# Reconstructing 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

**June 22, 2010**

**Abstract.** A logical framework consisting of a polymorphic call-by-value functional language and a first-order logic on the values is presented, which is a reconstruction of the logic of the verification system VeriFun. The reconstruction uses contextual semantics to define the logical value of equations. It equates undefinedness and nontermination, which is a standard semantical approach. The main results of this paper are: Meta-theorems about the globality of several classes of theorems in the logic, and proofs of global correctness of transformations and deduction rules. The deduction rules of VeriFun are globally correct if rules depending on termination are appropriately formulated. The reconstruction also gives hints on generalizations of the VeriFun framework: reasoning on nonterminating expressions and functions, mutual recursive functions and abstractions in the data values, and formulas with arbitrary quantifier prefix could be allowed.

## 1 Introduction

Proving properties of recursively defined functions by induction is a powerful method for validating properties of programs that operate on inductively defined data structures. Some influential early work is [BM75,KM86]. There are a couple of tools that are designed to perform this task automatically, or to give support in constructing a proof. Such a system is `VeriFun` (see [WS05b,SWGA07,Wal94]

and [Ver]). The logical framework of `VeriFun` 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, usually by induction. Since one is 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 the append-function on lists, or even more complex (encoded) theorems like the undecidability of the Halting Problem (see [Ver] and [BM84] for an early automated proof).

In general the system `VeriFun` requires that functions are proved to terminate before any other lemma about the function can be proved. This also holds for other prominent theorem provers like Isabelle/HOL [NPW02] and Coq [BC04]. Nevertheless, these systems provide special mechanisms to allow partial functions (see e.g. [Kra06,BK08]), like the selector-functions *head* and *tail* for lists, which are undefined for the empty list. In `VeriFun` these functions are seen as terminating since undefined expressions are seen as terminating. Details of this approach in `VeriFun` can be found in [WS05b,WS05a,Sch07] and for different work on the integration of partially defined functions into logics see e.g. [Far96,MS97]. The semantics of the logics of `VeriFun` in [WS05b,SWGA07,Ade09] is based on term-algebras over the data constructors and on the other hand on evaluation in a strict functional language. Undefined functional expressions are treated in a non-standard way as having any value of an appropriate type, which validates the intuitively correct theorems, but also validates nonsense theorems like $\mathit{tail}(\mathtt{Nil}) = \mathit{tail}(\mathit{tail}(\mathtt{Nil})) \implies \mathit{tail}(\mathtt{Nil}) = \mathtt{Nil}$.

The main goal of this paper is to provide a reconstruction (and adjustment and generalization) of the semantics of the logical system of `VeriFun` by treating undefined and nonterminating expressions in a standard semantics way. The goal is to obtain a logical framework of general interest that is at least as expressive as `VeriFun` and can handle partial functions and non-terminating programs. Our reconstruction starts from a programming language semantics view: based on the operational semantics of an ML-like core language which comprises polymorphic (and recursive) function definitions and data types, we use the well-established notion of contextual equivalence as semantics for programs (see e.g. [Plo77,BH99,Pit00,Pie02]). Contextual equivalence equates expressions if they have the same observational behavior (i.e. termination) if they are plugged inside any surrounding program context. An advantage is that contextual equivalence smoothly integrates the semantics of higher-order functions. Partially defined data-selectors are represented using `case`-expressions where nontermination is the result instead of an "undefined" in the case that the result is not defined, for example in $\mathit{tail}(\mathtt{Nil})$.

For the logical level we use a two-valued logic of predicate logic formulas where equalities between expressions are the atomic formulas. The semantics of these equations is given by contextual equivalence. The formulas are monomorphic where the quantification is over closed data values of the appropriate type.

An important observation is that only theorems are of interest that are invariant under any extension of the program by further function definitions or data types, i.e., conservativity is important. These theorems are called *global*, otherwise we call the theorems *local*.
Our main results are (i) conservativity theorems showing that several forms of local theorems are already global theorems; (ii) proofs of global correctness of deduction and transformations of almost all rules employed in `VeriFun`. In particular, call-by-value (beta) and (case) reductions are globally correct (Theorem 5.6), almost all deduction rules of `VeriFun` are also globally correct with respect to our semantics with the exception of call-by-name beta-reduction. In addition, several deduction rules concerning undefined expressions are valid, which are missing in `VeriFun`, and adapted call-by-name reductions and further deduction rules are correct (Theorem 5.6). Also several classes of theorems are shown to be conservative under extensions. An important class are universally quantified equations (Theorem 6.3) and general monomorphic theorems if functions do not occur in the data (Theorem 6.6).
A side effect of the reconstruction are the following generalizations: higher-order values may also occur in data objects, in the logical level functions that may not terminate on certain arguments are permitted, and mutual recursive function definitions are possible on the top level.
To establish these results we introduce proof techniques for contextual equality in combination with polymorphic types (for a call-by-need calculus see also [SSSH09]), which allow to prove a CIU-Theorem from context lemmas (for a general approach see also [SSS10]), and an adaptation of the subterm property of simply-typed lambda-calculi.
An interesting generalization of the logic expressiveness are polymorphic formulas (the quantified type may have type variables). These are expressible in `VeriFun`, provided the quantifier prefix is $\forall^*$. Conservativity of polymorphic theorems is false in general. We conjecture that polymorphic theorems that are universally quantified polymorphic equations also hold in 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 explain the different variants of the CIU-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 certain preconditions hold. Finally, in Section 6 we explain the logic, its semantics and analyze some conservativity properties and state open questions. Missing proofs can be found in the appendix.

## 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 6. 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 $\mathcal{D} = \bigcup_{K \in \mathcal{K}} D_K$. Since terms are constructed under polymorphic typing restrictions, we first define types, data and type constructors and then the expression level.

### 2.1 Syntax of Types

Types $T$ are defined by: $T ::= X \mid (T_1 \to T_2) \mid (K\ T_1 \ldots T_{ar(K)})$, 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 $T_1 \to T_2$ are called *arrow types*, and types $(K\ T_1 \ldots T_{ar(K)})$ are called *constructed types*. We also will use *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 $typeOf(c_{K,i})$ 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 the variables $X_i$ occur as free type variables in $T_{K,i,1}, \ldots, T_{K,i,m_i}$.

### 2.2 Syntax of Expressions of *P*

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

$$\begin{aligned} s, s_i, t \in \mathit{Expr} &::= x \mid f \mid (s\ t) \mid \lambda x.s \mid (c_i\ s_1 \ldots s_{ar(c_i)}) \\ &\quad \mid (\mathtt{case}_K\ s\ \mathit{Alt}_1 \ldots \mathit{Alt}_n) \qquad \text{where } n = |D_K| \\ \mathit{Alt}_i &::= ((c_i\ x_1 \ldots x_{ar(c_i)})\ \texttt{->}\ s_i) \end{aligned}$$

Note that data constructors can only be used with all their arguments present. We assume that there is a $\mathtt{case}_K$ for every type constructor $K \in \mathcal{K}$. The $\mathtt{case}_K$-construct is assumed to have a case-alternative $((c_i\ x_1 \ldots x_{ar(c_i)})\ \texttt{->}\ s_i)$ 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. E.g. for the context $C = \lambda x.[\cdot]$ it holds $C[\lambda y.x] = \lambda x.\lambda y.x$.
A *value* $v$ is defined as $v, v_i \in \mathit{Val} ::= x \mid \lambda x.s \mid (c\ v_1 \dots v_n)$, i.e. a variable, an abstraction, or a constructor-expression $(c\ v_1 \dots v_n)$, where the immediate subexpressions are also values. For instance, $\lambda x.\mathtt{True} : (\lambda y.\mathtt{False} : \mathtt{Nil})$ is a value while the list $((\lambda x.\mathtt{True})\ \mathtt{False}) : \mathtt{Nil}$ is not a value, since the subexpression $((\lambda x.\mathtt{True})\ \mathtt{False})$ is not a value.
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*.

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

1. *a signature $(\mathcal{F}, \mathcal{K}, \mathcal{D})$ where $\mathcal{K} \neq \emptyset$.*
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 $f = d_f$.*

*Accordingly for a given program $\mathcal{P}$ we call the expressions* $\mathcal{P}$-expressions*, the values* $\mathcal{P}$-values*, the contexts* $\mathcal{P}$-contexts*, and the types* $\mathcal{P}$-types*.*

For instance, the identity function can be defined as $id = \lambda x.x$ where $id \in \mathcal{F}$. Note that it is allowed that functions are defined mutually recursive. For example, if $map, head, bot \in \mathcal{F}$, these functions can be defined as:

$$\begin{aligned}
map &= \lambda f, xs.\mathtt{case}_{\mathrm{List}}\ xs\ ((y : ys)\ \texttt{->}\ (f\ y : map\ f\ xs))\ (\mathtt{Nil}\ \texttt{->}\ \mathtt{Nil})\\
head &= \lambda xs.\mathtt{case}_{\mathrm{List}}\ xs\ (y : ys\ \texttt{->}\ y)\ (\mathtt{Nil}\ \texttt{->}\ (bot\ \mathtt{Nil}))\\
bot &= \lambda x.\ (bot\ x)
\end{aligned}$$

### 2.3 Typing of Expressions

We extend expressions now with type labels and distinguish between usual expressions and expressions in function definitions: We assume that the definitional expressions $d_f$ are polymorphically typed in a standard way, where every subexpression is annotated with a type, and that the $\mathcal{P}$-expressions are monomorphically typed. For every $f \in \mathcal{F}$ the pair $(f, d_f)$ is labeled with a perhaps quantified type. We assume that occurrences of defined function symbol $f$ are labeled with an instance type of $f$. All the rules of the monomorphic system *MonoTp* are standard (see Appendix A). For instance, for case-expressions the rule is

$$\left.\begin{array}{l}
(\mathtt{case}_K\ s :: S\ ((c_{K,1}\ x_{1,1} \dots x_{1,n_1}) :: S\ \texttt{->}\ t_1 :: T)\\
\qquad\qquad \dots\\
\qquad\qquad ((c_{K,m}\ x_{m,1} \dots x_{m,n_m}) :: S\ \texttt{->}\ t_m :: T))
\end{array}\right\} \mapsto T$$

**Definition 2.2.** *We say a program $\mathcal{P}'$* extends *the program $\mathcal{P}$ (denoted with $\mathcal{P}' \sqsupseteq \mathcal{P}$), 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 $\mathcal{P}'$.*

(beta) $R[((\lambda x.s)\ v)] \to R[s[v/x]]$ where $v$ is a value

(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) $R[(\texttt{case}\ (c\ v_1 \dots v_n)\ \dots ((c\ y_1 \dots y_n)\ \texttt{->}\ s) \dots)]$
$\to R[s[v_1/y_1, \dots, v_n/y_n]]$ where $v_1, \dots, v_n$ are values

**Fig. 1.** Standard Reduction Rules

## 3 Operational Semantics

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

$$R \in \mathcal{R} ::= [\cdot] \mid (R\ s) \mid (v\ R) \mid \texttt{case}_K\ R\ alts \mid (c\ v_1\ \dots\ v_i\ R\ s_{i+2} \dots\ s_n)$$

where $s$, $s_i$ are $\mathcal{P}$-expressions and $v$, $v_i$ are $\mathcal{P}$-values.
Standard reduction rules are defined in Fig. 1 without mentioning all types.

**Definition 3.1.** *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 a value by its evaluation, denoted by $s\downarrow$. Otherwise, we say $s$* diverges*, denoted by $s\Uparrow$.*

By induction on the term structure it is easy to verify that for every expression, there is at most one standard reduction possible. One can also verify that reduction is type-safe: reduction of expressions preserves the type of the expressions, i.e. $t :: T$ and $t \to t'$ implies that $t' :: T$, and a progress lemma holds, i.e. every closed and well-typed expression without reduction is a closed value.

### 3.1 Assumptions on Valid Programs

**Assumption 3.2.** *We assume that for every (monomorphic) type $T$ of every program $\mathcal{P}$ there is at least one closed value of type $T$.*

*Remark 3.3.* This excludes types like the type `Foo` with one constructor `foo` : `Foo` $\to$ `Foo`. The only potentially closed value would be an infinitely nested expression `foo(foo(...))`, which of course does not exist.

**Assumption 3.4.** *We assume that for every program $\mathcal{P}$ and for every $\mathcal{P}$-type $\tau$ there is a closed diverging expression, denoted as $\bot^\tau$.*

The second assumption is satisfied if there is a single definition $f = (\lambda x.f\ x) :: \forall a, b.a \to b$. Then the expressions $\bot^\tau := (f\ v)^\tau$ do not converge, where $v$ is any closed value. This also allows us to construct values $\lambda x.\bot^\tau$ of any given function-type. Thus the assumptions can easily be satisfied in a finite program. The expressions $\bot^\tau$ allow us to define partial functions. For instance, the function *tail* could be defined as

$$tail = \lambda xs.\texttt{case}_{\text{List}}\ xs\ (y : ys\ \texttt{->}\ ys)\ (\texttt{Nil}\ \texttt{->}\ \bot^{\texttt{List}}).$$

### 3.2 Equivalence of Expressions

The conversion relation defined by applying (beta), (case) and (delta) in every context is too weak to justify sufficiently many equations. E.g., only expressions of the same asymptotic complexity class are equated (see [SGM02]). So we will use contextual equivalence that observes termination in all closing contexts, where we define a local (for $\mathcal{P}$), and a global variant (for all extensions of $\mathcal{P}$).

**Definition 3.5.** *Assume 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 $C[s]\downarrow \implies C[t]\downarrow$. We also define $s \sim_{\mathcal{P}\forall,T} t$ 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 $\sim_{\mathcal{P},T}$.*

It is easy to verify that $\leq_{\mathcal{P},T}$ and $\leq_{\mathcal{P}\forall,T}$ are precongruences, and $\sim_{\mathcal{P},T}$ and $\sim_{\mathcal{P}\forall,T}$ are congruences.

*Example 3.6.* 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 $s = \texttt{Nil}, t = (\texttt{case}_{\text{List}}\ x\ ((y : z)\,\texttt{->}\,\texttt{Nil})\ (\texttt{Nil}\,\texttt{->}\,\texttt{Nil}))$. 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 $s \sim_{\mathcal{P},T} t$.

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 equation $s \sim_{\mathcal{P},T} t$ holds. $\mathcal{T}$ is *globally correct* iff for all $(s, t) \in \mathcal{T}$ of type $T$ the equation $s \sim_{\mathcal{P}\forall,T} t$ holds.

## 4 A CIU-Theorem

In this section we assume that $\mathcal{P}$ is a fixed program and argue that a so-called CIU-Theorem (for other calculi see e.g. [MT91,FH92]) holds, which allows easier proofs of contextual equivalence, i.e. it is sufficient to take only closed reduction contexts and closing value substitutions into account, in order to show contextual equality. In the subsequent section we will extend these results to all programs extending the program $\mathcal{P}$. The following theorem is formulated in stronger form for *F-free* expressions and substitutions, which means that they may contain $\bot$-symbols, but do not contain other function constants from $\mathcal{F}$.

**Theorem 4.1 (CIU-Theorem F-free).** *For $\mathcal{P}$-expressions $s, t :: T$: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ for all F-free $\mathcal{P}$-value substitutions $\sigma$ and for all 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.*

In the appendix (Proposition B.11) we show:

**Proposition 4.2.** *The reductions (beta), (delta), and (case) are correct program transformations in $\mathcal{P}$. I.e., if $s \to t$ by (beta), (delta), or (case), then $C[s] \to C[t]$ is a correct transformation.*

Note that ordinary (i.e. call-by-name) beta-reduction is in general not correct, for instance $(\lambda x.\mathtt{True})\ \bot$ is equivalent to $\bot :: \mathtt{Bool}$, however, using a call-by-name beta-reduction results in `True`, which is obviously not equivalent to $\bot$. Note also that in `VeriFun` call-by-name beta-reduction is used. This use is correct, since the `VeriFun`-logic assumes termination of all functions.
We end this section by analyzing so-called $\Omega$-expressions, i.e. terms that diverge after closing them by an arbitrary value substitution.

**Definition 4.3.** *We say an expression $s$ is an* $\Omega$-expression *iff for all value substitutions $\sigma$ 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 $\Omega$-expression of the corresponding type.*

The property of being an $\Omega$-expression inherits to reduction contexts, i.e. if $s :: \tau$ is an $\Omega$-expression, and $R[\cdot :: \tau]$ a reduction context, then $R[s]$ is also an $\Omega$-expression (see Appendix B.4, Proposition B.17). The CIU-Theorem also implies that $\Omega$-expressions are least elements w.r.t. contextual ordering, and that $\Omega$-expressions of the same type form a single equivalence class:

**Corollary 4.4.** *Let $s, t :: \tau$ and let $s$ be an $\Omega$-expression. Then $s \leq_{\mathcal{P},\tau} t$. If also $t$ is an $\Omega$-expression, then $s \sim_{\mathcal{P},\tau} t$.*

## 5 Global Correctness of Program Transformations

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.
We will show that the local CIU-equivalence, i.e. testing only $\mathcal{P}$-value substitutions, and $\mathcal{P}$-reduction contexts which are additionally $F$-free, coincides with the (global) contextual equivalence taking into account all extensions of programs.
As a first step we show that it is sufficient to take into account closed expressions:

**Lemma 5.1.** *Let $s, t$ be (open) $\mathcal{P}$-expressions of type $T$. Then $s \leq_{\mathcal{P},T} t$ iff for all closing $\mathcal{P}$-value-substitutions $\sigma$: $\sigma(s) \leq_{\mathcal{P},T} \sigma(t)$.*

*Proof.* If $s \leq_{\mathcal{P},T} t$, then $\sigma(s) \leq_{\mathcal{P},T} \sigma(t)$ for closing value substitutions $\sigma$ holds, since beta-reduction is correct. The converse follows from the CIU-Theorem. □

We provide criteria on contextual approximation for closed expressions:

**Lemma 5.2.** *Let $s, t$ be closed expressions of constructed type $T$. Then $s \leq_{\mathcal{P},T} t$ iff $s\Uparrow$, or $s \xrightarrow{*} c\ v_1 \ldots v_n$ and $t \xrightarrow{*} c\ w_1 \ldots w_n$ for some constructor $c$, and $v_i \leq_{\mathcal{P},T_i} w_i$ for $i = 1, \ldots, n$.*

*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 a closed value 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 precongruence and due to Corollary 4.4. □

**Proposition 5.3.** *For closed expressions $s, t$ of function type $T$: $s \leq_{\mathcal{P},T} t$ iff $s\Uparrow$, or $s \xrightarrow{*} \lambda x.s'$ and $t \xrightarrow{*} \lambda x.t'$ and $s'[v/x] \leq_{\mathcal{P},T} t'[v/x]$ for all closed F-free $\mathcal{P}$-values $v$.*

Now we are able to extend the CIU-Theorem to all extensions $\mathcal{P}'$ of $\mathcal{P}$ where only F-free $\mathcal{P}$-values substitutions and reduction contexts need to be taken into account. Note that the difficult part of the proof (see Appendix C) is to show that type and data constructors of the extended program $\mathcal{P}'$ need not be considered. Then the local CIU-Theorem 4.1 implies the following theorem:

**Theorem 5.4 (CIU-Theorem F-free and global).** *Let $\mathcal{P}'$ be an extension of $\mathcal{P}$. For $\mathcal{P}$-expressions $s, t :: T$, the implication $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ holds for all F-free $\mathcal{P}$-value substitutions $\sigma$ 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.*

Hence, on $\mathcal{P}$-expressions the local and global contextual approximations coincide:

**Theorem 5.5 (Local preorder is global).** *Let $\mathcal{P}$ be a program and $s, t :: T$ be $\mathcal{P}$-expressions. Then $s \leq_{\mathcal{P},T} t$ iff $s \leq_{\mathcal{P}\forall,T} t$.*

*Proof.* This follows from Theorem 5.4, since the conditions mention only $\mathcal{P}$-expressions and substitutions independent of the extensions $\mathcal{P}'$ of $\mathcal{P}$. □

We end this section by proving (global) correctness of some program transformations. In Figs. 2, 3 and 4 the so-called *VN-reductions* are defined, where a sequentializing construct is used as $(s; r)$, called seq-expression, which means $((\lambda_.r)\ s)$. Operationally, this means to first evaluate $s$ and if it evaluates to a value, then evaluate $r$ and return its value.
These rules can be used as normalization rules for open expressions and values (see Appendix C), especially in the deduction system, but they are also of interest as program transformations. In the following we will argue that the VN-reductions and the reduction rules of Fig. 1 are globally correct. Using the arguments above and the F-free CIU-Theorem 5.4, the following is obtained:

**Theorem 5.6.** *The transformations (beta), (delta), and (case), i.e. the call-by-value reduction rules, and the transformations in Figs. 2, 3 and 4 are globally correct program transformations in $\mathcal{P}$.*

*Proof.* Global correctness of the call-by-value reduction rules follows from Proposition 4.2 and Corollary 5.5. The other reductions are proved correct in the appendix (Theorem C.16). □

| | |
|---|---|
| $\texttt{Bot}\ s \to \texttt{Bot}$ | $(c\ \ldots \texttt{Bot} \ldots) \to \texttt{Bot}$ |
| $s\ \texttt{Bot} \to \texttt{Bot}$ | $(t; \texttt{Bot}) \to \texttt{Bot}$ |
| $\texttt{case}_K\ s\ (p_1 \to \texttt{Bot}) \ldots\ (p_n \to \texttt{Bot}) \to \texttt{Bot}$ | $(\texttt{Bot}; t) \to \texttt{Bot}$ |

**Fig. 2.** Bot-reduction rules

| | | | |
|---|---|---|---|
| seqlam | $((\lambda x.s); t) \to t$ | seqc | $((c\ s_1 \ldots s_n); s) \to (s_1; (\ldots (s_n; s) \ldots))$ |
| seqx | $(x; s) \to s$ | caseseq | $(\texttt{case}_K\ (r; s)\ alts) \to (r; (\texttt{case}_K\ s\ alts))$ |
| seqseq | $((s_1; s_2); s_3) \to (s_1; (s_2; s_3))$ | VNbeta | $((\lambda x.s)\ t) \to (t; s[t/x])$ |
| seqapp | $((s_1; s_2)\ s_3) \to (s_1; (s_2\ s_3))$ | | |

VNcase $\texttt{case}_K\ (c\ s_1 \ldots s_n)\ \ldots\ (c\ x_1 \ldots x_n) \texttt{->}\ t\ \ldots$
$\to (s_1; (\ldots (s_n; t[s_1/x_1, \ldots, s_n/x_n])))$

**Fig. 3.** Adapted call-by-name-reduction rules

caseapp $((\texttt{case}_K\ t_0\ (p_1 \texttt{->} t_1) \ldots (p_n \texttt{->} t_n))\ r)$
$\to (\texttt{case}_K\ t_0\ (p_1 \texttt{->} (t_1\ r)) \ldots (p_n \texttt{->} (t_n\ r)))$

casecase $(\texttt{case}_K\ (\texttt{case}_{K'}\ t_0\ (p_1 \texttt{->} t_1) \ldots (p_n \texttt{->} t_n))\ \ (q_1 \texttt{->} r_1) \ldots (q_m \texttt{->} r_m))$
$\to (\texttt{case}_{K'}\ t_0\ (p_1 \texttt{->} (\texttt{case}_K\ t_1\ (q_1 \texttt{->} r_1) \ldots (q_m \texttt{->} r_m)))$
$\ldots$
$(p_n \texttt{->} (\texttt{case}_K\ t_n (q_1 \texttt{->} r_1) \ldots (q_m \texttt{->} r_m))))$

seqcase $((\texttt{case}_K\ t\ (q_1 \texttt{->} r_1) \ldots (q_m \texttt{->} r_m)); r)$
$\to (\texttt{case}_K\ t\ (q_1 \texttt{->} (r_1; r)) \ldots (q_m \texttt{->} (r_m; r)))$

**Fig. 4.** Case-Shifting Transformations

## 6 The Logic and Induction

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

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

### 6.1 The Semantics

Let $\mathcal{P}'$ be an extension of the program $\mathcal{P}$ and $T$ be a $\mathcal{P}$-type. The set $\mathcal{M}_{\mathcal{P}',T}$ is the set of all closed $\mathcal{P}'$-values of type $T$. Note that our assumptions imply that for every $T$ there is a value of this type and thus also $\mathcal{M}_{\mathcal{P}',T} \neq \emptyset$.

**Definition 6.1.** *Let $\mathcal{P}$ be a program and $\mathcal{P}' \sqsupseteq \mathcal{P}$. The interpretation function $I_{\mathcal{P}'}$ w.r.t. $\mathcal{P}'$ of closed monomorphic $\mathcal{P}$-formulas is defined as follows:*

$I_{\mathcal{P}'}(s = t) = \texttt{True}$ *if* $s \sim_{\mathcal{P}',\tau} t$ *for expressions* $s, t :: \tau$
$I_{\mathcal{P}'}(s = t) = \texttt{False}$ *if* $s \not\sim_{\mathcal{P}',\tau} t$ *for expressions* $s, t :: \tau$
$I_{\mathcal{P}'}(A \wedge B) = I_{\mathcal{P}'}(A) \wedge I_{\mathcal{P}'}(B)$
$I_{\mathcal{P}'}(A \vee B) = I_{\mathcal{P}'}(A) \vee I_{\mathcal{P}'}(B)$
$I_{\mathcal{P}'}(\neg A) = \neg I_{\mathcal{P}'}(A)$
$I_{\mathcal{P}'}(\forall x :: \tau.F) = \texttt{True}$ *if for all* $a \in \mathcal{M}_{\mathcal{P}',\tau} : I_{\mathcal{P}'}(F[a/x]) = \texttt{True}$
$I_{\mathcal{P}'}(\exists x :: \tau.F) = \texttt{True}$ *if for some* $a \in \mathcal{M}_{\mathcal{P}',\tau} : I_{\mathcal{P}'}(F[a/x]) = \texttt{True}$

*A closed monomorphic $\mathcal{P}$-formula $F$ is a $\mathcal{P}'$*-tautology *(*$\mathcal{P}'$-theorem*,* monomorphic $\mathcal{P}'$-theorem*) iff $I_{\mathcal{P}'}(F) =$ `True` and it is a* global tautology *iff it is a $\mathcal{P}'$*-tautology *for all extensions $\mathcal{P}'$ of program $\mathcal{P}$.*

Of course only global tautologies are of interest. However, a verifier cannot compute all program extensions, and thus we will show that for a large set of monomorphic formulas it is sufficient to only take the program $\mathcal{P}$ into account for the (global) tautology proof. Since local and global equivalences coincide (Theorem 5.5), it is promising to look for classes of formulas where it is sufficient to test the values $M_{\mathcal{P},T}$ instead of all the values of $M_{\mathcal{P}',T}$ for every $\mathcal{P}' \sqsupseteq \mathcal{P}$.

*Example 6.2.* Given appropriate definitions of the data type `nat` with two constructors $0, \texttt{succ}$, where `pred`, defined as $\lambda x.\texttt{case}_{\texttt{nat}}\ x\ (0 \rightarrow \bot)\ (\texttt{succ}\ y \rightarrow y)$, is a function that acts like a selector for `succ`, and where also addition $+$ is recursively defined, the formula $\forall x :: \texttt{nat}.\exists y :: \texttt{nat}.x + \texttt{succ}(0) = y$ is a tautology: 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.

## 6.2 Universally Quantified Formulas: Conservativity

The following theorem shows that it not always necessary to consider all extensions of a program: Monomorphic formulas of the form $\forall x_1 :: T_1, \ldots, x_n :: T_n\ .\ s = t$ are global tautologies iff they are $\mathcal{P}$-tautologies, i.e.:

**Theorem 6.3.** *Let $\mathcal{P}$ be a program and $F := \forall x_1 :: T_1, \ldots, x_n :: T_n\ .\ s = t$ be a monomorphic $\mathcal{P}$-theorem. Then for all $\mathcal{P}' \sqsupseteq \mathcal{P}$, 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 5.4. □

Thus 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 4.2 and 5.3. In the following we investigate extensions of Theorem 6.3. First we look for values without higher-order subexpressions, like Peano-numbers, Booleans and lists of Peano-numbers.

**Definition 6.4.** *A type $T$ is a* DT-type*, if every closed value of type $T$ is only built from data constructors.*

**Lemma 6.5.** *Let $\mathcal{P}'$ be an extension of $\mathcal{P}$. If $v :: T$ is a $\mathcal{P}'$-value, where $T$ is a DT-type and a $\mathcal{P}$-type. Then $v$ is a $\mathcal{P}$-expression.*

**Theorem 6.6.** *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 6.5, and from Theorem 5.4, which among others shows that all closed $\sim$-equalities are global. □

**Proposition 6.7.** *Let $\mathcal{P}$ be a program such that every computable function on DT-types can be expressed as abstraction using the functions of $\mathcal{P}$ and let $\mathcal{P} \sqsubseteq \mathcal{P}'$. Then for every $\mathcal{P}'$-value $v$ of $\mathcal{P}$-type $\tau = \tau_1 \to \ldots \to \tau_n$ where all $\tau_i$ 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 $\tau_i$ 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 6.8.** *If there is a polymorphic fixpoint function* $\text{fix} : (\alpha \to \alpha) \to (\alpha \to \alpha)$ *with* $\text{fix} = \lambda f.\lambda x.(f\ (\lambda x.\text{fix}\ f\ x)\ x)$ *in $\mathcal{P}$ then the expressivity-assumption in Proposition 6.7 is satisfied and thus the claim of Proposition 6.7 holds.*

**Theorem 6.9.** *Let $\mathcal{P}$ be a program such that there is a fixpoint function as in Corollary 6.8 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 $\tau_i$ 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 6.8. □

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

### 6.3 Conservativity by Adding Definedness

In this section we consider formulas which ensure that all expressions in equations are defined. The intention is to cover the monomorphic formulas which are in scope of the `VeriFun`-system (where termination is an apriori requirement). Given a program $\mathcal{P}$ we define $\mathcal{P}_D \sqsupset \mathcal{P}$, including for every DT-type $T$ a binary function $eq_T :: T \to T \to \texttt{Bool}$ such that for all closed values $v, w :: T$: $v \sim w \implies eq_T\ v\ w \xrightarrow{*} \texttt{True}$ and $v \not\sim w \implies eq_T\ v\ w \xrightarrow{*} \texttt{False}$. Also, functions *and*, *or*, *not* on the Boolean values `True`, `False` are defined in $\mathcal{P}_D$. The function $\lambda x^T.\texttt{True}$, abbreviated as $\mathit{defined}_T$, has the following property: $(\mathit{defined}_T\ s) \xrightarrow{*} \texttt{True}$ for every converging expression of DT-type $T$. The function never produces `False`, but does not terminate if the argument is not terminating.

**Definition 6.10.** *The translation $B$ is defined as: $B(\wedge) \equiv \lambda x, y.\mathit{and}\ x\ y$, $B(\vee) \equiv \lambda x, y.\mathit{or}\ x\ y$, $B(\neg) \equiv \lambda x.\mathit{not}\ x$, and $B(s =_T t) \equiv eq_T\ s\ t$. A quantifier-free formula $F$ is translated into the equation ($eq_{\texttt{Bool}}\ B(F)\ \texttt{True}$).*

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 deMorgan's law. 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 all expressions $s, t$ in every equation $s = t$ in the formula evaluate to a closed value, 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:
For a $\mathcal{P}$-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 (w.r.t. $\mathcal{P}$) be $\forall x_1, \ldots, x_n.(Def(F) \implies F)$, where $Def(F)$ is the formula $defined(s_1) = \mathtt{True} \wedge \ldots \wedge defined(s_n) = \mathtt{True}$, where $s_i, i = 1, \ldots, n$ are all the expressions that occur as top-expressions in equations of $F$. We show that the theorems in the scope of `VeriFun` are global:

**Theorem 6.11.** *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 $\mathcal{P}' \sqsupseteq \mathcal{P}$, the formula $\forall x_1 :: T_1, \ldots, x_n :: T_n.Def(F) \implies F$ is also a theorem; i.e. it is a 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 $\lambda x_1, \ldots, x_n.B(Def(F) \implies F) \sim_{\mathcal{P}_D,T} \lambda x_1, \ldots, x_n.B(Def(F))$: If $\sigma(s_i)$ is equivalent to a value for all $i$, then the claim is obvious. If some $\sigma(s_i)$ is undefined, then the equation $defined(s_1) = \mathtt{True}$ 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 5.4 for any extension $\mathcal{P}'_D$ of $\mathcal{P}_D$. The latter again implies that $\forall x_1 :: T_1, \ldots, x_n :: T_n\ .Def(F) \implies F$ is a closed monomorphic theorem w.r.t. $\mathcal{P}'$. Now the CIU-Theorem implies that the formula is a global $\mathcal{P}$-tautology. □

### 6.4 Polymorphic Formulas

In this section we consider polymorphic formulas and show that their conservativity does in general not hold. Then the section illustrates how to prove global polymorphic theorems directly. *Polymorphic formulas* are like monomorphic formulas, where type variables are permitted in the type of the quantified variables, and the expressions are polymorphically typed (as in the defining values), polymorphic expressions are permitted in the formulas, where in equations $s = t$, the expressions $s, t$ must be of the same polymorphic type. The semantics has to be extended as follows:

**Definition 6.12.** *For a program $\mathcal{P}$ and an extension $\mathcal{P}'$ of $\mathcal{P}$ a polymorphic $\mathcal{P}$-formula $F$ is a $\mathcal{P}'$-tautology (a* polymorphic $\mathcal{P}'$-theorem*), if for every $\mathcal{P}'$-type substitution $\rho$ that instantiates every type variable in $F$ with a monomorphic $\mathcal{P}'$-type, the formula $\rho(F)$ is a global tautology. $F$ is a* global $\mathcal{P}$-theorem*, iff it is a polymorphic $\mathcal{P}'$-theorem for all extensions $\mathcal{P}'$ of $\mathcal{P}$.*

In general, not 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, but no other data types. Then the following formula $F$ is a polymorphic $\mathcal{P}$-theorem:

$$F := \forall x_1 :: a, x_2 :: a, x_3 :: a.((x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3) \implies \exists x :: a.x \neq x_1 \wedge x \neq x_2 \wedge x \neq x_3),$$

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 the set $\{\texttt{red}, \texttt{blue}, \texttt{green}\}$ as data constructors. Then $F$ is false in $\mathcal{P}'$. Hence, $F$ is not a global theorem.
An example for a global polymorphic theorem is associativity of the append-function on lists of any type:

$$F := \forall xs :: \texttt{List}\ a, ys :: \texttt{List}\ a, zs :: \texttt{List}\ a. \texttt{append}(xs, (\texttt{append}(ys, zs))) = \texttt{append}(\texttt{append}(xs, ys), zs)$$

We sketch the tautology proof: Let $F$ be a $\mathcal{P}$ formula and $\mathcal{P}' \sqsupseteq \mathcal{P}$. For all $\mathcal{P}'$-type substitutions $\rho$ and for all $u \in \mathcal{M}_{\mathcal{P}',\rho(a)}$ and all $r, s, t \in \mathcal{M}_{\mathcal{P}',\texttt{List}\ \rho(a)}$ it holds

i) $\texttt{append}(\texttt{Nil}, (\texttt{append}(s, t))) \sim_{\mathcal{P}',\texttt{List}\ \rho(a)} \texttt{append}(\texttt{append}(\texttt{Nil}, s), t)$
ii) $\texttt{append}(r, (\texttt{append}(s, t))) \sim_{\mathcal{P}',\texttt{List}\ \rho(a)} \texttt{append}(\texttt{append}(r, s), t)$
$\implies \texttt{append}(u : r, (\texttt{append}(s, t))) \sim_{\mathcal{P}',\texttt{List}\ \rho(a)} \texttt{append}(\texttt{append}(u : r, s), t)$

This follows by global correctness of the reduction rules where the proof never computes the value of any $\rho(T)$. Items i) and ii) are the usual parts in an induction scheme and thus for all $\mathcal{P}'$-type substitutions $\rho$ and all $r, s, t \in \mathcal{M}_{\mathcal{P}',\texttt{List}\ \rho(a)}$ it holds: $\texttt{append}(r, (\texttt{append}(s, t))) \sim_{\mathcal{P}',\texttt{List}\ \rho(a)} \texttt{append}(\texttt{append}(r, s), t)$.
Theorem 5.5 now shows that $\rho(F)$ is a global monomorphic theorem for any $\mathcal{P}'$-type substitution $\rho$. Hence, $F$ is a global polymorphic theorem.
In summary this illustration shows that a universally quantified polymorphic formula is a global theorem, if the induction and the induction measure are "independent" of the type variables and only global $\mathcal{P}$-theorems and globally correct $\mathcal{P}$-transformations are used to prove the induction base and hypothesis.
However, we have to leave open whether the following holds:

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

## 7 Conclusion

A reconstruction and adjustment of the logic and semantics used by the prover VeriFun is presented. The analysis exhibits that a contextual semantics is only a slight change in the intended semantics proposed by the developers yet permits more correct program transformations that could also be used in the proof system and permits also several generalizations like higher-order data and a treatment of nonterminating expressions. Conservativity theorems are shown for interesting

classes of formulas, and also several open questions are formulated, like globality of monomorphic theorems and globality of universally quantified polymorphic equations which are local theorems. Besides working on the open questions, a possible direction of future work is to apply the techniques to other systems of typed functional programming languages.

## References


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

[Bar84] H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York, 1984.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BH99] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. *Inf. Comput.*, 155(1-2):3–63, 1999.

[BK08] Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in Coq. In *PPDP '08: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 89–96, New York, NY, USA, 2008. ACM.

[BM75] Robert S. Boyer and J. Strother Moore. Proving theorems about lisp functions. *J. ACM*, 22(1):129–144, 1975.

[BM84] Robert S. Boyer and J. Strother Moore. A mechanical proof of the unsolvability of the halting problem. *J. ACM*, 31(3):441–458, 1984.

[Far96] W. M. Farmer. Mechanizing the traditional approach to partial functions. In W. Farmer, M. Kerber, and M. Kohlhase, editors, *CADE-13 Workshop on the Mechanization of Partial Functions*, pages 27–32, 1996.

[FH92] Matthias Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.*, 103:235–271, 1992.

[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.

[KM86] Deepak Kapur and David R. Musser. Inductive reasoning with incomplete specifications (preliminary report). In *Proceedings, Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, Massachusetts, USA*, pages 367–377. IEEE Computer Society, 1986.

[Kra06] Alexander Krauss. Partial recursive functions in higher-order logic. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer, 2006.

[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.

[MS97] Olaf Müller and Konrad Slind. Treating partiality in a logic of total functions. *Comput. J.*, 40(10):640–652, 1997.

[MSS10] Matthias Mann and Manfred Schmidt-Schauß. Similarity implies equivalence in a class of non-deterministic call-by-need lambda calculi. *Information and Computation*, 208(3):276 – 291, 2010.

[MT91] Ian Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *J. Funct. Programming*, 1(3):287–327, 1991.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Pit00] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Math. Structures Comput. Sci.*, 10:321–359, 2000.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5:223–255, 1977.

[Sch07] Dirk Stephan Schweitzer. *Symbolische Auswertung und Heuristiken zur Verifikation funktionaler Programme*. PhD thesis, TU Darmstadt, Juni 2007.

[SGM02] David Sands, Jörgen Gustavsson, and Andrew Moran. Lambda calculi and linear speedups. In *The Essence of Computation 2002*, pages 60–84, 2002.

[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.

[SSS10] Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.*, 411(11-13):1521 – 1541, 2010.

[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.

[Ver] VeriFun Website. www.inferenzsysteme.informatik.tu-darmstadt.de/verifun/.

[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.

[WS05a] Christoph Walther and Stephan Schweitzer. Automated termination analysis for incompletely defined programs. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, volume 3452 of *Lecture Notes in Comput. Sci.*, pages 332–346. Springer, 2005.

[WS05b] Christoph Walther and Stephan Schweitzer. Reasoning about incompletely defined programs. In *12. LPAR '05*, LNCS 3835, pages 427–442, 2005.

# A Typing of Expressions

## A.1 Type Derivation System

The type of unlabeled expressions is defined by using the inference system shown in Fig. 5. 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 $\Gamma$ 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 $\Gamma$. The notation $\Gamma, x :: \tau$ means a new environment where $x \notin \mathrm{Dom}(\Gamma)$. Given a quantified type $\forall \mathcal{X}.T$, a *(type-)substitution* $\rho$ for $\forall \mathcal{X}.T$ substitutes types for type variables X, such that $\rho(T)$ is an (unquantified) type. We call $\rho(T)$ an *instance* of type $\forall \mathcal{X}.T$. The types of function symbols in $\mathcal{F}$ may also have a quantifier-prefix.

*Example A.1.* Let $T$ be the type $\forall a, b.a \to b$. Then $\mathtt{Int} \to \mathtt{Int}$ is an instance of $T$, as well as $a \to \mathtt{Int}$, where the latter has a variable name in common with $T$.

(Var) $$\Gamma, x :: S \vdash x :: S$$

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

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

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

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

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

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

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

**Fig. 5.** The type-derivation rules

**Definition A.2.** *Given a program, the types $\Gamma$ 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.

*Example A.3.* The polymorphic type of $\lambda x.x$ is $\forall a.a \to a$. The type of the function composition $\lambda f, g, x.f\ (g\ x)$ is $\forall a, b, c.(b \to c) \to (a \to b) \to a \to c$.

From a typing point of view, the derivation system and the type-labeling (see Section 2.3) are equivalent mechanisms.
Note 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.

**Assumption A.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 above.*

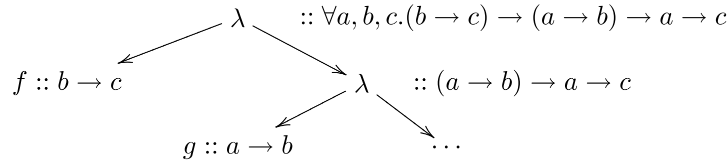
### A.2 Type Consistency Rules

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 (see Section 2.3). 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 A.5.* 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:

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

An illustration is as follows:



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

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

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

Case-expression

$$\left.\begin{array}{l} (\mathtt{case}_K\ s :: S\ ((c_{K,1}\ x_{1,1} \ldots x_{1,n_1}) :: S \texttt{->} t_1 :: T) \\ \quad \ldots \\ \quad ((c_{K,m}\ x_{m,1} \ldots x_{m,n_m}) :: S \texttt{->} t_m :: T)) \end{array}\right\} \mapsto T$$

**Fig. 6.** Computation of *MonoTp*

**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 $c$ is 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 Fig. 6 based on the type labels of the subexpressions.

**Definition A.6.** *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*.*

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 A.7 (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 A.8 (Progress Lemma).** *A closed and well-typed expression without reduction is a value.*

*Proof.* Checking the rules, it is easy to see that for a closed expression $t$, the only possibility for $t$ to be irreducible, and not a WHNF is to be of the form $R[((c\ t_1 \ldots t_n)\ t')]$, where $c$ is a constructor, $R[\mathtt{case}_K\ (\lambda x.r)\ Alts]$, or $R[\mathtt{case}_K\ (c\, t_1 \ldots\ t_n)\ Alts]$ where $c$ is a constructor that does not belong to data type $K$. All these cases are ruled out, since they are not well-typed. □

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

**Fig. 7.** Searching the redex in the let-language $L_{let}$

# B Proof of Theorem 4.1

In this section we prove Theorem 4.1. 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 $\leq_{\mathcal{P}\forall,T}$.

## B.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:

$$\begin{array}{lll}
s, s_i, t \in Expr & ::= & x \mid f \mid (s\ t) \mid \lambda x.s \mid (c_i\ s_1 \ldots s_{ar(c_i)}) \\
& & \mid\ (\mathtt{case}_K\ s\ Alt_1 \ldots Alt_{|D_K|}) \mid (\mathtt{let}\ x = v\ \mathtt{in}\ s) \\
Alt_i & ::= & ((c_i\ x_1 \ldots x_{ar(c_i)})\ \mathtt{->}\ s_i)
\end{array}$$

where $v$ is a value, i.e. $v, v_i \in Val ::= x \mid (c\ v_1\ \ldots v_N) \mid \lambda x.s$. The `let`-construct is non-recursive, i.e. the scope of $x$ in $(\mathtt{let}\ x = v\ \mathtt{in}\ s)$ is only $s$. The *type-constraints* for the `let`-construct are as follows: in $(\mathtt{let}\ x = v\ \mathtt{in}\ s)$, 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 $(\mathtt{let}\ x :: T_1 = v :: T_1\ \mathtt{in}\ s :: T_2) :: T_2$ is a correct typing. We use a label-shift to determine the reduction position. For an expression $s$ the label-shift algorithm starts with $s^{\mathsf{lr}}$ and then exhaustively applies the shifting rules shown in Fig. 7. During shifting we assume that the label is not removed, however, in the right hand sides of the rules in Fig. 7 only the new labels are shown. The two labels $\mathsf{sub}$ and $\mathsf{lr}$ are used to prevent searching inside `let`-expressions which are below application and constructor applications. The standard reduction rules for $L_{let}$ are defined in Fig. 8, which can be applied after performing the label shift algorithm where an additional condition is that rule (cp) is only applicable if rule ($\text{case}_{\mathtt{let}}$) is not applicable. 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. With (lll) we denote the union of the rules (lapp), (lrapp), (lcapp), and (lcase).
The *answers* of reductions are values – but not variables – that may be embedded in lets. I.e., expressions of the form $(\mathtt{let}\ x_1 = v_1\ \mathtt{in}\ (\mathtt{let}\ x_2 = v_2\ \mathtt{in}\ \ldots (\mathtt{let}\ x_n =$

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

(delta$_{\texttt{let}}$) $C[f^{\mathsf{sub}} :: T] \to C[d_f]$ 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$_{\texttt{let}}$) $C[(\texttt{case}\ (c\ v_1 \ldots v_n)^{\mathsf{sub}}\ \ldots ((c\ y_1 \ldots y_n)\ \texttt{->}\ s) \ldots)] \to C[\texttt{let}\ y_1 = v_1\ \texttt{in}\ \ldots \texttt{let}\ y_n = v_n\ \texttt{in}\ s]$

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

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

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

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

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

**Fig. 8.** Standard Reduction rules in the let-language $L_{let}$

$v_n$ in $v) \ldots))$ 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 $(\texttt{let}\ x = v\ \texttt{in}\ t)$, then the hole must be within an abstraction of $v$. Contextual approximation and contextual equivalence for $L_{\texttt{let}}$ are defined accordingly, where we use the symbols $\leq_{\texttt{let},\mathcal{P},T}$ and $\sim_{\texttt{let},\mathcal{P},T}$ 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 $(\texttt{let}\ x = [\cdot]\ \texttt{in}\ t)$.
For a reduction sequence $RED$ the function $\mathrm{rl}(RED)$ computes the length of the reduction sequence $RED$.

**Definition B.1.** *For well-typed $\mathcal{P}$-expressions $s, t :: T$, the inequation $s \leq_{\texttt{let},\mathcal{P},R,T} t$ holds iff for all $\rho$ where $\rho$ is a variable-permutation such that variables are renamed, the following holds: $\forall\mathcal{P}$-reduction contexts $R[\cdot :: T]$: if $R[\rho(s)], R[\rho(t)]$ are closed, then $(R[\rho(s)]\downarrow \implies R[\rho(t)]\downarrow)$)*

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 B.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 B.3 (Context Lemma).** *The following holds:* $\leq_{\mathtt{let},\mathcal{P},R,T} \subseteq \leq_{\mathtt{let},\mathcal{P},T}$.

*Proof.* We prove a more general claim:
For all $n \geq 0$ and for all $\mathcal{P}$-multicontexts $C[\cdot_1 :: T_1, \ldots, \cdot_n :: T_n]$ and for all well-typed $\mathcal{P}$-expressions $s_1 :: T_1, ..., s_n :: T_n$ and $t_1 :: T_1, ..., t_n :: T_n$:
If for all $i = 1, \ldots, n$: $s_i \leq_{\mathtt{let},\mathcal{P},R,T_i} t_i$, and if $C[s_1, \ldots, s_n]$ and $C[t_1, \ldots, t_n]$ are closed, then $C[s_1, \ldots, s_n]\downarrow \implies C[t_1, \ldots, t_n]\downarrow$.
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 $\forall i : s_i \leq_{\mathtt{let},\mathcal{P},R,T_i} t_i$. 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 B.2 implies that there is a hole $\cdot_i$ such that $R_1 = C[s_1, \ldots, s_{i-1}, \cdot_i :: T_i, s_{i+1}, \ldots, s_n]$ and $R_2 = C[t_1, \ldots, t_{i-1}, \cdot_i :: T_i, t_{i+1}, \ldots, t_n]$ are both reduction contexts. Let $C_1 = C[\cdot_1 :: T_1, \ldots, \cdot_{i-1} :: T_{i-1}, s_i, \cdot_{i+1} :: T_{i+1}, \ldots, \cdot_n :: T_n]$. From $C[s_1, \ldots, s_n] = C_1[s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n]$ 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

$C[s_1, \dots, s_n] \xrightarrow{ls,a} C'[s'_1, \dots, s'_m]$, then $C[t_1, \dots, t_n] \xrightarrow{ls,a} C'[t'_1, \dots, t'_m]$, such that $s'_i \leq_{\mathtt{let},\mathcal{P},R,T'_i} t'_i$. We go through the cases of which reduction step is applied to $C[s_1, \dots, s_n]$ 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$_{\mathtt{let}}$) reduction, the holes $\cdot_i$ may change their position.
- For a (case$_{\mathtt{let}}$) 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 $s'_i \leq_{\mathtt{let},\mathcal{P},R,T'_i} t'_i$.
- If the standard reduction is a (delta$_{\mathtt{let}}$)-reduction, then $s_i, t_i$ cannot be influenced, since within $d_f$, there are no holes.

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

### B.2 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 on translations in [SSNSS08]. Let $\Phi$ be the translation from $L$ to $L_{\mathtt{let}}$ 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 $L_{\mathtt{let}}$ into $L$. The translation is defined as $\overline{\Phi}(\mathtt{let}\ x = v\ \mathtt{in}\ s) := \overline{\Phi}(s)[\overline{\Phi}(v)/x]$ 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 B.4.** *Let $t \in L_{\mathtt{let}}$ 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 $\xrightarrow{ls,lll}$ and $\xrightarrow{ls,cp}$ are finite.*

**Lemma B.5.** *Let $t$ be an expression of $L_{\mathtt{let}}$ 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 an 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 B.6.** *Let $t$ be an $L_{\texttt{let}}$ 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 B.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 $t$:

$$\begin{array}{l}\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[(\lambda x.r_0)\ v_0] \\ \quad \xrightarrow{ls,beta_{\texttt{let}}} \texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[\texttt{let}\ x = v_0\ \texttt{in}\ r_0].\end{array}$$

Now it is easy to verify that $\overline{\Phi}(t') = s$ holds. □

**Lemma B.7.** *The following properties hold:*

1. *For all $t \in L_{\texttt{let}}$: if $t$ is an answer, then $\overline{\Phi}(t)$ is a value for $L$, and if $\overline{\Phi}(t)$ is a value (but not a variable), then $t \xrightarrow{ls,*} t'$ where $t'$ is an answer for $L_{\texttt{let}}$.*
2. *For all $t \in L$: $t$ is a non-variable value 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 $\overline{\Phi}(t_1) \to \overline{\Phi}(t_2)$.*
4. *Let $t_1 \in L_{\texttt{let}}$ with $\overline{\Phi}(t_1) \to t_2'$. Then $t_1 \xrightarrow{ls,+} t_2$ with $\overline{\Phi}(t_2) = t_2'$.*

*Proof.* Part 1 and 2 follow by definition of values and answers in $L$ and $L_{\texttt{let}}$ and the definitions of $\Phi$, $\overline{\Phi}$. Note that it may be possible that $\overline{\Phi}(t)$ is a value, 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 $\overline{\Phi}(t_1) \to \overline{\Phi}(t_2)$ by the reduction with the same name. Part 4 follows from Lemma B.4 and B.6. □

**Lemma B.8.** *$\Phi$ 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 B.7, part 1. The induction step follows by Lemma B.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 B.7, part 1 shows that if $\overline{\Phi}(t)$ is a (non-variable) value, then $t\downarrow$. For the induction step let $\overline{\Phi}(t) \to t'$ such that $t'\downarrow$. Lemma B.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 B.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 $\Phi$ imply adequacy, i.e.:

**Corollary B.9 (Adequacy of $\Phi$).** $\Phi(s) \leq_{\texttt{let},\mathcal{P},T} \Phi(t) \implies s \leq_{\mathcal{P},T} t$.

**Lemma B.10 (CIU-Lemma).** *Let $s,t :: T$ be two expressions of $L$ such that for all $\mathcal{P}$-value substitutions $\sigma$ and for all $\mathcal{P}$-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 $s \leq_{\mathcal{P},T} t$ holds.*

*Proof.* Let $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ hold for all $\mathcal{P}$-value substitutions $\sigma$ and $\mathcal{P}$-reduction contexts $R$, such that $R[\sigma(s)], R[\sigma(t)]$ are closed. We show that $\Phi(s) \leq_{\texttt{let},\mathcal{P},R,T} \Phi(t)$ holds. Then the context lemma B.3 shows that $\Phi(s) \leq_{\texttt{let},\mathcal{P},T} \Phi(t)$ and the previous corollary implies $s \leq_{\mathcal{P},T} t$.
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 $R_{let} = \texttt{let}\ x_1 = v_1\ \texttt{in}\ (\texttt{let}\ x_2 = v_2\ \texttt{in}\ (\ldots(\texttt{let}\ x_n = v_n\ \texttt{in}\ R'_{let}))))$ 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 $\sigma$ 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 B.11.** *The transformation (beta), (delta), and (case) are correct program transformations in $L$.*

*Proof.* We use the CIU-Lemma B.10: Let $a \in \{(\text{beta}), (\text{delta}), (\text{case})\}$. Let $s \xrightarrow{a} t$, $R$ be a reduction context, and $\sigma$ 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. $R[\sigma(s)] \to t_1 \xrightarrow{*} t_n$ where $t_n$ is a value. Since standard reduction is unique one can verify that then $R[\sigma(s)] \xrightarrow{a} R[\sigma(t)] = t_1$ must hold, i.e. $R[\sigma(t)] \downarrow$. □

**Theorem B.12 (CIU-Theorem).** *For $\mathcal{P}$-expressions $s, t :: T$: $R[\sigma(s)] \downarrow \Longrightarrow R[\sigma(t)] \downarrow$ for all $\mathcal{P}$-value substitutions $\sigma$ 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 B.10. For the other direction, let $s \leq_{\mathcal{P},T} t$ hold and $R[\sigma(s)] \downarrow$ for a $\mathcal{P}$-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 $\mathcal{P}$-reduction context. Since (beta) is correct, we have $R[(\lambda x_1.\ldots.x_n.s)\ v_1 \ldots v_n] \sim_{\mathcal{P},T} R[\sigma(s)]$. Thus, $R[(\lambda x_1.\ldots.x_n.s)\ v_1 \ldots v_n] \downarrow$ and applying $s \leq_{\mathcal{P},T} t$ we derive $R[(\lambda x_1.\ldots.x_n.t)\ v_1 \ldots v_n] \downarrow$. Correctness of (beta) shows $R[\sigma(t)] \downarrow$. □

Applied to extensions $\mathcal{P}'$ of $\mathcal{P}$, we obtain the following corollary, since every $\mathcal{P}$-expression is also an $\mathcal{P}'$-expression:

**Corollary B.13.** *Let $\mathcal{P}$ be a program. For $\mathcal{P}$-expressions $s, t :: T$: $R[\sigma(s)] \downarrow \Longrightarrow R[\sigma(t)] \downarrow$ for all extensions $\mathcal{P}'$ of $\mathcal{P}$ and all $\mathcal{P}'$-value substitutions $\sigma$ 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.*

### B.3 Local CIU-Theorems

In this subsection we show that the CIU-theorem can be made stronger by restricting $R$ and $\sigma$ 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 3.4.
We will use the lambda-depth-measure for subexpression-occurrences $s$ of some expression $t$: it is the number of lambdas and pattern-alternatives that are crossed by the position of the subexpression.

**Lemma B.14 (CIU-Lemma F-free).** *Let $s, t :: T$ be two $\mathcal{P}$-expressions of $L$ such that for all F-free $\mathcal{P}$-value substitutions $\sigma$ and all F-free $\mathcal{P}$-reductions contexts $R$ such that $R[\sigma(s)], R[\sigma(t)]$ are closed: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$. Then $s \leq_{\mathcal{P},T} t$ 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 $\sigma$ 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 $\sigma$ 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 a value. We construct F-free reduction contexts $R'$ and F-free value substitutions $\sigma'$ as follows: apply $n+1$ times a delta-step for every occurrence of function symbols in $R$ and $\sigma$. 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 $R'[\sigma'(t)] \leq_{\mathcal{P},T} R[\sigma(t)]$, since delta-reduction is correct and the insertion of $\bot$ makes the expression smaller w.r.t. the contextual ordering. Hence $R[\sigma(t)]\downarrow$. Then we can use the CIU-Theorem B.12. □

We are now able to prove Theorem 4.1. The claim is:

> For $s, t :: \tau \in L$: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ for all F-free $\mathcal{P}$-value substitutions $\sigma$ and F-free $\mathcal{P}$-reduction contexts $R$, where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P},\tau} t$ holds.

*Proof (of Theorem 4.1).* One direction is the F-free CIU-Lemma B.14. The other direction is the same as in the proof of the local CIU-theorem B.12. □

**Corollary B.15.** *Let $s, t :: \tau \in L$. If for all closing F-free $\mathcal{P}$-value substitutions $\sigma$, we have $\sigma(s) \leq_{\mathcal{P},\tau} \sigma(t)$, then $s \leq_{\mathcal{P},\tau} t$.*
*If for all closing F-free $\mathcal{P}$-value substitutions $\sigma$, we have $\sigma(s) \sim_{\mathcal{P},\tau} \sigma(t)$, then $s \sim_{\mathcal{P},\tau} t$.*

*Proof.* Follows from the F-free CIU-theorem 4.1. □

**Corollary B.16.** *Let $s, t :: \tau \in L$. If for all closing F-free $\mathcal{P}$-value substitutions $\sigma$, $\sigma(s)$ and $\sigma(t)$ standard reduce to the same value, then $s \sim_{\mathcal{P},\tau} t$.*

*Proof.* Follows from the F-free CIU-theorem 4.1, since reduction of $R[\sigma(s)]$ (respectively $R[\sigma(t)]$ first evaluates the expression $\sigma(s)$ (respectively $\sigma(t)$). □

### B.4 Properties of $\Omega$-Expressions

We show that the property of being an $\Omega$-expression inherits to reduction contexts:

**Proposition B.17.** *Let $s :: \tau$ be an $\Omega$-expression. Then for every reduction context $R[\cdot :: \tau]$, the expression $R[s]$ is an $\Omega$-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 reduction context $R_1$ with $R = R_1[([\cdot]\ t)]$, $R = R_1[(v\ [\cdot])]$, $R = R_1[(\texttt{case}\ [\cdot]\ alts)]$, 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 $\sigma$ the expression $\sigma(s\ t)$, $\sigma(v\ s)$, $\sigma(\texttt{case}\ s\ alts)$, or $\sigma(c\ v_1\ \dots\ v_i\ s\ s_{i+1}\ \dots\ s_n)$, respectively, cannot be evaluated to a value, since $\sigma(s)\Uparrow$. Hence, $(s\ t)$, $(v\ s)$, $(\texttt{case}\ s\ alts)$, or $(c\ v_1\ \dots\ v_i\ s\ s_{i+1}\ \dots\ s_n)$, respectively, is an $\Omega$-expression. Thus, the induction hypothesis can be applied to $R_1$ which shows that $R[s]$ is an $\Omega$-expression. □

**Corollary B.18.** *Let $s, t :: \tau$ and let $s$ be an $\Omega$-expression. Then $s \leq_{\mathcal{P},\tau} t$. If also $t$ is an $\Omega$-expression, then $s \sim_{\mathcal{P},\tau} t$.*

*Proof.* We only prove $s \leq_{\mathcal{P},\tau} t$, since the other direction is symmetric. We use the CIU-Theorem B.12: Let $R$ be a reduction context, $\sigma$ be a value substitution such that $\sigma(s), \sigma(t)$ are closed. Then $\sigma(s)$ must be an $\Omega$-expression, and by Proposition B.17 $R[\sigma(s)]$ is an $\Omega$-expression, too. Thus $R[\sigma(s)]\Uparrow$, and $s \leq_{\mathcal{P},\tau} t$ holds. The second claim follows by symmetry. □

## C Proof of Theorem 5.4

In the following we intend to show that $\leq_{\mathcal{P},T}$ and $\sim_{\mathcal{P},T}$ 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.

In the following, we will add a sequentializing construct that always comes with two arguments. The expression $(s; t)$ can be seen as an abbreviation for $((\lambda_.t)s)$, where we assume that the seq-expressions are labelled such that they can be distinguished from other applications. 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. $((\lambda x. \dots)\ (\texttt{case}\ y\ \dots))$ 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 $\Omega$-expressions. The extended set of VN-reductions is in Figs. 2, 3 and 4.

**Lemma C.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.

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 $\mathtt{case}_T\ s_1'\ (c\, x_1 \ldots x_n) \to r \ldots$, 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 $s_1$ of non-$\mathcal{P}$-type that is not within the scope of a non-$\mathcal{P}$-binder.

1. $s_1$ cannot be an application. Assume otherwise. Then $s_1 = s_1'\ s_2'\ \ldots s_n'$ 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.
2. $s_1$ cannot be in function position in an application $(s_1\ s_2)$. Due to the previous item, $s_1\ s_2$ must have a $\mathcal{P}$-type, and $s_1$ is not an application. 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.

Now we choose an $s_1$ such that it has maximal size. Note that $s_1$ is irreducible, has a $\mathcal{P}'$-type, and it cannot be the top expression $v'$, since $v'$ has a $\mathcal{P}$-type. We check all the remaining 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 $(r; s_1)$ 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, and not in function position as shown above.
- $s_1$ cannot be the result expression of an alternative due to maximality, but may be the first argument of a `case`.

Now we analyze the last cases:

1. $s_1$ cannot be the first argument of a seq-expression: We scan all syntactic cases of $s_1$. Since $v'$ is irreducible, $s_1$ cannot be a seq-expression, a `case`-expression, a constructor-expression, `Bot`, an abstraction, nor a variable. An application is not possible as shown above.
2. $s_1$ cannot be the first argument of a `case`: We scan all syntactic cases of $s_1$. Since $v'$ is irreducible, $s_1$ cannot be seq-expression, a `case`-expression, a constructor-expression, `Bot`. Due to typing, an abstraction is impossible. A variable is impossible since there are only $\mathcal{P}$-scopes. An application is not possible as shown above.

### C.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. The arguments require several steps. Since we did not a find a proof that VN-reduction is strongly terminating, we use approximation-methods.
Note that our proof only works, since we need to take only $F$-free value substitutions and $F$-free reduction contexts into account. Note also, that even if we could prove termination of the VN-reduction, this would not imply that every monomorphic $\mathcal{P}$-theorem is a global tautology, since the object sets $M_{\mathcal{P}',T}$ may also include non-$F$-free values.

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

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

**Lemma C.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: $4css(t_0) + 2\max(css(t_i)) + 2css(r) > 2css(t_0) + \max(2css(t_i) + 2css(r))$.
- 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 C.3.** *All VN-reduction rules are (locally) correct.*

*Proof.* Correctness of the bot-reduction-rules can be shown by using the CIU-Theorem: Let $s, t$ be $\mathcal{P}$ expressions of type $T$ with $s \to t$ by a bot reduction, $R$ be a reduction context and $\sigma$ be a closing value substitution for $s$ and $t$. Then it easy to verify that both, $R[\sigma(s)]$ and $R[\sigma(t)]$, diverge.
Correctness of the rules (seqlam), (seqx) follows from the correctness of beta-reduction. Let $s \to t$ by a rule in

| | | | |
|---|---|---|---|
| $(s\ t)^{VNS}$ | $\rightarrow (s^{VNS}\ t)$ | $(s\ t)^{VNS}$ | $\rightarrow (s\ t^{VNS})$ |
| $(s;t)^{VNS}$ | $\rightarrow (s^{VNS};t)$ | $(s;t)^{VNS}$ | $\rightarrow (s;t^{VNS})$ |
| $(\texttt{case}\ s\ \ldots)^{VNS}$ | $\rightarrow (\texttt{case}\ s^{VNS}\ \ldots)$ | $(c\ s_1\ \ldots\ s_n)^{VNS}$ | $\rightarrow (c\ s_1\ \ldots\ s_i^{VNS} \ldots\ s_n)$ |

**Fig. 9.** The *VNS*-label-shifting rules

{seqc, seqseq, seqapp, caseseq, caseapp, casecase, seqcase} then clearly every evaluation of $R[\sigma(s)]$ can be transformed into an evaluation of $R[\sigma(t)]$ and vice versa, where additionally the correctness of (beta) is needed.
Finally we consider the rules VNbeta and VNcase. Let $s, t$ be $\mathcal{P}$-expressions of type $T$ with $s = (\lambda x.s')\ t'$ and $t = (t'; s'[t'/x])$, i.e. $s \xrightarrow{VNbeta} t$. Let $\sigma$ be a closing value substitution for $s, t$. We distinguish two cases:

- $\sigma(t')$ is an $\Omega$-expression. Then obviously $\sigma(s), \sigma(t)$ are $\Omega$-expressions, i.e. $\sigma(s) \sim_{\mathcal{P},T} \sigma(t)$ by Corollary 4.4.
- $\sigma(t')$ is not an $\Omega$-expression. Then there exists a value $v$ such that $\sigma(t') \xrightarrow{*} v$. Correctness of the standard reduction rules implies $v \sim_{\mathcal{P},T} \sigma(t')$. Now we can transform $\sigma(s)$ into $\sigma(t)$ using (beta)-reductions: $\sigma(s) = (\lambda x.\sigma(s'))\ \sigma(t') \sim_{\mathcal{P},T} (\lambda x.\sigma(s'))\ v \sim_{\mathcal{P},T} \sigma(s')[v/x] \sim_{\mathcal{P},T} \sigma(s')[\sigma(t')/x] \sim_{\mathcal{P},T} (\sigma(t'); s'[t'/x])$.

We have shown that for all closing value substitutions $\sigma$: $\sigma(s) \sim_{\mathcal{P},T} \sigma(t)$. Hence Corollary B.15 implies $s \sim_{\mathcal{P},T} t$. Correctness of VNcase can be shown in a similar way. □

Now we want to show that infinite VN-reduction sequences for an expression indicate that this expression can only be equal to `Bot`. Therefore, we define a standard reduction that is usually applied to subexpressions of $v$, which are in general open expressions.

**Definition C.4.** *Let $t$ be a (perhaps open) F-free expression. A* VN-standard-reduction *of $t$ is defined as follows: Apply the VNS-label-shift in Fig. 9 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. The outermost-leftmost VN-reduction according to Figs. 2, 3 and 4 is applied to a labelled redex, where in case of a conflict the bot-reduction is preferred. The reduction is denoted as $\xrightarrow{VNsr}$. If the VNsr-reduction is not a Bot-reductions, then it is denoted as $\xrightarrow{VNNBsr}$.*

Note that there may be multiple redexes with *VNS*-labels, but due to the above priority rules, the VN-standard-reduction is uniquely defined.
The standard-reduction $\xrightarrow{sr}$ treats the seq-expressions $(s;t)$ as an application of the lambda-expressions $((\lambda_.t)\ s)$. For counting the length of reduction sequences, we assume that this lambda-expression is labelled to distinguish it from other abstractions. The seq-reduction $(v;s) \rightarrow s$, where $v$ is a closed value (which

| | | | |
|---|---|---|---|
| seqcbv | $(v; s)$ | $\to s$ | if $v$ is a value |
| betacbv | $((\lambda x.s)\ v)$ | $\to s[v/x])$ | if $v$ is a value |
| casecbv | $\texttt{case}_K\ (c\ v_1 \dots v_n)\ \dots\ (c\ x_1 \dots x_n)\ \texttt{->}\ t\ \dots$ | $\to t[v_1/x_1, \dots, v_n/x_n]$ | if $(c\ v_1 \dots v_n)$ is a value |

**Fig. 10.** Call-by-value-reduction rules

corresponds to 1 beta-reduction) is not counted as a beta-reduction in the length of standard-reductions, but as a seqlam-, seqx-, or seqc-reduction.
We repeat the reduction rules of our logic adapted to the seq-notation and restricted to F-free expressions:

**Definition C.5.** *The call-by-value reduction rules are the rules in Fig. 10. The call-by-value reduction is denoted as* $\xrightarrow{cbv}$*. Note that a value* $v$ *is defined as* $v, v_i \in Val ::= x \mid \lambda x.s \mid (c\ v_1 \dots v_n)$*, i.e. a variable, an abstraction, or a constructor-expression* $(c\ v_1 \dots v_n)$*, where the immediate subexpressions* $v_i$ *are also values. We also need the parallel reduction of several* $\xrightarrow{cbv}$*-reductions, which is defined like the 1-reduction in [Bar84], and is denoted as* $\xrightarrow{par}$*.*
*The parallel reduction is used w.r.t. further restrictions. Let* $s$ *be an F-free expression and* $\sigma$ *be a value-substitution. Then the notation* $\sigma(s) \xrightarrow{par,s} r$ *means that the reduction is parallel as above, and that for all free variables* $x$ *of* $s$*, the reduction within the subexpressions* $\sigma(x)$ *is the same.*

Now we analyse in a series of lemmas the relation between call-by-value reduction and the VNsr-reduction for F-free expressions. Since the analysis can be restricted to VNsr-reductions without Bot-reductions, we consider the VNNBsr-reduction in the following lemmas.

**Lemma C.6.** *Let* $s$ *be an F-free expression and* $\sigma$ *be a value-substitution. If* $\sigma(s)$ *is a value, then* $s$ *is a value.*

*Proof.* This follows from the possible structures of values.

We distinguish between internal and noninternal-reductions within the parallel reduction: Let $s$ be an F-free expression and $\sigma$ be a value-substitution. A parallel cbv-reduction of $\sigma(s)$ is called $s$-internal if there is no position $p$ in $\sigma(s)$ that is a prefix of the VNsr-redex of $s$. The notation is $\xrightarrow{par,s,int}$. This notation is also extended to expressions $s'$ such that every position of $s$ is also a position in $s'$.

**Lemma C.7.** *Let* $s$ *be an F-free expression and* $\sigma$ *be a value-substitution. If* $\sigma(s) \xrightarrow{par,s} r$ *is not* $s$*-internal and is free of Bot-reductions, then the reduction can be split into* $s \xrightarrow{VNNBsr,*} s'$ *and* $\sigma(s') \xrightarrow{par,s',int} r$*.*

*Proof.* First we show that for a non-internal paralle reduction, thatthere is an $s'$, such that $s \xrightarrow{VNNBsr} s'$ and $\sigma(s') \xrightarrow{par,s'} r$: It is not possible that a position in the parallel reduction is a proper prefix of the VNsr-redex position. If the VNsr-redex position is also a position in $\xrightarrow{par}$, then there are two possibilities:

1. There is no VN-reduction at the position in $s$, i.e. $s$ is VN-irreducible. This is not possible if the cbv-reduction is a (seqcbv). In the cases of a (betacbv) or a (casecbv), the subexpression at $p$ is VNNBsr-irreducible. Hence this case cannot occur.
2. There is VNsr-reduction at this position in $s$. Scanning all possibilities of VNsr-reductions shows that in the case of VN-reductions there is some $s'$ with $s \xrightarrow{VNNBsr,*} s'$ and $\sigma(s') \xrightarrow{par,s'} r$.

Repeating this splitting will terminate, since the number of positions in $\xrightarrow{par,s}$ is properly decreased in every step, hence we will arrive at an internal reduction, i.e.: $s \xrightarrow{VNNBsr,*} s'$ and $\sigma(s') \xrightarrow{par,s',int} r$.

**Lemma C.8.** *Let $s$ be an F-free expression and $\sigma$ be a value-substitution. If $\sigma(s) \xrightarrow{(par,s,int),*} v$, where $v$ is a value, then $s$ is VNNBsr-irreducible.*

*Proof.* The $\xrightarrow{(par,s,int),*}$-reduction does not remove $s$-positions: it only may change the labels of the positions that are leaves in $s$. If $s$ is VNNBsr-reducible, then there is a position that is also cbv-reducible in all expressions that are derived from $\sigma(s)$, hence the final expression cannot be a value.

Now we inspect how internal parallel reductions on $\sigma(s)$ and VNsr-reductions on $s$ can overlap using a diagram notation.

**Lemma C.9.** *Let $s$ be an F-free expression and $\sigma$ be a value-substitution. If $\sigma(s) \xrightarrow{par,s,int} r$, then there is some $s'$ with $s \xrightarrow{par,s,int} s'$, and a substitition $\sigma'$ such that $\sigma(x) \xrightarrow{par} \sigma'(x)$ for all $x$, and $r = \sigma'(s')$.*

*Proof.* This follows from the compatibility of the parallel reduction with the substitution.

**Lemma C.10.** *Let $s_1$ be an F-free expression and $\sigma$ be a value-substitution. If $\sigma(s_1) \xrightarrow{par,s_1,int} \sigma'(s_2)$, and $s_2 \xrightarrow{VNNBsr} s_3$, then there is a reduction $s_1 \xrightarrow{VNNBsr} s_4$, and either $\sigma(s_4) \xrightarrow{par,s_4,int} \sigma'(s_3)$ or $\sigma(s_4) \xrightarrow{par,s_4,int} s_2$, but only if $s_1 \xrightarrow{VNNBsr} s_4$ is not a VNbeta nor a VNcase. The diagrams are as follows:*

$$\begin{array}{ccc} \sigma(s_1) & \xrightarrow{par,s_1,int} & \sigma'(s_2) \\ {\scriptstyle VNNBsr,\ (s_1\to s_4)} \downarrow & & \downarrow {\scriptstyle VNNBsr\ (s_2\to s_3)} \\ \sigma(s_4) & \xrightarrow{par,s_4,int} & \sigma'(s_3) \end{array}$$

$$\begin{array}{ccc} \sigma(s_1) & \xrightarrow{par,s_1,int} & \sigma'(s_2) \\ {\scriptstyle VNNBsr,\ (s_1\to s_4)} \downarrow {\scriptstyle not\ VNbeta, VNcase} & \nearrow {\scriptstyle par,s_4,int} & \downarrow {\scriptstyle VNNBsr\ (s_2\to s_3)} \\ \sigma(s_4) & & \sigma'(s_3) \end{array}$$

*Proof.* The position of the VNsr-redex in $s$ is not influenced by $\xrightarrow{par,s,int}$-reductions. Hence it is sufficient to check the VN-reductions. This is done in the following for a selection of the rules, where we mention a selection of the essential cases.
Rule (seqapp). Let $s_1 = (t_1; t_2)\ t_3$ where $\sigma(t_1)$ is a value. Let $s_2 = (t_2'\ t_3')$,

$$\begin{array}{ccc} \sigma((t_1;t_2)\ t_3) & \xrightarrow{par,s_1,int} & \sigma'(t_2'\ t_3') \\ {\scriptstyle VNNBsr\ (s_1 \to s_4)} \downarrow & \nearrow {\scriptstyle par,s_4,int} & \\ \sigma(t_1;(t_2\ t_3)) & & \end{array}$$

Rule (VNbeta). Let $s_1 = ((\lambda x.s)\ t)$, $s_2 = ((\lambda x.s')\ t')$, $s_3 = (t'; s'[t'/x])$, $s_4 = t; s[t/x]$.

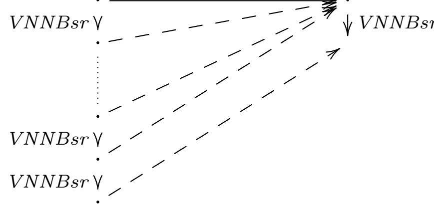
$$\begin{array}{ccc} \sigma((\lambda x.s)\ t) & \xrightarrow{par,s_1,int} & \sigma'((\lambda x.s')\ t') \\ {\scriptstyle VNNBsr\ (s_1 \to s_4)} \downarrow & & \downarrow {\scriptstyle VNNBsr\ (s_2 \to s_3)} \\ \sigma(t; s[t/x]) & \xrightarrow{par,s_4,int} & \sigma'(t'; s'[t'/x]) \end{array}$$

The rule (VNcase) leads to a diagram similar to (VNbeta) and the rules (seqseq), (caseapp),(casecase), (seqcase) lead to a triangle-diagram as above for (seqapp).

**Lemma C.11.** *Let $t$ be an expression. If for some closing value-substitution $\sigma$ the evaluation $\sigma(t) \xrightarrow{cbv,*} v$ is valid for some value $v$, then the VNsr-evaluation of $t$ terminates, i.e., $t \xrightarrow{VNsr,*} t'$ for some VNsr-irreducible $t'$ and $\sigma(t') \xrightarrow{sr,*} v'$ with $v' \sim v$.*

*Proof.* Note that if the *VNsr*-reduction sequence of $t$ includes a `Bot`-reduction, then the final result will be `Bot`, and hence $\sigma(t)$ cannot reduce to a value. Hence no `Bot`-reduction could be used in any reduction sequence $t \xrightarrow{VNsr,*} t'$.
The reduction $\sigma(t) \xrightarrow{cbv,*} v$ can also be written as a sequence $\sigma(t_1) \xrightarrow{par,t_1} \sigma(t_2) \xrightarrow{par,t_2} \ldots \sigma(t_n) \xrightarrow{par,t_n} \ldots v$. This needs some standard reasoning on the length of reduction sequences: if in one of the subexpressions $\sigma(x)$ a reduction is applied, which is missing in the subepxresssions of the other subexpressions $\sigma(x)$ then all the others could also be reduced at the same position, and there exists a cbv-reduction sequence to a value that is not longer than before.
We show by induction on the number of parallel reductions that there is a terminating VNNBsr-reduction of $t_1$. Lemma C.7 shows that the parallel reductions can be split into a prefix of VNsr-reductions and a subsequent internal parallel reduction.
Using Lemma C.10, we can show that the VNNBsr-reductions can be shifted to the start of the sequence, where the induction ordering can be construted using the number of parallel-reductions to the right of VNNBsr-reduction sequences. The square-diagrams are well-behaved. For the triangle-diagrams, Lemma C.2

shows that these cannot applied infinitely often. In this case the construction can be illustrated as follows:




The final situation is that of Lemma C.8, which tells us that the VNNBsr-reduction sequence of the starting expression terminates with an VNsr-irreducible expression. Since there also holds a standardization theorem for the usual standard reduction, and reductions are correct, also the final claim of the lemma holds.

**Corollary C.12.** *If $t$ has an infinite VNsr-reduction, then for every closing value-substitution $\sigma$: $\sigma(t)\Uparrow$, i.e. $t$ is an $\Omega$-expression.*

*Proof.* Assume that for some $\sigma$: $\sigma(t)\downarrow$. Then Lemma C.11 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, that, given a depth $n$ cuts the expressions at lambda-depth $n$ by replacing subexpressions by `Bot`.

- $\mathit{ValueConstr}_n(t)$: 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` not changing its type.
- Apply the same construction to the bodies of the maximal abstractions of $t'$ using parameter $n-1$ and replace these subterms with the results.

**Lemma C.13.** *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 $v' \leq_{\mathcal{P}',T} v$.*

*Proof.* The (mathematical) construction terminates and results in a value. Lemma C.1 shows that the result is a $\mathcal{P}$-value.

**Lemma C.14.** *Let $t$ be an expression. If for some closing value-substitution $\sigma$ the reduction $\sigma(t) \xrightarrow{sr,n} v$ holds for some value $v$, and $t'$ is constructed from $t$ using ValueConstr for binder-depth $n+1$, then $\sigma(t') \xrightarrow{\leq n,sr} v' \leq_{\mathcal{P}',T} v$.*

*Proof.* This follows from Lemma C.11, and since the `Bot`-insertions are below binder-depth $n$, and since $\Omega$-expressions are smaller than other expressions w.r.t. $\leq_{\mathcal{P}',T}$.

**Lemma C.15.** *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 substitutions $\sigma'$ and all closed $\mathcal{P}'$-reduction contexts $R'$, also the implication $R'[\sigma'(s)] \downarrow \implies R'[\sigma'(t)] \downarrow$ holds.*

*Proof.* Let $\sigma'$ 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 $R'' = (R'; \texttt{True})$, 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 $n$, i.e. $r = \mathit{ValueConstr}_n(r')$. 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 C.13. By the assumption, we also have $R[\sigma(t)] \downarrow$, and since $r \leq_{\mathcal{P}} r'$ and $\sigma(t) \leq_{\mathcal{P}} \sigma'(t)$, we also obtain $R''[\sigma'(t)] \downarrow$.

We are now able to prove Theorem 5.4. The claim is:

> 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 $\sigma$ 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 (of Theorem 5.4).* This follows from the F-free CIU-theorem 4.1 and from Lemma C.15.

The VN-reductions in Figs. 2, 3 and 4 are globally correct reductions:

**Theorem C.16.** *The transformations in Figs. 2, 3 and 4, 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 C.3 shows that the transformations in Figs. 2, 3 and 4 are correct if only $\mathcal{P}$-reduction contexts and $\mathcal{P}$-value-substitutions are used. Then Theorem 5.5 (which is a direct consequence of Theorem 5.4) shows that the transformations are also globally correct. □

# D 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,MSS10].
A substitution $\sigma$ 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 $\nu$ 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 $\mu$ to the type $T$ is usually indicated by an extra suffix $T$: i.e. $\mu_T$. 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 $\nu$ be a binary relation on closed expressions. Then $s\ \nu^o\ t$ for any expressions $s, t$ iff for all closing value substitutions $\sigma$: $\sigma(s)\ \nu\ \sigma(t)$. Conversely, for binary relations $\mu$ on open expressions, $\mu^c$ is the restriction to closed expressions.

**Lemma D.1.** *For a relation $\nu$ on closed expressions, the equality $((\nu)^o)^c = \nu$ holds. For a relation $\mu$ on open expressions: $s\ \mu\ t \implies \sigma(s)\ (\mu)^c\ \sigma(t)$ for all closing value substitutions $\sigma$ 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 $\tau$, which may be `case`, application, a constructor or $\lambda$, and $\theta$ for an operator that may be the head of a value i.e. a constructor or $\lambda$. Note that $\theta$ may represent also the binding $\lambda$ using $\theta(x.s)$ as representing $\lambda x.s$. Abstract syntax expressions $x.s$ only occur in relational formulas, where we permit $\alpha$-renaming and follow the convention that $x.s\ \mu\ x.t$ means $s\ \mu\ t$ for open expressions $s, t$.
A relation $\mu$ is *operator-respecting*, iff $s_i\ \mu\ t_i$ for $i = 1, \dots, n$ implies $\tau(s_1, \dots, s_n)\ \mu\ \tau(t_1, \dots, t_n)$.

**Definition D.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 $\nu$ over closed expressions: $s\ [\nu]\ t$ if $s\Uparrow$ or $s \downarrow (c\ s_1 \dots s_n)$ and $t \downarrow (c\ t_1 \dots t_n)$ 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 $\nu$ on closed expressions with $\nu \subseteq [\nu]$, we derive $\nu \ \subseteq \leq_b$. This obviously also implies $\nu^o \ \subseteq \leq_b^o$.

**Lemma D.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 D.4.** *For closed values $(c\ s_1 \dots s_n), (c\ t_1 \dots t_n)$ of equal type, we have $(c\ s_1 \dots s_n) \leq_b (c\ t_1 \dots t_n)$ 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 D.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 D.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. *$\tau(s_1, \ldots, s_n) \widehat{\leq_b^o} s$ if there is some expression $\tau(s'_1, \ldots, s'_n) \leq_b^o s$ with $s_i \widehat{\leq_b^o} s'_i$.*

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

**Lemma D.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 D.1.

**Lemma D.8.** *The middle expression in the definition of $\widehat{\leq_b^o}$ can be chosen as closed, if $s, t$ are closed: Let $s = \tau(s_1, \ldots, s_{ar(\tau)})$, such that $s \widehat{\leq_b^o} t$ holds. Then there are operands $s'_i$, such that $\tau(s'_1, \ldots, s'_{ar(\tau)})$ is closed, $\forall i : s_i \widehat{\leq_b^o} s'_i$ and $\tau(s'_1, \ldots, s'_{ar(\tau)}) \leq_b^o s$.*

*Proof.* The definition of $\widehat{\leq_b^o}$ implies that there is a expression $\tau(s''_1, \ldots, s''_{ar(\tau)})$ such that $s_i \widehat{\leq_b^o} s''_i$ for all $i$ and $\tau(s''_1, \ldots, s''_{ar(\tau)}) \leq_b^o t$. Let $\sigma$ be the substitution with $\sigma(x) := v_x$ for all $x \in FV(\tau(s''_1, \ldots, s''_{ar(\tau)}))$, where $v_x$ is the closed value for the type of $x$ that exists by Assumption 3.2.
Lemma D.7 now shows that $s_i = \sigma(s_i) \widehat{\leq_b^o} \sigma(s''_i)$ holds for all $i$. The relation $\sigma(\tau(a''_1, \ldots, a''_{ar(\tau)})) \leq_b^o t$ holds, since $t$ is closed and due to the definition of an open extension. The requested expression is $\tau(\sigma(a''_1), \ldots, \sigma(a''_{ar(\tau)}))$.

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

**Theorem D.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 D.7 (6) implies $\widehat{\leq_b^o} \subseteq ((\widehat{\leq_b^o})^c)^o \subseteq \leq_b^o$.

**"2 $\implies$ 1":** Lemma D.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 D.7 and a precongruence. □

### D.1 Determining the Congruence Candidate

**Lemma D.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 D.11.** *If $s \ \widehat{\leq_b^o}\ t$ and $t \to t'$, then $s \ \widehat{\leq_b^o}\ t'$*

*Proof.* Follows from Lemma D.10.

**Definition D.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 D.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 $\sigma$: $\sigma(s) \leq_b \sigma(t)$ by definition of open extensions. This implies that for all closed contexts $C$ and all closing value substitutions $\sigma$: $\forall C : C[\sigma(s)] \leq_b C[\sigma(t)]$, since $\leq_b^o$ is a precongruence. Hence $s \leq_{\mathcal{P}} t$. The other direction follows from Lemma D.3.

**Lemma D.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 D.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 $s' \widehat{\leq_b^o} t''$.

If $\theta$ is a constructor, then there is a closed expression $c(t'_1, \ldots, t'_n)$ with $s_i \widehat{\leq_b^o} t'_i$ for all $i$ and $c(t'_1, \ldots, t'_n) \leq_b t$. By applying the induction hypothesis to $s_i \widehat{\leq_b^o} t'_i$ 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 $s_i \widehat{\leq_b^o} t''_i$ by Lemma D.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 $s_i \widehat{\leq_b^o} t_i^{(3)}$ for all $i$.

**Proposition D.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 $s \widehat{\leq_b^o} t$. 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 D.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 D.16.** *If $\widehat{\leq_b^o}$ is stable, then $\widehat{\leq_b^o} = \leq_b^o = \leq_{\mathcal{P}}$.*

*Proof.* Lemma D.11, Propositions D.15, D.13 and Theorem D.9 show the claim.

It remains to show stability:

**Proposition D.17.** *Let $s, t$ be closed expressions, $s \widehat{\leq_b^o} t$ and $s \to s'$ where $s$ is the redex. Then $s' \widehat{\leq_b^o} t$.*

*Proof.* Let $s, t$ be closed expressions, $s \widehat{\leq_b^o} t$ and $s \to s'$ where $s$ is the redex. The relation $s \widehat{\leq_b^o} t$ implies that $s = \tau(s_1, \ldots, s_n)$ and that there is some closed $t' = \tau(t'_1, \ldots, t'_n)$ with $s_i \widehat{\leq_b^o} t'_i$ 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 D.14 shows that $t'_1 \xrightarrow{*} \lambda x.t''_1$ with $\lambda x.s'_1 \widehat{\leq_b^o} \lambda x.t''_1$ and also $s_1 \widehat{\leq_b^o} t''_1$. From $s_2 \widehat{\leq_b^o} t'_2$ and since $s_2$ is a value, we obtain the next part of the standard reduction $t'_2 \xrightarrow{*} t''_2$ with $s_2 \widehat{\leq_b^o} t''_2$. From $t' \xrightarrow{*} t''_1[t''_2/x]$ we obtain $t''_1[t''_2/x] \leq_b t$. Lemma D.7 now shows $s'_1[s_2/x] \widehat{\leq_b^o} t''_1[t''_2/x]$. Hence $s'_1[s_2/x] \widehat{\leq_b^o} t$, again using Lemma D.7.

- Similar arguments apply to the case-reduction.
- Suppose, the reduction is a $\delta$-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 $s \to s'$ means also $s' \sim_b^o s$, we also have $s' \leq_b^o t$. By Lemma D.7, this implies $s' \mathrel{\widehat{\leq_b^o}} t$.

**Proposition D.18.** *Standard reduction is stable in surface contexts*

*Proof.* We use induction on the structure of contexts. The base case is proved in Proposition D.17. Let $S[s], t$ be closed, $S[s] \mathrel{\widehat{\leq_b^o}} t$ and $S[s] \to S[s']$, where we assume that the redex is not at the top level. The relation $S[s] \mathrel{\widehat{\leq_b^o}} t$ implies that $S[s] = \tau(s_1, \dots, s_n)$ and that there is some $t' = \tau(t'_1, \dots, t'_n) \leq_b^o t$ with $s_i \mathrel{\widehat{\leq_b^o}} t'_i$ for all $i$. If $s_j \to s'_j$, then by induction hypothesis, $s'_j \mathrel{\widehat{\leq_b^o}} t'_j$. Since $\widehat{\leq_b^o}$ is operator-respecting, we obtain also $S[s'] = \tau(s_1, \dots, s_{j-1}, s'_j, s_{j+1}, \dots, s_n) \mathrel{\widehat{\leq_b^o}} \tau(t'_1, \dots, t'_{j-1}, t'_j, t'_{j+1}, \dots, t'_n)$.

**Theorem D.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 D.17, D.18 and from Theorem D.16.