *is stable, then* $\widehat{\leq_b^o} = \leq_b^o = \leq_{\mathcal{P}}$.

Proof. Lemma 7.11, Propositions 7.15, 7.13 and Theorem 7.9 show the claim.

It remains to show stability:

Proposition 7.17. *Let* s,t *be closed expressions,* $s \mathrel{\widehat{\leq_b^o}} t$ *and* $s \rightarrow s'$ *where* s *is the redex. Then* $s' \mathrel{\widehat{\leq_b^o}} t$.

Proof. Let s,t be closed expressions, $s \mathrel{\widehat{\leq_b^o}} t$ and $s \rightarrow s'$ where s is the redex. The relation $s \mathrel{\widehat{\leq_b^o}} t$ implies that $s = \tau(s_1,\ldots,s_n)$ and that there is some closed $t' = \tau(t'_1,\ldots,t'_n)$ with $s_i \mathrel{\widehat{\leq_b^o}} t'_i$ for all i and $t' \leq_b^o t$.

- For the (beta)-reduction, $s = s_1\ s_2$, where $s_1 = (\lambda x.s'_1)$, s_2 is a closed value, and $t' = t'_1\ t'_2$. Lemma 7.14 shows that $t'_1 \xrightarrow{*} \lambda x.t''_1$ with $\lambda x.s'_1 \mathrel{\widehat{\leq_b^o}} \lambda x.t''_1$ and also $s_1 \mathrel{\widehat{\leq_b^o}} t''_1$. From $s_2 \mathrel{\widehat{\leq_b^o}} t'_2$ and since s_2 is a value, we obtain the next part of the standard reduction $t'_2 \xrightarrow{*} t''_2$ with $s_2 \mathrel{\widehat{\leq_b^o}} t''_2$. From $t' \xrightarrow{*} t''_1[t''_2/x]$ we obtain $t''_1[t''_2/x] \leq_b t$. Lemma 7.7 now shows $s'_1[s_2/x] \mathrel{\widehat{\leq_b^o}} t''_1[t''_2/x]$. Hence $s'_1[s_2/x] \mathrel{\widehat{\leq_b^o}} t$, again using Lemma 7.7.
- Similar arguments apply to the case-reduction.
- Suppose, the reduction is a δ-reduction. Then $s \mathrel{\widehat{\leq_b^o}} t$ and s is a function name. By the definition of $\widehat{\leq_b^o}$, this means $s \leq_b^o t$. Since $s \rightarrow s'$ means also $s' \sim_b^o s$, we also have $s' \leq_b^o t$. By Lemma 7.7, this implies $s' \mathrel{\widehat{\leq_b^o}} t$.

Proposition 7.18. *Standard reduction is stable in surface contexts*

Proof. We use induction on the structure of contexts. The base case is proved in Proposition 7.17. Let $S[s], t$ be closed, $S[s] \ \widehat{\leq_b^o}\ t$ and $S[s] \to S[s']$, where we assume that the redex is not at the top level. The relation $S[s] \ \widehat{\leq_b^o}\ t$ implies that $S[s] = \tau(s_1, \ldots, s_n)$ and that there is some $t' = \tau(t'_1, \ldots, t'_n) \leq_b^o t$ with $s_i \ \widehat{\leq_b^o}\ t'_i$ for all i. If $s_j \to s'_j$, then by induction hypothesis, $s'_j \ \widehat{\leq_b^o}\ t'_j$. Since $\widehat{\leq_b^o}$ is operator-respecting, we obtain also $S[s'] = \tau(s_1, \ldots, s_{j-1}, s'_j, s_{j+1}, \ldots, s_n) \ \widehat{\leq_b^o}\ \tau(t'_1, \ldots, t'_{j-1}, t'_j, t'_{j+1}, \ldots, t'_n)$.

Theorem 7.19. *The following equalities hold:* $\widehat{\leq_b^o} = \leq_b^o = \leq_{\mathcal{P}}$.

Proof. Follows from stability of $\widehat{\leq_b^o}$ using Propositions 7.17, 7.18 and from Theorem 7.16.

8 Constructive Logic and Induction

We assume in this section that a program $\mathcal{P}$, including the set of types, constructors, and function symbols is given and fixed. Of course we assume that all the assumptions (i.e. Assumptions 2.4, 3.4, and 3.6) on $\mathcal{P}$ are satisfied.

8.1 The Syntax

The syntax of monomorphic formulas (w.r.t. a program $\mathcal{P}$) is:

$$\begin{array}{ll} \text{atoms}: & A ::= \texttt{True} \mid \texttt{False} \mid (s \ = \ t) \\ \text{formulas}: & F ::= A \mid F \vee F \mid F \wedge F \mid \neg F \\ & \qquad \mid \forall x :: T.F \mid \exists x :: T.F \\ & \qquad \text{where } T \text{ is a monomorphic } \mathcal{P}\text{-type} \\ & \qquad \text{and } s, t \text{ are } \mathcal{P}\text{-expressions} \end{array}$$

8.2 The Semantics

There are the usual logical values `True`, and `False`. An important reference set for quantification is the set of closed values for a given type T of some program $\mathcal{P}$:

Definition 8.1. *The set $M_{\mathcal{P},T}$ is defined to be the set of all closed $\mathcal{P}$-values of monomorphic type T.*

Note that we have assumed that for every T, the set $M_{\mathcal{P},T}$ is not empty, and that for every monomorphic type T, there is an undefined expression of this type.

Definition 8.2. *Let $\mathcal{P}$ be a program. The semantics of closed monomorphic formulas is as follows, where I is an interpretation.*

$$\begin{array}{lll}
I(s = t) = \texttt{True} & \textit{if } s \sim_{\mathcal{P},\tau} t & \textit{for expressions } s, t :: \tau \\
I(s = t) = \texttt{False} & \textit{if } s \not\sim_{\mathcal{P},\tau} t & \textit{for expressions } s, t :: \tau \\
I(A \wedge B) & = I(A) \wedge I(B) & \\
I(A \vee B) & = I(A) \vee I(B) & \\
I(\neg(A) & = \neg I(A) & \\
I(\forall x :: \tau.F) & = \texttt{True} & \textit{if for all } a \in M_{\mathcal{P},\tau} : I(F[a/x]) = \texttt{True} \\
I(\exists x :: \tau.F) & = \texttt{True} & \textit{if for some } a \in M_{\mathcal{P},\tau} : I(F[a/x]) = \texttt{True}
\end{array}$$

A $\mathcal{P}$-tautology *(*$\mathcal{P}$-theorem, monomorphic $\mathcal{P}$-theorem*) is a closed monomorphic $\mathcal{P}$-formula F, such that $I(F) = \texttt{True}$. F is called a* global *$\mathcal{P}$-tautology, iff it holds for all extensions $\mathcal{P}'$ of $\mathcal{P}$.*

Example 8.3. Given appropriate definitions of the data type `nat` with two constructors $0, \texttt{succ}$, where `pred`, defined as $\lambda x.\texttt{case}_{\texttt{nat}}\ x\ (0 \to \bot)\ (\texttt{succ}\ y \to y)$, is a function that acts like a selector for `succ`, and where also addition $+$ is inductively defined, the following formula is a tautology:

$$\forall x :: \texttt{nat}.\exists y : \texttt{nat}.x + \texttt{succ}(0) = y$$

The closed formula $\exists x :: \texttt{nat}.\texttt{pred}(0) = x$ is not a tautology, since only `nat`-values for x are permitted, and since $\bot \not\sim n$ for every `nat`-value n.
The formula $\neg(\exists x :: \texttt{nat}.\texttt{pred}(0) = x)$ is a tautology.

8.3 Universally Quantified Formulas: Conservativity

Theorem 8.4. *Let $\mathcal{P}$ be a program and $F := \forall x_1 :: T_1, \ldots, x_n :: T_n\ .\ s = t$ be a closed monomorphic $\mathcal{P}$-theorem. Then for all extensions $\mathcal{P}'$ of $\mathcal{P}$, the formula F is also a theorem, i.e., the formula is a global $\mathcal{P}$-theorem.*

Proof. The claim is equivalent to $\lambda x_1, \ldots, x_n.s \sim_{\mathcal{P},T} \lambda x_1, \ldots, x_n.t \iff \lambda x_1, \ldots, x_n.s \sim_{\mathcal{P}',T} \lambda x_1, \ldots, x_n.t$, which holds by Theorem 6.10 for any extension $\mathcal{P}'$ of $\mathcal{P}$.

Thus we can say that universally quantified equations between (monomorphically typed) expressions that hold for a program $\mathcal{P}$ are global (for $\mathcal{P}$). This also holds for the correct program transformations (seen as equations) that we already exhibited in Proposition 4.11 and 5.5.
In the following we extend Theorem 8.4 to formulas, where $s = t$ is replaced by a quantifier-free formula F, provided the type T is restricted.

Definition 8.5. *A type T is a* DT-type*, if every closed value of type T is only built from data constructors.*

Examples for DT-types are Peano-integers, Boolean values and lists of Peano-numbers.

Lemma 8.6. *Let $\mathcal{P}'$ be an extension of $\mathcal{P}$. If $v :: T$ is a value, where T is a DT-type and a $\mathcal{P}$-type. Then v is a $\mathcal{P}$-expression.*

Proof. This follows from the type restriction of data constructors.

Theorem 8.7. *Let $\mathcal{P}$ be a program and F be a closed monomorphic formula, such that all quantified variables have a DT-type. Then F is a $\mathcal{P}$-tautology iff it is a global $\mathcal{P}$-tautology.*

Proof. This follows from the definition of DT-type: the sets $\mathcal{M}_{\mathcal{P},T}$ do not change when the program is extended, from Lemma 8.6, and from Theorem 6.10, which among others shows that all closed $\sim$-equalities are global.

We show a stronger claim on the existence of values than the approximation techniques used by the proof techniques for the CIU-theorem-

Proposition 8.8. *Let $\mathcal{P}$ be a program that is sufficiently expressive, such that in particular every computable function on DT-types can be programmed in $\mathcal{P}$. Let $\mathcal{P}'$ be an extension of $\mathcal{P}$. Then for every $\mathcal{P}'$-value v of $\mathcal{P}$-type $\tau = \tau_1 \to \ldots \to \tau_n$ where all τ_i are DT-types, there exists a "local" $\mathcal{P}$-value w with $v \sim_{\mathcal{P}\forall,\tau} w$.*

Proof. A $\mathcal{P}'$-value v of $\mathcal{P}$-type $\tau_1 \to \ldots \to \tau_n$ where all τ_i are DT-types defines a computable function on DT-types, hence by assumption this can be programmed in $\mathcal{P}$, and the corresponding expression is such a $\mathcal{P}$-value w.

Corollary 8.9. *If there is a polymorphic fixpoint function* $\mathrm{fix} : (\alpha \to \alpha) \to (\alpha \to \alpha)$ *with* $\mathrm{fix} = \lambda f.\lambda x.(f\ (\lambda x.\mathrm{fix}\ f\ x)\ x)$ *in $\mathcal{P}$ then the expressivity-assumption in Proposition 8.8 is satisfied and thus the claim of Proposition 8.8 holds.*

Theorem 8.10. *Let $\mathcal{P}$ be a program such that there is a fixpoint function as in Corollary 8.9 and let F be a closed monomorphic formula, such that all quantified variables have a DT-type or a type $\tau_1 \to \ldots \to \tau_n$, where all τ_i are DT-types. Then F is a $\mathcal{P}$-tautology iff it is a global $\mathcal{P}$-tautology.*

Proof. This follows from the definition of DT-types: the sets $\mathcal{M}_{\mathcal{P},T}$ do not change when the program is extended, and from Corollary 8.9.

We have to leave open the question whether every monomorphic tautology is also a global $\mathcal{P}$-tautology. The obstacle is that we could not prove that for any closed $\mathcal{P}'$-value of $\mathcal{P}$-type there is an equivalent $\mathcal{P}$-value.

8.4 Conservativity by Adding Definedness

Lemma 8.11. *For every DT-type T, we can add a binary function $eq_T :: T \to$ `Bool` to $\mathcal{P}$ such that for all closed values $v, w :: T$: $v \sim w \implies eq_T\ v\ w \xrightarrow{*}$ `True` and $v \not\sim w \implies eq_T\ v\ w \xrightarrow{*}$ `False`.*

Proof. Corollary 5.8 shows that the contextual equality does not change when $\mathcal{F}$ is extended, in particular an equality-test function can be added. It is sufficient to use `case`-expressions and recursion to define the equality function on DT-types with an obvious programming. Since values of type T only consist of data constructors, the comparison will terminate for values.

A quantifier-free formula F that is built from $\wedge, \vee, \neg$ and equations over DT-types can be internalized (i.e. represented by functions) using the Boolean data type and a translation B as follows, where and, or, not are functions on the Boolean values `True`, `False`, programmed using `case`, and which are strict. The behavior, using the Boolean values T, F and `Bot` for non-termination (or undefined values), is as follows:

not	T	F	`Bot`
	F	T	`Bot`

or	T	F	`Bot`
T	T	T	`Bot`
F	T	F	`Bot`
`Bot`	`Bot`	`Bot`	`Bot`

and	T	F	`Bot`
T	T	F	`Bot`
F	F	F	`Bot`
`Bot`	`Bot`	`Bot`	`Bot`

Definition 8.12. *The translation B is defined as:*

$$\begin{array}{ll} B(\wedge) & \equiv \lambda x, y.and\ x\ y \\ B(\vee) & \equiv \lambda x, y.or\ x\ y \\ B(\neg) & \equiv \lambda x.not\ x \\ B(s =_T t) & \equiv eq_T\ s\ t \end{array}$$

A quantifier-free formula F is translated into the equation $(eq_T\ B(F)\ \texttt{True})$.

Note that the Boolean functions are defined to be symmetric in order to reflect the properties of the logical connectives $\vee, \wedge$ like correctness of double negation elimination and the law of deMorgan. However, if an expression is undefined, then the B-translation of a formula also evaluates to undefined, whereas the formula `Bot` $=$ `Bot` is interpreted as `True`. Thus, quantifier-free formulas can only be correctly translated, if every expressions s, t in every equation $s = t$ in the formula evaluates to an answer, since otherwise, the expression $(eq_T\ s\ t)$ does not terminate and is equivalent to `Bot`. Special kinds of formulas that take care of definedness can be translated correctly:
Let $defined_T$ be the function $\lambda x^T.\texttt{True}$ having the following property: $defined_T(s) \xrightarrow{*} \texttt{True}$ for every converging expression of DT-type T. The function never produces `False`, but does not terminate if the argument is not terminating.
Given a program $\mathcal{P}$ that includes the Boolean data type, the extension $\mathcal{P}_D$ is constructed by adding the Boolean functions *and*, *or*, and *not* and for a given finite set of types the functions eq_T, the functions $defined_T$.
For a formula $\forall x_1, \ldots, x_n.F$, where F is quantifier-free and every equation is of a DT-type, let the definedness-formula be $\forall x_1, \ldots, x_n.(Def(F) \implies F)$, where $Def(F)$ is the formula $defined(s_1) = \texttt{True} \wedge \ldots \wedge defined(s_n) = \texttt{True}$, where $s_i, i = 1, \ldots, n$ are all the expressions that occur as top-expressions in equations of F.
The following theorem shows that the theorems in the scope of VeriFun are global:

Theorem 8.13. *Let $\mathcal{P}$ be a program and F be a quantifier-free formula, where every equation in F is of a DT-type, and let $\forall x_1 :: T_1, \ldots, x_n :: T_n\ .(Def(F) \implies F)$ be a closed monomorphic theorem. Then for all extensions $\mathcal{P}'$ of $\mathcal{P}$, the* *formula* $\forall x_1 :: T_1, \ldots, x_n :: T_n.Def(F) \implies F$ *is also a theorem; i.e. it is global* $\mathcal{P}$*-tautology.*

Proof. The formula $\forall x_1 :: T_1, \ldots, x_n :: T_n .Def(F) \implies F$ is a closed monomorphic theorem w.r.t. $\mathcal{P}_D$ if and only if $\lambda x_1, \ldots, x_n.B(Def(F) \implies F) \sim_{\mathcal{P}_D,T} \lambda x_1, \ldots, x_n.B(Def(F))$, which can be seen as follows: If some $\sigma(s_i)$ is undefined, then the equation $defined(s_1) = \texttt{True}$ is false under the interpretation, hence the whole formula is true. For the corresponding substitution, both functions are equivalent to `Bot`. The claim is equivalent to $\lambda x_1, \ldots, x_n.B(Def(F) \implies F) \sim_{\mathcal{P}'_D,T} \lambda x_1, \ldots, x_n.B(Def(F))$, which holds by Theorem 6.10 for any extension $\mathcal{P}'$ and by Lemma 8.11. Constructing the extensions is no problem by keeping names different if necessary). The latter again implies that $\forall x_1 :: T_1, \ldots, x_n :: T_n .Def(F) \implies F$ is a closed monomorphic $\mathcal{P}'$-theorem. Now the CIU-theorem implies that the formula is a global $\mathcal{P}$-tautology.

It is not clear how to extend Theorem 8.4 and Theorem 8.13 to formulas with arbitrary quantifiers and formulas for any extension $\mathcal{P}'$: The semantics changes, since there are more $\mathcal{P}'$-values of type T than $\mathcal{P}$-values of type T, and since existential quantifiers and quantifier-nesting cannot be translated into a programmable function like eq_T.

8.5 Polymorphic Formulas

Polymorphic formulas are like monomorphic formulas, where type variables are permitted in the type of the quantified variables, and in expressions in formulas. The semantics has to be extended as follows:

Definition 8.14. *Given a program* $\mathcal{P}$*, a polymorphic* $\mathcal{P}$*-formula* F *is a* $\mathcal{P}$*-tautology (a* polymorphic $\mathcal{P}$-theorem*), if for every* $\mathcal{P}$*-type substitution* ρ *that instantiates every type variable in* F *with a monomorphic* $\mathcal{P}$*-type, the formula* $\rho(F)$ *is a monomorphic* $\mathcal{P}$*-theorem.*
F *is a* global $\mathcal{P}$-theorem*, iff it holds also for all extensions* $\mathcal{P}'$ *of* $\mathcal{P}$*.*

Example 8.15. In general it is not the case that every polymorphic $\mathcal{P}$-theorem is also global. E.g. let $\mathcal{P}$ be a program where the data type `Bool`, Peano-numbers and lists are defined as data structures, but no other data types. Then the following polymorphic theorem holds:
$(\exists x_1 :: a, x_2 :: a, x_3 :: a.x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3)$
$\implies \exists x :: a.x \neq x_1 \wedge x \neq x_2 \wedge x \neq x_3)$, which expresses that if there are three different values of a certain type, then there is another value of this type. This is true in $\mathcal{P}$. However, it is easy to extend $\mathcal{P}$ to $\mathcal{P}'$ by adding a type T_3 having exactly the set $\{\texttt{red}, \texttt{blue}, \texttt{green}\}$ as data constructors. Then the formula is false for this type T_3 in $\mathcal{P}'$.

A similar example can be constructed if F is an inequation:

Example 8.16. Let $\mathcal{P}$ be the program containing Boolean, Peano-numbers and lists. Let the formula be: $\forall x :: a.\exists y :: a.x \neq y$. This formula is a $\mathcal{P}$-tautology. However, after adding a unit-type with exactly one value giving the program $\mathcal{P}'$, this will no longer hold.

Nevertheless, we believe that there are classes of polymorphic formulas, where being a $\mathcal{P}$-tautology is equivalent to being a global $\mathcal{P}$-tautology: e.g. universally quantified equations; and as a generalization, perhaps also universally quantified formulas.

8.6 Inductively Proved Polymorphic Theorems

An example for a global polymorphic theorem is associativity of the append-function on lists of any type: $\forall xs :: \texttt{List (a)}, ys :: \texttt{List (a)}, zs :: \texttt{List (a)}.\texttt{append}(xs, (\texttt{append}(ys, zs))) = \texttt{append}(\texttt{append}(xs, ys), zs)$. This, for example, also holds for list-elements of function type or if the elements are from an extension of $\mathcal{P}$.

Lemma 8.17. *Let $\mathcal{P}$ be a program and $\mathcal{P}'$ be an extension of $\mathcal{P}$. Every $\mathcal{P}'$-value v of a (non-quantified) polymorphic $\mathcal{P}$-type τ where all occurring type variables are in $\{\alpha_1, \ldots, \alpha_n\}$ is built using the following grammar:*
$W ::= \lambda x.E \mid c_{\mathcal{P}}\ W_1 \ldots W_n \mid E :: \alpha_i$ where $c_{\mathcal{P}}$ is a $\mathcal{P}$-constructor.

Proof. By induction on the size. The base case is included in the following case analysis:

- If v is an abstraction, then the claim holds.
- If $v = c\ v_1 \ldots v_n$, and c is a constructor from $\mathcal{P}$, then the claim also holds by induction hypothesis.
- If $v = c\ v_1 \ldots v_n$, and c is a constructor from $\mathcal{P}'$, but not a $\mathcal{P}$-constructor, then the type of v cannot contain a $\mathcal{P}'$-type constructor, due to the variable condition of the type of type-constructors. Hence the only possibility is that v has type α_i.

If an induction scheme for the proof of a universally quantified polymorphic equation is used, where the induction measure is "independent" of the type variables and only global theorems and globally correct transformations are used to prove the induction base and hypothesis, then also the universally quantified equation will be a global $\mathcal{P}$-theorem. For example, associativity of append is thus provable to be a global theorem (see [Wal09]).

We have to permit polymorphically typed expressions, i.e. where the type may contain type variables. Then the equality is defined as equality under all monomorphic type-substitutions and value-substitutions. The correctness of call-by-value δ, (beta) and `case`-reductions holds for this equality.

A simple induction scheme for global theorems is as follows, where we allow polymorphic types for the subexpressions.

- Assume a fixed $\mathcal{P}$.
- Assume there is a measure μ on values giving natural numbers, such that the subexpressions whose type is a type variable, do not contribute to the measure. This may be e.g. a weighted sum of the symbols not counting the values whose type is a type variable.
- Given a formula $\forall x_1, \dots, x_n.F$, perform the following two proof steps:
  - (base case) Prove $F[v_1, \dots, v_n]$ for all values v_i with $\mu(v_i) = 0$.
  - (induction step) For all $n > 0$ prove the following implication: If $F[v_1, \dots, v_n]$ holds for all v_i with $\mu(v_i) < n$, then $F[v_1, \dots, v_n]$ also holds for all v_i with $\mu(v_i) = n$, where only globally correct proof steps are permitted.

  Then the formula holds and is global for $\mathcal{P}$.

It is open whether the following holds:

> Let $\mathcal{P}$ be a program and F be a polymorphic theorem of the form $\forall x_1, \dots, x_n.s = t$. Then for all extensions $\mathcal{P}'$ of $\mathcal{P}$, the formula F is also a $\mathcal{P}'$-theorem.

References

[Ade09] Markus Axel Aderhold. *Verification of Second-Order Functional Programs.* PhD thesis, Computer Science Department, Technische Universität Darmstadt, Germany, 2009.

[Gor99] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoret. Comput. Sci.*, 228(1-2):5–47, October 1999.

[How89] D. Howe. Equality in lazy computation systems. In *4th IEEE Symp. on Logic in Computer Science*, pages 198–203, 1989.

[How96] D. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.

[KTU93] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–1018, 1993.

[Man05] Matthias Mann. Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. *Electron. Notes Theor. Comput. Sci.*, 128(1):81–101, 2005.

[MSS07] Matthias Mann and Manfred Schmidt-Schauß. How to prove similarity a precongruence in a broad class of non-deterministic call-by-need lambda calculi, 2007. submitted.

[SSNSS08] Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*, volume 273 of *IFIP*, pages 521–535. Springer, 2008.

[SSNSS09] Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. Frank report 33, Inst. f. Informatik, Goethe-University, Frankfurt, 2009.

[SSS09] Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for for higher-order calculi with sharing. *Theoret. Comput. Sci.*, In Press, Corrected Proof, 2009. DOI 10.1016/j.tcs.2009.12.001.

[SSSH09] David Sabel, Manfred Schmidt-Schauß, and Frederik Harwath. Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *INFORMATIK 2009, Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28.9 - 2.10.2009 in Lübeck*, volume 154 of *GI Edition - Lecture Notes in Informatics*, pages 369; 2931–45, October 2009. (4. Arbeitstagung Programmiersprachen (ATPS)).

[SWGA07] Andreas Schlosser, Christoph Walther, Michael Gonder, and Markus Aderhold. Context dependent procedures and computed types in verifun. *ENTCS*, 174(7):61–78, 2007.

[Wal94] Christoph Walther. Mathematical induction. In Dov M. Gabbay, Christopher J. Hogger, J. A. Robinson, and Jörg H. Siekmann (Eds.), editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–228. Oxford University Press, 1994.

[Wal09] Christoph Walther. VeriFun website, 2009. `http://www.inferenzsysteme.informatik.tu-darmstadt.de/verifun/`.

[WS05] Christoph Walther and Stephan Schweitzer. Reasoning about incompletely defined programs. In *12. LPAR '05*, LNCS 3835, pages 427–442, 2005.

A Type Derivation System

The type of unlabeled expressions is defined by using the inference system shown in figure 9. The explicit typing of variables is placed into a type environment, i.e. variables have no built-in type for this derivation system. An environment Γ is a (partial) mapping from variables and function symbols $f \in \mathcal{F}$ to types, where we assume that every function f is mapped to a type. The notation $\mathrm{Dom}(\Gamma)$ is the set of variables (and function names) that are mapped by Γ. The notation $\Gamma, x :: \tau$ means a new environment where $x \notin \mathrm{Dom}(\Gamma)$. The types of function symbols in $\mathcal{F}$ may also have a quantifier-prefix.

Definition A.1. *Given a program, the types Γ of the functions in f are called admissible, and all the functions are called* derivationally well-typed*, iff for every $f \in \mathcal{F}$ and the type $f :: T \in \Gamma$, we have $\Gamma \vdash d_f :: T$.*

Using the rules of the derivation system, a standard polymorphic type system can be implemented that computes types as greatest fixpoints using iterative processing. By standard reasoning, there is a most general type of every expression, From a typing point of view, the derivation system and the type-labeling are equivalent mechanisms.

Not that typability using the iterative procedure is undecidable, since the semi-unification problem [KTU93] can be encoded. Stopping the iteration, like in Milner's type system, leads to a decidable, but incomplete type system.

(Var) $$\mathit{\Gamma}, x :: S \vdash x :: S$$

(Fn) $$\mathit{\Gamma}, f :: S \vdash f :: S \quad \text{for } f \in \mathcal{F}$$

(App) $$\frac{\mathit{\Gamma} \vdash s :: S_1 \to S_2 \quad \mathit{\Gamma} \vdash t :: S_1}{\mathit{\Gamma} \vdash (s\ t) :: S_2}$$

(Abs) $$\frac{\mathit{\Gamma}, x :: S_1 \vdash s :: S_2}{\mathit{\Gamma} \vdash (\lambda x.s) :: S_1 \to S_2}$$

(Cons) $$\frac{\begin{array}{c}\mathit{\Gamma} \vdash s_1 :: S_1\ ; \ldots ; \mathit{\Gamma} \vdash s_n :: S_n \\ \mathit{\Gamma}, y :: \mathit{typeOf}(c) \vdash (y\ s_1 \ldots s_n) :: T\end{array}}{\mathit{\Gamma} \vdash (c\ s_1 \ldots s_n) :: T} \quad \text{if } ar(c) = n$$

(Case) $$\frac{\begin{array}{ll}\mathit{\Gamma} & \vdash s :: K\ S_1 \ldots S_m \\ \mathit{\Gamma}, x_{1,1} :: T_{1,1}, \ldots x_{1,n_1} :: T_{1,n_1} & \vdash t_1 :: T \\ \mathit{\Gamma}, x_{1,1} :: T_{1,1}, \ldots x_{1,n_1} :: T_{1,n_1} & \vdash (c_1\ x_{1,1} \ldots x_{1,n_1}) :: K\ S_1 \ldots S_m \\ \ldots & \ldots \\ \mathit{\Gamma}, x_{k,1} :: T_{k,1}, \ldots x_{k,n_k} :: T_{k,n_k} & \vdash t_k :: T \\ \mathit{\Gamma}, x_{k,1} :: T_{k,1}, \ldots x_{k,n_k} :: T_{k,n_k} & \vdash (c_k\ x_{k,1} \ldots x_{k,n_1}) :: K\ S_1 \ldots S_m\end{array}}{\mathit{\Gamma} \vdash (\texttt{case}_K\ s\ ((c_1\ x_{1,1} \ldots x_{1,n_1})\texttt{->}\,t_1) \ldots) :: T}$$

(Generalize) $$\frac{\mathit{\Gamma} \vdash t :: T}{\mathit{\Gamma} \vdash t :: \forall \mathcal{X}.T} \quad \begin{array}{l}\text{if } \mathcal{X} = FTV(T) \setminus \mathcal{Y} \\ \text{where } \mathcal{Y} = \bigcup\limits_{x \in FV(t)} \{FTV(S) \mid (x :: S) \in \mathit{\Gamma}\}\end{array}$$

(Instance) $$\frac{\mathit{\Gamma} \vdash t :: \forall \mathcal{X}.S_1}{\mathit{\Gamma} \vdash t :: S_2} \quad \text{if } \rho(S_1) = S_2 \text{ with } \mathrm{Dom}(\rho) \subseteq \mathcal{X}$$

Fig. 9. The type-derivation rules