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

January 26, 2011

1 Introduction

The goal of this paper is to describe a programming logic for functional programs which permits specifying data values, recursive typed functions, logical propositions on the functions and values, and proof rules that allow function evaluation and induction proofs. A motivating source was the ease of use of some proof systems and our discontent with some of their restrictions and compromises. Principles for the construction of the components of the logic are to retain the ease of use and to be close to existing programming logics in order to support them.

One of the restrictions of existing logics is the underspecification mechanism used in several systems to deal with error or undefinedness like division by zero, 1/0, or asking for the first element of an empty list, (`head Nil`). Closely related is the often used restriction that defined functions must terminate, where either a proof has to be provided before any other logical formula makes sense, or the definition of nonterminating functions is prevented by the definition mechanism. We remove both restrictions and treat errors as is usual in semantics of programming languages: all errors and undefinedness are indiscriminatingly viewed as nontermination. Note that our view is that typing errors are real programming mistakes and thus untyped programs should be rejected by an appropriate type system.

There is a history of logics and proof systems. Inductive proof systems are [BM75, KM86]. A system with a strong influence on our logic is the automated

proof system `VeriFun` (see [WS05, SWGA07, Wal94] and ([Ver11])). Standard, classical and prominent examples of proofs are commutativity of addition (on Peano-numbers), associativity of the append-function on lists, or even more complex (encoded) theorems like the undecidability of the Halting Problem (see [Ver11] and [BM84] for an early automated proof).
Most proof systems require termination of the defined functions (Isabelle/HOL: [NPW02], Coq: [BC04], and `VeriFun`). There may be special extensions to allow partial functions (see e.g. [Kra06, BK08]), like the selector-functions *head* and *tail* for lists, or the treatment of error-elements as underspecification.
An interesting overview and discussion on partiality, many-valued logics and underspecification is in [Häh05], who proposes two-valued logics, and underspecification to be as exact as possible, but also contains a warning that the underspecification mechanism may lead in certain cases to "overspecification".
Detailing on this, consider as example the proposition

$$(tail\ \texttt{Nil}) = (tail\ (tail\ \texttt{Nil})) \implies (tail\ \texttt{Nil}) = \texttt{Nil}.$$

The mechanism of underspecification in some systems (see e.g. [WS05, SWGA07, Ade09]) treats this as follows: there is one undefined object of type list, which has a value, but the value is not known. Let $\star$ denote such an unknown value of list type. Then the approaches internally use the following definition of the *tail*-function:

$$\begin{aligned} tail\ (x : xs) &= xs \\ tail\ \texttt{Nil} \quad &= \star \end{aligned}$$

Thus we obtain that $(tail\ \texttt{Nil})$ is an error, but in the three occurrences in the proposition, it is *the same* list $\star$. Thus there are the cases that $\star = \texttt{Nil}$, and both equations are true, or $\star$ is a nonempty list, but since lists are finite, $\star = (tail\ \star)$ is false in this case. Thus the proposition above is a theorem under underspecification, which is clearly an artifact of the method.
Other related work on specifications and a logic of functional programs also allowing polymorphism and partiality is for example in ([SM09]). Program transformations in higher-order term rewriting with constructors using an operational correctness criterion and also permitting induction principles is in ([CAT05]).

Our logic consists of a functional programming part and a logical part. The programming part is a strict functional programming language allowing the definition of data types like lists and the definition of recursive functions. Defined functions may have a polymorphic type. The logical part is a two-valued first order logic where the only predicate is equality (=). The formulas are monomorphic where the quantification is over closed data values of the appropriate type. Since the quantifier restriction fixes the domain it is like a second order logic and thus allows induction principles. The semantics of equality is contextual semantics: i.e. $s = t$ is valid, iff for all contexts C such that $C[s]$ and $C[t]$ are closed, the evaluation of $C[s]$ terminates with a value if and only if the evaluation of $C[t]$ terminates with a value. Formulas are *theorems* of the logic iff they hold under any extension of the program by further function definitions or data

types (that is they are *(locally) valid* for every extension), which makes the logic "monotonous" w.r.t. such extensions.
For example, this allows to specify appending lists, formulate the associativity statement as

$$\forall x, y, z :: (\texttt{List Nat}) : (\texttt{append}\ x\ (\texttt{append}\ y\ z)) = (\texttt{append}\ (\texttt{append}\ x\ y)\ z)$$

and also to prove it using induction proof rules. In our logic $(\mathit{tail}\ \texttt{Nil})$ is interpreted as nontermination, and thus the formula $(\mathit{tail}\ \texttt{Nil}) = (\mathit{tail}\ (\mathit{tail}\ \texttt{Nil})) \implies (\mathit{tail}\ \texttt{Nil}) = \texttt{Nil}$ is not valid in our logic, since it evaluates to $\bot = \bot \implies \bot = \texttt{Nil}$, where $\bot$ is a representative for nonterminating expressions, and $\bot = \bot$ is valid while $\bot = \texttt{Nil}$ is invalid.
Other predicates than equality like $\leq$ on Peano-numbers could be specified as functions with Boolean result, and transitivity of $\leq$ is the formula $\forall x, y, z :: \texttt{Nat}.(x \leq y = \texttt{True}) \wedge (y \leq z = \texttt{True}) \implies (x \leq z = \texttt{True})$.
The semantics renders equality undecidable, however, this is not a problem, since – as we show – a sufficient set of proof rules and transformations can be proved as correct in the logic.
Besides the construction and definition of a program logic our main results are (i) proofs of global correctness of deduction and transformations of almost all rules employed in `VeriFun`, and also further rules; (ii) conservativity theorems showing that for several forms of formulas their local validity already implies that the formulas are theorems.
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 locally valid formulas are shown to hold in all extensions, i.e. they are theorems of the logic. An important class are universally quantified equations (Theorem 7.1) and general monomorphic theorems if functions do not occur in the data (Theorem 7.4). In general, the local validity of formulas does not imply that they are also tautologies (see Theorem 7.10).
As a side effect of the general logic the following generalizations for `VeriFun` (see [Ver11]) would be possible: 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 ([MT91]) from context lemmas (for a general approach see also [SSS10a]), and an adaptation of the subterm property of simply-typed lambda-calculi.
An interesting generalization of the logic's expressiveness are polymorphic formulas (the quantified type may have type variables). Local validity of polymorphic theorems does in general not imply that these are theorems. We conjecture that

polymorphic formulas that are universally quantified polymorphic equations and which are locally valid also hold in all 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. In Section 6 we explain the logic, its semantics and analyze some conservativity properties and state open questions. In Section 8 we consider polymorphic formulas. We discuss the differences of our approach in contrast to the logic used by `VeriFun` in Section 9 and finally conclude in Section 10. 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 $K_1, K_2 \in \mathcal{K}$ we assume $D_{K_1} \cap D_{K_2} = \emptyset$ 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: $T ::= X \mid (T_1 \to T_2) \mid (K\ T_1 \dots T_{ar(K)})$, where the symbols X, X_i are type variables, T, T_i stand for types, and $K \in \mathcal{K}$ is a type constructor. As usual we assume function types to be right-associative, i.e. $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$. Types of the form $T_1 \to T_2$ are called *arrow types*, and types $(K\ T_1 \dots T_{ar(K)})$ are called *constructed types*. T represent poymorphic types, we will also 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 $\forall X_1, \dots, X_{ar(K)}.T_{K,i,1} \to \dots \to T_{K,i,m_i} \to K\ X_1 \dots X_{ar(K)}$, where $m_i = ar(c_{K,i})$, $X_1, \dots, X_{ar(K)}$ are distinct type variables, and only the variables X_i occur as free type variables in $T_{K,i,1}, \dots, T_{K,i,m_i}$.

2.2 Syntax of Expressions

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:

$$\begin{aligned} s, s_i, t \in Expr ::= & \; x \mid f \mid (s\ t) \mid \lambda x.s \mid (c_i\ s_1 \ldots s_{ar(c_i)}) \\ & \mid (\mathtt{case}_K\ s\ Alt_1 \ldots Alt_n) \text{ where } n = |D_K| \\ Alt_i ::= & \; ((c_i\ x_1 \ldots x_{ar(c_i)})\ \mathtt{->}\ s_i) \end{aligned}$$

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 $((c_i\ x_1 \ldots x_{ar(c_i)})\ \mathtt{->}\ s_i)$ 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 $v, v_i \in Val ::= x \mid \lambda x.s \mid (c\ v_1 \ldots v_n)$, i.e. a variable, an abstraction, or a constructor-expression $(c\ v_1 \ldots v_n)$, where the immediate subexpressions are also values. For 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:

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

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

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

Fig. 1. Reduction Rules

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. For every $f \in \mathcal{F}$ the pair (f, d_f) is labeled with a perhaps quantified type. We also assume that $\mathcal{P}$-expressions, which are used for evaluation and in formulas, are monomorphically typed. For $\mathcal{P}$-expressions we assume that variables x have a built-in type (denoted by x^τ if x has built-in type τ), and 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

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

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 $\mathcal{P}'$.*

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:

$$\begin{array}{rl} R \in \mathcal{R} ::= & [\cdot] \mid (R\ s) \mid (v\ R) \mid \texttt{case}_K\ R\ alts \\ & \mid\ (c\ v_1\ \dots\ v_{i-1}\ R\ s_{i+1} \dots\ s_n)\end{array}$$

where s, s_i are $\mathcal{P}$-expressions and v, v_i are $\mathcal{P}$-values.

Definition 3.1. Reduction rules *are defined in Fig. 1 without mentioning all types. The* standard reduction $\to$ *(sometimes also denoted as $\xrightarrow{sr}$) is defined as a reduction using one of the standard reduction rules in a reduction context, i.e. $s \xrightarrow{sr} t$ iff $s = R[s'], t = R[t']$ where $s' \to t'$ by a rule of Fig. 1 and $R \in \mathcal{R}$.*

The evaluation *of an expression s 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 (standard) reduction is a closed value.

Remark 3.2. Note that the standard reduction does not rely on the typing of the expressions. An equivalent definition of the standard reduction of a typed expression s would be to apply the untyped reduction rules in reduction contexts to the type erasure of s. Nevertheless, our definition has the advantage that type safety is easy to verify and that correctness proofs for program transformations are sometimes easier, since they work on type labeled expressions.

Example 3.3. Let *map* be defined as before and let *not* be defined as $not = \lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})\ (\mathtt{True} \mathtt{->} \mathtt{False})$. An example of an evaluation is (type-labels are omitted):

$$
\begin{array}{ll}
 & map\ not\ (\mathtt{True} : \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{delta}} & (\lambda f, xs.\mathtt{case}_{\mathrm{List}}\ xs\ ((y:ys) \mathtt{->} (f\ y : map\ f\ ys))(\mathtt{Nil} \mathtt{->} \mathtt{Nil})) \\
 & \quad not\ (\mathtt{True} : \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{delta}} & (\lambda f, xs.\mathtt{case}_{\mathrm{List}}\ xs\ ((y:ys) \mathtt{->} (f\ y : map\ f\ ys))(\mathtt{Nil} \mathtt{->} \mathtt{Nil})) \\
 & \quad (\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False})) \\
 & \quad (\mathtt{True} : \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{beta}} & (\lambda xs.\mathtt{case}_{\mathrm{List}}\ xs\ \ \ldots\ \ \ldots \\
\xrightarrow{sr,\mathrm{beta}} & \mathtt{case}_{\mathrm{List}}\ (\mathtt{True} : \mathtt{Nil}) \\
 & \quad ((y:ys) \mathtt{->} ((\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False}))\ y) \\
 & \quad\quad : (map\ (\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False}))\ ys)) \\
 & \quad (\mathtt{Nil} \mathtt{->} \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{case}} & ((\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False}))\ \mathtt{True}) \\
 & \quad : (map\ (\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False}))\ \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{beta}} & (\mathtt{case}_{\mathtt{Bool}}\ \mathtt{True}\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False})) \\
 & \quad : (map\ (\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False}))\ \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{case}} & \mathtt{False} : (map\ (\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False}))\ \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{delta}} & \mathtt{False} : ((\lambda f, xs.\mathtt{case}_{\mathrm{List}}\ xs\ \ldots) \\
 & \qquad\qquad (\lambda b.\mathtt{case}_{\mathtt{Bool}}\ b\ (\mathtt{False} \mathtt{->} \mathtt{True})(\mathtt{True} \mathtt{->} \mathtt{False}))\ \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{beta}} & \mathtt{False} : ((\lambda xs.\mathtt{case}_{\mathrm{List}}\ xs\ \ldots)\ \mathtt{Nil}) \\
\xrightarrow{sr,\mathrm{beta}} & \mathtt{False} : (\mathtt{case}_{\mathrm{List}}\ \mathtt{Nil}\ \ldots (\mathtt{Nil} \mathtt{->} \mathtt{Nil})) \\
\xrightarrow{sr,\mathrm{case}} & \mathtt{False} : \mathtt{Nil}
\end{array}
$$

3.1 Assumptions on Valid Programs

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

Remark 3.5. This excludes types like the type `Foo` with one constructor `foo` :: `Foo` $\rightarrow$ `Foo`. The only potentially closed value would be an infinitely nested expression `foo`(`foo`(...)), which does not exist.

Definition 3.6. *We say an expression s is an Ω-*expression *iff for all value substitutions σ where $\sigma(s)$ is closed, $\sigma(s)\Uparrow$ holds.*

Assumption 3.7 *We assume that for every program $\mathcal{P}$ and for every $\mathcal{P}$-type τ there is a closed Ω-expression, denoted as $\bot^\tau$.*

The second assumption is satisfied if there is a single definition $f = (\lambda x.f\ x) :: \forall a, b.a \rightarrow b$. 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

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

Remark 3.8. In the following we will sometimes assume that there are constants $\mathtt{Bot}^T$ of every type T that represent the expressions $\bot^T$. We assume that these constants are divergent. There are reduction rules (see 2) for these constants. The use of these constants and the reduction rules does not change the equivalence of expressions, as proved below.

3.2 Equivalence of Expressions

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

Definition 3.9. *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 3.10. 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 $s \sim_{\mathcal{P},T} t$.

Definition 3.11. *A* program transformation $\mathcal{T}$ *is defined as 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* locally correct *iff for all $(s,t) \in \mathcal{T}$ of type T the equation $s \sim_{\mathcal{P},T} t$ holds. $\mathcal{T}$ is called* 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 that extend 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 σ 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.*

The proof can be found in the appendix B.
In the appendix (Proposition B.12) we show:

Proposition 4.2. *The reductions (beta), (delta), and (case) are locally 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 locally correct transformation.*

Note that ordinary (i.e. call-by-name) beta-reduction is in general not correct, for instance $(\lambda x.\texttt{True})\ \bot$ is equivalent to $\bot :: \texttt{Bool}$, 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.
Now we analyze so-called Ω-expressions, i.e. terms that diverge after closing them by an arbitrary value substitution.
The property of being an Ω-expression inherits to reduction contexts, i.e. if $s :: T$ is an Ω-expression, and $R[\cdot :: T]$ a reduction context, then $R[s]$ is also an Ω-expression (see Appendix B.4, Proposition B.18). The CIU-Theorem also

implies that Ω-expressions are least elements w.r.t. contextual ordering, and that Ω-expressions of the same type form a single equivalence class w.r.t. $\sim_{\mathcal{P},T}$:

Corollary 4.3. *Let $s, t :: T$ and let s be an Ω-expression. Then $s \leq_{\mathcal{P},T} t$. If also t is an Ω-expression, then $s \sim_{\mathcal{P},T} t$.*

5 Global Correctness of Program Transformations

This section proves criteria for (global) contextual equality of expressions that are easier to use than the definition. In particular, it is shown that contextual 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 $s, t :: T$ be (open) $\mathcal{P}$-expressions. Then $s \leq_{\mathcal{P},T} t$ iff for all closing $\mathcal{P}$-value-substitutions σ: $\sigma(s) \leq_{\mathcal{P},T} \sigma(t)$.*

Proof. If $s \leq_{\mathcal{P},T} t$, then $\sigma(s) \leq_{\mathcal{P},T} \sigma(t)$ for closing value substitutions σ 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 $s\downarrow$ and $t\downarrow$. 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 4.3. □

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 $\mathcal{P}$-value 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 σ and F-free $\mathcal{P}$-reduction contexts R, where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P}',T} t$ holds.*

$$
\begin{array}{l|l}
\texttt{Bot}\ s \to \texttt{Bot} & (c\ \ldots \texttt{Bot} \ldots) \to \texttt{Bot} \\
s\ \texttt{Bot} \to \texttt{Bot} & (t; \texttt{Bot}) \to \texttt{Bot} \\
\texttt{case}_K\ s\ (p_1 \to \texttt{Bot}) \ldots\ (p_n \to \texttt{Bot}) \to \texttt{Bot} & (\texttt{Bot}; t) \to \texttt{Bot} \\
\texttt{case Bot}\ \mathit{Alts} \to \texttt{Bot} &
\end{array}
$$

Fig. 2. `Bot`-reduction rules

$$
\begin{array}{lll}
\text{(seq)} & v; s & \to s \quad \text{if } v \text{ is a value} \\
\text{(seqseq)} & ((s_1; s_2); s_3) & \to (s_1; (s_2; s_3)) \\
\text{(seqapp)} & ((s_1; s_2)\ s_3) & \to (s_1; (s_2\ s_3)) \\
\text{(seqc)} & ((c\ s_1 \ldots s_n); s) & \to (s_1; (\ldots (s_n; s) \ldots)) \\
\text{(caseseq)} & (\texttt{case}_K\ (r; s)\ \mathit{alts}) & \to (r; (\texttt{case}_K\ s\ \mathit{alts})) \\
\text{(VNbeta)} & ((\lambda x.s)\ t) & \to (t; s[t/x]) \\
\text{(VNcase)} & \texttt{case}_K\ (c\ s_1 \ldots s_n)\ \ldots\ ((c\ x_1 \ldots x_n)\ \texttt{->}\ t)\ \ldots & \\
 & \to (s_1; (\ldots (s_n; t[s_1/x_1, \ldots, s_n/x_n]))) &
\end{array}
$$

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

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

Fig. 4. Case-Shifting Transformations

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

Main 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 $s \leq_{\mathcal{P}\forall,T} t$.*
Hence, also $s \sim_{\mathcal{P},T} t$ iff $s \sim_{\mathcal{P}\forall,T} t$.

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

This main theorem states a central result, which is the foundation of several results on the localization of tautologies.
Now we prove (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 Main Theorem 5.5. The other reductions are proved correct in the appendix (Theorem C.22). □

A direct consequence of Main Theorem 5.5 and Corollary 4.3 is that all Ω-expressions are in a single equivalence class of the global equivalence, i.e.:

Corollary 5.7. *Let $s, t :: T$ be expressions, such that s is an Ω-expression. Then $s \leq_{\mathcal{P}\forall,T} t$ holds. If t is also an Ω-expression, then $s \sim_{\mathcal{P}\forall,T} t$.*

We prove some further helpful lemmas.

Lemma 5.8. *For all data constructors c_i and expressions s_i, t_i:*

$$(c\ s_1\ \ldots\ s_{ar(c_i)}) \sim_{\mathcal{P},T} (c\ t_1\ \ldots\ t_{ar(c_i)}) \iff \forall i : s_i \sim_{\mathcal{P},T} t_i$$

Proof. One direction holds, since $\sim_{\mathcal{P},T}$ is a congruence. For the other direction assume that the equation $(c\ s_1\ \ldots\ s_{ar(c_i)}) \sim_{\mathcal{P},T} (c\ t_1\ \ldots\ t_{ar(c_i)})$ holds. Consider the context

$$C = \mathsf{case}\ [\cdot]\ \ldots\ ((c\ y_1\ \ldots\ y_{ar(c_i)})\ \texttt{->}\ y_i)\ \ldots.$$

Correctness of the reduction rule (case) implies $t_i \sim_{\mathcal{P},T'} C[(c\ t_1\ \ldots\ t_{ar(c_i)})] \sim_{\mathcal{P},T'} C[(c\ s_1\ \ldots\ s_{ar(c_i)})] \sim_{\mathcal{P},T'} s_i$. □

Lemma 5.9. *For all expressions s, t of the same type T: $\lambda x.s \sim_{\mathcal{P},T'} \lambda x.t \iff s \sim_{\mathcal{P},T} t$*

Proof. Since $\sim_{\mathcal{P},T}$ is a congruence, one direction is trivial. For the other direction let $\lambda x.s \sim_{\mathcal{P},T'} \lambda x.t$. Correctness of (beta) shows $s \sim_{\mathcal{P},T} ((\lambda x.s)\ x) \sim_{\mathcal{P},T} ((\lambda x.t)\ x) \sim_{\mathcal{P},T} t$. □

6 The Logic

In this section we introduce the logic, i.e. we define the syntax of (monomorphic) formulas, and the interpretation of formulas. Then we introduce the notion of valid formulas and of tautologies.
The syntax of monomorphic formulas (w.r.t. a program $\mathcal{P}$) is as follows: We use true and false for the *logical* truth values. *Atoms* are given by the grammar:

$$A ::= \mathsf{true} \mid \mathsf{false} \mid (s\ =\ t)$$

The syntax of *formulas* is defined by the grammar:

$$F ::= A \mid F \vee F \mid F \wedge F \mid \neg F \mid \forall x :: T.F \mid \exists x :: T.F$$

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 type 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:*

$$
\begin{array}{ll}
I_{\mathcal{P}'}(\mathsf{true}) & = \mathsf{true} \\
I_{\mathcal{P}'}(\mathsf{false}) & = \mathsf{false} \\
I_{\mathcal{P}'}(s = t) & = \mathsf{true}, \textit{ if } s \sim_{\mathcal{P}',\tau} t \quad \textit{for expressions } s,t :: \tau \\
I_{\mathcal{P}'}(s = t) & = \mathsf{false}, \textit{ if } s \not\sim_{\mathcal{P}',\tau} t \quad \textit{for expressions } s,t :: \tau \\
I_{\mathcal{P}'}(A \wedge B) & = I_{\mathcal{P}'}(A) \wedge I_{\mathcal{P}'}(B) \\
I_{\mathcal{P}'}(A \vee B) & = I_{\mathcal{P}'}(A) \vee I_{\mathcal{P}'}(B) \\
I_{\mathcal{P}'}(\neg A) & = \neg I_{\mathcal{P}'}(A)
\end{array}
$$

$$
I_{\mathcal{P}'}(\forall x :: \tau.F) = \begin{cases} \mathsf{true}, & \textit{if for all } a \in \mathcal{M}_{\mathcal{P}',\tau} : I_{\mathcal{P}'}(F[a/x]) = \mathsf{true} \\ \mathsf{false}, & \textit{otherwise} \end{cases}
$$

$$
I_{\mathcal{P}'}(\exists x :: \tau.F) = \begin{cases} \mathsf{true}, & \textit{if for some } a \in \mathcal{M}_{\mathcal{P}',\tau} : I_{\mathcal{P}'}(F[a/x]) = \mathsf{true} \\ \mathsf{false}, & \textit{otherwise} \end{cases}
$$

A closed monomorphic $\mathcal{P}$-formula F is valid for $\mathcal{P}'$ *iff $I_{\mathcal{P}'}(F) = \mathsf{true}$ and it is a* $\mathcal{P}$-tautology *(or a* $\mathcal{P}$-theorem*) iff it is valid for all extensions $\mathcal{P}'$ of program $\mathcal{P}$.*

Note that the definition of tautologies implies that our logic is "monotonous" w.r.t. program extensions. I.e. if F is a $\mathcal{P}$-tautology and $\mathcal{P}'$ is an extension of $\mathcal{P}$, then F is also a $\mathcal{P}'$-tautology, which is a trivial consequence of the definition. An automated verification system cannot compute all program extensions $\mathcal{P}' \sqsupseteq \mathcal{P}$ to prove a $\mathcal{P}$-tautology. As a potential remedy, we will show that for a large set of monomorphic formulas $\mathcal{P}$-validity already implies the $\mathcal{P}$-tautology property. In the following we call this property "conservativity of extensions". In Section 7.3 we show that conservativity of extensions does not hold in general.
A first consequence of the definition is

Proposition 6.2. *Let $\mathcal{P}$ be a program and F be a $\mathcal{P}$-formula. Then F is $\mathcal{P}$-valid iff $\neg(F)$ is not $\mathcal{P}$-valid.*

Note that the proposition is wrong for $\mathcal{P}$-tautologies (see Theorem 7.10). Since local and global equivalences coincide (see the Main 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 $\mathcal{P}' \sqsupseteq \mathcal{P}$. An easy case is:

Theorem 6.3. *Let F be a closed monomorphic and quantifier-free $\mathcal{P}$-formula. Then F is $\mathcal{P}$-valid iff F is a $\mathcal{P}$-tautology.*

Proof. If the formula is an atom $s = t$, then $I_{\mathcal{P}'}(s = t) = I_{\mathcal{P}}(s = t)$ for any $\mathcal{P}' \sqsupseteq \mathcal{P}$, since local and global correctness of equations coincide (see Main Theorem 5.5). For complex formulas the equivalence follows by induction over the structure of the quantifier-free formula. □

Example 6.4. Given appropriate definitions of the data type `Nat` with two constructors $0, \texttt{Succ}$, where `pred`, defined as $\lambda x.\texttt{case}_{\texttt{Nat}}\ x\ (0 \to \bot)\ ((\texttt{Succ}\ y) \to y)$, is a function that acts like a selector for `Succ`, and where also addition $+$ is 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.

7 Proof Methods for Tautologies

In this section we show several cases where local $\mathcal{P}$-validity of a formula F implies that F is a tautology. But we also prove that this property does not hold in general. Finally, we provide an induction scheme and present some proof rules.

7.1 Universally Quantified Formulas: Conservativity

The following theorem shows that it is 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 $\mathcal{P}$-tautologies iff they are valid for $\mathcal{P}$, i.e.:

Theorem 7.1. *Let $\mathcal{P}$ be a program and*

$$F := \forall x_1 :: T_1, \ldots, x_n :: T_n\ .\ s = t$$

be a $\mathcal{P}$-valid formula. Then for all $\mathcal{P}' \sqsupseteq \mathcal{P}$, the formula F is also valid for $\mathcal{P}'$, i.e., the formula is a $\mathcal{P}$-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 $\mathcal{P}$-tautologies. This also holds for the correct program transformations (seen as equations) that we already exhibited in Propositions 4.2 and 5.3. In the following we investigate extensions of Theorem 7.1. First we look for values without higher-order subexpressions, like Peano-numbers, Booleans and lists of Peano-numbers.

Definition 7.2. *A type T is a* DT-type*, if every closed value of type T is only built from data constructors.*

Lemma 7.3. *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 7.4. *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 valid for $\mathcal{P}$ iff it is a $\mathcal{P}$-tautology.*

Proof. The definition of DT-types implies that the sets $\mathcal{M}_{\mathcal{P},T}$ do not change when the program is extended. Lemma 7.3 and Theorem 5.4 show that all closed $\sim$-equalities are globally correct. □

7.2 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 the scope of the `VeriFun`-system (where termination is an a priori requirement). Given a program $\mathcal{P}$ we define $\mathcal{P}_D \supset \mathcal{P}$, 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 $defined_T$, has the following property: $(defined_T\ s) \xrightarrow{*} \texttt{True}$ for every converging expression of DT-type T. The function can only produces `True`, and does not terminate if the argument is not terminating.

Definition 7.5. *The translation B is defined as: $B(\mathsf{true}) = \texttt{True}$, $B(\mathsf{false}) = \texttt{False}$, $B(\wedge) \equiv \lambda x, y.and\ x\ y$, $B(\vee) \equiv \lambda x, y.or\ x\ y$, $B(\neg) \equiv \lambda x.not\ x$, 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 $\texttt{Bot} = \texttt{Bot}$ 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 $defined(s_1) = \texttt{True} \wedge \ldots \wedge defined(s_n) = \texttt{True}$, where $s_i, i = 1, \ldots, n$ are all the expressions that occur as top-expressions in equations of F.

Theorem 7.6. *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 valid for $\mathcal{P}$. Then the formula $\forall x_1 :: T_1, \ldots, x_n :: T_n.Def(F) \implies F$ is also a $\mathcal{P}$-theorem.*

Proof. The formula $\forall x_1 :: T_1, \ldots, x_n :: T_n\ .Def(F) \implies F$ is valid for $\mathcal{P}_D$ if and only if $\lambda x_1, \ldots, x_n.B(Def(F) \implies F) \sim_{\mathcal{P}_D,T} \lambda x_1, \ldots, x_n.B(Def(F))$: Let $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ be a substitution of closed values for the x_i, and let s_j be the toplevel expressions in the formula 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 $\mathit{defined}(s_1) = \texttt{True}$ is false under the interpretation using σ, hence the whole formula is true. For the corresponding substitution, both abstractions are equivalent to `Bot`. The claim is equivalent to $\lambda x_1, \ldots, x_n.B(\mathit{Def}(F) \implies F) \sim_{\mathcal{P}'_D,T} \lambda x_1, \ldots, x_n.B(\mathit{Def}(F))$, which holds by Theorem 5.4 for any extension $\mathcal{P}'_D$ of $\mathcal{P}_D$. The latter again implies that $\forall x_1 :: T_1, \ldots, x_n :: T_n\ .\mathit{Def}(F) \implies F$ is valid for $\mathcal{P}'$. Now the CIU-Theorem implies that the formula is a tautology. □

7.3 Valid Monomorphic Formulas Might not be Theorems

We show in this subsection that in general, monomorphic $\mathcal{P}$-formulas that are valid for $\mathcal{P}$ might not be $\mathcal{P}$-tautologies. This is shown for a program $\mathcal{P}_{simple}$ that has lists, Booleans, and Peano-numbers as data structures, there is a $\bot$ for every type, but no other defined functions. Then the following holds:

Lemma 7.7. *It is decidable, whether an arbitrary $\mathcal{P}_{simple}$-expression terminates.*

Proof. The argument is that reduction of $\mathcal{P}_{simple}$-expressions strongly terminates, if we assume that `Bot`s of different types are given as constants. The full proof requires several pages (see e.g. [SSS10b]), but can easily be derived from the standard proofs, where the changes are described in the following paragraph. But note that it is not true that the reduction of monomorphic expressions (of any program $\mathcal{P}$) always terminates.
The basic definition is that of strongly computable (SC) expressions, which is as follows: An expression t is SC, if it is strongly normalizable (i.e. its reduction terminates, SN), and if it is of type $T_1 \to \ldots \to T_n \to T_{n+1}$ where T_{n+1} is not a function type, then for all SC-expressions $s_1, \ldots s_n$ with $s_i :: T_i$, $(t\ s_1 \ldots s_n)$ is SN and if $(t\ s_1 \ldots s_n) \xrightarrow{*} (c\ t'_1 \ldots t'_m)$, where c is a data constructor, then also $t'_1, \ldots, t'_m$ are SC.
Then it can be shown that every SC-expression is strongly normalizing and that all $\mathcal{P}_{simple}$-expressions are SC. □

Now we extend the program $\mathcal{P}_{simple}$ with a new functional symbol g which encodes a universal Turing machine. We denote the extended program with $\mathcal{P}'$. We only sketch the encoding of the Turing machine: The definitional equation is of the form $g = \lambda lt, rt, st.r$ where lt and rt are parameters for two lists encoding the tape to the left and the right of the head (we assume that these are Boolean lists), and st is the parameter for state of the Turing machine (encoded as a Peano-number). The body r now checks if st is a final state. If this is true, then the result is `True`. Otherwise, r performs one step of the Turing machine and then recursively calls g.
Now we construct a formula that states the existence of an encoding of a universal Turing machine. Let the (closed) formula F_T be defined as $F_T :=$

$$\exists g\ \forall\ lt :: (\texttt{List Bool}), rt :: (\texttt{List Bool}), st :: \texttt{Nat} : g\ lt\ rt\ st = r$$

Note that in the formula the symbol g (and all occurences of g in r) is a variable representing a value.
In case such a value g exists, it is undecidable, given any lt, rt, st whether $g\ lt\ rt\ st \sim \texttt{Bot}$, since this is exactly the Halting Problem for Turing machines. Since in $\mathcal{P}_{simple}$ the termination of every expression is decidable, we have:

Proposition 7.8. *The formula F_T is not valid in $\mathcal{P}_{simple}$.*

However, for the extended program $\mathcal{P}'$, the value $(\lambda lt, rt, st.r)$ validates the formula. Thus the following holds:

Proposition 7.9. *The formula F_T is valid in the extension $\mathcal{P}'$ of $\mathcal{P}_{simple}$.*

Since $\mathcal{P}'$ is an extension of $\mathcal{P}_{simple}$, $\mathcal{P}_{simple}$-validity does in general not imply the tautology property, and thus we have:

Theorem 7.10. *For closed monomorphic formulas F, in general $\mathcal{P}$-validity of F does not imply that F is also a $\mathcal{P}$-tautology.*

Proof. The formula $\neg F_T$ is $\mathcal{P}_{simple}$-valid according to Propositions 7.8 and 6.2, but not in a certain extension of $\mathcal{P}_{simple}$ according to Proposition 7.9. Thus it cannot be a $\mathcal{P}_{simple}$-tautology. □

7.4 Open Question

It remains open whether every $\mathcal{P}$-valid monomorphic formula of the form $\forall^* : A$, where A is quantifier-free is also a $\mathcal{P}$-tautology.
The previous considerations show that this holds for several special forms of A. The obstacle for a proof attack are that in case of a quantified variable $x :: T$ where T has a function type as subtype, the extension of a program $\mathcal{P}$ to $\mathcal{P}'$ might lead to an extended set of $\mathcal{P}'$-values of type T.

7.5 Proof by Induction

Induction proofs of (universally quantified) tautologies are mostly done over the structure of the data, if the type of the quantified variables corresponds to data. Values of DT-types are finite and only consist of data constructors, hence induction can be applied.
If the to-be-proved $\mathcal{P}$-formula is universally quantified, i.e.,

$$F = \forall x_1 :: T_1, \ldots x_n :: T_n.A(x_1, \ldots, x_n),$$

where $A(x_1, \ldots, x_n)$ denotes a formula with occurrences of the variables $x_i, i = 1, \ldots, n$, and if T_i are DT-types, then induction on the structure of the values of type T_i is possible. Instead of the structure of the values, other well-founded orderings on the tuples in $\mathcal{M}_{\mathcal{P},T_1} \times \ldots \times \mathcal{M}_{\mathcal{P},T_n}$ may be used. The subformula $A(x_1, \ldots, x_n)$ is usually quantifier-free, but the induction principle is independent of the form of A, so it can also be used for arbitrary forms of A.
There are at least two variants of proving a formula to be a $\mathcal{P}$-tautology:

1. Use induction to prove that the formula is $\mathcal{P}$-valid, and then use one of the conservativity theorems 6.3, 7.1, 5.5, 7.4, and 7.6, to show that the formula is a $\mathcal{P}$-tautology.
2. Use induction to prove that the formula is a $\mathcal{P}$-tautology, where the formula for the base and the induction step have to proved to be $\mathcal{P}$-tautologies.

As a prototypical example, we describe a more automatable induction scheme for inductively defined data structures. Let T be a monomorphic type and let $(\texttt{List}\ T)$ have the two constructors $\texttt{Nil} :: (\texttt{List}\ T)$ and $\texttt{Cons} :: T \to (\texttt{List}\ T) \to (\texttt{List}\ T)$.

Definition 7.11 (Induction Scheme for Lists). *Let $\forall x :: (\texttt{List}\ T).F$ be a closed $\mathcal{P}$-formula. Assume that the following holds:*

1. *The formula $F[\texttt{Nil}/x]$ is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology).*
2. *The following formula is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology):*
   $\forall z :: (\texttt{List}\ T).(F[z/x] \implies \forall y :: T.F[(\texttt{Cons}\ y\ z)/x])$

Then the formula $\forall x :: (\texttt{List}\ T).F$ is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology).

This scheme can be generalized and adapted to other type constructors and their data constructors.

7.6 Proof Rules for Monomorphic Formulas

In this section we summarise our investigation on monomorphic formulas by providing proof rules. The notation $\mathcal{P} \vdash F$ means that F is provable to be $\mathcal{P}$-valid, where it is assumed in this section that F is a closed monomorphic formula, and that F can be formulated in $\mathcal{P}$. The notation $\forall\mathcal{P} \vdash F$ means that F is a $\mathcal{P}$-tautology, and can be formulated in $\mathcal{P}$. The following proof rules are sound for closed monomorphic $\mathcal{P}$-formulas.
There are proof rules for $\mathcal{P}$-validity and for $\mathcal{P}$-tautology.
We omit the usual proof rules for predicate logic, which are correct in pur logic.

Quantifier-free and Propositional formulas.

$$\frac{\begin{array}{l}F(x_1,\ldots,x_n) \text{ is a propositional tautology}\\ \text{with the propositional variables } x_1,\ldots x_n\\ \text{and } F_1,\ldots F_n \text{ are closed monomorphic formulas}\end{array}}{\forall\mathcal{P} \vdash F(F_1,\ldots,F_n)}$$

$\mathcal{P}$-valid and negation. For a closed formula F:

$$\frac{\mathcal{P} \nvdash F}{\mathcal{P} \vdash \neg F}$$

Atoms, Congruence and Ω-Expressions. We now consider rules for atoms, where we omit types in the notation. The following rules are sound, since $\sim_{\mathcal{P},T}$ is a congruence, due to Main Theorem 5.5, and due to the results on Ω-expressions.

$$\frac{\mathcal{P} \vdash s = t}{\forall\mathcal{P} \vdash s = t}$$

$$\forall\mathcal{P} \vdash s = s \qquad \frac{\mathcal{P} \vdash s = t}{\mathcal{P} \vdash t = s}$$

$$\frac{\mathcal{P} \vdash s = t \quad \mathcal{P} \vdash t = r}{\mathcal{P} \vdash s = r} \qquad \frac{\mathcal{P} \vdash s = t}{\mathcal{P} \vdash C[s] = C[t]}$$

$$\frac{s,t \text{ are } \Omega\text{-expressions}}{\forall\mathcal{P} \vdash s = t} \qquad \frac{s \text{ is an } \Omega\text{-expression}, t \in \mathcal{M}_{\mathcal{P},T}}{\forall\mathcal{P} \vdash \neg(s = t)}$$

Correctness of Term-Transformations. Correctness of the call-by-value reductions and the VN-reductions implies the soundness of the rule, where $\xrightarrow{trans}$ is the union of (beta), (case), (delta) and the VN-reductions.

$$\frac{s \xrightarrow{trans} t}{\forall\mathcal{P} \vdash s = t}$$

Argument Selection and Abstractions. Lemmas 5.8 and 5.9 imply the soundness of the following rules, where c, c_i are data constructors:

$$\frac{\mathcal{P} \vdash (c\ s_1\ \ldots\ s_n) = (c\ t_1\ \ldots\ t_n)}{\mathcal{P} \vdash s_i = t_i}$$

$$\mathcal{P} \vdash \neg(c_i\ s_1\ \ldots\ s_n = c_j\ t_1\ \ldots\ t_m) \qquad \text{if } c_i \neq c_j$$

$$\frac{\mathcal{P} \vdash \lambda x_1.\ldots\lambda x_n.s = \lambda x_1.\ldots\lambda x_n.t}{\mathcal{P} \vdash \forall x_1,\ldots,x_n.s = t}$$

$$\frac{\mathcal{P} \vdash \forall x_1,\ldots,x_n.s = t}{\mathcal{P} \vdash \lambda x_1.\ldots\lambda x_n.s = \lambda x_1.\ldots\lambda x_n.t}$$

Tautologies.

$$\frac{\forall\mathcal{P} \vdash F}{\mathcal{P} \vdash F}$$

$$\frac{\mathcal{P} \vdash \forall x_1,\ldots x_n.s = t}{\forall\mathcal{P} \vdash \forall x_1,\ldots x_n.s = t} \quad \text{if } \forall x_1,\ldots x_n.s = t \text{ is closed}$$

$$\frac{\mathcal{P} \vdash F}{\forall\mathcal{P} \vdash F} \quad \text{if all quantified variables in } F \text{ have a DT-type}$$

$$\frac{\mathcal{P} \vdash F}{\forall\mathcal{P} \vdash F} \quad \text{if } F \text{ is a definedness-formula}$$

Structural Induction Let K be a type constructor with data constructors $c_1, \ldots, c_n$, let $T = K(T')$ be a monomorphic type and let F be a formula with occurrences of the free variable $x :: T$. For a constructor c_i let $Ind_T(c_i)$ be the set of indices of the arguments of c_i that are of type T. The slightly generalized induction scheme is as follows, where all constructors c_i must be covered:

$$\frac{\begin{array}{l}\mathcal{P} \vdash F[c_i/x] \quad \text{for all 0-ary constructors } c_i \\ \mathcal{P} \vdash \forall y_1, \ldots, y_m.F[y_1/x] \wedge \ldots \wedge F[y_m/x] \\ \qquad \Longrightarrow \forall y_{m+1}, \ldots, y_k.F[(c_i\ y_{\rho(1)} \ldots y_{\rho(k)})/x] \\ \quad \text{for all constructors } c_i, \text{ where } ar(c_i) = k, \\ \quad |Ind_T(c_i)| = m, \ \rho \text{ is a permutation on } \{1, \ldots, k\} \text{ such that} \\ \quad \rho(i) \in \{1, \ldots, m\} \text{ for all } i \in Ind_T(c)\end{array}}{\mathcal{P} \vdash \forall x :: T.F}$$

8 Polymorphic Formulas

In this section we consider polymorphic formulas, adapt validity and tautology to polymorphic formulas, and show that validity of a polymorphic formula for a fixed program does in general not imply that the formula is a tautology. Then the section illustrates how to prove polymorphic theorems directly.

Definition 8.1. Polymorphic $\mathcal{P}$-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 8.2. *For a program $\mathcal{P}$ and an extension $\mathcal{P}'$ of $\mathcal{P}$ a polymorphic $\mathcal{P}$-formula F is* valid for $\mathcal{P}'$*, if for every $\mathcal{P}'$-type substitution ρ that instantiates every type variable in F with a monomorphic $\mathcal{P}'$-type, the formula $\rho(F)$ is a $\mathcal{P}'$-tautology.*
A polymorphic $\mathcal{P}$-formula F is a (polymorphic) $\mathcal{P}$-theorem*, iff it is valid for all extensions $\mathcal{P}'$ of $\mathcal{P}$.*

Lemma 8.3. *A $\mathcal{P}$-formula F is a polymorphic $\mathcal{P}$-theorem if, and only if for all $\mathcal{P}' \sqsupseteq \mathcal{P}$ and for every $\mathcal{P}'$-type substitution ρ that instantiates every type variable in F with a monomorphic $\mathcal{P}'$-type, the monomorphic formula $\rho(F)$ is $\mathcal{P}'$-valid.*

Proof. The "only if"-direct is trivial. For the other direction let F be a $\mathcal{P}$-formula. In the rest of this proof, $\rho_{\mathcal{P}'}$ means type substitutions that substitute all type variables of a formula by (monomorphic) $\mathcal{P}'$ types. Note that any $\rho_{\mathcal{P}'}$ is also a $\rho_{\mathcal{P}''}$ for any program $\mathcal{P}'' \sqsupseteq \mathcal{P}'$. We assume that for all $\mathcal{P}' \sqsupseteq \mathcal{P}$ and any $\rho_{\mathcal{P}'}$ it holds: $\rho_{\mathcal{P}'}(F)$ is $\mathcal{P}'$-valid. Now let us assume $\mathcal{P}'$ and $\rho_{\mathcal{P}'}$ to be fixed. We have to show $\rho_{\mathcal{P}'}(F)$ is $\mathcal{P}''$-valid for all programs $\mathcal{P}'' \sqsubseteq \mathcal{P}'$. Since every $\rho'_{\mathcal{P}}$ is also a substitution that substitutes all types variables of F with $\mathcal{P}''$-types, the claim holds. □

Proposition 8.4. *$\mathcal{P}$-validity of a polymorphic formula F does not imply its $\mathcal{P}$-tautology property.*

Proof. 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 valid for $\mathcal{P}$:

$$\begin{aligned} F := \forall x_1 :: a, x_2 :: a, x_3 :: a. \\ ((x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3) \\ \implies \exists x :: a. x \neq x_1 \wedge x \neq x_2 \wedge x \neq x_3), \end{aligned}$$

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 $\mathcal{P}$, since it is true for `Bool`, Peano-numbers and lists, and also for function types. 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 polymorphic $\mathcal{P}$-theorem. □

8.1 Examples of Induction Schemes for Polymorphic Tautologies

Let $\mathcal{P}$ be a fixed program. For universally quantified closed polymorphic $\mathcal{P}$-formulas $F = \forall x_1 :: T_1(\overline{\alpha}), \ldots x_n :: T_n(\overline{\alpha}).A$, where A is a quantifier-free formula, and $\overline{\alpha}$ a tuple of type variables α_i, the induction proof scheme as well as the permitted inference rules have to be "independent" of the type variables $\overline{\alpha}$. However, there is one obstacle: since the validity of the formula F is defined over all instantiations ρ of types for $\overline{\alpha}$, the sets $\mathcal{M}_{\mathcal{P},\rho(T_i)}$ depend on the instantiation ρ.
We provide two examples for induction schemes for polymorphic formulas, where the first one is almost the same as for monomorphic formulas.

Definition 8.5 (Polymorphic Induction Scheme for Lists).
Let $\forall x :: (\texttt{List}\ a).F$ be a closed polymorphic $\mathcal{P}$-formula.
Assume that the following holds for every monomorphic type T.

1. *The formula $F[(\texttt{Nil} :: (\texttt{List}\ T))/x]$ is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology).*
2. *The following formula is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology):*
   $\forall y : T.(F[y/x] \implies \forall z :: (\texttt{List}\ T).F[\texttt{Cons}(y, z)/x])$.

Then the formula $\forall x :: (\texttt{List}\ T).F$ is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology).

An example for a polymorphic theorem that can be proved by induction is associativity of the append-function (called `app` in the following) on lists of any type, where the definition in $\mathcal{P}$ is:
$\texttt{app} = \lambda x, y.\texttt{case}_{\texttt{List}}\ x\ (\texttt{Nil -> }\ y)\ ((x_1\ :\ x_2)\ \texttt{->}\ x_1\ :\ (\texttt{app}\ x_2\ y)$
The formula stating associativity is:

$$\begin{aligned} F := \forall xs :: \texttt{List}\ a, ys :: \texttt{List}\ a, zs :: \texttt{List}\ a. \\ (\texttt{app}\ xs\ (\texttt{app}\ ys\ zs))) = (\texttt{app}\ (\texttt{app}\ xs\ ys)\ zs) \end{aligned}$$

We sketch the tautology proof: Let $\mathcal{P}' \sqsupseteq \mathcal{P}$. Let ρ be a $\mathcal{P}'$-type substitution. Then for all $u \in \mathcal{M}_{\mathcal{P}',\rho(a)}$ and all $r, s, t \in \mathcal{M}_{\mathcal{P}',\texttt{List}\ \rho(a)}$ we have to verify

1. (Base case) $(\texttt{app Nil } (\texttt{app } s\ t)) \sim_{\mathcal{P}',\texttt{List } \rho(a)} (\texttt{app } (\texttt{app Nil } s)\ t)$.
   This holds by global correctness of the reduction rules and the definition of app. The left and right hand side are $\sim_{\mathcal{P}',\texttt{List } \rho(a)} (\texttt{app } s\ t)$.
2. (Induction step) $(\texttt{app } r\ (\texttt{app } s\ t)) \sim_{\mathcal{P}',\texttt{List } \rho(a)} (\texttt{app } (\texttt{app } r\ s)\ t)$
   $\Longrightarrow (\texttt{app } (u\texttt{:}r\ (\texttt{app } s\ t))) \sim_{\mathcal{P}',\texttt{List } \rho(a)} (\texttt{app } (\texttt{app } (u\texttt{:}r)\ s)\ t)$.
   This follows from global correctness of the reduction rules and the definition of app. The implicant can be reduced and transformed to $u : (\texttt{app } r\ (\texttt{app } s\ t)) \sim_{\mathcal{P}',\texttt{List } \rho(a)} (u : (\texttt{app } (\texttt{app } r\ s)\ t))$.

Main Theorem 5.5 now shows that $\rho(F)$ is a monomorphic theorem for any $\mathcal{P}'$-type substitution ρ. Hence, F is a polymorphic theorem.
Note that the reasoning was independent of the type variable a and could be automated.

Another example are binary trees where nodes and leaves are labelled with values of different types. Let the type constructor be $\texttt{Tree}(.,.)$ with three constructors

1. $\texttt{Empty}:: \texttt{Tree } a_1\ a_2$.
2. $\texttt{Leaf}:: a_1 \to \texttt{Tree } a_1\ a_2$.
3. $\texttt{Node}:: a_2 \to \texttt{Tree } a_1\ a_2 \to \texttt{Tree } a_1\ a_2 \to \texttt{Tree } a_1\ a_2$.

Then one induction scheme is as follows:

Definition 8.6 (Polymorphic Induction Scheme for Trees).
Let $\forall x :: \texttt{Tree}(a_1, a_2).F$ be a closed polymorphic $\mathcal{P}$-formula.
Assume that the following holds for all monomorphic types T_1, T_2.

1. *The formula $F[(\texttt{Empty} :: (\texttt{Tree } T_1\ T_2))/x]$ is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology).*
2. *The following formula is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology):*
   $\forall z :: T_1.F[((\texttt{Leaf } z) :: (\texttt{Tree } T_1\ T_2))/x])$.
3. *The following formula is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology):*
   $\forall y_1, y_2 : (\texttt{Tree } T_1\ T_2).(F[y_1/x] \wedge F[y_2/x] \implies \forall z :: T_2.F[(\texttt{Node } z\ y_1\ y_2)/x])$

Then the formula $\forall x :: (\texttt{Tree } T_1\ T_2).F$ is $\mathcal{P}$-valid (a $\mathcal{P}$-tautology).

In general, the following type constructors allow induction schemes as for lists and the trees above.

Definition 8.7. *A type constructor $K \in \mathcal{K}$ is* inductive*, if the polymorphic types of all data constructors of D_K are of the form $T_1 \to \ldots \to T_h \to K(\alpha_1, \ldots, \alpha_k)$ where T_i for $i \leq h$ is either some α_i, or a closed DT-type, or $K(\alpha_1, \ldots, \alpha_k)$.*

In summary this illustration of induction schemes shows that a universally quantified polymorphic formula is a polymorphic theorem, if the induction and the induction measure are "independent" of the type variables and only $\mathcal{P}$-tautologies and globally correct $\mathcal{P}$-transformations are used to prove the induction base and the induction step.
However, we have to leave open whether the following holds:

Open Problem 8.8 *Let $\mathcal{P}$ be a program and F be a polymorphic formula of the form $\forall x_1 \ldots x_n.s = t$ that is valid for $\mathcal{P}$.*
Issue: is F a $\mathcal{P}$-tautology?

9 Note on the Differences to Verifun

We discuss the differences between our semantics and the semantics underlying `VeriFun`, and the consequences for the proof rules of an automated system. There are two main differences:

- We allow nonterminating functions whereas `VeriFun` only allows total functions.
- undefined expressions are semantically equal to nonterminating expressions and different from all values.

First note that `VeriFun`-total means that a function either terminates with a value or with the constant `undefined` that indicates an unknown value, whereas our notion of termination means termination to a value.
The consequences of the differences are:

- *Nonterminating functions:* The induction schemes are the same since they are in general over the structure of DT-values. As long as expressions are known to terminate to a value, there is no difference. If terms appear in formulas for which termination is not proved yet, then the only precaution is that the call-by-name beta- and case-reductions cannot be used and instead (VNbeta) and (VNcase) have to be used. This is only necessary if the termination-status of the argument of a beta-redex or the to-be-cased term of a case-expression is unclear.
- *Undefined expressions are semantically equal to nonterminating expressions.* This permits a set of deduction rules that allow to simplify expressions that contain undefined expressions or the constant `undefined`. It also allows to evaluate equations. For example (`tail Nil`) = `undefined` results in the logical truth value `true`, and (`tail Nil`) = `Nil` in false. Note that these equations often occur in `VeriFun`-subproofs that try to show preconditions of lemmas.
- *Undefined expressions are semantically different from all values.* The formula (*tail* `Nil`) = (*tail* (*tail* `Nil`)) $\Longrightarrow$ (*tail* `Nil`) = `Nil` can be analyzed, which is a Verifun-theorem due to the adopted underspecification, but not a theorem in our logic. The `VeriFun`-assumption that (*tail* `Nil`) is a certain unknown value is responsible for proving this nonsense theorem. The `VeriFun`-prover is permitted to make a case distinction whether (*tail* `Nil`) is equal to `Nil` or to a nonempty list and then arrives at the conclusion after some steps. Clearly, in our logic this evaluates to `True` $\Longrightarrow$ `False`, and hence is not a theorem.
  The correct proof rule for list expressions w.r.t our logic is to make a case analysis where the three possibilities are: nontermination, nil, non-empty list.

10 Conclusion

We presented a logical framework for inductive proofs of functional programs without a total-termination restriction for defined functions and with an exact

function equality semantics. This also provides a reconstruction of the logic and semantics used by the prover `VeriFun`, and proposals for generalizing the formulas and adapting the semantics and the proof rules.
For several interesting classes of formulas it was shown that local validity implies that they are theorems. Two open questions are: Is every locally valid universally quantified monomorphic formula also a tautology? and: Is every locally valid universally quantified polymorphic equation also a theorem?

Acknowledgments

The authors thank Jan Schwinghammer for pointing out that the VN-reduction is not strongly normalizing and providing Example C.2. We also thank the anonymous referees for observations and remarks that improved the paper.

Bibliography

[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.

[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.

[CAT05] Yuki Chiba, Takahito Aoto, and Yoshihito Toyama. Program transformation by templates based on term rewriting. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 59–69. ACM, 2005.

[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.

[Häh05] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4):415–433, 2005.

[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.

[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.

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

[SM09] Lutz Schröder and Till Mossakowski. HasCasl: Integrated higher-order specification and program development. *Theor. Comput. Sci.*, 410(12-13):1217–1260, 2009.

[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.

[SSS10a] 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.

[SSS10b] Manfred Schmidt-Schauß and David Sabel. A termination proof of reduction in a simply typed calculus with constructors. Frank report 42, Inst. für Informatik. Goethe-Universität Frankfurt, 2010.

[SSSH09] David Sabel, Manfred Schmidt-Schauß, and Frederik Harwath. Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In *INFORMATIK 2009*, volume 154 of *GI Edition - Lecture Notes in Informatics*, pages 369; 2931–45, October 2009.

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

[Ver11] VeriFun Website. www.inferenzsysteme.informatik.tu-darmstadt.de/verifun/, 2011.

[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.

[WS05] 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 Γ is a (partial) mapping from variables and function symbols $f \in \mathcal{F}$ to types, where we assume that every function f is mapped to a type. The notation $\mathrm{Dom}(\Gamma)$ is the set of variables (and function names) that are mapped by Γ. The notation $\Gamma, x :: \tau$ means a new environment where $x \notin \mathrm{Dom}(\Gamma)$. Given a quantified type $\forall \mathcal{X}.T$, a *(type-)substitution* ρ for $\forall \mathcal{X}.T$ substitutes types for type variables $\mathcal{X}$, 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 $\texttt{Int} \to \texttt{Int}$ is an instance of T, as well as $a \to \texttt{Int}$, 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) $$\frac{\Gamma \vdash s_1 :: S_1\ ; \ldots ; \Gamma \vdash s_n :: S_n \quad \Gamma, y :: \mathit{typeOf}(c) \vdash (y\ s_1 \ldots s_n) :: T}{\Gamma \vdash (c\ s_1 \ldots s_n) :: T} \text{ if } ar(c) = n$$

(Case) $$\frac{\begin{array}{l}\Gamma \vdash s :: K\ S_1 \ldots S_m \\ \text{for } i = 1, \ldots, k: \\ \quad \Gamma, x_{i,1} :: T_{i,1}, \ldots x_{i,n_i} :: T_{i,n_i} \vdash t_i :: T \\ \text{for } i = 1, \ldots, k: \\ \quad \Gamma, x_{i,1} :: T_{i,1}, \ldots x_{i,n_i} :: T_{i,n_i} \vdash (c_i\ x_{i,1} \ldots x_{i,n_i}) :: K\ S_1 \ldots S_m\end{array}}{\Gamma \vdash (\texttt{case}_K\ s\ ((c_1\ x_{1,1} \ldots x_{1,n_1})\ \texttt{->}\ t_1) \ldots) :: T}$$

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

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

Fig. 5. The type-derivation rules

Definition A.2. *Given a program, the types Γ of the functions in f are called* admissible*, and all the functions are called* derivationally well-typed*, iff for every $f \in \mathcal{F}$ and the type $f :: T \in \Gamma$, we have $\Gamma \vdash d_f :: T$.*

Using the rules of the derivation system, a standard polymorphic type system can be implemented that computes types as greatest fixpoints using iterative processing. By standard reasoning, there is a most general type of every expression.

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 Sections 2.3 and A.2) are equivalent mechanisms.
Note that typability using the iterative procedure is undecidable, since the semi-unification problem (see [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.*

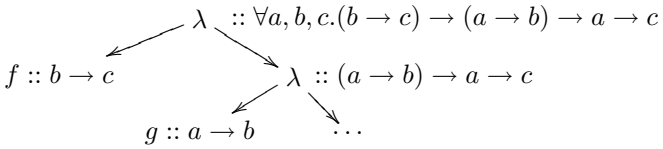
A.2 Type Consistency Rules

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:

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

An illustration is as follows:



We define the following *type-constraints*:

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

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

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

Case-expression

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

Fig. 6. Computation of *MonoTp*

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, $R[\texttt{case}_K\ (\lambda x.r)\ \mathit{Alts}]$, or $R[\texttt{case}_K\ (c\ t_1 \ldots\ t_n)\ \mathit{Alts}]$ 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. □

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

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 the following we assume that a fixed program $\mathcal{P}$ is given. We are interested in the contextual semantics of $\mathcal{P}$-expressions. In this section we denote the set of $\mathcal{P}$-expression as the language L. In order to prove a CIU-Lemma for L, we first have to prove a context lemma for L extended with a `let`.

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:

$$
\begin{array}{lll}
s, s_i, t \in \mathit{Expr} & ::= & x \mid f \mid (s\ t) \mid \lambda x.s \mid (c_i\ s_1 \ldots s_{ar(c_i)}) \\
& & \mid\ (\mathtt{case}_K\ s\ \mathit{Alt}_1 \ldots \mathit{Alt}_{|D_K|}) \mid (\mathtt{let}\ x = v\ \mathtt{in}\ s) \\
\mathit{Alt}_i & ::= & ((c_i\ x_1 \ldots x_{ar(c_i)})\ \mathtt{->}\ s_i)
\end{array}
$$

where v is a value, i.e. $v, v_i \in \mathit{Val} ::= x \mid (c\ v_1\ \ldots v_N) \mid \lambda x.s$. 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^{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 sub and lr are used to prevent searching inside `let`-expressions which are below application and constructor applications. It is easy to verify that the label shifting is deterministic and always terminates. 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 ($\mathrm{case}_{\mathtt{let}}$) 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$_{\texttt{let}}$) $C[((\lambda x.s)^{\texttt{sub}}\ v)] \to C[\texttt{let}\ x = v\ \texttt{in}\ s]$

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

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

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

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

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

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

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

$v_n\ \texttt{in}\ v)\ldots))$ 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 $(\texttt{let}\ x = [\cdot]\ \texttt{in}\ t)$.
In the following we sometimes denote sequences of reductions $s_1 \to \ldots \to s_n$ with the meta-symbol RED. For a reduction sequence RED the function 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 ρ where ρ 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. □

Remark B.3. Note that i and j in the previous lemma may be different. For instance, consider the two-hole context $C := ([\cdot_1]\ [\cdot_2])$. Then $C[\lambda.x.x, \cdot_2]$ is a reduction context, but $C[t, \cdot_2]$ is only a reduction context, if t is value. Nevertheless the context $C[\cdot_1, t]$ is a reduction context for any expression t.

Lemma B.4 (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 $C[s_1, \ldots, s_n] \xrightarrow{ls,a} C'[s'_1, \ldots, s'_m]$, then $C[t_1, \ldots, t_n] \xrightarrow{ls,a} C'[t'_1, \ldots, t'_m]$, such that $s'_i \leq_{\mathtt{let},\mathcal{P},R,T'_i} t'_i$. We go through the cases of which reduction step is applied to $C[s_1, \ldots, s_n]$ to figure out how the expressions s_i (and t_i) are modified by the reduction step, where we only mention the interesting cases.

- For a (lapp), (lrapp), (lcapp), (lcase), and (beta$_{\mathtt{let}}$) 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 $C'[s'_1, \ldots, s'_m]$ has a terminating sequence of standard reductions of length $l - 1$, we also have $C'[t'_1, \ldots, t'_m]{\Downarrow}$.
With $C[t_1, \ldots, t_n] \xrightarrow{ls,a} C'[t'_1, \ldots, t'_m]$ we have $C[t_1, \ldots, t_n]{\Downarrow}$. □

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 Φ 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.5. *Let $t \in L_{\texttt{let}}$ and $t \xrightarrow{ls,lll} t'$ or $t \xrightarrow{ls,cp} t'$. Then $\overline{\Phi}(t') = \overline{\Phi}(t)$. Furthermore, all reduction sequences consisting only of $\xrightarrow{ls,lll}$ and $\xrightarrow{ls,cp}$ are finite.*

Lemma B.6. *Let t be an expression of $L_{\texttt{let}}$ such that $\overline{\Phi}(t) = R[s]$, where (ls,cp)- and (ls,lll)-reductions are not applicable to t, and R is a reduction context. Let t be represented as $t = \texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t_1$ where t_1 is not a* let*-expression. Then there is some reduction context R' and 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.7. *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.6 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 ($\text{beta}_{\texttt{let}}$)-reduction to t:

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

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

Lemma B.8. *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 Φ, $\overline{\Phi}$. 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 ($\text{beta}_{\texttt{let}}$), ($\text{delta}_{\texttt{let}}$), or ($\text{case}_{\texttt{let}}$), then $\overline{\Phi}(t_1) \to \overline{\Phi}(t_2)$ by the reduction with the same name. Part 4 follows from Lemma B.5 and B.7. □

Lemma B.9. *Φ and $\overline{\Phi}$ are convergence equivalent.*

Proof. We have to show four parts:

- $t\!\downarrow \implies \overline{\Phi}(t)\!\downarrow$: This follows by induction on the length of the evaluation of t. The base case is shown in Lemma B.8, part 1. The induction step follows by Lemma B.8, 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.8, 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.8, 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.8, part 2. For the induction step let $t \xrightarrow{a} t'$, where $t'\!\downarrow$ and $a \in \{(\text{beta}), (\text{delta}), (\text{case})\}$. If $a = (\text{delta})$ then $\Phi(t) \xrightarrow{ls,delta_{\texttt{let}}} \Phi(t')$, and hence the induction hypothesis shows $\Phi(t')\!\downarrow$ and thus $\Phi(t)\!\downarrow$. For the other two cases we have $\Phi(t) \xrightarrow{ls,a} t''$, with $\overline{\Phi}(t'') = t'$. The second part of this proof shows that $t'\!\downarrow$ implies $t''\!\downarrow$. Hence, $\Phi(t)\!\downarrow$.
- $\Phi(t)\!\downarrow \implies t\!\downarrow$: This follows, since the first part of this proof shows $\Phi(t)\!\downarrow$ implies $\overline{\Phi}(\Phi(t))\!\downarrow$, and since $\overline{\Phi}(\Phi(t)) = t$. □

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

Corollary B.10 (Adequacy of Φ). $\Phi(s) \leq_{\texttt{let},\mathcal{P},T} \Phi(t) \implies s \leq_{\mathcal{P},T} t$.

Lemma B.11 (CIU-Lemma). *Let* $s, t :: T$ *be two expressions of* L *such that for all* $\mathcal{P}$*-value substitutions* σ *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 σ 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.4 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 σ is a value substitution, convergence equivalence of $\overline{\Phi}$ shows $R'[\sigma(s)]\downarrow$. Since $R'[\sigma(s)]$ and $R'[\sigma(t)]$ are closed, the precondition of the lemma now implies $R'[\sigma(t)]\downarrow$. Since $R'[\sigma(t)] = R'[\sigma(\overline{\Phi}(\Phi(t)))] = \overline{\Phi}(R_{let}[\Phi(t)])$ and since $\overline{\Phi}$ is convergence equivalent, we have $R[\Phi(t)]\downarrow$.

Proposition B.12. *The transformation (beta), (delta), and (case) are correct program transformations in L.*

Proof. We use the CIU-Lemma B.11: Let $a \in \{(\text{beta}), (\text{delta}), (\text{case})\}$. Let $s \xrightarrow{a} t$, R be a reduction context, and σ be a value substitution, such that $R[\sigma(s)]$ is closed. If $R[\sigma(t)]\downarrow$, then $R[\sigma(s)] \xrightarrow{a} R[\sigma(t)]$ by a standard reduction, and thus $R[\sigma(s)]\downarrow$.
For the other direction let $R[\sigma(s)]\downarrow$, i.e. $R[\sigma(s)] \to t_1 \xrightarrow{*} t_n$ 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.13 (CIU-Theorem). *For $\mathcal{P}$-expressions $s, t :: T$: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ for all $\mathcal{P}$-value substitutions σ and $\mathcal{P}$-reduction contexts R where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P},T} t$ holds.*

Proof. One direction is the CIU-Lemma B.11. 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.14. *Let $\mathcal{P}$ be a program. For $\mathcal{P}$-expressions $s, t :: T$: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ for all extensions $\mathcal{P}'$ of $\mathcal{P}$ and all $\mathcal{P}'$-value substitutions σ and $\mathcal{P}'$-reduction contexts R where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P}\forall,T} t$ holds.*

B.3 Local CIU-Theorems

In this subsection we show that the CIU-theorem can be made stronger by restricting R and σ to be free of function symbols from $\mathcal{F}$.
Let an *F-free* expression, value, or context be an expression, value, or context that is built over the language without function-symbols, but where $\bot$-symbols of every type are allowed according to Assumption 3.7.
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.15 (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 σ 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 σ and all F-free reductions contexts R where $R[\sigma(s)], R[\sigma(t)]$ are closed: $R[\sigma(s)]{\downarrow} \implies R[\sigma(t)]{\downarrow}$. Let R be any reduction context and σ be any value substitution such that $R[\sigma(s)], R[\sigma(t)]$ are closed, and assume $R[\sigma(s)]{\downarrow}$. Let n be the number of reductions of $R[\sigma(s)]$ to a value. We construct F-free reduction contexts R' and F-free value substitutions σ' as follows: apply $n+1$ times a delta-step for every occurrence of function symbols in R and σ. As a last step, replace every remaining function symbol by $\bot$ of the appropriate type. Note that a single reduction step can shift the bot-symbols at most one lambda-level higher. By standard reasoning and induction, we obtain that $R'[\sigma'(s)]{\downarrow}$, by using the reduction sequence of $R[\sigma(s)]$ also for $R'[\sigma'(s)]$, where the induction is by the number of reduction steps. The assumption now implies that $R'[\sigma'(t)]{\downarrow}$, We have $R'[\sigma'(t)] \leq_{\mathcal{P},T} R[\sigma(t)]$, 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.13.

We are now able to prove Theorem 4.1.

Proof of 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 σ and F-free $\mathcal{P}$-reduction contexts R, where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P},\tau} t$ holds.*

One direction is the F-free CIU-Lemma B.15. The other direction is the same as in the proof of the local CIU-theorem B.13. □

Corollary B.16. *Let $s, t :: \tau \in L$. If for all closing F-free $\mathcal{P}$-value substitutions σ, 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 σ, 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.17. *Let $s, t :: \tau \in L$. If for all closing F-free $\mathcal{P}$-value substitutions σ, $\sigma(s)$ 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 Ω-Expressions

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

Proposition B.18. *Let $s :: \tau$ be an Ω-expression. Then for every reduction context $R[\cdot :: \tau]$, the expression $R[s]$ is an Ω-expression.*

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

Corollary B.19. *Let $s, t :: \tau$ and let s be an Ω-expression. Then $s \leq_{\mathcal{P},\tau} t$. If also t is an Ω-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.13: Let R be a reduction context, σ be a value substitution such that $\sigma(s), \sigma(t)$ are closed. Then $\sigma(s)$ must be an Ω-expression, and by Proposition B.18 $R[\sigma(s)]$ is an Ω-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.
To establish this result we will use the VN-reduction to "normalize" the values, i.e. we will eliminate all subexpressions which are not of a $\mathcal{P}$-type. This requires to prove several properties of the VN-reduction and we require more notation.
Therefore, we will add a sequentializing construct that always comes with two arguments. The seq-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. $((\lambda x. \ldots)\ (\texttt{case}\ y\ \ldots))$ may be irreducible, but not a value. However, for the lemma below it is necessary to be able to apply a general kind of beta-reduction to this expression. We also permit the symbol `Bot`, labeled with a type, for Ω-expressions. The extended set of VN-reductions is given 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 expression 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 the variable is bound by a lambda-binder, then we choose the corresponding abstraction. If the variable is bound by a pattern in a case-expression, then the case-expression is of the form $\texttt{case}_T\ s_1'\ (c\ x_1 \ldots x_n) \to r \ldots$, 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 in one step. 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 VN-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 the variable binders above s_1 only bind $\mathcal{P}$-type. The expression s_1 can also not be an abstraction, `Bot`, a seq-expression, or a case-expression, since v' is VN-irreducible. It cannot be a constructor application, due to typing. Hence this case is impossible.
3. s_1 cannot be an argument in an application. The reason is that s_1 occurs in a term $(\ldots(r_1 \ldots r_2)\ldots s_1)$ where r_1 is of a non-$\mathcal{P}$-type, which was already shown as impossible.

Now we choose an s_1 such that it has maximal size. Note that s_1 is VN-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 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 VN-irreducible, s_1 cannot be a seq-expression, a constructor-expression, `Bot`, an abstraction, nor a variable. An application is not possible as shown above. It can also not be a `case`-expression, since v' is VN-irreducible.
2. s_1 cannot be the first argument of a `case`: We scan all syntactic cases of s_1. Since v' is VN-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.
Note that VN-reduction is not strongly normalizing even for F-free expressions as the following example shows:

Example C.2. Assume there is a type U with $D_U = \{\texttt{Unit}\}$ and a type F with $D_F = \{\texttt{Fold}\}$ such that: $\mathit{typeOf}(\texttt{Unit}) = U$ and $\mathit{typeOf}(\texttt{Fold}) = (F \to U) \to F$. Then it is possible to define a typeable fixpoint combinator $y :: (U \to U) \to U$ as follows

$$y := \lambda f.(\lambda x.f\ (\mathit{unfold}\ x\ x))\ (\texttt{Fold}\ (\lambda x.f\ (\mathit{unfold}\ x\ x)))$$

where *unfold* is the expression $\lambda w.\texttt{case}_F\ w\ ((\texttt{Fold}\ y) \to y)$.
In particular there is an infinite VN-reduction sequence for $(y\ (\lambda w.w))$:

$$(y\ \lambda w.w) \xrightarrow{VN,*} (\lambda x.(\mathit{unfold}\ x\ x))\ (\texttt{Fold}\ (\lambda x.(\mathit{unfold}\ x\ x)))$$

and

$$(\lambda x.(\mathit{unfold}\ x\ x))\ (\texttt{Fold}\ \lambda x.(\mathit{unfold}\ x\ x)) \xrightarrow{VN,*} (\lambda x.(\mathit{unfold}\ x\ x))\ (\texttt{Fold}\ \lambda x.(\mathit{unfold}\ x\ x)),$$

hence the second sequence can be performed infinitely often.

Since VN-reduction is not 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.

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.

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

As a measure for the expressions s we use the multiset $mcss(.)$ of all numbers $css(s')$ for all subexpressions of s. It is well-known that this is a well-founded measure.

Lemma C.3. *Every VN-reduction sequence without the (VNcase)- and (VNbeta)-reduction steps is finite.*

Proof. We check that for every possible reduction rule, the measure $mcss(.)$ is strictly decreased: Since proper subexpression have a strictly smaller measure, it is sufficient to check the reduction without a context.

- The reduction rules that reduce to `Bot` strictly reduce the measure.
- seqc: reduces the size by 2.
- seq: strictly reduces 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.4. *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 σ 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 rule (seq) follows from the correctness of (beta)-reduction. Let $s \to t$ by a rule in $\{\texttt{seqc}, \texttt{seqseq}, \texttt{seqapp}, \texttt{caseseq}, \texttt{caseapp}, \texttt{casecase}, \texttt{seqcase}\}$ 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 σ be a closing value substitution for s, t. We distinguish two cases:

$$
\begin{array}{lcl}
(s\ t)^{VNS} & \to & (s^{VNS}\ t) \\
(s;t)^{VNS} & \to & (s^{VNS};t) \\
(\texttt{case}\ s\ \ldots)^{VNS} & \to & (\texttt{case}\ s^{VNS}\ \ldots) \\
(c\ \ldots v_i\ s_{i+1}\ldots)^{VNS} & \to & (c\ \ldots v_i\ s_{i+1}^{VNS}\ \ldots)
\end{array}
$$

where v_i are values.

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

- $\sigma(t')$ is an Ω-expression. Then obviously $\sigma(s), \sigma(t)$ are Ω-expressions, i.e. $\sigma(s) \sim_{\mathcal{P},T} \sigma(t)$ by Corollary 4.3.
- $\sigma(t')$ is not an Ω-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 σ: $\sigma(s) \sim_{\mathcal{P},T} \sigma(t)$. Hence Corollary B.16 implies $s \sim_{\mathcal{P},T} t$. Correctness of VNcase can be shown in a similar way. □

C.2 VN standard reduction and termination properties

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

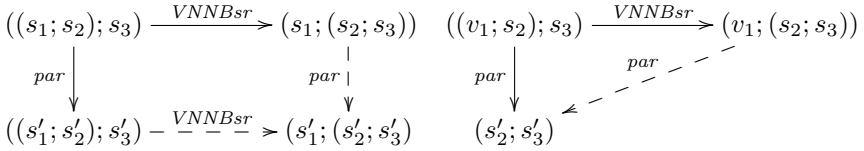
Definition C.5. *Let t be a (perhaps open) F-free expression. A* VN-standard-reduction *of t is defined as follows: Apply the VNS-label-shift in Fig. 9 to t, starting with t^{VNS} and where no other subexpression is labelled VNS. The outermost-leftmost VN-reduction according to Figs. 2, 3 and 4 is applied 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 we denoted it 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 $(v;s) \to s$, where v is a value (which corresponds to 1 beta-reduction) is not counted as a beta-reduction in the length of *VN*-standard-reductions, but as a seq or seqc-reduction.
In the following we denote the usual call-by-value standard reduction rules as $\xrightarrow{sr}$ (see also Definition 3.1). Note that we apply them here only to F-free expressions,

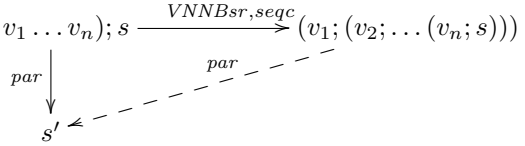
hence δ is not necessary. However, `Bot`-symbols are permitted. The reduction for the seq-expressions is as for the beta-reduction.
We also introduce a parallel version of the $\xrightarrow{sr}$-reduction which is defined like the 1-reduction in [Bar84], and is denoted as $\xrightarrow{par}$.
Now we analyze 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. *The following forking diagrams hold for forkings of* $\xrightarrow{par}$ *and* $\xrightarrow{VNNBsr}$*-reductions, provided the reductions are different.*

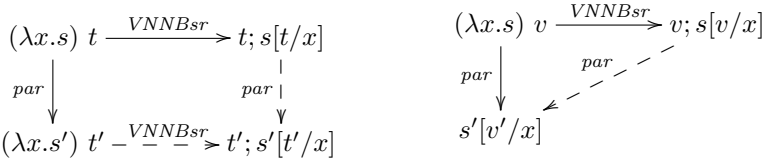
Proof. The proof is by inspecting all the possibilities. We explain the typical and the exceptional case. The diagrams for (seqapp) are a typical case:



The following forking diagram is the triangle diagram for (seqc), where the seq-redexes have to be added to the parallel reduction.



The following shows typical cases for VNbeta.



The cases for VNcase are similar to VNbeta. □

Lemma C.7. *Let s be an (open) expression. If s is sr-irreducible, then the VNNBsr-reduction of s is finite.*

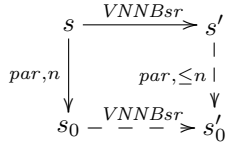
Proof. If s is a value, then there is no VNNBsr-reduction. Otherwise, $s = R[x]$ for a reduction context R. Now induction on the following depth of the hole of $R[x]$: It is the sum of the following numbers: if the path to the hole crosses

an application, case, or constructor, this counts as 2, whereas seq-expressions count as 1, where the subexpressions $(s_1; (s_2; (\ldots (s_{n-1}; s_n)\ldots)))$ are treated as flattened.
We prove also that the result of the VNNBsr-reduction creates a final expression $R''[x]$, where the hole-depth is not greater than the hole-depth of R.
If $R = [\cdot]$, then s is a value, and we are done. R may be $\texttt{case}\ R'\ alts$, $(R'\ s')$, $((\lambda y.s')\ R')$, $(c\ v_1 \ldots v_i\ R'\ alts)$, or $R'; s'$.

1. First assume that the VNNBsr-redex is at the top. Then there are the following possibilities:
   (a) $s = \texttt{case}\ (\texttt{case}\ R''[x]\ alts) alts')$. Then the casecase-reduction results in $(\texttt{case}\ R''[x]\ alts'')$, and the depth of the hole is decreased by 2.
   (b) The reduction is caseseq. Then the depth is also decreased by 2.
   (c) $s = \texttt{case}\ (c\ s_1 \ldots s_n)\ alts)$ and the reduction is VNcase. Then the hole of the reduction context is in some s_i and after the reduction the hole is at a smaller depth. Note that we assumed right-flattened seq-expressions.
   (d) seqapp: Again the hole of the reduction context after the reduction has strictly smaller depth.
   (e) VNbeta: The depth is decreased by 1.
   (f) caseapp: $s = (\texttt{case}\ t_0\ alts)\ r)$. The hole can only be in t_0, hence the depth is strictly decreased after one reduction.
   (g) In the case $s = (c\ s_1 \ldots s_n)$, we can use induction on the depth of R.
   (h) seqseq: The depth is decreased by 1.
   (i) seq: The depth is decreased by 1.
   (j) seqc: $((c\ s_1 \ldots s_n); s)$: the hole is in some s_i, and after the reduction, the depth is decreased by 2.
   (k) seqcase: The depth is decreased by 1.
2. If the VNNBsr-redex is not at the top, then we use induction, since the depth of the reduction hole within the subexpression is strictly smaller, hence we can use the claim that the depth is not increased, and then we can use part 1. □

In the following the notation $s \xrightarrow{\leq n} s'$ means that there are $k \leq n$-reductions from s to s', where we also may attach more information to the reduction arrow.

Lemma C.8. *Let s be an (open) expression. If $s \xrightarrow{VNNBsr} s'$ and $s \xrightarrow{par,n} s_0$ where s_0 is sr-irreducible, then there is some sr-irreducible s'_0, such that $s' \xrightarrow{par,\leq n} s'_0$:*



Proof. This follows by induction on n: For the base case one has to observe that if s is sr-irreducible, and $s \xrightarrow{VNNBsr} s'$, then s' is sr-irreducible, too. This follows by inspecting the VN-reductions. The induction step $n > 0$ follows by applying one of the diagrams of Lemma C.6 and then using the induction hypothesis. □

For the proof of Lemma C.11 we need some properties of the *par*-reduction, which are provable using standard methods. Nevertheless, we give a sketch since we are considering extensions like case and seq-expressions.

Lemma C.9. *Let $s \xrightarrow{par} s'$. Then the reduction can be split into $s \xrightarrow{sr} s'' \xrightarrow{par} s'$, where s'' does not contain an sr-redex.*

Proof. The argument is that the sr-redex is not contained in another redex of the parallel reduction, and that an upper bound for the development (i.e. a replacement of the parallel reduction by a single-step reduction) can be explicitely given. It is a function ϕ on expressions as follows, where only the nontrivial cases are given:

- If the subexpression $((\lambda x.s)\ t)$ is reduced in the parallel reduction, then $\phi((\lambda x.s)\ t) = 1 + \phi(s) + \phi(t)\dot{\#}(x, s)$, where $\#(x, s)$ is the number of occurrences of x in s.
- If the subexpression $(\mathtt{case}_T\ (c\ s_1 \ldots s_n) \ldots (c\ x_1 \ldots x_k) \to r)$ is reduced in the parallel reduction, then the ϕ-result is $1 + \phi(r) + \Sigma(\phi(s_i)\dot{\#}(x_i, r))$,

Since the split $s \xrightarrow{par} s'$ into $s \xrightarrow{sr} s'' \xrightarrow{par} s'$ implies that the ϕ-value of s'' is strictly smaller then the ϕ-value of s, this splitting terminates. □

Lemma C.10. *Let $s \xrightarrow{par,*} s'$ where s' is sr-irreducible. Then there is a terminating reduction $s \xrightarrow{sr,*} s''$, where s'' is sr-irreducible.*

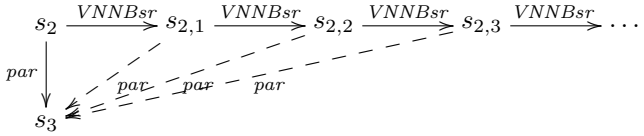
Proof. This follows from Lemma C.9 and since the internal parallel-reduction can be commuted with sr-reductions, and by induction.

Lemma C.11. *Let s be an (open) expression. Then termination of s (i.e., $s \xrightarrow{sr} s_0$ for some sr-irreducible s_0) implies that the VNNBsr-reduction of s is finite.*

Proof. We show a slightly generalized claim by induction on n: If $s \xrightarrow{par,n} s_0$ where s_0 is irreducible, then the VNNBsr-reduction of s is finite.
Assume that $s \xrightarrow{par,n} s_0$, where s_0 is irreducible.
The base case $n = 0$ is proved in Lemma C.7.
Now assume that $n > 0$, and that $s \xrightarrow{par} s_1$. If also $s \xrightarrow{VNNBsr} s_1$, then we can use induction, since $s_1 \xrightarrow{par,n-1} s_0$. The other cases permit one of the diagrams in Lemma C.6:

Lemma C.8 shows that the reduction length of s'_1 w.r.t. $\xrightarrow{par}$ is also $\leq n - 1$, hence the induction hypothesis applies and shows that s'_1 has a finite *VNNBsr*-reduction. Now we look for the transformation of the *VNNBsr*-reduction of s into the *VNNBsr*-reduction of s_1. If the *VNNBsr*-reduction of s is finite, then

the proof is finished. Assume that the *VNNBsr*-reduction of s is infinite. Then there must be some s_2 in the reduction, such that only triangle-diagrams are applicable.



The triangle diagrams are special: they only apply if the expression has also an sr-redex. There are the cases of a (beta), (case) and (seq)-reduction: $R[(\lambda x.s)\ v] \xrightarrow{VNNBsr} R[v; s[v/x]] \xrightarrow{VNNBsr} R[s[v/x]]$. Similar for the case-reduction and the seq-reduction.

Now we see that $R[(\lambda x.s)\ v] \xrightarrow{VNNBsr,*} R[s[v/x]]$, and also $R[(\lambda x.s)\ v] \xrightarrow{sr} R[s[v/x]]$. This also holds for the other cases of triangle-diagrams. We use Lemma C.10 to obtain a maximal number of sr-reductions of s_2 to an sr-irreducible expression. Since the number of sr-reductions strictly decreases in the $\xrightarrow{VNNBsr,*}$-reduction of s_3, and s_3 has a finite sr-reduction to an irreducible expression by Lemma C.8, the $\xrightarrow{VNNBsr,*}$-reduction using only triangle-diagrams must be finite. □

Lemma C.12. *Let s be an (open) expression. If for some closing value-substitution σ: $\sigma(s)\downarrow$, then the VNNBsr-reduction of s is finite.*

Proof. Note that if the *VNsr*-reduction sequence of s includes a `Bot`-reduction, then the final result will be `Bot`: This follows since for this case the $\sigma(s)$ cannot reduce to a value, since the *VNsr*-reductions are correct-. Hence `Bot`-reductions are not used in any reduction sequence $s \xrightarrow{VNsr,*} s'$.

The reduction $\sigma(s) \xrightarrow{sr,*} v$ consists of sr-reductions that first reduce s until it is irreducible, i.e. there is some sr-irreducible s_1 with $s \xrightarrow{sr,*} s_1$. Then Lemma C.11 shows that the *VNNBsr*-reduction of s is finite and ends with an *VNNBsr*-irreducble expression. □

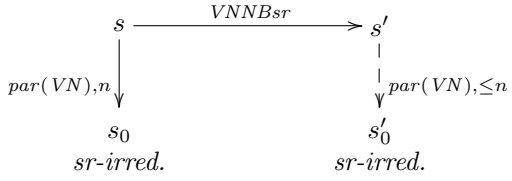
This obviously implies:

Corollary C.13. *If t has an infinite VNsr-reduction, then for every closing value-substitution σ: $\sigma(t)\Uparrow$, i.e. t is an Ω-expression.*

We also need more information about the necessary parallel reduction

Definition C.14. *The following specialized parallel reduction $\xrightarrow{par(VN)}$ is defined as the following specialized variant of the parallel reduction $\xrightarrow{par}$: A single parallel reduction is derived from a single beta- or case-redex, combined with arbitrary parallel (seq)-reductions. Note that a single sr-reduction is also a $\xrightarrow{par(VN)}$-reduction.*

Lemma C.15. *Let s be an (open) expression such that $s \xrightarrow{par(VN),n} s_0$, where s_0 is sr-irreducible, and $s \xrightarrow{VNNBsr} s'$. Then there is some s'_0 such that $s' \xrightarrow{par(VN),\leq n} s'_0$, where s'_0 is sr-irreducible.*



Proof. That the constructed parallel reduction is a $s \xrightarrow{par(VN),\leq n} s'$-reduction follows from an extension of the proof of the forking diagrams in Lemma C.6, where we have to scan all cases and check the construction of the to-be-constructed parallel reduction. The upper bound n for the length of the parallel reduction sequence follows by induction. □

Definition C.16. *Let t be an expression and p be a position in t. Then the constructor-lambda-depth of p, denoted $cl\text{–}depth(p)$ is defined as follows:*

- *If p is empty, then $cl\text{–}depth(p) := 0$.*
- *If $p = i.p'$, then $cl\text{–}depth(p) := cl\text{–}depth(p')$ in the following cases: $t|_p$ is an application; $t|_p$ is a case-expression and $i = 1$; $t|_p$ is a seq-expression.*
- *If $p = i.p'$, then $cl\text{–}depth(p) := cl\text{–}depth(p') + 1$ in the following cases: $t|_p$ is a lambda-expression; $t|_p$ is a constructor-expression; $t|_p$ is a case-expression and i points into an alternative*

The `Bot`*–cl–depth of an expression is the minimum of all cl–depths of all occurrences of* `Bot`*. If there are no* `Bot`*-occurrences, then it is ∞.*

Lemma C.17. *Let s be an (open) expression such that $s \xrightarrow{par(VN)} s'$. If n, (or n'. resp.) are the* `Bot`*–cl–depths of s (or s'. resp.), then $n - n' \leq 1$.*

Proof. This holds, since a single parallel reduction step is composed from copies of a single redex. If this contains a `Bot`, then the contribution to the `Bot`*–cl–depth* comes from a topmost `Bot`. The case of seq-reductions does not change the `Bot`*–cl–depth*. □

Now we can justify the following mathematical (non-effective) construction $ValueConstr_n$ of a $\mathcal{P}$-value for a $\mathcal{P}'$-value v of $\mathcal{P}$-type, that cuts the expressions for a paramater n by replacing subexpressions whose *cl–depth* (see below) exceeds n by `Bot`.

Definition C.18 (Value construction with depth cut).

- *$ValueConstr_n(t)$: 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 cl–depth of the subexpression exceeds $n+1$, then replace the subexpression by* `Bot` *not changing its type.*

We see that if an expression $\mathit{ValueConstr}_n(t)$ does not contain an occurrence of `Bot`, then it is an exact construction without approximation, i.e., $\mathit{ValueConstr}_n(t) \sim t$.

Lemma C.19. *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.20. *Let t be an expression. If for some closing value-substitution σ the reduction $\sigma(t) \xrightarrow{sr,n} v$ holds for some value v, and t' is constructed from t using ValueConstr for binder-depth $n+1$, then $\sigma(t') \xrightarrow{par(VN),\leq n} v' \leq_{\mathcal{P}',T} v$.*

Proof. This follows from Lemmas C.12, C.15, and C.17 since the `Bot`-insertions are below *cl–depth* n, and since Ω-expressions are smaller than other expressions w.r.t. $\leq_{\mathcal{P}',T}$. Also Corollory C.13 is required, since this shows that the replacement of expressions with infinite *VNNBsr*-reductions are Ω-expressions and thus do not interfere. □

Lemma C.21. *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 σ' and all closed $\mathcal{P}'$-reduction contexts R', also the implication $R'[\sigma'(s)]{\downarrow} \implies R'[\sigma'(t)]{\downarrow}$ holds.*

Proof. Let σ' be a $\mathcal{P}'$-closing value substitution and R' be a closed $\mathcal{P}'$-reduction context, such that $R'[\sigma'(s)]{\downarrow}$ holds. If the type of R' is a $\mathcal{P}'$-type, then we use $R'' = (R'; \lambda x.x)$. 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, and also every $\xrightarrow{par(VN)}$-step reduces the `Bot`–*cl–depth* at most by one, by Lemma C.19, by Lemma C.20 and by Lemma C.17. 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.

Proof of Theorem 5.4.
The claim is:

> *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 σ and F-free $\mathcal{P}$-reduction contexts R, where $R[\sigma(s)], R[\sigma(t)]$ are closed if, and only if $s \leq_{\mathcal{P}',T} t$ holds.*

This follows from the F-free CIU-theorem 4.1 and from Lemma C.21. □

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

Theorem C.22. *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.*

Proof. Lemma C.4 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 Applicative Bisimulation

We show in this section that equality of expressions can be determined by applicative bisimulation. First we illustrate using two examples, what the method can do and how it can be applied, using examples.

1. In our language, for every program it holds that $\Omega \not\sim \lambda.\Omega$. The reason is that both expressions are closed and that Ω does not terminate, whereas $\lambda.\Omega$ terminates. Thus looking for the convergence behavior in an empty context distinguishes the two expressions.
2. Let $\mathcal{P}$ be a program where the boolean data type is defined. Let $s_1 = \texttt{True}$ and $s_2 = \texttt{if}\ x\ \texttt{then True else True}$. Then $s_1 \sim s_2$. Using applicative bisimulation, this is detected as follows: First a closing value substitution replacing x by some value v is applied where every well-typed possibility has to be tried. Then the two expressions have to evaluated. Only s_2 will be affected and will be $s_2' = \texttt{if}\ v\ \texttt{then True else True}$. Only `True` and `False` are possible for v. In any case the result of evaluation will be `True`. Thus applicative bisimulation tells us that s_1, s_2 are equal.
3. A slightly more complex example is derived from a let-over-lambda shift with a converging term. Let $s_1 = (\lambda x.\lambda y.x)\ t$, and $s_2 = \lambda y.((\lambda x.x)\ t)$, where t is a converging expression without free occurrences of y. Then let σ be a value substitution such that $\sigma(t)$ is closed. Assume that $\sigma(t)$ evaluates to a value v_t. Evaluation of $\sigma(s_1)$ and $\sigma(s_2)$ leads to $s_1' = \lambda y.v_t$, and $s_2' = \lambda y.((\lambda x.x)\ \sigma(t))$. Both are closed, and values, but syntactically different. Trying them on a further closed value v as argument and evaluating them leads to $s_1'\ v \xrightarrow{*} v_t$, and to $s_2'\ v \xrightarrow{*} v_t$, hence they behave the same, and thus applicative bisimulation will answer that both are equal.

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 data types, data constructors, a case-construct and recursive definition of functions. The approach was also worked out for a call-by-need non-deterministic calculi in a similar way in [Man05, MSS10].

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

Lemma D.1. *For a relation ν on closed expressions, the equality $((\nu)^o)^c = \nu$ holds. For a relation μ on open expressions: $s\ \mu\ t \implies \sigma(s)\ (\mu)^c\ \sigma(t)$ for all closing value substitutions σ is equivalent to $\mu\ \subseteq ((\mu)^c)^o$.*

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

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 ν over closed expressions: $s\ [\nu]\ t$ if $s{\Uparrow}$ or $s{\downarrow}(c\ s_1 \ldots s_n)$ and $t{\downarrow}(c\ t_1 \ldots t_n)$ and $s_i\ \nu\ t_i$ for all i or $s{\downarrow}\lambda x.s'$ and $t{\downarrow}\lambda x.t'$ and $s'\ \nu^o\ t'$*

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

Lemma 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 $\leq_{\mathcal{P}} \subseteq \leq_b^o$ by Lemma D.1.

Lemma D.4. *For closed values $(c\ s_1 \ldots s_n), (c\ t_1 \ldots t_n)$ of equal type, we have $(c\ s_1 \ldots s_n) \leq_b (c\ t_1 \ldots t_n)$ iff $s_i \leq_b t_i$. For abstractions $\lambda x.s, \lambda x.t$ of equal type, we have $\lambda x.s \leq_b \lambda x.t$ iff $s \leq_b^o\ t$.*

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

Lemma D.5. *The relations $\leq_b$ and $\leq_b^o$ are reflexive and transitive*

Proof. Transitivity follows by showing that $\nu := \quad \leq_b \cup (\leq_b \circ \leq_b)$ 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 $\leq_b^o = \leq_{\mathcal{P}}$.

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 σ 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 3.4.
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 $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ 1": Lemma D.7.(3) 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. □

Proposition D.10. *If $\leq_b$ is a precongruence, then $\leq_b = \leq_{\mathcal{P}}$.*

Proof. Let $s \leq_b^o t$. Then for all closing contexts C: $C[s] \leq_b C[t]$, since $\leq_b^o$ is a precongruence. Hence $s \leq_{\mathcal{P}} t$. The other direction follows from Lemma D.3.

D.1 Determining the Congruence Candidate

Lemma D.11. *If $s \to s'$, then $s \sim_b^o s'$, i.e., $s \leq_b^o s'$ and $s \leq_b^o s'$.*

Proof. This holds, since standard reduction is deterministic which implies that s, s' reduce to the same value, and by the definition of $\leq_b^o$.

Lemma D.12. *If $s \ \widehat{\leq_b^o} \ t$ and $t \to t'$, then $s \ \widehat{\leq_b^o} \ t'$*

Proof. Follows from Lemma D.11.

Definition D.13. *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 $s' \ (\widehat{\leq_b^o})^c \ t$.*

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 $s_i \mathrel{\widehat{\leq_b^o}} t'_i$ 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 $s' \mathrel{\widehat{\leq_b^o}} t'$. The relation $\lambda x.t' \leq_b^o t$ implies that $t \xrightarrow{*} \lambda x.t''$. Lemma D.11 now implies $\lambda x.s' \mathrel{\widehat{\leq_b^o}} \lambda x.t''$. Definition of $\widehat{\leq_b^o}$ now shows that there is some closed $\lambda x.t^{(3)}$ with $s' \mathrel{\widehat{\leq_b^o}} t^{(3)}$ and $\lambda x.t^{(3)} \leq_b \lambda x.t''$. The latter relation implies $t^{(3)} \leq_b^o t''$, which also shows $s' \mathrel{\widehat{\leq_b^o}} t''$.

If θ is a constructor, then there is a closed expression $c(t'_1,\ldots,t'_n)$ with $s_i \mathrel{\widehat{\leq_b^o}} t'_i$ for all i and $c(t'_1,\ldots,t'_n) \leq_b t$. By applying the induction hypothesis to $s_i \mathrel{\widehat{\leq_b^o}} t'_i$ we obtain that $t'_i \xrightarrow{*} t''_i$, where t''_i are values, and hence $c(t''_1,\ldots,t''_n)$ is a value. It follows that $s_i \mathrel{\widehat{\leq_b^o}} t''_i$ by Lemma D.12 and $c(t''_1,\ldots,t''_n) \leq_b t$, by arranging the reduction $c(t'_1,\ldots,t'_n) \xrightarrow{*} c(t''_1,\ldots,t''_n)$ from left to right to obtain a standard reduction. The definition of $\leq_b$ implies that $t \xrightarrow{*} \theta(t_1^{(3)},\ldots,t_n^{(3)})$ with $t''_i \leq_b t_i^{(3)}$ for all i. By definition of $\widehat{\leq_b^o}$, we obtain $s_i \mathrel{\widehat{\leq_b^o}} t_i^{(3)}$ for all i.

Proposition 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 $s \mathrel{\widehat{\leq_b^o}} t$. Let $s{\downarrow}\theta(s_1,\ldots,s_n)$. Then $\theta(s_1,\ldots,s_n)\ (\widehat{\leq_b^o})^c\ t$ using the stability assumption. By Lemma D.14 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.12, 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.12, Propositions D.15, D.10 and Theorem D.9 show the claim.

It remains to show stability:

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

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

- For the (beta)-reduction, $s = s_1\ s_2$, where $s_1 = (\lambda x.s'_1)$, s_2 is a closed value, and $t' = t'_1\ t'_2$. Lemma D.14 shows that $t'_1 \xrightarrow{*} \lambda x.t''_1$ with $\lambda x.s'_1 \mathrel{\widehat{\leq_b^o}} \lambda x.t''_1$ and also $s_1 \mathrel{\widehat{\leq_b^o}} t''_1$. From $s_2 \mathrel{\widehat{\leq_b^o}} t'_2$ and since s_2 is a value, we obtain the next part of the standard reduction $t'_2 \xrightarrow{*} t''_2$ with $s_2 \mathrel{\widehat{\leq_b^o}} t''_2$. From $t' \xrightarrow{*} t''_1[t''_2/x]$ we obtain $t''_1[t''_2/x] \leq_b t$. Lemma D.7 now shows $s'_1[s_2/x] \mathrel{\widehat{\leq_b^o}} t''_1[t''_2/x]$. Hence $s'_1[s_2/x] \mathrel{\widehat{\leq_b^o}} t$, again using Lemma D.7.

- Similar arguments apply to the case-reduction.
- Suppose, the reduction is a δ-reduction. Then $s \mathrel{\widehat{\leq_b^o}} t$ and s is a function name. By the definition of $\widehat{\leq_b^o}$, this implies $s \leq_b^o t$. 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 $S[s] = \tau(s_1, \ldots, s_n)$ and that there is some $t' = \tau(t'_1, \ldots, t'_n) \leq_b^o t$ 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 $S[s'] = \tau(s_1, \ldots, s_{j-1}, s'_j, s_{j+1}, \ldots, s_n) \mathrel{\widehat{\leq_b^o}} \tau(t'_1, \ldots, t'_{j-1}, t'_j, t'_{j+1}, \ldots, t'_n)$.

Remark D.19. Note that the correctness of applicative bisimulation as proved in this section does not imply (without the help of other theorems) that every locally valid equation is also a tautology. This reasoning can only be used for expressions, where the closing value substitutions can be restricted to DT-types since otherwise, the local applicative bisimulation definition and the global applicative bisimulation differ, since there may be more values of a certain type if the program is extended.
However, applicative bismulation together with Main Theorem 5.5 provides a strong tool for detecting and proving equalites.