Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A Two-Valued Logic for Properties of Strict Functional Programs allowing Partial Functions

David Sabel · Manfred Schmidt-Schauß

Received: date / Accepted: date

Abstract A typed program logic LMF for recursive specification and verification is presented. It comprises a strict functional programming language with polymorphic and recursively defined partial functions and polymorphic data types. The logic is two-valued with the equality symbol as only predicate. Quantifiers range over the values, which permits inductive proofs of properties. The semantics is based on a contextual (observational) semantics, which gives a consistent presentation of higher-order functions. Our analysis also sheds new light on the the role of partial functions and loose specifications. It is also an analysis of influence of extensions of programs on the tautologies. The main result is that universally quantified equations are conservative, which is also the base for several other conservative classes of formulas.

Keywords Verification · Functional Programming · Logics · Semantics · Contextual Equivalence

1 Introduction

Clearly, programming and reasoning about the properties of programs is a core task of computer science. A great variety of program logics for this purpose are proposed in the computer science literature.

In this paper we investigate a logic for a strict higher-order functional programming language with polymorphically typed, recursive function definitions and data structures. Instead of starting with the logical view we start from

David Sabel and Manfred Schmidt-Schauß
Institut für Informatik
Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
Tel.: +49 69 798 22890
Fax: +49 69 798 28919
E-mail: {sabel,schauss}@ki.informatik.uni-frankfurt.de

the perspective of a programming language semanticist. The semantics of our programming language is given by the operational semantics which defines the evaluation of programs. Equality of basic data is canonical; equality of functions on data (say natural numbers) is naturally defined as extensional equality, but equality of recursively defined (partial) higher-order functions is in general not rigorously treated in most logics; for example in lambda-calculi often convertibility is used as definition of equality, which is usually too syntactic, and hence too fine-grained. We choose the natural notion of contextual equivalence (see e.g. [35,38,37]) as program equivalence which follows Leibniz' law to identify the indiscernible programs. Contextual equivalence is based on the operational semantics; two programs are contextually equivalent if their (operational) behavior can not be distinguished if the first program is replaced by the other program in any surrounding larger program.

Now our next step was to use this notion of contextual equivalence as equality in logical formulas on programs. Our proposed logic is a rather simple first-order, two-valued logic comprising equality, truth values, Boolean connectives, and universal and existential quantification over values of some type. The logic can be completely described in a few pages. However, this approach raises several questions which we try to answer throughout this paper.

From the semanticist's perspective, undefined, partial, and non-terminating functions are natural, since they occur in a lot of programs and are in extreme cases unavoidable. Some examples are the tail function which is a partial function, since it is undefined for the empty list, or the implementation of an interpreter for a (Turing-complete) programming language which of course cannot be a total function. The notion of contextual equivalence can deal with those undefined expressions without any problems. On the other hand in the literature on program logics and on automated reasoning tools (e.g. theorem provers) partial functions and undefined expressions are often forbidden or require special modelling and treatment.

So one question is how to classify our approach in relation to other existing approaches. At this point we only give a short description of the existing approaches, but later in Section 1.1. we will extend this discussion and provide more details of our logic.

The logics in the literature and in use in deduction systems employ different methods for dealing with *partial functions* (overviews can be found e.g. in [10,41,16]). The methods are: (i) All functions must terminate and partial functions are represented as total functions plus a domain of definition, and reasoning about the functions requires reasoning over their domain. (ii) The partiality of functions and undefinedness of expressions is spread into the logic, i.e. there is a *logical* value for undefined and consequently the logic is three-valued. (iii) All functions must terminate, and partial functions are modeled using loose specification: If function f is applied to an argument a outside of its domain, then $f(a)$ is seen as having any value, and theorems must be provable for any such possibility. In this approach partiality stemming from undefined patterns like tail of a list can then be treated as termination. (iv) Nontermination is seen as an extra expression $\bot$, also as in our logic.

Contextual equivalence usually is hard to prove, since all program contexts need to be taken into account. Hence the question arises, whether a useful program logic can be constructed upon this notion with an appropriate set of equalities and whether and how proofs of equalities are possible. We answer these questions in this work and show that there are strong proof methods which can be used to prove contextual equivalence and we show that a lot of equivalences indeed hold. The equivalences are mainly program transformations which can be used for partial evaluation of programs.

A practically relevant and important question for deduction system built upon LMF is the question whether proofs of theorems and lemmas can be automated. We will show the correctness of a lot of deduction rules, and also that induction proofs over data structures are possible in our logic. The corresponding proof rules and operations are very similar to usual first-order deduction systems.

We also consider the problem whether correctness proofs in our logic can be performed *locally* for a given program and formula, i.e. without considering any extensions of the defined functions and extensions of the type and data constructors. We will show that for several classes of formulas this can indeed be performed, but we will also show that this is not possible in general. These results may also be interesting for other logics (e.g., logics which forbid partial functions) since our counter-examples may be applicable to those logics, too.

Related Logical Systems

There is a history of logics and proof systems for proving theorems about programs. Early and influential inductive systems proving properties of programs are [8,29]. A system which was a source of inspiration for our work is the automated proof system `VeriFun` (see [49,42,48] and [47]). Standard textbook examples of proofs are concerned with the properties of addition on Peano-numbers like commutativity and associativity. Even more complex (encoded) computer science problems could be treated like the undecidability of the Halting Problem ([9]) and the formal verification of compilers (see e.g. [33]). For a whole archive of such proofs e.g. [30].

Roughly speaking most of the state of the art theorem provers are either built on constructive type theory [13], like e.g. Coq [5,12], Agda [2] or on higher-order logic [22], like e.g. Isabelle/HOL [36,27], HOL4 [24], or also on first-order logic like e.g. ACL2 [1], and `VeriFun` [47]. Most of these proof systems require termination of the defined functions (and also a corresponding termination proof) before any proposition can be proved about these functions.

To treat partiality in such logics, several papers argue for loose specification as encoding method [15,19,23,42]. There are several extensions and approaches to partial functions for the mentioned systems (see e.g. [7] for an excellent overview, and e.g. [31,32] for Isabelle/HOL, [6] for Coq, [49,42] for `VeriFun`), [21] for induction provers; where the core logic can only deal with total functions. In contrast, our logic *LMF* is not restricted to total functions.

More related are systems and logics where partiality of functions is allowed in the logic. An overview of several approach can e.g. be found in [10]. We will briefly compare some of the existing approaches with our logic. In the subsequent section we will discuss the properties of our logic in detail.

An early example of an automatable logic dealing with partiality is Scott's LCF [46]. As in our approach, LCF separates between the logic for reasoning on programs and the programs itself. In contrast to our approach LCF is built on Scott's domain theory and thus it uses a denotational semantics, while we use contextual semantics which is directly derivable from the operational semantics. Another difference is that quantification in formulas of LCF range over values and the diverging expression $\bot$, while in our logic the quantification only ranges over values.

Another approach is used in the system IMPS [17] which implements the logic PF* [14] and allows partial functions, but requires that every logic formula is either true or false, i.e. the truth value of a formula is always defined. This property is established by setting every formula including a nonterminating expression as false. For example the equation $\bot = \bot$ is wrong in this logic. In contrast in our logic this equation is a tautology.

Another system with partial functions is the system VDM [28] which is based on the logic LPF [4]. In their approach the partiality is also allowed on the logical level and thus the law of the excluded middle does not hold. There are numerous papers arguing for a three valued logic, examples are ([41,20]). Our approach is different, since our logic is two-valued.

Other related work on specifications and a logic of functional programs also allowing polymorphism and partiality is e.g. in [45]. Work on program transformations in higher-order term rewriting with constructors using an operational correctness criterion and permitting induction principles can be found in [11].

1.1 Our Logic *LMF*

We now describe our logic in detail and discuss several properties of the logic and give justifications for the chosen approach. Our logic consists of a functional programming part and a logical part. The programming part is a typed strict functional higher-order programming language with lambda abstraction, data constructors, like natural numbers (in the Peano-encoding) and lists and recursive function definitions. A *program* fixes the definition of data constructors and named recursive functions which have a polymorphic type.

In the logical part one can express formulas on *equations between expressions* which may contain data constructors and functions of a fixed program. The logic is two-valued and first-order where the only predicate is equality ($=$) of expressions. Formulas are monomorphic where quantification is over closed data values of the appropriate type. Quantification is over the values, like `True`, `False`, lists, and also over abstractions. The semantics of equality of program expressions is contextual semantics: i.e. $s = t$ is valid, iff for all program 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. This will turn out to be equivalent to an extensional view of the equality of functions (Proposition 3.18). We use $\bot$ as an expression representing nontermination. Formulas are *tautologies* of the logic iff they hold under any extension of the program by further function definitions or data types (i.e. they are *(locally) valid* for every extension), which makes the logic "monotonic" w.r.t. such extensions.

As an example, this allows the specification of a function that computes the tail of a list, and a function computing list concatenation as follows:

$$\begin{aligned}\texttt{tail} &= \lambda xs.\texttt{case}\, xs\, \texttt{of}\, (\texttt{Nil} \to \bot)\, (\texttt{Cons}\, z\, zs \to zs)\\ \texttt{append} &= \lambda xs, ys.\texttt{case}\, xs\, \texttt{of}\, (\texttt{Nil} \to ys)\, (\texttt{Cons}\, z\, zs \to \texttt{Cons}\, z\, (\texttt{append}\, zs\, ys))\end{aligned}$$

In the logic the associativity statement of list concatenation can be formulated as $\forall x, y, z :: (\texttt{List Nat}) : \texttt{append}\ x\ (\texttt{append}\ y\ z) = \texttt{append}\ (\texttt{append}\ x\ y)\ z$. This formula can then be proved correct using induction proof rules.

Another example for a formula is $\forall x :: \texttt{Nat}.\neg(x = 0) \implies (\mathit{div}\ x\ x) = (S\ 0)$ where we assume that *div* is a defined (partial) function for division of numbers in the Peano-encoding. The formula is a tautology, where for example for $x = 0$, since $(\mathit{div}\ 0\ 0) = \bot \neq (S\ 0)$ holds in our logic, the formula is $\texttt{ff} \implies \texttt{ff}$, which results in $\texttt{tt}$.

We now discuss different aspects of our approach, give details of its features, and also compare it with other approaches.

Expressions that are permitted in the equations are program expressions of a typed strict higher-order functional language. Recursive definitions of functions and data types with recursive constructors (positive and negative recursion) are permitted. Functions are polymorphically typed, whereas expressions in equations are monomorphically typed. Evaluation of (closed) expressions is a deterministic call-by-value evaluation, which is like innermost reduction, but does not reduce inside the body of abstractions. Instead of function definitions by argument pattern, the language has a case-construct, and functions are defined recursively as a single abstraction.

Values and the Bottom Element. Values in our logic are special expressions, for example Peano numbers $0, (S\ 0), S\ (S\ 0)$, Boolean constants `True`, `False`, lists of values, and lambda-abstractions. A nonterminating expression is definable, which we denote as $\bot$, which is not considered as a value. The definition of our equality implies that for every monomorphic type, all nonterminating expressions and undefined results of function applications of this type are equal. We can reason about $\bot$ on the logical level, but the evaluation of functions cannot compute with $\bot$.

Results of Partial Functions. In our approach a partial function, say f, can be represented, where in case the argument is outside of the domain of f, the result is $\bot$. Nontermination of f on argument a is equivalent to a being not

in the domain of f. We employ so-called strong equality, i.e. $\bot = \bot$ is true as a formula. Thus for two partial functions $f, g :: \texttt{Nat} \to \texttt{Nat}$, the formula $\forall x :: \texttt{Nat}.f(x) = g(x)$ is equivalent to the mathematical equality of f, g as functions: The domains must be equal, since values are not equal to $\bot$, and for arguments x not in the domain, the result is always $\bot$, hence $\bot = \bot$ shows that $f(x) = g(x)$ in this case (see Proposition 3.18 for extensionality). Moreover, in our approach, the equation $f = g$ is also a syntactically valid formula with the same logical truth value. Now let f, g be two (proper) partially defined functions with the same domain $D_{f,g} \subseteq D_{\texttt{Nat}}$, and which are equal on all arguments in $D_{f,g}$. Then $\forall x :: \texttt{Nat}.f(x) = g(x)$ is a tautology in our logic. Note that under loose specification this does not hold, since for arguments a outside of the domain, $f(a)$ and $g(a)$ may be different. In three-valued logics, the formula $\forall x :: \texttt{Nat}.f(x) = g(x)$ is also not a tautology, since the logical value is $\bot$, the third logical value, except if $\bot = \bot$ is defined as logically true.

In our logic *LMF* the equation $s = s$ is logical true for all expressions s, moreover, our logic validates all the equational axioms, like reflexivity, symmetry, transitivity, and congruence, for all expressions. This is in contrast to the approach of e.g. [15], where all equations with $\bot$ are false, and hence $s = s$ is not correct in [15] for nonterminating closed expressions s.

Another problem of the loose specification approach is that in case of a one-element range $\{d\}$ a partial function and a constant function cannot be distinguished, whereas in our approach a partial function $\{d\} \to \{d\}$ is $f = \lambda x.\bot$, and the constant function is $g = \lambda x.x$, which are different in our logic.

Decidability. It is undecidable whether $f(x) = \bot$ holds for a given function f and argument x, but this does not make our logic harder than others from a complexity point of view. Only the place where the complexity appears is different. Functions that produce an error like *tail* for the argument `Nil` allow an easy decision algorithm, if the argument is a value.

Equality is defined in our logic using all observations. This results in so-called "strong" equality; i.e. $s = t$ of a type like `Nat` is valid for closed s, t, if s evaluates to the value v, t evaluates to the value w, and v is syntactically the same as w, or if $s = t = \bot$. For higher-order functions we do not use syntactic equality, but observational equality. For example if f, g are two defined functions, then these are equal, if f and g cannot be distinguished by observing the termination / non-termination of reductions in any program context. Of course, this equality is not a computable function.

This notion of equality is the expected one on concrete domains of values, like natural numbers, lists, and so forth, and scales up in a well-behaved way to higher-order functions. The equality of two abstractions v, w holds, if these represent the same partial functions (up to contextual equality), i.e. we have extensional equality. This is again different to the approach, where conversion (i.e. the reflexive-symmetric-transitive closure of the reduction relation in any context) is defined as equality. However, our equality includes conversion, i.e. if $s \overset{*}{\leftrightarrow} t$, then $s = t$ also holds.

Note that the logical equality = cannot be used in function definitions, due to the separation of the expression and logical level. An operational equality $\mathtt{eq}_T$ for type T on the expression level and for values that do not contain abstractions can easily be defined. This equality $\mathtt{eq}_T$ is, however, partial and does not terminate for all arguments: for example $(\mathtt{eq}_T\ \bot\ s)$ does not terminate.

Valid formulas (or tautologies) of the form $\forall x_1, \ldots x_n.s = t$ can be used as rules for replacing s by t. They can also be used as rewriting rules, if the substitutions are restricted to value substitutions.

Freeness of Constructors within the Values holds, due to the definition of equality: $c\ v_1 \ldots v_n = c'\ w_1 \ldots w_m$ is false whenever c, c' are constructors (of the same result type), $c \neq c'$, and v_i, w_j are values. Also, $c\ v_1 \ldots v_n = c\ w_1 \ldots w_m \iff (v_1 = w_1 \wedge \ldots \wedge v_n = w_n)$ is logically valid for all closed values v_i, w_j. Also, $c\ v_1 \ldots v_n \neq \bot$, $c\ v_1 \ldots v_n \neq \lambda x.r$, and $\lambda x.r \neq \bot$ in our logic, where v_i, w_j are values.

Consistency of the Logic and of Equations. Our logic is consistent for any set of data constructors and function definitions. The reason is that the logical value of equations is unambiguously defined.

The (equational) sublogic that only consists of the ground equations is also consistent, since there are at least two different elements of every type, for example $\bot \neq \mathtt{Nil}$. The equational axioms like substitutivity, reflexivity, transitivity, symmetry also hold for functional expressions, for example the equation $s = s$ is logically true, for any expressions s, independent of termination or non-termination of s, which is not the case for several other logics.

Loose Specification, Partiality and Recursive Definitions. In general, the combination of loose specification, partiality, and recursive function definitions may lead to *inconsistencies* in a logic. An example for this is contained in [21]. We give a variation of this example: let f be a function defined as $f := \lambda x.S\ (f\ x)$. Then this function does not terminate. Loose specification means that $f\ 0 = a$ for some natural number a, but also the fixpoint equation for f must hold, in particular $f\ 0 = S\ (f\ 0)$ must hold, which leads to the equation $a = S\ a$, and hence to the inconsistency that all numbers are equal. In our logic this is resolved as follows: $fx = \bot$ for all x, and since $S\ \bot = \bot$, the logic remains consistent.

If loose specification is only used for dealing with undefined patterns, then the logic remains consistent; and in the case of ATD-formulas (see Definition 5.10), where an example is $\forall x :: \mathtt{Nat}.\neg(x = 0) \implies (div\ x\ x) = (S\ 0)$, our logic defines the same tautologies as the loose specification semantics (see Section 5.1.6 and Theorem 5.14).

Induction on the Data is the usual proof mechanism for infinite sets of arguments. This is also valid for partial functions. However, induction on the function evaluation, i.e. on the number of reductions, is valid only if termination of the involved functions can be proved.

Boolean Values, Boolean Operations and Predicates. These are on the logical level and not on the expression level. We assume that there is also a Boolean type and constructors $\texttt{True}, \texttt{False}$ on the functional level, where the Boolean functions are defined ones and may produce errors depending on their definition and arguments, i.e. may result in $\bot$.

Predicates other than $=$ on the logical level are not allowed in *LMF*, but can be simulated by the logical formula $p(s) = \texttt{True}$. Our representation of the formula $s \neq \texttt{Nil} \implies \texttt{head}\ s \geq 0$ is $s \neq \texttt{Nil} \implies (\texttt{head}\ s \geq 0) = \texttt{True}$, which treats undefinedness on the expression level. Transitivity can be formulated as

$$\forall x, y, z : \texttt{Nat} : x \neq \bot \wedge y \neq \bot \wedge z \neq \bot \\ \implies \big(x \leq y = \texttt{True} \wedge y \leq z = \texttt{True} \implies x \leq z = \texttt{True}\big)$$

Truth of Formulas under Extensions. This is the question whether formulas remain valid if data structures or if independent function definitions are added. This is a topic which is often neglected. Our view is that only formulas that remain valid under all such extensions are tautologies. Technically, if a formula holds w.r.t. a fixed program $\mathcal{P}$, then we say it is locally valid (for $\mathcal{P}$), and if the formula also holds for all extensions of $\mathcal{P}$ by functions and data types, then it is a tautology of the logic (w.r.t. $\mathcal{P}$). If the functions only operate on "basic data" like natural numbers or lists of natural numbers, then this is not an issue. The conservativity under extensions is an issue if we ask for the tautology property of formulas that quantify over higher-order values, for example equations between higher-order functions which operate on data that may contain abstractions. The reason is that an extension in general increases the set of abstractions of a given type.

We will show that several classes of locally valid formulas like universally quantified single equations remain valid under extensions and that this does not hold in general for formulas with a complex quantifier prefix.

One important result is that a lot of formula transformations derived from almost all intuitively correct equality transformations and reductions preserve the property that these are tautologies. All of them can thus be used in mechanical proofs of tautologies. This is in conformance with other provers and approaches that take for granted that program reductions are valid steps of a mechanical proof.

Nested Quantifiers are of course permitted. However, the usual interesting tautologies in verification are of the form $\forall^* F$, where F is quantifier-free. Also existential formulas $\exists^* F$, where F is quantifier-free are of interest: they state the definability of certain functions, but do not give a construction method. Other formulas with nested and alternating quantification of variables may be of interest as long as they speak about data values, but if quantification is in addition over function types the correctness of these formulas tend to speak more about peculiarities of the logic and the functional programming language than over real programming problems.

Strict vs. Lazy Functional Programming Languages. As explained before, quantification in strict functional programming languages can be done over the values, i.e. $\forall x : \texttt{Nat} \ldots$ quantifies over $0, (S\ 0), \ldots$ and thus exactly over the natural numbers. In contrast, using a lazy functional programming language, quantification would have to be over all expressions. For example for type `Nat`, this means that quantification is not only over $0, (S\ 0), \ldots$, but also over $\bot, (S\ \bot), S\ (S\ \bot)$, and even $(S\ (S\ \ldots))$, where nesting is infinitely deep. The restriction to a strict functional language with strict constructors (as in *LMF*) has the drawback, that our logic cannot speak about infinite lists (because those are not definable). Investigation on extensions is left to future work.

1.2 Results

Besides the construction and definition of a program logic our main results are
(i) proofs of correctness of several deduction and transformations rules, and also further rules;
(ii) conservativity theorems showing that for several kinds of formulas their local validity already implies that the formulas are tautologies.

In particular, call-by-value (beta)- and (case)-reductions are correct (Theorem 3.14). Almost all deduction rules of a verification system like `VeriFun` are also 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, and adapted call-by-name reductions and further deduction rules are correct (Theorem 3.14). Also several classes of locally valid formulas are shown to hold in all extensions, i.e. they are tautologies of the logic: An important class are universally quantified equations (Theorem 5.2) and general monomorphic theorems if functions do not occur in the data (Theorem 5.8). In general, the local validity of formulas does not imply that they are also tautologies (see Theorem 5.20). The relation to loose specifications is clarified in Theorem 5.14.

To establish these results we introduce proof techniques for contextual equality in combination with polymorphic types (for a call-by-need calculus see also [39]), which allow the proof of a CIU-Theorem ([34]) from context lemmas (for a general approach see also [43]), 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 formulas does in general not imply that these are tautologies. We conjecture that polymorphic formulas that are universally quantified polymorphic equations and which are locally valid also hold in all extensions.

1.2.1 Structure of the Paper.

In Section 2 we define the syntax 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 2.3). We show in Section 3 that equality is conservative if programs are extended by new function definitions and new data types, provided certain preconditions hold. In Section 4 we explain the logic, and its semantics. In Section 5 we analyze some conservativity properties and state open questions. In Section 6 we consider polymorphic formulas. We finally conclude in Section 7. 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 now focus on (i), whereas (ii) is postponed to Section 4.

2.1 Syntax and Semantics

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, then the expression level, and later the typing of symbols and expressions.

2.1.1 Syntax

In Fig. 1 the syntax of types and type schemes is given. 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 *function types*, and types $(K\ T_1 \dots T_{ar(K)})$ are called *constructed types*. 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 $T_{K,i,1}, \dots, T_{K,i,m_i}$.

The (type-free) syntax of expressions *Expr* and of values *Val* over a signature $(\mathcal{F}, \mathcal{K}, \mathcal{D})$ is shown in Fig.1. Note that data constructors can only be used with all their arguments present and that there is a $\texttt{case}_K$ for every type constructor $K \in \mathcal{K}$. The $\texttt{case}_K$-construct has a case-alternative $((c_i\ x_1 \dots x_{ar(c_i)})\ \texttt{->}\ s_i)$ for every constructor $c_i \in D_K$, where the variables in a pattern have to be distinct. The scoping rules in expressions are as usual. We assume that expressions satisfy the distinct variable convention before reduction is applied, which can be achieved by a renaming of bound variables. An

Types: $\tau, \tau_i \in \mathit{Typ} ::= X \mid (\tau_1 \to \tau_2) \mid (K\ \tau_1 \dots \tau_{ar(K)})$, where $K \in \mathcal{K}, X \in \mathit{TVar}$

Type schemes: $\forall \mathcal{X}.\tau$, where $\tau \in \mathit{Typ}$ and $\mathcal{X}$ are all types variables of τ

Expressions: $s, s_i, t \in \mathit{Expr} ::= x \mid f \mid (s\ t) \mid \lambda x.s \mid (c_i\ s_1 \dots s_{ar(c_i)})$
$\mid (\mathtt{case}_K\ s\ ((c_1\ x_{1,1} \dots x_{1,ar(c_1)})\ \mathtt{->}\ s_1)$
$\dots$
$((c_n\ x_{n,1} \dots x_{n,ar(c_n)})\ \mathtt{->}\ s_n))$ with $\{c_1, \dots, c_n\} = D_K$
where $f \in \mathcal{F}$, $K \in \mathcal{K}, c_i \in \mathcal{D}, D_K \subseteq \mathcal{D}$, and $x, x_i \in \mathit{Var}$

Values: $v, v_i \in \mathit{Val} ::= x \mid \lambda x.s \mid (c\ v_1 \dots v_n)$ where $s \in \mathit{Expr}$, $x \in \mathit{Var}$

Contexts: $C \in \mathit{Ctxt} ::= [\cdot] \mid \lambda x.C \mid (C\ s) \mid (s\ C) \mid (c_i\ s_1 \dots s_j\ C\ s_{j+2} \dots s_{ar(c_i)})$
$\mid\ (\mathtt{case}_K\ C\ \mathit{alts}) \mid (\mathtt{case}_K s\ \dots (c_i\ x_1 \dots x_{ar(c_i)}\ \mathtt{->}\ C) \dots)$
where $s, s_i \in \mathit{Expr}$, $x, x_i \in \mathit{Var}$

Reduction contexts:
$R \in \mathit{RCtxt} ::= [\cdot] \mid (R\ s) \mid (v\ R) \mid \mathtt{case}_K\ R\ \mathit{alts} \mid (c\ v_1\ \dots\ v_{i-1}\ R\ s_{i+1} \dots\ s_n)$
where $s, s_i \in \mathit{Expr}$, $v, v_i \in \mathit{Val}$

Fig. 1 Syntax of types, expressions, values, contexts, and reduction contexts over a signature $(\mathcal{F}, \mathcal{K}, \mathcal{D})$, a set of variables Var, and a set of type variables TVar.

Reduction rules:

(beta) $((\lambda x.s)\ v) \to s[v/x]$ where $v \in \mathit{Val}$

(delta) $f :: T \to d_f$ if $f = d_f :: T' \in \mathit{Def}_{\mathcal{F}}$
The reduction is accompanied by a type instantiation $\rho(d_f)$ where $\rho(T') = T$

(case) $(\mathtt{case}\ (c\ v_1 \dots v_n)\ \dots ((c\ y_1 \dots y_n)\ \mathtt{->}\ s) \dots) \to s[v_1/y_1, \dots, v_n/y_n]$
where $v_1, \dots, v_n \in \mathit{Val}$

Standard reduction: $R[s] \xrightarrow{sr} R[t]$ iff $s \to t$ by a reduction rule and $R \in \mathit{RCtxt}$.

Fig. 2 Reduction rules and standard reduction for $\mathcal{P}$-expressions ($\mathcal{P} = ((\mathcal{F}, \mathcal{K}, \mathcal{D}), \mathit{Def}_{\mathcal{F}})$)

explicit renaming in implementations could be optimized by using de Bruijn-indices. For our examples, 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 in Fig. 1 the syntax of *contexts* $C \in \mathit{Ctxt}$ is defined, where $[\cdot]$ denotes the context hole. The notation $C[s]$ means the expression that results from replacing the hole in C by s. Special contexts are the *reduction contexts* (see Fig. 1) which indicate the position of a call-by-value redex.

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*. The same notation is used for contexts. A value-substitution σ substitutes variables by values, it is called *closing* for expressions $s_1, \dots, s_n$ iff $\sigma(s_1), \dots, \sigma(s_n)$ are closed.

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 $\mathit{Def}_{\mathcal{F}} := \{(f, d_f) \mid f \in \mathcal{F}\}$, 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 $id \in \mathcal{F}$, and if $map, head, fbot \in \mathcal{F}$, these functions can be defined as:

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

2.1.2 Typing of Expressions

We now extend expressions with type labels and distinguish between usual expressions and expressions in function definitions:

The definitional expressions d_f are typed by a standard polymorphic type system, like the Hindley-Milner type system, where we assume that every subexpression is annotated with a type. For instance, the definitions of *head* fully annotated is as follows:

$$\begin{array}{l} head^{\mathrm{List}\ \alpha\to\alpha} = (\lambda xs^{\mathrm{List}\ \alpha}.(\mathtt{case}_{\mathrm{List}}\ xs^{\mathrm{List}\ \alpha} \\ \qquad\qquad ((y^{\alpha} : ys^{\mathrm{List}\ \alpha})^{\mathrm{List}\ \alpha}\ \mathtt{->}\ y^{\alpha}) \\ \qquad\qquad (\mathtt{Nil}^{\mathrm{List}\ \alpha}\ \mathtt{->}\ (fbot^{\mathrm{List}\ \alpha\to\alpha}\ \mathtt{Nil}^{\mathrm{List}\ \alpha})^{\alpha}))^{\alpha})^{\mathrm{List}\ \alpha\to\alpha} \end{array}$$

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 (see Section 4), 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. We omit the concrete rules of the monomorphic type system, since they are standard. For convenience we assume that every subexpression carries its monomorphic type label, which is not so standard, but it is necessary, since correctness of program transformations requires a given typing of expressions.

Definition 2.2 We say a program $\mathcal{P}' = ((\mathcal{F}', \mathcal{K}', \mathcal{D}'), Def_{\mathcal{F}'})$ *extends* the program $\mathcal{P} = ((\mathcal{F}, \mathcal{K}, \mathcal{D}), Def_{\mathcal{F}})$ (denoted with $\mathcal{P}' \sqsupseteq \mathcal{P}$), if $\mathcal{F}' \supseteq \mathcal{F}, \mathcal{K}' \supseteq \mathcal{K}, \mathcal{D}' \supseteq \mathcal{D}, Def'_{\mathcal{F}'} \supseteq Def_{\mathcal{F}}$, such that $\mathcal{D}' = \mathcal{D} \cup \bigcup_{K \in \mathcal{K}' \setminus \mathcal{K}} D_K$.

2.1.3 Operational Semantics

We will now define the standard reduction for $\mathcal{P}$-expressions which is a usual call-by-value reduction, which uses call-by-value beta- and case-reduction and does not reduce inside the body of abstractions. The reduction is performed on the type labeled expressions and thus it must inherit the type labels. However, the reduction does not rely on the typing of the expression, i.e. it could also be performed (and defined) on the type erasure of the type labeled expressions. Nevertheless for the formal treatment (e.g. in correctness proofs of program transformation) keeping track of the type labels is more advantageous.

Definition 2.3 *Reduction rules* are defined in Fig. 2 without mentioning all types. The *standard reduction* $\xrightarrow{sr}$ applies a reduction rule inside a reduction context (see Fig. 2). 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 standard reduction is deterministic. 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.

2.2 Valid Programs

We now add requirements which must be satisfied by any program $\mathcal{P}$, i.e. all other programs are excluded and thus are defined to be invalid. As a prerequisite we define Ω-expressions.

Definition 2.4 We say an expression s is an *Ω-expression* iff for all value substitutions σ where $\sigma(s)$ is closed, $\sigma(s){\Uparrow}$ holds.

By structural induction on the reduction context, it is easy to show that the property of being an Ω-expression extends to reduction contexts:

Proposition 2.5 *Let $s :: \tau$ be an Ω-expression. Then for every reduction context $R[\cdot :: \tau]$, the expression $R[s]$ is an Ω-expression.*

Definition 2.6 (Valid Program) A program $\mathcal{P}$ is *valid* if it fulfills the following two properties for every (monomorphic) $\mathcal{P}$-type T:

- There is at least one closed value of type T.
- There is a closed Ω-expression of type T, denoted as $\bot^T$.

Note that the first property on valid programs excludes types like the type `Foo` with one constructor `foo` :: `Foo` $\to$ `Foo`. The only potentially closed value would be an infinitely nested expression `foo`(`foo`(...)), which does not exist.

The second property of valid programs is satisfied if there is a single definition $\mathit{fbot} = (\lambda x.\mathit{fbot}\ x) :: \forall a, b.a \to b$. Then the expressions $(\mathit{fbot}\ v) :: \tau$ do not converge, where v is any closed value. In the following we will use $\bot$ as an abbreviation of $(\mathit{fbot}\ v)$. Inside function definitions the expression $(\mathit{fbot}\ v)$ can be used as a *polymorphic* Ω-expression of any polymorphic type. This also allows us to construct values $\lambda x.\bot^\tau$ of any given function type.

The expressions $\bot^\tau$, where τ may also be a polymorphic type in function definitions, allow us to define partial functions. For instance, the function *tail* could be defined as $\mathit{tail} = \lambda xs.\mathtt{case}_{\mathrm{List}}\ xs\ (y : ys \texttt{->} ys)\ (\texttt{Nil} \texttt{->} \bot^{\texttt{List}\ a})$.

Remark 2.7 In the following we will sometimes assume that there are constants $\texttt{Bot}^T$ of every type T that represent the expressions $\bot^T$. We assume that these constants are divergent. There are reduction rules (see Figure 3) for these constants. The use of these constants and the reduction rules do not change the equivalence of expressions, as proved below.

In the rest of the paper all programs are assumed valid.

2.3 Equivalence of Expressions

The conversion relation defined by applying the reduction rules 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. [40]). 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 $\mathcal{P}$).

Note that the bisimulation ala Howe [25,26] is equivalent to contextual equivalence, but Howe's method is insufficient to prove the main theorem, which requires an approximation method.

Definition 2.8 Assume given a program $\mathcal{P}$. Let s, t be two $\mathcal{P}$-expressions of (ground) type T. Then $s \leq_{\forall\mathcal{P},T} t$ 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 $s \sim_{\forall\mathcal{P},T} t$ iff $s \leq_{\forall\mathcal{P},T} t$ and $t \leq_{\forall\mathcal{P},T} s$. If contexts $C[\cdot]$ are restricted to be $\mathcal{P}$-contexts, then we denote the relations as $\leq_{\mathcal{P},T}$ and $\sim_{\mathcal{P},T}$.

The relation $\leq_{\forall\mathcal{P},T}$ is called the (global) *contextual preorder*, the relation $\sim_{\forall\mathcal{P},T}$ is called the (global) *contextual equivalence*. The relations $\leq_{\mathcal{P},T}$ and $\sim_{\mathcal{P},T}$ are called *local contextual preorder* and *local contextual equivalence*

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

Note that although contextual equivalence tests for convergence, only, it is a strong notion of equality and distinguishes many expressions. The reason is that all contexts can be used to test the expressions. For instance, `True` and `False` are distinguished by plugging them in the context $C := \texttt{case}_{Bool}\ [\cdot]\ (\texttt{True} \to \texttt{True})\ (\texttt{False} \to \texttt{Bot}_{Bool})$, since $C[\texttt{True}]$ converges, but $C[\texttt{False}]$ diverges.

Example 2.9 Note that in call-by-value calculi there is a difference between looking for termination in all contexts vs. termination in closing contexts.
The $\leq_{\mathcal{P},T}$-relation defined in terms of 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 2.10 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 $s \sim_{\forall\mathcal{P},T} t$ holds.

A helpful tool to prove the correctness of program transformation is a so-called CIU-Theorem (for other calculi see e.g. [34,18]). The CIU-Theorem says that it is sufficient to take only closed reduction contexts and closing value substitutions into account, in order to show (local) contextual equivalence. Moreover, we can strengthen the proposition by using only those reduction contexts and value substitutions which are *F-free*, i.e. they do not contain any function symbols except for the $\bot$-symbols:

Definition 2.11 (F-free expressions, values, contexts) An *F-free* expression, value, or context is 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 the definition of valid programs (Definition 2.6).

The proof of the CIU-Theorem for local contextual equivalence is fairly standard and it follows the method used in [43] (see also Appendix A).

The F-free CIU-preorder on $\mathcal{P}$-expressions is defined like the local contextual preorder where the tests are restricted to (F-free) reduction contexts and (F-free) closing value substitutions.

Definition 2.12 (CIU-preorder, F-Free) For $\mathcal{P}$-expressions, the *F-free CIU-preorder* $\leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}}$ is defined as follows. For $\mathcal{P}$-expressions $s,t :: T$ the inequation $s \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}} t$ holds, iff for all F-free closing $\mathcal{P}$-value substitutions σ and for all closed F-free $\mathcal{P}$-reduction contexts $R[\cdot_T]$, the implication $R[\sigma(s)]{\downarrow} \implies R[\sigma(t)]{\downarrow}$ is valid.

The CIU-Theorem now states that local contextual preorder and the F-free CIU-preorder coincide. Its proof can be found in Appendix A.

Theorem 2.13 (CIU-Theorem F-free) $\leq_{\mathcal{P},T} \;=\; \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}}$

Using the CIU-Theorem it is easy to show that all reduction rules are locally correct program transformations, i.e.:

Proposition 2.14 *Reductions (beta), (delta), and (case) are locally correct program transformations for $\mathcal{P}$-expressions. I.e., if $s \to t$ by (beta), (delta), or (case), then for all $C \in Ctxt$: $C[s] \to C[t]$ is a locally correct transformation.*

Proof The proof can be found in Proposition A.13.

The CIU-Theorem together with Proposition 2.5 implies that Ω-expressions are (locally) contextually equivalent and that they are least elements w.r.t. the contextual preorder:

Corollary 2.15 *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 $s \sim_{\mathcal{P},\tau} t$.*

2.4 Criteria for Contextual Approximation and Equivalence

A consequence of Theorem 2.13 and local correctness of (beta)-reduction is that is sufficient to take into account closed expressions only, i.e.:

Corollary 2.16 *Let $s, t :: \tau$ be $\mathcal{P}$-expressions. If, and only if for all closing F-free $\mathcal{P}$-value substitutions σ, we have $\sigma(s) \leq_{\mathcal{P},\tau} \sigma(t)$, then $s \leq_{\mathcal{P},\tau} t$.*

Corollary 2.17 *Let $s, t :: \tau$ be $\mathcal{P}$-expressions. 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 This follows from the Theorem 2.13, since reduction of $R[\sigma(s)]$ (respectively $R[\sigma(t)]$) first evaluates the expression $\sigma(s)$ (respectively $\sigma(t)$).

The following proposition provides criteria to show contextual preorder for closed expressions (see Appendix A.4).

Proposition 2.18 *For closed $\mathcal{P}$-expressions s, t of constructed type T: $s \leq_{\mathcal{P},T} t$ iff $s{\Uparrow}$, or $s \xrightarrow{*} c\ v_1 \dots v_n$ and $t \xrightarrow{*} c\ w_1 \dots w_n$ for some constructor c, and $v_i \leq_{\mathcal{P},T_i} w_i$ for $i = 1, \dots, n$.*

For closed $\mathcal{P}$-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.

3 Conservativity of Contextual Equivalence

The goal of this section is to show that local contextual equivalence and (global) contextual equivalence coincide (Main Theorem 3.13), i.e. local contextual equivalence is conservative w.r.t. extending programs. This result will be very helpful when we consider the logical formulas and tautologies. First we show that the F-free CIU-preorder is conservative w.r.t. program extensions, i.e. $s \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}} t \implies s \leq^{\neg F}_{\mathcal{P}',T,\mathsf{ciu}} t$ for any program $\mathcal{P}'$ that extends $\mathcal{P}$. The (local) F-free CIU-Theorem 2.13 then implies that $\sim_{\mathcal{P},T}$ and $\sim_{\forall\mathcal{P},T}$ coincide. The conservativity of the F-free CIU-preorder w.r.t. an extended program will be shown by using an approximation mechanism which removes all non-$\mathcal{P}$-subexpressions and non-$\mathcal{P}$-types of an (F-free) expression s of $\mathcal{P}$-type without changing the convergence behavior of s.

3.1 Normalizing Values

The goal of this section is to show that for programs $\mathcal{P}' \sqsupseteq \mathcal{P}$ any F-free $\mathcal{P}'$-value of $\mathcal{P}$-type can be approximated by a $\mathcal{P}$-value. That is, if v is a $\mathcal{P}'$-value of $\mathcal{P}$-type, then there exists a $\mathcal{P}$-value v' with $v' \leq_{\mathcal{P}',T} v$ (sufficiently close to v). A hurdle is that abstractions of $\mathcal{P}$-type in an extension $\mathcal{P}'$ might not be $\mathcal{P}$-expressions, since they may contain data constructors, types, etc. of $\mathcal{P}'$. We will introduce the VN-reduction to "normalize" values, i.e. the reduction will eliminate all subexpressions of $\mathcal{P}$-typed values which are not $\mathcal{P}$-expressions.

Informally, the normalization is possible, since the value has a $\mathcal{P}$-type and thus only proper subexpressions which are not visible in the result (of evaluation) can have a $\mathcal{P}'$-type. However, the call-by-value evaluation rules are not appropriate for this normalization; for instance, consider the value $\lambda x.((\lambda y.x)\ (\mathtt{case}_{Bool}\ x\ (\mathtt{True} \to \mathtt{False})\ (\mathtt{False} \to \mathtt{True})))$ where the variable y has a $\mathcal{P}'$ type. Call-by-value beta reduction cannot reduce the subexpression $(\lambda y.x)\ (\mathtt{case}_{Bool}\ x\ (\mathtt{True} \to \mathtt{False})\ (\mathtt{False} \to \mathtt{True}))$, since the argument is not a value. Hence, the VN-reduction will use a fully substituting call-by-name reduction, but it will respect the termination property of call-by-value reduction. This is ensured by adding strictness constraints, and that is why we slightly extend the expression syntax and allow seq-expressions $(s;t)$, i.e. the expression syntax, the reduction contexts and the standard reduction is:

$$\begin{array}{ll} s, s_i, t \in \mathit{Expr} & ::= (s;t) \mid \ldots \text{(as before)} \\ R, R_i \in \mathit{RCtxt} & ::= (R;s) \mid \ldots \text{(as before)} \\ \text{(seq)} & (v;s) \to s \text{ if } v \text{ is a value} \end{array}$$

The language with the sequentializing construct is isomorphic to the language without the construct, since any expression $(s;t)$ can be encoded as the application $(\lambda\_.s)\ t$ (moreover, $(s;t) \sim_{\mathcal{P},T} ((\lambda\_.s)\ t)$). For the standard reduction, a reduction sequence for an extended expression corresponds after the encoding to a reduction sequence for the encoded expression where every (seq)-reduction is replaced by a (beta)-reduction. This shows that convergence w.r.t. this encoding is unchanged, and thus contextual equivalence is also unchanged, and since (beta)-reduction is locally correct, also the (seq)-reduction is locally correct. We will use all the results of the previous section (especially the F-free CIU-theorem) also for the language with seq-expressions, since the isomorphism allows us to transfer all these results.

We also permit the symbol `Bot`, labeled with a type, for Ω-expressions.

3.1.1 The VN-Reduction

Definition 3.1 The set of *VN-reduction rules* (value normalization) is given in Figs. 3, 4 and 5. A VN-reduction $\xrightarrow{VN}$ is defined as $C[s] \xrightarrow{VN} C[t]$ whenever $s \to t$ by a VN-reduction rule where C is any context.

Lemma 3.2 *All VN-reduction rules are (locally) correct.*

Proof Local correctness of the bot-reduction-rules follows from Corollary 2.15 since the rules transform Ω-expressions into Ω-expressions. Local correctness of the rule (seq) follows from local correctness of (beta)-reduction. Let $s \to t$ by a rule in $\{\text{seqc}, \text{seqseq}, \text{seqapp}, \text{caseseq}, \text{caseapp}, \text{casecase}, \text{seqcase}\}$ then clearly every evaluation of $R[\sigma(s)]$ can be transformed into an evaluation of $R[\sigma(t)]$ and vice versa, where additionally the local 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:

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

Fig. 3 Bot-reduction rules (subset of VN-reductions)

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

Fig. 4 Adapted call-by-name-reduction rules (subset of VN-reductions)

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

Fig. 5 Case-Shifting Transformations (subset of VN-reductions)

- $\sigma(t')$ 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 2.15.
- $\sigma(t')$ is not an Ω-expression. Then there exists a value v such that $\sigma(t') \xrightarrow{*} v$. Local 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 2.16 implies $s \sim_{\mathcal{P},T} t$. Local correctness of VNcase can be shown in a similar way. □

Now we show that the VN-reduction is indeed normalizing. However, a requirement is that the reduction terminates:

Lemma 3.3 *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 of v' of non-$\mathcal{P}$-type. Under all those subexpressions we can choose a subexpression s_1 that is of

non-$\mathcal{P}$-type, and that is not in scope of a binder that binds a variable of non-$\mathcal{P}$-type. This is possible, since if there is a subexpression s_1' of non-$\mathcal{P}$-type within such a scope, then we can choose another subexpression 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 $\texttt{case}_T\ t\ (c\ x_1 \dots x_n) \to r \dots$, where T is a $\mathcal{P}'$-type and s_1' is contained in r. In this case we choose t and so on. 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 $s_1 = s_1'\ s_2'\ \dots s_n'$ with $n \geq 2$, such that s_1' is not an application. Obviously, s_1' is also of non-$\mathcal{P}$-type. Now s_1' cannot be a variable, since all bound variables above s_1 have $\mathcal{P}$-type. The expression s_1' can also not be an abstraction, `Bot`, a seq-expression, or a case-expression, since v' is 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 $(\dots (r_1 \dots r_2) \dots s_1)$ 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:

- Due to maximality, s_1 cannot be an argument of a constructor, the body of an abstraction, the second argument in $(r; s_1)$, or the result expression of a case-alternative.
- s_1 cannot be an argument in an application and not in function position as shown above.
- 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.
- 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. □

Example 3.4 VN-reduction is not strongly normalizing even for F-free expressions: Assume there is a type U with $D_U = \{\texttt{Unit}\}$ and a type U' with $D_{U'} = \{\texttt{Fold}\}$ such that: $\mathit{typeOf}(\texttt{Unit}) = U$ and $\mathit{typeOf}(\texttt{Fold}) = (U' \to U) \to U'$. Then it is possible to define a typeable fixpoint combinator $y :: (U \to U) \to U$ as $y := \lambda f.(\lambda x.f\ (\mathit{unf}\ x\ x))\ (\texttt{Fold}\ (\lambda x.f\ (\mathit{unf}\ x\ x)))$ where unf is the expression $\lambda w.\texttt{case}_F\ w\ ((\texttt{Fold}\ y) \to y)$. In particular there is an infinite VN-reduction sequence for the expression $(y\ (\lambda w.w))$: $(y\ \lambda w.w) \xrightarrow{VN,*} (\lambda x.(\mathit{unf}\ x\ x))\ (\texttt{Fold}\ (\lambda x.(\mathit{unf}\ x\ x)))$ and $(\lambda x.(\mathit{unf}\ x\ x))\ (\texttt{Fold}\ \lambda x.(\mathit{unf}\ x\ x)) \xrightarrow{VN,*} (\lambda x.(\mathit{unf}\ x\ x))\ (\texttt{Fold}\ \lambda x.(\mathit{unf}\ x\ x))$, hence the second sequence can be performed infinitely often.

3.1.2 VN standard reduction and termination properties

Since VN-reduction is not strongly normalizing, we use approximation methods. The two essential parts of this approximation are:

1. If the VN-reduction of a subexpression does not terminate, then replace this subexpression by `Bot`.
2. If subexpressions are deep enough (such that they cannot be reached by a successful evaluation), then the subexpressions are replaced by `Bot`.

The second part will require some specific depth measures. For the first part we require a deterministic strategy for the VN-reduction, and thus we will introduce a VN standard reduction, and we also have to show that nontermination of the VN standard reduction also implies nontermination of the usual standard reduction, to show validity of the approximation. This requires to analyze the overlappings between standard reduction and VN standard reduction.

Definition 3.5 Let t be a (perhaps open) F-free expression. A *VN-standard-reduction* of t is defined as follows: Let $t = R[s]$ for a reduction context R such that s is the outermost-leftmost VN-redex according to Figs. 3, 4 and 5. Apply the VN-reduction rule to s 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-reduction, then we denote it as $\xrightarrow{VNNBsr}$.

Note that the above priority rules ensures that the VN-standard-reduction is uniquely defined. In the following lemmas we assume that expressions are F-free, such that we can apply the VN-standard reduction. We come back to arbitrary expressions (i.e. expressions which may contain function symbols) in Lemma 3.11.

In Appendix C we analyze the *VNsr*-reduction and the overlappings between the *VNsr*-reduction and a parallel call-by-value reduction (like the 1-reduction in [3]) which is appropriate for the $\xrightarrow{sr}$-reduction. This enables us to show the following theorem:

Theorem 3.6 *If t has an infinite VNsr-reduction, then for every F-free closing value-substitution σ: $\sigma(t)\Uparrow$, i.e. t is an Ω-expression.*

3.1.3 Approximating the Values

Definition 3.7 Let t be an F-free 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

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

Definition 3.8 (Value construction with depth cut) Let t be an F-free expression, then the value construction with depth cut for t proceeds as follows:

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

Lemma 3.9 *Let $\mathcal{P}'$ be an extension of the program $\mathcal{P}$. Given an F-free $\mathcal{P}'$-value v of $\mathcal{P}$-type, the construction $ValueConstr_n(v)$ results in an F-free $\mathcal{P}$-value v' with $v' \leq_{\mathcal{P}',T} v$.*

Proof The (mathematical) construction terminates and results in a value. Lemma 3.3 shows that the result is a $\mathcal{P}$-value. By construction, $v' \leq_{\mathcal{P}',T} v$ holds, since the *VN*-reductions are correct (Lemma 3.2), the replaced nonterminating subexpressions are Ω-terms (Theorem 3.6), and Ω-expressions are least elements w.r.t. contextual preorder (Proposition 2.5) □

In Appendix C we prove the following lemma, where the key argument in the proof is to show that `Bot`-symbols inserted in $cl\text{–}depth\ n+1$ can only appear at the top of the expression after more than n standard reductions.

Lemma 3.10 *Let s be an F-free expression such that $s \xrightarrow{sr,n} v$ where v is a value. Let s' be an expression constructed from s where (some) subexpressions t are replaced by $ValueConstr_n(t)$. Then $s'{\downarrow}$.*

3.2 The Main Theorem

Using the value construction of the last section we prove in this section our main result that local contextual equivalence coincides with (global) contextual equivalence. We first show conservativity of the F-free CIU-preorder:

Lemma 3.11 *Let s, t be expressions, such that $s \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}} t$ holds. Then also $s \leq^{\neg F}_{\mathcal{P}',T,\mathsf{ciu}} t$ for any program $\mathcal{P}' \sqsupseteq \mathcal{P}$ holds.*

Proof Let σ' be an F-free closing $\mathcal{P}'$-value substitution and R' be an F-free 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 evaluation 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 let $v_i := \mathit{ValueConstr}_n(v'_i)$, and let $\sigma := \{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m\}$. Also let $r := \mathit{ValueConstr}_n(r')$. Then with $R[\cdot] := r\ [\cdot]$, we have $R[\sigma(s)]\downarrow$ by Lemma 3.10. By the assumption $s \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}} t$, 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 $R''[\sigma'(t)]\downarrow$. □

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. The F-free CIU-Theorem 2.13 and Lemma 3.11 imply:

Theorem 3.12 (CIU-Theorem F-free and global) *Let $\mathcal{P}'$ be an extension of $\mathcal{P}$. For $\mathcal{P}$-expressions $s, t :: T$ the inequation $s \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}} t$ holds if, and only if $s \leq_{\mathcal{P}',T} t$ holds.*

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

Main Theorem 3.13 (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 $s \leq_{\forall\mathcal{P},T} t$.*
Hence, also $s \sim_{\mathcal{P},T} t$ iff $s \sim_{\forall\mathcal{P},T} t$.

Proof This follows from Theorem 3.12 and Theorem 2.13. □

The Main Theorem is the foundation of several conservativity under extensions results for classes of formulas (Theorems 5.2, 5.8, and 5.12).

3.3 Correctness of Program Transformations

Main Theorem 3.13 implies that the program transformations shown already locally correct are correct in general:

Theorem 3.14 *The transformations (beta), (delta), and (case), i.e. the call-by-value reduction rules, and the transformations in Figs. 3, 4 and 5 are correct program transformations in $\mathcal{P}$.*

Proof Local correctness of the transformations was shown in Proposition 2.14 and in Lemma 3.2. Thus, Main Theorem 3.13 implies global correctness. □

A direct consequence of Main Theorem 3.13 and Corollary 2.15 is that all Ω-expressions are in a single equivalence class of the global equivalence, i.e.:

Corollary 3.15 *Let $s, t :: T$ be expressions, such that s is an Ω-expression. Then $s \leq_{\forall\mathcal{P},T} t$ holds. If t is also an Ω-expression, then $s \sim_{\forall\mathcal{P},T} t$.*

The following propositions show a form of extensionality:

Proposition 3.16 *For all data constructors c_i and values v_i, w_i:*

$$(c\ v_1\ \ldots\ v_{ar(c_i)}) \sim_{\forall\mathcal{P},T} (c\ w_1\ \ldots\ w_{ar(c_i)}) \iff \forall i : v_i \sim_{\forall\mathcal{P},T} w_i$$

Proof One direction holds, since $\sim_{\forall\mathcal{P},T}$ is a congruence. For the other direction assume that the equation $(c\ v_1\ \ldots\ v_{ar(c_i)}) \sim_{\forall\mathcal{P},T} (c\ w_1\ \ldots\ w_{ar(c_i)})$ holds. Consider the context $C = \texttt{case}\ [\cdot]\ \ldots\ ((c\ y_1\ \ldots\ y_{ar(c_i)})\ \texttt{->}\ y_i)\ \ldots$. Correctness of the reduction rule (case) implies $w_i \sim_{\forall\mathcal{P},T'} C[(c\ w_1\ \ldots\ w_{ar(c_i)})] \sim_{\forall\mathcal{P},T'} C[(c\ v_1\ \ldots\ v_{ar(c_i)})] \sim_{\forall\mathcal{P},T'} v_i$. □

Note that $(\texttt{Cons}\ \bot\ \texttt{Nil}) \sim_{\forall\mathcal{P},T} \bot$, since constructors are strict, and reduction is correct, hence the proposition cannot be extended to non-values.

Proposition 3.17 *For all expressions s, t of the same type T: $\lambda x.s \sim_{\forall\mathcal{P},T'} \lambda x.t \iff s \sim_{\forall\mathcal{P},T} t$*

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

Proposition 3.18 *Let $\mathcal{P}$ be a program. For all $\mathcal{P}$-abstractions $\lambda x.s$, $\lambda x.t$ of the same type $T_1 \to T_2$ the following statements are equivalent:*

- $\lambda x.s \sim_{\mathcal{P},T} \lambda x.t$
- $\lambda x.s \sim_{\forall\mathcal{P},T} \lambda x.t$
- *For all $\mathcal{P}$-values v of type T_1: $s[v/x] \sim_{\forall\mathcal{P},T_2} t[v/x]$.*
- *For all $\mathcal{P}$-values v of type T_1: $s[v/x] \sim_{\mathcal{P},T_2} t[v/x]$.*

Proof This follows from Theorem 3.13 and Proposition 2.18. □

4 The Logic *LMF*

In this section we introduce the logic *LMF* (logic over monomorphic formulas of functional programs), as a class of logics $LMF(\mathcal{P})$ and $LMF(\forall\mathcal{P})$. Therefore we define the syntax of (monomorphic) formulas, and the interpretation of formulas. Then we introduce the notion of valid formulas and of tautologies. While $LMF(\mathcal{P})$ only interprets a formula w.r.t. a fixed program $\mathcal{P}$, the logic $LMF(\forall\mathcal{P})$ takes all possible extensions of the program $\mathcal{P}$ into account. From the semantics perspective only the logic $LMF(\forall\mathcal{P})$ is interesting, but from the view of automated deduction the "local" logic $LMF(\mathcal{P})$ is more useful, since it is like a first-order logic. Hence, in the subsequent section we will look for

$$
\begin{array}{lll}
I(\mathsf{tt}) = \mathsf{tt} & I(\mathsf{ff}) = \mathsf{ff} & \\
I(A \wedge B) = I(A) \wedge I(B) & I(A \vee B) = I(A) \vee I(B) & I(\neg A) = \neg I(A) \\
I(s = t) = \mathsf{tt}, \text{if } s \sim_{\mathcal{P},\tau} t \quad \text{for } s,t :: \tau & & \\
I(s = t) = \mathsf{ff}, \text{if } s \not\sim_{\mathcal{P},\tau} t \quad \text{for } s,t :: \tau & &
\end{array}
$$

$$
I(\forall x :: \tau.F) = \begin{cases} \mathsf{tt}, \text{ if for all } a \in \mathcal{M}_{\mathcal{P},\tau} : I(F[a/x]) = \mathsf{tt} \\ \mathsf{ff}, \text{ otherwise} \end{cases}
$$

$$
I(\exists x :: \tau.F) = \begin{cases} \mathsf{tt}, \text{ if for some } a \in \mathcal{M}_{\mathcal{P},\tau} : I(F[a/x]) = \mathsf{tt} \\ \mathsf{ff}, \text{ otherwise} \end{cases}
$$

Fig. 6 The interpretation function $I_{LMF(\mathcal{P})}$

cases where proving or disproving properties inside the logic $LMF(\mathcal{P})$ suffices to transfer the properties to the logic $LMF(\forall\mathcal{P})$.

We use tt and ff for the *logical* truth values, which are different from the constants `True`, `False` of type `Bool` in programs.

Let $\mathcal{P}$ be a program. *Atoms* A and *formulas* F are given by the grammars:

$$
\begin{array}{lll}
A ::= \mathsf{tt} \mid \mathsf{ff} \mid (s \ = \ t) & & \text{where } s,t \text{ are } \mathcal{P}\text{-expressions of} \\
& & \text{the same monomorphic type.} \\
F ::= A \mid F \vee F \mid F \wedge F \mid \neg F & & \\
\quad \mid \forall x :: T.F \mid \exists x :: T.F & & \text{where } T \text{ is a monomorphic } \mathcal{P}\text{-type.}
\end{array}
$$

Sometimes we write $F \implies G$ for the formula $\neg(F) \vee G$.

4.1 The Semantics of *LMF*

In the following definition we introduce the semantics of $LMF(\mathcal{P})$. It uses the set $\mathcal{M}_{\mathcal{P},T}$, which is the set of all closed $\mathcal{P}$-values of type T. Note that our definition of valid programs implies that for every monomorphic type T there is a closed value of this type and thus $\mathcal{M}_{\mathcal{P}',T} \neq \emptyset$.

Definition 4.1 (Semantics of $LMF(\mathcal{P})$) Let $\mathcal{P}$ be a program. The interpretation function $I_{LMF(\mathcal{P})}$ (often written without the suffix) of closed monomorphic $\mathcal{P}$-formulas is defined in Fig. 6. A closed monomorphic $\mathcal{P}$-formula F is $\mathcal{P}$-*valid* iff $I(F) = \mathsf{tt}$, which we write as $LMF(\mathcal{P}) \models F$.

A closed monomorphic $\mathcal{P}$-formula F is a $\mathcal{P}$-*tautology* iff it is $\mathcal{P}'$-valid for all extensions $\mathcal{P}'$ of program $\mathcal{P}$, written as $LMF(\forall\mathcal{P}) \models F$.

Example 4.2 Let $\mathcal{P}$ be the program with 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. Then the formula $\forall x :: \texttt{Nat}.\exists y :: \texttt{Nat}.x + (\texttt{Succ}\ 0) = y$ is a $\mathcal{P}$-tautology. The formula $\exists x :: \texttt{Nat}.(\texttt{pred}\ 0) = x$ is invalid for every program $\mathcal{P}' \sqsupseteq \mathcal{P}$ (and thus also 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 $\mathcal{P}$-tautology.

The definition of tautologies implies that the logic is "monotonic" 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. A first consequence of the semantics is that the (local) logics $LMF(\mathcal{P})$ are complete, which is obvious since the interpretation is a deterministic function:

Proposition 4.3 *Let $\mathcal{P}$ be a program and F be a closed $\mathcal{P}$-formula. Then F is $\mathcal{P}$-valid iff $\neg(F)$ is not $\mathcal{P}$-valid.*

Completeness does not hold w.r.t. $LMF(\forall\mathcal{P})$, since there exist programs $\mathcal{P}' \sqsupseteq \mathcal{P}$ and a $\mathcal{P}$-formula F such that $LMF(\mathcal{P}') \models F$, but $LMF(\mathcal{P}) \models \neg F$. Hence neither $LMF(\forall P) \models F$ nor $LMF(\forall P) \models \neg F$ holds (see Theorem 5.20).

An automated verification system cannot scan all program extensions $\mathcal{P}' \sqsupseteq \mathcal{P}$ and prove that F is $\mathcal{P}'$-valid in order to prove that F is $\mathcal{P}$-tautology. Thus, we propose several automatable approaches for proving that F is a $\mathcal{P}$-tautology; (i) The first method is to exhibit (syntactic) classes of monomorphic formulas where $\mathcal{P}$-validity already implies the $\mathcal{P}$-tautology property. For these formulas an automated proof of $\mathcal{P}$-validity, i.e. reasoning only in $LMF(\mathcal{P})$, is sufficient. (ii) The second method is to modify the formula by adding definedness-tests of the occurring expressions, and (iii) the third method is to show that proving $\mathcal{P}$-validity, but using only a selected set of reasoning principles already proves the $\mathcal{P}$-tautology property. Among these are almost all logical rules as well as correct evaluation rules of expressions.

Definition 4.4 Let $\mathcal{P}$ be a program. A class $\mathcal{C}$ of formulas w.r.t $\mathcal{P}$ is *conservative under extensions*, iff for every formula $F \in \mathcal{C}$, if $LMF(\mathcal{P}) \models F$, then also $LMF(\forall\mathcal{P}) \models F$.

Although in Section 5.2 we show that conservativity of extensions does not hold in general, in the next section we will look for several classes of formulas and show their conservativity under extensions.

5 Proof Methods for Tautologies

In this section we exhibit several classes of (closed) formulas that are conservative under extension. But we also prove that this property does not hold in general. Finally, we provide an induction scheme and present several correct proof rules.

5.1 Conservativity of some Classes of Formulas

Since local and global equivalences coincide (see the Main Theorem 3.13), it is promising to look for classes of formulas where it is sufficient to test the values $\mathcal{M}_{\mathcal{P},T}$ instead of all the values of $\mathcal{M}_{\mathcal{P}',T}$ for every $\mathcal{P}' \sqsupseteq \mathcal{P}$.

5.1.1 Quantifier-Free Formulas

A first case of a class that is conservative under extensions is the class of *MQF-formulas* where a MQF-formula is any closed formula F which is a monomorphic and quantifier-free $\mathcal{P}$-formula for some program $\mathcal{P}$.

Theorem 5.1 *The class of MQF-formulas is conservative under extensions.*

Proof If the MQF-formula is an atom $s = t$, then $I_{LMF(\mathcal{P}')}(s = t) = I_{LMF(\mathcal{P}')}(s = t)$ for any $\mathcal{P}' \sqsupseteq \mathcal{P}$, since local correctness and correctness of equations coincide (Main Theorem 3.13). For complex MQF-formulas the equivalence follows by induction over the structure of the quantifier-free formula. □

5.1.2 Universally Quantified Equations

We now consider specific universally quantified formulas. An *AEQ-formula* is any closed $\mathcal{P}$-formula F of the form $\forall x_1 :: T_1, \ldots, x_n :: T_n \;.\; s = t$

Theorem 5.2 *The class of AEQ-formulas is conservative under extensions.*

Proof 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 AEQ-formula. Then for all $\mathcal{P}' \sqsupseteq \mathcal{P}$, the formula F is also $\mathcal{P}'$-valid, i.e., the formula is a $\mathcal{P}$-tautology: This follows since 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 3.13. □

Examples for such tautologies are provided by the correct program transformations (seen as equations) that we already exhibited in Propositions 2.14 and 2.18. Other examples are well-known laws for computing with lists, e.g. associativity of append, the self inverse law of the reverse function, asf.

5.1.3 Termination Proof Formulas

A universally quantified special negated equation claiming the termination of the function $\lambda x_1 \ldots x_n.s$ is also conservative under extensions. A closed $\mathcal{P}$-formula of the form $\forall x_1 :: T_1, \ldots, x_n :: T_n \;.\; \neg(s = \bot)$ where $FV(s) = \{x_1, \ldots, x_n\}$ is called a *termination-proof formula (TP-formula, for short).*

Theorem 5.3 *The class of TP-formulas is conservative under extensions.*

Proof Let $\mathcal{P}$ be a program and $F := \forall x_1 :: T_1, \ldots, x_n :: T_n \;.\; \neg(s = \bot)$ be a $\mathcal{P}$-valid formula. We have to show that for all $\mathcal{P}' \sqsupseteq \mathcal{P}$, the formula F is also $\mathcal{P}'$-valid, i.e., the formula is a $\mathcal{P}$-tautology. Assume the claim is false, i.e. the formula $F := \forall x_1 :: T_1, \ldots, x_n :: T_n \;.\; \neg(s = \bot)$ is $\mathcal{P}$-valid, but there exists $\mathcal{P}' \sqsupseteq \mathcal{P}$ such that F is not $\mathcal{P}'$-valid. Then there exist $\mathcal{P}'$-values $v_i' :: T_i$ for $i = 1, \ldots, n$ such that $s[v_1'/x_1, \ldots, v_n'/x_n] \sim_{\mathcal{P}',T} \bot$ where T is the type of s. We use the value construction $\mathit{ValueConstr}_n(\cdot)$ to construct a $\mathcal{P}$-value v_i for every value v_i' (see Definition 3.8). Lemma 3.9 shows

that $v_i \leq_{\mathcal{P}',T_i} v_i'$ for every i. Since $\leq_{\mathcal{P}',T_i}$ are precongruences, this implies $s[v_1/x_1,\ldots,v_n/x_n] \leq_{\mathcal{P}',T} s[v_1'/x_1,\ldots,v_n'/x_n]$. Since $\bot$ is a least element w.r.t. $\leq_{\mathcal{P}',T}$, we also have $s[v_1/x_1,\ldots,v_n/x_n] \sim_{\mathcal{P}',T} \bot$. Main Theorem 3.13 now implies $s[v_1/x_1,\ldots,v_n/x_n] \sim_{\mathcal{P},T} \bot$ which contradicts $\mathcal{P}$-validity of F. Hence our assumption was wrong and the claim holds.

Definition 5.4 Let $\mathcal{P}$ be a program, f be a function or closed abstraction, and let $T = T_1 \to \ldots \to T_n \to T_{n+1}$ be a monomorphic type of f where T_{n+1} is a constructed type. Then f is $(\mathcal{P},T)$-*terminating*, if $(f\ v_1 \ldots v_n)\!\downarrow$ (i.e. terminates) for all closed $\mathcal{P}$-values $v_i :: T_i$, $i = 1,\ldots n$.

$(\mathcal{P},T)$-termination of f is equivalent to $\mathcal{P}$-validity of the (closed) TP-formula $\forall x_1 :: T_1,\ldots,x_n :: T_n.\neg(f\ x_1 \ldots x_n = \bot)$. In this terminology, addition and multiplication of Peano-numbers are terminating, whereas `tail`, `head`, and `map` on lists are nonterminating. Theorem 5.3 implies that the following holds:

Proposition 5.5 *Termination of a function at a monomorphic type T is conservative under extensions.*

5.1.4 Datatype Quantified Formulas

We now look for values without higher-order subexpressions.

Definition 5.6 A type T is a *DT-type*, if every closed value of type T is only built from data constructors. The class of *DT-formulas* consists of all closed formulas F where all quantified variables have a DT-type.

Examples of DT-types are Peano-numbers, Booleans, lists of Peano-numbers, lists of lists of Booleans. On the contrary, lists of functions of type $\texttt{Nat} \to \texttt{Nat}$ are not DT-types.

Lemma 5.7 *Let $\mathcal{P}'$ be an extension of $\mathcal{P}$. Let T be a $\mathcal{P}$-type that is also a DT-type. Then every $\mathcal{P}'$-value $v :: T$ is also a $\mathcal{P}$-value. Moreover, $\mathcal{M}_{\mathcal{P},T} = \mathcal{M}_{\mathcal{P}',T}$.*

Theorem 5.8 *The class of DT-formulas is conservative under extensions.*

Proof By definition the sets $\mathcal{M}_{\mathcal{P},T}$ do not change for DT-types T if the program $\mathcal{P}$ is extended (Lemma 5.7). Theorem 3.13 implies that all closed $\sim$-equalities are conservative under extension. Hence the interpretation function for any *DT*-formula does not change under extension. □

5.1.5 Formulas with Terminating Expressions

In this section we show that so-called ATD-formulas are conservative under extension. In formulas F, the expressions s,t occurring in equations $s = t$ in F are called *top-expressions* of F. ATD-formulas are closed universally quantified formulas $F = \forall \ldots . F' \implies F''$, where all top-expressions of equations $s = t$

are of DT-type, in F' all those top-expressions terminate, and F' is the condition that guarantees that all top-expressions in F'' terminate. An example for an ATD-formula is $\forall x :: \texttt{Nat}.\neg(x = 0) \implies (div\ x\ x) = (S\ 0)$.

ATD-formulas provide a link to proof methods that deal only with DT-types (i.e. positive recursion in data constructors) and expect all functions to terminate. On the other hand, in the special case of universally quantified formulas, the obtained theorem can be seen as splitting the proof obligations for $\mathcal{P}$-tautology into a proof of termination (or termination under certain conditions) of the involved functions (or expressions) and a proof of $\mathcal{P}$-validity.

ATD-formulas are in the scope of other logics for verification like the `VeriFun`-system, where termination of functions is an a priori requirement. But note that our termination notion means termination with a value, whereas under a loose specification regime certain partial functions are viewed as terminating. For example `tail` is non-terminating (due to our definition), whereas it is seen as terminating under loose specification.

We will also show that the tautology-property is invariant if we switch to a loose specification approach for the semantics of partial functions.

Proposition 5.9 *Let $\mathcal{P}$ be a program that contains the Boolean type and let $F = \forall x_1 :: T_1 \ldots x_n :: T_n.F'$ be a closed $\mathcal{P}$-formula where F' is quantifier-free, and such that all equations are of DT-type. Let $\mathcal{P}'$ be $\mathcal{P}$ extended by a finite set of functions. Then F is $\mathcal{P}$-valid iff it is $\mathcal{P}'$-valid.*

For the proof see Appendix D.

In the following we assume that the program $\mathcal{P}$ already contains monomorphic functions $\texttt{eq}_T$ for all ground DT-types: For a ground DT-type T the function $\texttt{eq}_T$ is a binary function *computing* the equality of two *terminating* expressions of type T, which can easily be programmed using case-expressions and recursion. This extension leaves the tautologies invariant by Proposition 5.9.

Note that in general there are infinitely many such functions, but for a given formula F finitely many are sufficient.

Definition 5.10 Let $\mathcal{P}$ be a program that contains the Boolean type and let $F = \forall x_1 :: T_1 \ldots x_n :: T_n.F' \implies F''$ be a closed $\mathcal{P}$-formula where F', F'' are quantifier-free. Assume the following holds:

- All top-expressions of F are of DT-type.
- All top-expressions of F' are terminating.
- For all $\mathcal{P}$-values $v_1, \ldots, v_n$: If $F'[v_1/x_1, \ldots, v_n/x_n]$ is $\mathcal{P}$-valid, then all top-expressions of $F''[v_1/x_1, \ldots, v_n/x_n]$ are terminating.

Then F is called an *ATD-formula.*

Note that quantification may also be over non-DT-types.

Lemma 5.11 *Let $\mathcal{P}$ be a program that contains the Boolean type and also sufficiently many functions $\texttt{eq}_T$ for DT-types T. Let F be an open quantifier-free formula with $FV(F) = \{x_1 :: T_1, \ldots, x_n :: T_n\}$ such that all top-expressions in equations are of a DT-type. Then there is a closed $\mathcal{P}$-abstraction $f :: T_1 \to$*

$\ldots \to T_n \to$ `Bool` *such that the following holds: For every* $\mathcal{P}' \sqsupseteq \mathcal{P}$*, and all* $\mathcal{P}'$*-values* $v_1 :: T_1, \ldots v_n :: T_n$ *such that all top-expressions in* $F[v_1/x_1, \ldots, v_n/x_n]$ *terminate:* $f\ v_1\ \ldots\ v_n$ *results in* `True`*, iff* $F[v_1/x_1, \ldots, v_n/x_n]$ *is* $\mathcal{P}'$*-valid.*

Proof The abstraction f can be constructed using case-expressions over the Boolean type implementing, logical and, logical or, and logical not, and computing the value of equations $s_1[v_1/x_1 \ldots v_n/x_n] = s_2[v_1/x_1 \ldots v_n/x_n]$ of type T by $\mathtt{eq}_T\ s_1[v_1/x_1 \ldots v_n/x_n]\ s_2[v_1/x_1 \ldots v_n/x_n]$. Note that the termination of the top-expressions guarantees that the $\mathtt{eq}_T$-expression terminate and result in either `True` or `False`, since top-expressions have a DT-type.

Let us call the function f from above the *evaluation function* for F.

Theorem 5.12 *The class of ATD-formulas is conservative under extension.*

Proof Let $\mathcal{P}_0$ be a program and let $F = \forall x_1 :: T_1, \ldots, x_n :: T_n.F' \implies F''$ be an ATD-formula. Proposition 5.9 shows that w.l.o.g. we can use an extension $\mathcal{P}$ which contains all functions $\mathtt{eq}_T$ for all DT-types T of the top-expressions of equations.
Assume that F is $\mathcal{P}$-valid. We have to show that F is also $\mathcal{P}'$-valid. Lemma 5.11 shows that there exist abstractions f', f'' such that: f' is an evaluation function for F' and f'' is an evaluation function for F''. Now consider the abstraction $g := \lambda x_1, \ldots x_n.(\mathtt{if}\ f'\ x_1 \ldots x_n\ \mathtt{then}\ f''\ x_1 \ldots x_n\ \mathtt{else}\ \mathtt{True})$. Then for all $\mathcal{P}'$-values $v_1, \ldots, v_n$: $g\ v_1\ \ldots\ v_n$ evaluates to `True`, iff $(F' \implies F'')[v_1/x_1, \ldots, v_n/x_n]$ is $\mathcal{P}'$-valid.

The formula $G := \forall x_1, \ldots x_n.g\ x_1\ \ldots\ x_n = \mathtt{True}$ is $\mathcal{P}$-valid, since g terminates for all $\mathcal{P}$-values $v_1, \ldots, v_n$. Theorem 5.2 implies that G is also $\mathcal{P}'$-valid, and since g corresponds exactly to F by Proposition 5.9, we now derive that F is also $\mathcal{P}'$-valid. We conclude that F is $\mathcal{P}$-tautology.
Thus the class of ATD-formulas is conservative. □

If we allow an iterated implication and then formulas of the form $\forall x_1, \ldots x_n.F_1 \implies (\ldots(F_2 \implies \ldots \implies F_m)\ldots)$ then we obtain a slightly more general formulas with the same properties as ATD-formulas.

5.1.6 The Relation to Loose Specifications.

In this section we clarify the relation to logics that only allow terminating functions and employ loose specifications to reason about partial functions. Since the loose specification approach has a certain popularity and is used as a foundation of several logic verification systems [15,19,21,23,42], the connection to our logic provides useful information for these systems. The article [42] based on loose specifications appears to restrict the formulas of interest already in a very similar way as the ATD-formulas.

In this section we show that ATD-formulas are *LMF*-tautologies iff they are loose specification-tautologies. The idea to model loose specification is

to replace the bot-alternatives in `case`-expressions by arbitrary closed values, such that there are no undefined alternatives. As semantics we accept a formula as valid under loose specification if it is valid for all such possibilities. An example is the function `tail`, which we may modify as $\texttt{tail}^* = \lambda xs.\texttt{case}\ xs\ (\texttt{Nil} \to \texttt{Nil})\ ((\texttt{Cons}\ x\ xs) \to xs)$, which generates a terminating function of the same polymorphic type, with the same results for all arguments of interest.

However, since we allow polymorphic functions, we have to be careful. For example consider the type $(\texttt{Either}\ a\ b)$ with constructors $\texttt{Right} :: (b \to \texttt{Either}\ a\ b)$ and $\texttt{Left} :: (a \to \texttt{Either}\ a\ b)$. Then there is no closed value of polymorphic type $\texttt{Either}\ a\ b$.

So we prove the claim only for programs where at least one loose specification modification exists, in particular for all programs with only monomorphic functions.

Definition 5.13 Given a program $\mathcal{P}$. Then $\mathcal{P}^*$ is a *loose specification modification* (LSM) of $\mathcal{P}$, if for every $\mathcal{P}$-function f, all the case alternatives in the function bodies that are $\bot$ are replaced by closed $\mathcal{P}^*$-values, resulting in a function f^*, provided that the types of the functions f and f^* are the same. Also, all occurrences of functions in the body are changed from g to g^*. For a formula F, the formula F^* is the formula where all occurrences of functions f are modified into f^*. If the program $\mathcal{P}$ has at least one LSM, then we define the following:
We say a (closed) $\mathcal{P}$-formula F is *loosely $\mathcal{P}$-valid* if for all LSMs $\mathcal{P}^*$ of $\mathcal{P}$, the formula F^* is $\mathcal{P}^*$-valid. We say a (closed) $\mathcal{P}$-formula F is a *loose $\mathcal{P}$-tautology*, iff it is loosely $\mathcal{P}'$-valid for every extension $\mathcal{P}' \sqsupseteq \mathcal{P}$, which has at least one LSM.

For monomorphic functions, such a modification is always possible. Also for example, for a polymorphically typed `tail` a modification exists. If the program has only monomorphic functions, then at least one LSM of $\mathcal{P}$ exists, since for every monomorphic type there is at least one closed value.

Theorem 5.14 *Let $\mathcal{P}$ be a program that contains the Boolean type and sufficiently many $\texttt{eq}_T$-functions, and at least one LSM $\mathcal{P}^*$ of $\mathcal{P}$ exists, and let F be a (closed) ATD-formula w.r.t. $\mathcal{P}$. Then F is $\mathcal{P}$-valid iff F is loosely $\mathcal{P}$-valid. Consequently, F is a $\mathcal{P}$-tautology iff F is a loose $\mathcal{P}$-tautology.*

The proof is in Appendix D.

Note that the assumption that $F = \forall x_1 :: T_1, \ldots, x_n :: T_n.F' \implies F''$ is an ATD-formula means that the termination of the top expressions of F' has to be verified without the LSM-modification. Thus for example, the formula $\forall xs, ys.xs = ys \implies \texttt{tail}\ xs = \texttt{tail}\ ys$ is not an ATD-formula: xs, ys terminate, but $\texttt{tail}(xs)$ is not terminating according to our definition. However, as already remarked above, under loose specifications `tail` may be viewed as terminating.

Remark 5.15 If an equation contains nonterminating top expressions, then the logical truth of formulas might change after a loose specification modification.

- The (non-ATD-) formula $(tail\ \texttt{Nil}) = (tail\ (tail\ \texttt{Nil})) \implies (tail\ \texttt{Nil}) = \texttt{Nil}$ is a tautology under any of the loose specification modifications, (there exists at least one), using a case analysis over the possibilities. However, it is not an LMF-tautology, since $LMF \models (tail\ \texttt{Nil}) = (tail\ (tail\ \texttt{Nil}))$, but $LMF \models (tail\ \texttt{Nil}) \neq \texttt{Nil}$.
- Suppose $\texttt{tail}$, $\texttt{tail}'$ are two definitions of the tail-function, then $LMF \models \texttt{tail} = \texttt{tail}'$, whereas $\texttt{tail} = \texttt{tail}'$ is not loosely $\mathcal{P}$-valid.

5.2 Valid Monomorphic Formulas Might not be Tautologies

We show in this subsection that in general, monomorphic $\mathcal{P}$-formulas that are $\mathcal{P}$-valid might not be $\mathcal{P}$-tautologies. The argument in the proof is, roughly speaking, that there are programs which are not expressive enough to express general recursion and thus not Turing-complete, however, these programs become Turing-complete after extending them.

Definition 5.16 Let $\mathcal{P}_{simple}$ be a program that has lists, Booleans, and Peano-numbers as data structures, there is a $\bot$-expression for every type, but no other defined functions (but abstractions are allowed).

We will show that in $\mathcal{P}_{simple}$ it is impossible to encode a universal Turing machine, since in $\mathcal{P}_{simple}$ it is decidable whether an expression terminates. Hence, the formula expressing the existence of a certain universal Turing machine is not valid in $\mathcal{P}_{simple}$. Of course we can extend the program $\mathcal{P}_{simple}$ such that the formula becomes valid in the extended program. This shows that conservativity under extension does in general not hold.

Note that due to recursive data types it is not obvious that every program terminates. For example, in [44] there is an example of a nonterminating program in a recursion-free language like $\mathcal{P}_{simple}$ (with more data types, also containing negative recursion in the constructors).

Lemma 5.17 *It is decidable, whether a $\mathcal{P}_{simple}$-expression terminates.*

Proof The argument is that reduction of $\mathcal{P}_{simple}$-expressions strongly normalizes (i.e. terminates), if we assume that $\texttt{Bot}$s of different types are given as constants. If the result of an evaluation is a value, then we detect termination, and if a redex touches $\texttt{Bot}$, then we detect nontermination. The proof is along standard methods for simply typed languages, but has to be extended for constructors. It can be found for example in [44], where the data types of $\mathcal{P}_{simple}$ fit the restrictions made there. □

The idea of our counterexample is as follows: Quantification ranges over the set of definable functions (as expressions in normal form). Since the language owes its power mainly to recursive function definitions, extending

the program may give rise to genuinely new functions, which can invalidate statements of the form $\neg\exists f.P(f)$. We believe that the addition function on Peano-numbers cannot be defined, which would be a nice counterexample, however, the proof appears to be too complex, since expressiveness of $\mathcal{P}_{simple}$ already includes to form abstractions which may use recursive constructed data like lists and Peano numbers. We can also not built on termination alone, since `Bot` is a permitted expression. Instead we will built upon the property of decidability of termination and show that we can encode a recursive function such that the termination of expressions becomes undecidable.

We extend the program $\mathcal{P}_{simple}$ with a new function 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 $h = \lambda lt, rt, st.r$ 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 the 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 h.

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

$$F_T := \exists g.\forall lt :: (\texttt{List Bool}), rt :: (\texttt{List Bool}), st :: \texttt{Nat}.g\ lt\ rt\ st = r[g/h]$$

Note that in the formula the symbol g (and all occurrences of g in $r[g/h]$) 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 5.18 *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 5.19 *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 5.20 *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 5.18 and 4.3, but not in a certain extension of $\mathcal{P}_{simple}$ according to Proposition 5.19. Thus it cannot be a $\mathcal{P}_{simple}$-tautology. □

Remark 5.21 (Open Questions) It remains open whether every $\mathcal{P}$-valid monomorphic formula of the form $\forall x_1 :: T_1, \ldots, x_n :: T_n.A$, 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.

It is also open how sufficient expressiveness like existence of a (definable) fixpoint function of a program influences the conservativity of $\mathcal{P}$-formulas.

5.3 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 closed $\mathcal{P}$-formula to be proved 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 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 3.13, 5.1, 5.2, 5.3, 5.8, and 5.12, 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 that can be used for proving $\mathcal{P}$-validity as well as for being a $\mathcal{P}$-tautology. Let T be a monomorphic type and let (`List` T) have the two constructors `Nil` :: (`List` T) and `Cons` :: $T \to$ (`List` T) $\to$ (`List` T).

Definition 5.22 (Induction Scheme for Lists) Let $\forall x ::$ (`List` T).F be a closed $\mathcal{P}$-formula. Assume that the following holds:

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

Then the formula $\forall x ::$ (`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.

5.4 Proof Rules for Monomorphic Formulas

In this section we summarize 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 proved to be a $\mathcal{P}$-tautology, and can be formulated in $\mathcal{P}$. The notation $\mathcal{P} \blacktriangleright s \sim t$ means that contextual equality of s and t holds, i.e. $s \sim t$ is a tautology (see Main Theorem 3.13), that s and t can be formulated in $\mathcal{P}$, it is considered as symmetric, and it allows also open expressions s, t. The notation $F[G]$ ($F[s]$, $C[s]$, resp.) means a formula which has the subformula G (subexpression s) at a specific position. Note that G (s, resp.) may also contain free variables. We also use the notation $\mathcal{P} \blacktriangleright s \not\sim t$ with the semantics that for all value substitutions $\sigma : \sigma(s) \not\sim_{\mathcal{P},T} \sigma(t)$. It is also symmetric and free variables are permitted. The following proof rules using $\vdash$ are sound for closed monomorphic $\mathcal{P}$-formulas and the rules using $\blacktriangleright$ for open equations, where we use the representation $\dfrac{\text{Premise}}{\text{Conclusion}}$ for the rules. If the premise is empty, we write only the conclusion. 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 our logic *LMF*.

Equations, Inequations and Formulas

$$\frac{\mathcal{P} \blacktriangleright s \sim t; \quad \mathcal{P} \vdash F[s]}{\mathcal{P} \vdash F[t]} \qquad \frac{\mathcal{P} \blacktriangleright s \sim t; \quad \mathcal{P} \vdash F[\mathsf{tt}]}{\mathcal{P} \vdash F[s = t]} \qquad \frac{\mathcal{P} \blacktriangleright s \not\sim t; \quad \mathcal{P} \vdash F[\mathsf{ff}]}{\mathcal{P} \vdash F[s = t]}$$

$$\frac{\mathcal{P} \blacktriangleright s \sim t}{\mathcal{P} \vdash \forall x_1, \ldots, x_n.s = t} \qquad \frac{\mathcal{P} \blacktriangleright s \not\sim t}{\mathcal{P} \vdash \forall x_1, \ldots, x_n.\neg(s = t)}$$

Equations: $\boxed{\mathcal{P} \blacktriangleright s \sim t, \text{ if } s \sim_{\forall\mathcal{P},T} t}$

$$\frac{\mathcal{P} \blacktriangleright s_1 \sim s_3; \quad \mathcal{P} \blacktriangleright s_3 \sim s_2}{\mathcal{P} \blacktriangleright s_1 \sim s_2} \qquad \frac{\mathcal{P} \blacktriangleright s \sim t}{\mathcal{P} \vdash C[s] \sim C[t]} \qquad \frac{\mathcal{P} \blacktriangleright s \sim t; \quad \mathcal{P} \blacktriangleright C[s] \sim r}{\mathcal{P} \blacktriangleright C[t] \sim r}$$

In the following rules, c, c_i are data constructors, v_i, w_i are values, and r, r_i are Ω-expressions.

$$\frac{\text{For all } i \in \{1, \ldots, n\} : \mathcal{P} \blacktriangleright v_i \sim w_i}{\mathcal{P} \blacktriangleright (c\ v_1\ \ldots\ v_n) \sim (c\ w_1\ \ldots\ w_n)} \qquad \mathcal{P} \blacktriangleright r_1 \sim r_2 \quad \mathcal{P} \blacktriangleright r_1 \sim (c\ \ldots r_2 \ldots)$$

Inequations $\dfrac{\text{For some } i \in \{1, \ldots, n\} : \mathcal{P} \blacktriangleright v_i \not\sim w_i}{\mathcal{P} \blacktriangleright (c\ v_1\ \ldots\ v_n) \not\sim (c\ w_1\ \ldots\ w_n)}$ $\qquad \mathcal{P} \blacktriangleright r \not\sim c\ v_1 \ldots v_n$

$$\mathcal{P} \blacktriangleright c_i\ v_1\ \ldots\ v_n \not\sim c_j\ w_1\ \ldots\ w_m \text{ if } c_i \neq c_j \qquad \mathcal{P} \blacktriangleright r \not\sim \lambda x.s$$

Note that – although the property of being an Ω-term is undecidable in general – several Ω-terms can be recognized by encoding the functions with undefined patterns like `tail` using the constant `Bot` for the error-case and perhaps applying the $\bot$-reduction rules in Fig. 3.

Tautologies.

$$\frac{\forall\mathcal{P} \vdash F}{\mathcal{P} \vdash F} \text{ for any formula } F \qquad \frac{\mathcal{P} \vdash F}{\forall\mathcal{P} \vdash F} \text{ if } F \text{ is an MQF-, AEQ-, TP-, DT-, or, ATD-formula}$$

Case Distinction. 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 $x :: T$.

$$\frac{\text{For all } i \in \{1, \ldots n\} : \mathcal{P} \vdash \forall y_1, \ldots, y_{ar(c_i)}.F[(c_i\ y_1 \ldots y_{ar(c_i)})/x]}{\mathcal{P} \vdash \forall x :: T.F}$$

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:

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

As a (very small) example (without induction) we show a proof derivation for the formula $\forall x :: \tau, xs :: \texttt{List}\ \tau.tail\ (x : xs) = xs$ where *tail* is defined as before, and τ is some monomorphic type.

$$\frac{\dfrac{\mathcal{P} \blacktriangleright tail\ (x : xs) \sim xs,\ \text{since } tail(x : xs) \sim_{\forall\mathcal{P},T} xs}{\mathcal{P} \vdash \forall x :: \tau, xs :: \texttt{List}\ \tau.tail\ (x : xs) = xs}}{\forall\mathcal{P} \vdash \forall x :: \tau, xs :: \texttt{List}\ \tau.tail\ (x : xs) = xs}\ \text{(AEQ formula)}$$

As another example, $\forall xs :: \texttt{List}\ \tau.\neg(xs = \texttt{Nil}) \implies \neg(tail\ xs = \bot)$ can be shown a tautolgy by the following derivation (where the implication is already removed), which uses the case distinction rule and is of the form:

$$\frac{\dfrac{\mathcal{P} \blacktriangleright \texttt{Nil} \sim \texttt{Nil},\ \dfrac{\mathcal{P} \vdash \texttt{tt}}{\mathcal{P} \vdash \texttt{tt} \vee \neg(tail\ \texttt{Nil} = \bot)}}{\mathcal{P} \vdash \texttt{Nil} = \texttt{Nil} \vee \neg(tail\ \texttt{Nil} = \bot)}, \quad \dfrac{\mathcal{P} \blacktriangleright (y : ys) \not\sim \texttt{Nil},\ \dfrac{\dfrac{\mathcal{P} \blacktriangleright tail\ (y : ys) \not\sim \bot}{\mathcal{P} \vdash \forall y :: \tau, ys :: \texttt{List}\ \tau.\neg(tail\ (y : ys) = \bot)}}{\mathcal{P} \vdash \forall y :: \tau, ys :: \texttt{List}\ \tau.\texttt{ff} \vee \neg(tail\ (y : ys) = \bot)}}{\mathcal{P} \vdash \forall y :: \tau, ys :: \texttt{List}\ \tau.(y : ys) = \texttt{Nil} \vee \neg(tail\ (y : ys) = \bot)}}{\mathcal{P} \vdash \forall xs :: \texttt{List}\ \tau.xs = \texttt{Nil} \vee \neg(tail\ (xs) = \bot)}}{\forall\mathcal{P} \vdash \forall xs :: \texttt{List}\ \tau.xs = \texttt{Nil} \vee \neg(tail\ (xs) = \bot)}$$

As a further example consider a program that defines the (nonterminating) function $f = \lambda x.\texttt{if}\ x\ \texttt{then}\ f(x)\ \texttt{else}\ x$. Then a proof derivation for the formula $\forall x :: \texttt{Bool}.\neg((f\ x) = \bot) \implies (f\ x) = \texttt{False}$ is as follows, where $f\ \texttt{True} \sim \bot$ can be proved by detecting a loop in the standard reduction of $f\ \texttt{True}$.

$$\dfrac{\dfrac{\mathcal{P} \blacktriangleright f\ \texttt{False} \sim \texttt{False};\ \dfrac{\mathcal{P} \vdash \texttt{tt}}{\mathcal{P} \vdash (f\ \texttt{False}) = \bot \vee \texttt{tt}}}{\mathcal{P} \vdash (f\ \texttt{False}) = \bot \vee (f\ \texttt{False}) = \texttt{False}}\ ;\ \dfrac{\mathcal{P} \blacktriangleright f\ \texttt{True} \sim \bot;\ \dfrac{\mathcal{P} \vdash \texttt{tt}}{\mathcal{P} \vdash \texttt{tt} \vee (f\ \texttt{True}) = \texttt{False}}}{\mathcal{P} \vdash (f\ \texttt{True}) = \bot \vee (f\ \texttt{True}) = \texttt{False}}}{\dfrac{\mathcal{P} \vdash \forall x :: \texttt{Bool}.(f\ x) = \bot \vee (f\ x) = \texttt{False}}{\forall \mathcal{P} \vdash \forall x :: \texttt{Bool}.(f\ x) = \bot \vee (f\ x) = \texttt{False}}}$$

Of course the chosen function f here is chosen artificially, but the example gives an impression, how the logic *LMF* can deal with partial functions. For instance, one can also show statements about the famous $3n + 1$-algorithm of Collatz, where an apriori termination proof of the algorithm is not required.

5.5 Relation to Induction Provers

We discuss some differences between our semantics and the semantics underlying other logics for induction like `VeriFun` ([47]), and the consequences for the proof rules of an automated system and possible improvements. A system that uses strict functional programming languages, loose specification semantics for undefined patterns and requires function to terminate on all arguments, can be improved by adopting our semantics as follows:

- *Permitting nonterminating functions:* The induction schemes are the same as long as induction is over the structure of DT-values. If expressions are known to terminate to a value for all value-substitutions, then there is no difference. If expressions appear for which termination is not proved yet, then the only precaution is that call-by-name beta- and case-reductions cannot be used for evaluation or as program transformations and instead (VNbeta) and (VNcase) have to be used.
- *Undefined expressions:* For the case of ATD-formulas, i.e. universally quantified formulas $F' \implies F''$, where every top-expression is of DT-type and every-top expression in F' terminates, and if F' is valid, also the top-expressions of equations in F'' terminate, the improvement is to use the Bot-reduction rules in Fig. 3, which are not available under loose specification; the justification is Theorem 5.14. Note that such expressions frequently occur in `VeriFun`-subproofs that try to show preconditions of lemmas. It is now possible to evaluate equations with $\bot$-expressions in F' like (`tail Nil`) = (`tail`(`tail Nil`)) (which is valid), and ease reasoning about formulas that are valid independent of the conditions in F'.

6 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. We will see that our polymorphism-definition serves its purpose as an extension of the logic *LMF*, but is not a true polymorphic logic, since for example quantifiers $\forall T$, where T is a type, are missing. We sketch how to prove a polymorphic tautology directly.

Definition 6.1 *Polymorphic $\mathcal{P}$-formulas* are like monomorphic formulas, where type variables are permitted in the type of the quantified variables, the expressions are polymorphically typed (as in the defining values), and polymorphic expressions are permitted in the formulas, where in equations $s = t$, the expressions s, t must be of the same polymorphic type.

The semantics has to be extended as follows:

Definition 6.2 For a program $\mathcal{P}$ and a polymorphic $\mathcal{P}$-formula F, we say F is *$\mathcal{P}$-valid*, 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 $\mathcal{P}$-valid.

A polymorphic $\mathcal{P}$-formula F is a *(polymorphic) $\mathcal{P}$-tautology*, iff it is $\mathcal{P}'$-valid for all extensions $\mathcal{P}'$ of $\mathcal{P}$.

Now we could start a research effort and analyze polymorphic equations, formulas and tautologies in the same way as for monomorphic formulas. However, there appear to be differences:

Proposition 6.3 *$\mathcal{P}$-validity of a polymorphic formula F in general does not imply its $\mathcal{P}$-tautology property.*

Proof Let $\mathcal{P}$ be a program where the data type `Bool` and Peano-numbers are defined, but no other data types. Then the following formula F is $\mathcal{P}$-valid:

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

which expresses that if there are three different values of a certain type, then there is another value of this type. This is true in $\mathcal{P}$, since it is true for `Bool`, Peano-numbers, 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 $\mathcal{P}$-tautology. □

6.1 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 an example for an induction schemes for polymorphic formulas, which is almost the same as for monomorphic formulas. But note that its correctness depends on the structure of the data type `List`.

Definition 6.4 (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 tautology that can be proved by induction is associativity of the append-function `app` 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:

$$\forall x :: \texttt{List}\ a, y :: \texttt{List}\ a, z :: \texttt{List}\ a.(\texttt{app}\ x\ (\texttt{app}\ y\ z)) = (\texttt{app}\ (\texttt{app}\ x\ y)\ z)$$

The induction proof is standard, so we omit it, but point to the proof details.

- It has to be proved for all $\mathcal{P}' \sqsupseteq \mathcal{P}$, and any $\mathcal{P}'$-type substitution ρ. This means to verify the claim for all values in $\mathcal{M}_{\mathcal{P}',\rho(a)}$. However, since the proof is independent of the type variable a, this can be ignored. The reason is the special form of the constructors for lists.
- Correct proof rules for validity have to be applied like partial evaluation.

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

Definition 6.5 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 tautology, if the induction and the induction measure are "independent" of the type variables and only $\mathcal{P}$-tautologies and 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 6.6 *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 $\mathcal{P}$-valid.*
Issue: is F a $\mathcal{P}$-tautology?

7 Conclusion

We presented a logical framework for proofs of properties of functional programs in a typed strict functional programming language 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`, at least for the class of DT-formulas. Our logic also proposes generalizations.

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 3.4. We thank Peter Schauß for discussions on proof traces of `VeriFun`. We also thank the anonymous referees for remarks that improved the paper.

References

1. ACL2 Website: `http://www.cs.utexas.edu/~moore/acl2` (2011)
2. Agda Website: `http://wiki.portal.chalmers.se/agda` (2011)
3. Barendregt, H.P.: The Lambda Calculus. Its Syntax and Semantics. North-Holland, Amsterdam, New York (1984)
4. Barringer, H., Cheng, J.H., Jones, C.B.: A logic covering undefinedness in program proofs. Acta Inform. **21**, 251–269 (1984)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)
6. Bertot, Y., Komendantsky, V.: Fixed point semantics and partial recursion in Coq. In: S. Antoy, E. Albert (eds.) Proc. PPDP '08, pp. 89–96. ACM, New York, NY, USA (2008)
7. Bove, A., Krauss, A., Sozeau, M.: Partiality and recursion in interactive theorem provers: An overview. Math. Structures Comput. Sci. (2011). To appear, preprint available at `http://www4.informatik.tu-muenchen.de/~krauss/papers/recursion.pdf`
8. Boyer, R.S., Moore, J.S.: Proving theorems about lisp functions. J. ACM **22**(1), 129–144 (1975)
9. Boyer, R.S., Moore, J.S.: A mechanical proof of the unsolvability of the halting problem. J. ACM **31**(3), 441–458 (1984)
10. Cheng, J.H., Jones, C.B.: On the usability of logics which handle partial functions. In: C. Morgan, J.C.P. Woodcock (eds.) 3rd Refinement Workshop, pp. 51–69. Springer (1991)
11. Chiba, Y., Aoto, T., Toyama, Y.: Program transformation by templates based on term rewriting. In: P. Barahona, A.P. Felty (eds.) Proc. PPDP '05, pp. 59–69. ACM (2005)
12. Coq Website: `http://coq.inria.fr/` (2011)
13. Coquand, T., Huet, G.P.: The calculus of constructions. Inform. and Comput. **76**(2/3), 95–120 (1988)
14. Farmer, W.M.: A simple type theory with partial functions and subtypes. Ann. Pure Appl. Logic **64**(3), 211–240 (1993)
15. Farmer, W.M.: Mechanizing the traditional approach to partial functions (1996). Available from `http://imps.mcmaster.ca/wmfarmer/`
16. Farmer, W.M.: Formalizing undefinedness arising in calculus. In: D.A. Basin, M. Rusinowitch (eds.) Proc. IJCAR '04, *Lecture Notes in Comput. Sci.*, vol. 3097, pp. 475–489. Springer (2004)
17. Farmer, W.M., Guttman, J.D., Thayer, F.J.: IMPS: An interactive mathematical proof system. J. Automat. Reason. **11**(2), 213–248 (1993)
18. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoret. Comput. Sci. **103**, 235–271 (1992)
19. Finn, S., Fourman, M.P., Longley, J.: Partial functions in a total setting. J. Automat. Reason. **18**(1), 85–104 (1997)
20. Fitzgerald, J., Jones, C.: The connection between two ways of reasoning about partial functions. Inform. Process. Lett. **107**(3–4), 128–132 (2008)
21. Giesl, J.: Induction proofs with partial functions. J. Automat. Reason. **26**(1), 1–49 (2001)
22. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, New York, NY, USA (1993)

23. Hähnle, R.: Many-valued logic, partiality, and abstraction in formal specification languages. Log. J. IGPL **13**(4), 415–433 (2005)
24. HOL4 Website: `http://hol.sourceforge.net/` (2011)
25. Howe, D.: Equality in lazy computation systems. In: R. Parikh (ed.) LICS '89, pp. 198–203 (1989)
26. Howe, D.: Proving congruence of bisimulation in functional programming languages. Inform. and Comput. **124**(2), 103–112 (1996)
27. Isabelle/HOL Website: `http://www.cl.cam.ac.uk/research/hvg/isabelle/` (2011)
28. Jones, C.B.: Systematic software development using VDM (2. ed.). Prentice Hall International Series in Computer Science. Prentice Hall (1991)
29. Kapur, D., Musser, D.R.: Inductive reasoning with incomplete specifications (preliminary report). In: A. Meyer (ed.) LICS '86, pp. 367–377. IEEE Computer Society (1986)
30. Klein, G., Nipkow, T., Paulson, L.: The archive of formal proofs (2011). `http://afp.sf.net`
31. Krauss, A.: Partial recursive functions in higher-order logic. In: U. Furbach, N. Shankar (eds.) Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, *Lecture Notes in Comput. Sci.*, vol. 4130, pp. 589–603. Springer (2006)
32. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Automat. Reason. **44**(4), 303–336 (2010)
33. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
34. Mason, I., Talcott, C.L.: Equivalence in functional languages with effects. J. Funct. Programming **1**(3), 287–327 (1991)
35. Morris, J.: Lambda-calculus models of programming languages. Ph.D. thesis, MIT (1968)
36. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *Lecture Notes in Comput. Sci.*, vol. 2283. Springer (2002)
37. Pitts, A.M.: Operationally-based theories of program equivalence. In: A.M. Pitts, P. Dybjer (eds.) Semantics and Logics of Computation. Cambridge University Press (1997)
38. Plotkin, G.D.: Call-by-name, call-by-value, and the lambda-calculus. Theoret. Comput. Sci. **1**, 125–159 (1975)
39. Sabel, D., Schmidt-Schauß, M., Harwath, F.: Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In: S. Fischer, E. Maehle, R. Reischuk (eds.) INFORMATIK 2009, *GI Edition - Lecture Notes in Informatics*, vol. 154, pp. 369; 2931–45 (2009)
40. Sands, D., Gustavsson, J., Moran, A.: Lambda calculi and linear speedups. In: T. Mogensen, D. Schmidt, I. Sudborough (eds.) The Essence of Computation 2002, pp. 60–84. Springer (2002)
41. Schieder, B., Broy, M.: Adapting calculational logic to the undefined. Comput. J. **42**(2), 73–81 (1999)
42. Schlosser, A., Walther, C., Gonder, M., Aderhold, M.: Context dependent procedures and computed types in verifun. Electron. Notes Theor. Comput. Sci. **174**(7), 61–78 (2007)
43. Schmidt-Schauß, M., Sabel, D.: On generic context lemmas for higher-order calculi with sharing. Theoret. Comput. Sci. **411**(11-13), 1521 – 1541 (2010)
44. Schmidt-Schauß, M., Sabel, D.: A termination proof of reduction in a simply typed calculus with constructors. Frank report 42, Inst. für Informatik. Goethe-Universität Frankfurt (2010)
45. Schröder, L., Mossakowski, T.: HasCasl: Integrated higher-order specification and program development. Theoret. Comput. Sci. **410**(12-13), 1217–1260 (2009)
46. Scott, D.: A type-theoretical alternative to CUCH, ISWIM, OWHY. Theoret. Comput. Sci. **121**, 411–440 (1993)
47. VeriFun Website: www.inferenzsysteme.informatik.tu-darmstadt.de/verifun/ (2011)
48. Walther, C.: Mathematical induction. In: D.M. Gabbay, C.J. Hogger, J.A. Robinson, J.H. Siekmann (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 2, pp. 127–228. Oxford University Press (1994)
49. Walther, C., Schweitzer, S.: Reasoning about incompletely defined programs. In: G. Sutcliffe, A. Voronkov (eds.) Proc. LPAR '05, *Lecture Notes in Comput. Sci.*, vol. 3835, pp. 427–442 (2005)

A The Local CIU-Theorem

In this section we prove the CIU-Theorem for local contextual equivalence. We assume that $\mathcal{P}$ is a fixed program. In [43] a context lemma was proved for a whole class of program calculi. Unfortunately, this framework does not fit for $\mathcal{P}$-expressions, since it cannot handle the polymorphic typing of functions. Nevertheless, our proof of the CIU-Theorem is similar to the method used in [43] and thus in order to prove the CIU-Theorem for $\mathcal{P}$-expressions, we first prove a context lemma for $\mathcal{P}$-expressions extended with a `let` in Section A.1. The reason is that our proof technique requires a reduction that only substitutes variables at reduction positions. This does not hold for (beta)-reduction (since it substitutes all occurrences of a variable in the body of an abstraction), in the extended language with `let` this substitution is avoided by constructing `let` expressions during reduction. After we have proved the context lemma for the extended language we will transfer it to `let`-free $\mathcal{P}$-expressions by an adequate translation in Section A.2. Finally in Section A.3 we show that we can strengthen the CIU-Theorem by considering only F-free closing value substitutions and F-free reduction contexts, i.e. the F-free CIU-preorder $\leq^{-F}_{\mathcal{P},T,\mathsf{ciu}}$ (Definition 2.12) coincides with the local contextual preorder.

A.1 Context Lemma for a Sharing Extension

For a fixed program $\mathcal{P}$ we define $\mathcal{P}$-`let`-*expressions* (or the `let`-language, for short) by the following grammar.

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

where v is a value, i.e. $v, v_i \in \mathit{Val} ::= x \mid (c\ v_1\ \ldots v_N) \mid \lambda x.s$. $\mathcal{P}$-`let`-expressions extend the language of $\mathcal{P}$-expressions by the `let`-construct which allows to share values. 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.

Instead of defining reduction contexts for $\mathcal{P}$-`let`-expressions by a grammar, we use a label-shifting algorithm to determine the reduction position. The algorithm uses the labels lr and sub. For an expression s the label-shift algorithm starts with s^{lr} and then exhaustively applies the shifting rules shown in Fig. 7 where $\mathsf{sub} \vee \mathsf{lr}$ means either label sub or lr. 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. Note that the label lr is switched into label sub when the label is moved into an application, a `case`-expression, or a constructor application and it can never be switched again into lr. The reason is that we do not want to reduce inside `let`-expressions which are below applications, constructor applications, and `case`-expressions.

It is easy to verify that the label shifting is deterministic and always terminates. The standard reduction rules for `let`-expressions 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. E.g. for the `let`-expression $\mathtt{let}\ x = \mathtt{True}\ \mathtt{in}\ \mathtt{case}\ (x : \mathtt{Nil})\ ((y : ys) \to y)\ (\mathtt{Nil} \to \mathtt{Nil})$ the standard reduction reduces the `case`-expression and thus it results in $\mathtt{let}\ x = \mathtt{True}\ \mathtt{in}\ x$. 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.

The intention of the reduction $\xrightarrow{ls,a}$ is to avoid substitution, i.e. instead of using substituting (beta)- and (case)-reduction now the sharing variants ($\mathrm{beta}_{\mathtt{let}}$) and ($\mathrm{case}_{\mathtt{let}}$) are used, which create new `let`-expressions to share the arguments instead of substituting them. With (lll) we denote the union of the rules (lapp), (lrapp), (lcapp), and (lcase). The rules are used to adjust the nesting of `let`-expressions.

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

Fig. 7 Searching the redex in the let-language

$$
\begin{array}{ll}
(\text{beta}_{\mathtt{let}}) & C[((\lambda x.s)^{\mathsf{sub}}\ v)] \to C[\mathtt{let}\ x = v\ \mathtt{in}\ s] \\
(\text{delta}_{\mathtt{let}}) & C[f^{\mathsf{sub}} :: T] \to C[d_f] \quad \text{if } f = d_f :: T' \text{ for the function symbol } f. \\
 & \text{The reduction is accompanied by a type instantiation } \rho(d_f), \text{ where } \rho(T') = T \\
(\text{case}_{\mathtt{let}}) & C[(\mathtt{case}\ (c\ v_1 \ldots v_n)^{\mathsf{sub}}\ \ldots((c\ y_1 \ldots y_n)\ \mathtt{->}\ s)\ldots)] \\
 & \quad \to C[\mathtt{let}\ y_1 = v_1\ \mathtt{in}\ \ldots \mathtt{let}\ y_n = v_n\ \mathtt{in}\ s] \\
(\text{cp}) & C[\mathtt{let}\ x = v\ \mathtt{in}\ C'[x^{\mathsf{sub}}]] \to C[\mathtt{let}\ x = v\ \mathtt{in}\ C'[v]] \\
(\text{lapp}) & C[((\mathtt{let}\ x = v\ \mathtt{in}\ s)^{\mathsf{sub}}\ t)] \to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (s\ t))] \\
(\text{lrapp}) & C[(v_1\ (\mathtt{let}\ x = v\ \mathtt{in}\ t)^{\mathsf{sub}})] \to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (v_1\ t))] \\
(\text{lcapp}) & C[(c\ v_1 \ldots v_{i-1}\ (\mathtt{let}\ x = v\ \mathtt{in}\ s_i)^{\mathsf{sub}}\ s_{i+1} \ldots s_n)] \\
 & \quad \to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (c\ v_1 \ldots v_{i-1}\ s_i \ldots s_n))] \\
(\text{lcase}) & C[(\mathtt{case}\ (\mathtt{let}\ x = v\ \mathtt{in}\ s)^{\mathsf{sub}}\ alts)] \to C[(\mathtt{let}\ x = v\ \mathtt{in}\ (\mathtt{case}\ s\ alts))]
\end{array}
$$

Fig. 8 Standard Reduction rules in the let-language

The *answers* of reductions are values – but not variables – that may be embedded in lets. I.e., expressions of the form $(\mathtt{let}\ x_1 = v_1\ \mathtt{in}\ (\mathtt{let}\ x_2 = v_2\ \mathtt{in}\ \ldots(\mathtt{let}\ x_n = v_n\ \mathtt{in}\ v)\ldots))$ 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 $(\mathtt{let}\ x = v\ \mathtt{in}\ t)$, then the hole must be within an abstraction of v. Local contextual approximation and local contextual equivalence for $\mathcal{P}$-$\mathtt{let}$-expressions are defined accordingly, where we use the symbols $\leq_{\mathtt{let},\mathcal{P},T}$ and $\sim_{\mathtt{let},\mathcal{P},T}$ for the corresponding relations. Now we can show the context lemma for $\mathcal{P}$-$\mathtt{let}$-expressions:

A $\mathcal{P}$-$\mathtt{let}$-*reduction context* $R[\cdot]$ is any context, where the label-shifting starting with $R[\cdot]^{\mathsf{lr}}$ ends successfully at the hole. Note that the hole cannot occur as $(\mathtt{let}\ x = [\cdot]\ \mathtt{in}\ t)$.

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

We now define the CIU-preorder, which is defined as the local contextual preorder, but uses the smaller class of reduction contexts for the tests in its definition.

Definition A.1 (CIU-preorder, $\mathtt{let}$-language) The *CIU-preorder* $\leq_{\mathtt{let},\mathcal{P},R,T}$ for $\mathcal{P}$-$\mathtt{let}$-expressions is defined as follows: For well-typed $\mathcal{P}$-$\mathtt{let}$-expressions $s, t :: T$, the inequation $s \leq_{\mathtt{let},\mathcal{P},R,T} t$ holds iff $\forall\mathcal{P}$-$\mathtt{let}$-reduction contexts $R[\cdot :: T]$: if $R[s], R[t]$ are closed, then $(R[s]{\downarrow} \implies R[t]{\downarrow})$

Our next target is to show a context lemma, which states that the CIU-preorder is included in the local contextual preorder in the $\mathtt{let}$-language. For the proof 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 A.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 $t_1 :: T_1,\ldots,t_n :: T_n$ the context $C[t_1,\ldots,t_{j-1},\cdot_j :: T_j, t_{j+1},\ldots,t_n]$ 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 A.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 A.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 $\mathcal{P}$-`let`-multicontexts $C[\cdot_1 :: T_1,\ldots,\cdot_n :: T_n]$ and for all well-typed $\mathcal{P}$-`let`-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 A.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 ($\mathrm{beta}_{\mathtt{let}}$) reduction, the holes $\cdot_i$ may change their position.

- For a ($\text{case}_{\texttt{let}}$) reduction, the position of $\cdot_i$ may be changed as in the previous item, or if the position of $\cdot_i$ is in an alternative of `case`, which is discarded by a (case)-reduction, then s_i and t_i are both eliminated.
- If the reduction is a (cp) reduction and there are some holes $\cdot_i$ inside the copied value, then there are variable permutations $\rho_{i,1}, \rho_{i,2}$ with $s'_i = \rho_{i,1}(s_i)$ and $t'_i = \rho_{i,2}(t_i)$. One can verify that we may assume that $\rho_{i,1} = \rho_{i,2}$ for all i. Now the precondition implies $s'_i \leq_{\texttt{let},\mathcal{P},R,T'_i} t'_i$, since it is easy to verify that for any expressions r_1, r_2 with $r_1 \leq_{\texttt{let},\mathcal{P},R,T} r_2$ also $\rho(r_1) \leq_{\texttt{let},\mathcal{P},R,T} \rho(r_2)$ for any variable permutation ρ holds.
- If the standard reduction is a ($\text{delta}_{\texttt{let}}$)-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 $C[t_1, \ldots, t_n]{\downarrow}$. □

A.2 The CIU-Theorem

Now we use the context lemma for the let-language and transfer the results to usual $\mathcal{P}$-expressions without `let`. There are two main steps to establish this goal. Firs we show that the CIU-equivalence $\leq_{\texttt{let},\mathcal{P},R,T}$ restricted on `let`-free $\mathcal{P}$-expressions implies local contextual preorder $\leq_{\mathcal{P},T}$ (for `let`-free $\mathcal{P}$-expressions). This part will be proved by showing adequacy of a translation: Given the identity translation Φ which translates $\mathcal{P}$-expressions into $\mathcal{P}$-`let`-expressions, we show *adequacy* of this translation, i.e. for all well-typed $\mathcal{P}$-expressions s, t the implication $\Phi(s) \leq_{\texttt{let},\mathcal{P},T} \Phi(t) \implies s \leq_{\mathcal{P},T} t$ holds. The framework of [**?**] gives us a sufficient criterion to show adequacy: We need to show that Φ is *compositional* (i.e. $\Phi(C[s]) = \Phi(C)[\Phi(s)]$ for all $\mathcal{P}$-expressions) and that Φ is convergence equivalent, i.e. $s{\downarrow} \iff \Phi(s){\downarrow}$.

The second step is to show that a CIU-preorder defined on (`let`-free) $\mathcal{P}$-expressions is included in the CIU-preorder $\leq_{\texttt{let},\mathcal{P},R,T}$ (restricted to $\mathcal{P}$-expressions). For this part we require convergence equivalence of a translation $\overline{\Phi}$ which translates $\mathcal{P}$-`let`-expressions into `let`-free $\mathcal{P}$-expressions by inlining all `let`-bindings.

Finally, we will show that the CIU-preorder and the local contextual preorder coincide.

Let Φ be the translation from $\mathcal{P}$-expressions to $\mathcal{P}$-`let`-expressions 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 $\mathcal{P}$-`let`-expressions into $\mathcal{P}$-expressions. The translation is defined as $\overline{\Phi}(\texttt{let } x = v \texttt{ in } s) := \overline{\Phi}(s)[\overline{\Phi}(v)/x]$ for the `let`-construct 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 A.5 *Let $t \in Expr_{\texttt{let}}$ 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 A.6 *Let t be an $\mathcal{P}$-`let`-expression 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`-construct. 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* $\mathcal{P}$-`let`*-reduction context.*

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 $\mathcal{P}$-`let`-reduction context. 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 $\mathcal{P}$-`let`-reduction contexts 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 A.7 *Let* $t \in Expr_{\texttt{let}}$ *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 A.6 shows that $\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ R'[]$ is a $\mathcal{P}$-`let`-reduction context. 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:

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

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

Lemma A.8 *The following properties hold:*

1. *For all* $t \in Expr_{\texttt{let}}$*: if* t *is an answer, then* $\overline{\Phi}(t)$ *is a value, and if* $\overline{\Phi}(t)$ *is a value (but not a variable), then* $t \xrightarrow{ls,*} t'$ *where* t' *is an answer for the* `let`*-language.*
2. *For all* $\mathcal{P}$*-expressions* t*:* t *is a non-variable value iff* $\Phi(t)$ *is an answer for the* `let`*-language.*
3. *Let* $t_1, t_2 \in Expr_{\texttt{let}}$ *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* $t_1 \in Expr_{\texttt{let}}$ *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 for the `let`-language and for the usual (`let`-free) language 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 the `let`-language.

3: If the reduction is a (ls,llll) 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 A.5 and A.7. □

Lemma A.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 A.8, part 1. The induction step follows by Lemma A.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 A.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 A.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 A.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 [**?**] shows that convergence equivalence and compositionality of Φ imply adequacy, i.e.:

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

We now define the CIU-preorder for $\mathcal{P}$-expressions. It is analogously defined to the local contextual preorder $\leq_{\mathcal{P},T}$, where only $\mathcal{P}$-reduction contexts are used in the tests, and where the expressions are closed by closing value substitutions.

Definition A.11 (CIU-preorder) For $\mathcal{P}$-expressions, the *CIU-preorder* $\leq_{\mathcal{P},T,\mathsf{ciu}}$ is defined as follows. For $\mathcal{P}$-expressions $s, t :: T$ the inequation $s \leq_{\mathcal{P},T,\mathsf{ciu}}$ holds, iff 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.

Lemma A.12 (CIU-Lemma) $\leq_{\mathcal{P},T,\mathsf{ciu}} \subseteq \leq_{\mathcal{P},T}$.

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 A.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 $\mathcal{P}$-`let`-reduction context 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`-constructs, $\overline{\Phi}(R_{let})$ 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`-construct 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 $\mathcal{P}$-`let`-reduction context 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 A.13 *The reductions (beta), (delta), and (case) are locally correct program transformations for $\mathcal{P}$-expressions. I.e., if $s \to t$ by (beta), (delta), or (case), then $C[s] \to C[t]$ is a locally correct transformation.*

Proof We use the CIU-Lemma A.12: 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. $R[\sigma(t)]\!\downarrow$.

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 requires termination of a function, before anything can be proved about the function.

Using the CIU-Lemma and the local correctness of (beta)-reduction we are now able to prove the local CIU-Theorem:

Theorem A.14 (CIU Theorem) *For $\mathcal{P}$-expressions $s, t :: T$: $R[\sigma(s)]\downarrow \implies R[\sigma(t)]\downarrow$ for all closing $\mathcal{P}$-value substitutions σ and $\mathcal{P}$-reduction contexts R if, and only if $s \leq_{\mathcal{P},T} t$ holds.*

Proof One direction is the CIU-Lemma A.12. 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 locally correct (Proposition 2.14), 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 $R[(\lambda x_1. \ldots . x_n.t)\ v_1 \ldots v_n]\downarrow$. Local 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 a $\mathcal{P}'$-expression:

Corollary A.15 *Let $\mathcal{P}$ be a program. For all $\mathcal{P}$-expressions $s, t :: T$ it holds:*
$(\forall \mathcal{P}' \sqsupseteq \mathcal{P} : s \leq_{\mathcal{P}',R,T} t)$ if, and only if $s \leq_{\forall \mathcal{P},T} t$ holds.

A.3 The F-Free CIU-Theorem

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

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 A.16 (CIU-Lemma F-free) $\leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}} \subseteq \leq_{\mathcal{P},T}$

Proof The proof is by an application of the CIU-lemma:
Let $s, t :: T$ be two $\mathcal{P}$-expressions with $s \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}} t$. 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 A.14.

We are now able to prove the following theorem.

Proof of Theorem 2.13 *We have to show $\leq_{\mathcal{P},T} = \leq^{\neg F}_{\mathcal{P},T,\mathsf{ciu}}$. One direction is the F-free CIU-Lemma A.16. The other direction is the same as in the proof of the local CIU-theorem A.14.* □

A.4 Criteria for Contextual Approximation

Proof of Proposition 2.18 *The claim is:*

For closed expressions s, t of constructed type T: $s \leq_{\mathcal{P},T} t$ 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 $i = 1, \ldots, n$.
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.

We first consider expressions s, t of constructed type: If $s \leq_{\mathcal{P},T} t$, then either $s{\Uparrow}$, or $s{\downarrow}$ and $t{\downarrow}$. If $s{\downarrow}$ and $t{\downarrow}$, then both expression must reduce to a closed value of constructed type T, hence both expressions must reduce to constructor applications. Assume that $s \xrightarrow{*} c\ v_1\ \ldots\ v_n$ and $t \xrightarrow{*} c'\ w_1\ \ldots\ w_{n'}$. The constructors c and c' must be identical, since otherwise the context $\texttt{case}\ [\cdot]\ (((c\ x_1\ \ldots\ x_n) \to \texttt{True})\ (c'\ y_1\ \ldots\ y'_n) \to \bot_{Bool}\ \ldots)$ *would refute the assumption $s \leq_{\mathcal{P},T} t$. Thus $c = c'$ and $n = n'$. To show $v_i \leq_{\mathcal{P},T_i} w_i$ we reason by contradiction: Assume that there exists a context C such that $C[v_i]{\downarrow}$ but $C[w_i]{\Uparrow}$. Then for the context $C' := C[\texttt{case}\ [\cdot]\ ((c\ x_1\ \ldots\ x_n) \to\ x_i)\ldots]$ we have $C'[s]{\downarrow}$, but $C'[t]{\Uparrow}$ which contradicts $s \leq_{\mathcal{P},T} t$. Thus our assumption was wrong and $v_i \leq_{\mathcal{P},T_i} w_i$ holds.*

For the other direction, we first consider the case that $s{\Uparrow}$. In this case s is an Ω-expression, since s is closed. Thus it is a least element w.r.t $\leq_{\mathcal{P},T}$ (see Corollary 2.15). Now assume that $s \xrightarrow{*} c\ v_1 \ldots v_n$ and $t \xrightarrow{*} c\ w_1 \ldots w_n$ and for all i: $v_i \leq_{\mathcal{P},T_i} w_i$. Local correctness of the reduction rules implies that $s \sim_{\mathcal{P},T} (c\ v_1 \ldots v_n)$ and $t \sim_{\mathcal{P},T} (c\ w_1 \ldots w_n)$. Hence it is sufficient to show that $(c\ v_1 \ldots v_n) \leq_{\mathcal{P},T} (c\ w_1 \ldots w_n)$ holds. This obviously follows from $v_i \leq_{\mathcal{P},T_i} w_i$ (for all i), since $\leq_{\mathcal{P},T}$ is a precongruence.

We now consider the second part of the proposition: If $s \leq_{\mathcal{P},T} t$, then either $s{\Uparrow}$, or $s{\downarrow}, t{\downarrow}$. Since T is a function type, the results must be abstractions. The conclusion holds since $\leq_{\mathcal{P},T}$ is a congruence and since (beta) is locally correct.
The other direction also holds, using Corollary 2.16 which implies $s' \leq_T t'$. Then we use that $\leq_{\mathcal{P},T}$ is a precongruence.

B Termination of a Subset of VN-Reductions

We show that VN-reduction without VNbeta- and VNcase-reductions terminates: Therefore we use two measures *css* and *mcss* of expressions, where $css(s) \in \mathbb{N}$ and $mcss(s)$ is a multiset of natural numbers:

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

The multiset $mcss(s)$ consists of the number $css(s)$ and all numbers $css(s')$, for any case-alternative s' which is a subterm of s. It is well-known [**?**] that multisets of numbers are well-founded w.r.t. the multiset-ordering: $M_1 << M_2$, iff there exists multisets X_1, X_2 such that $\emptyset \neq X_2 \subseteq M_2$, $M_1 = (M_2 \setminus X_2) \cup X_1$, and $\forall x_1 \in X_1 . \exists x_2 \in X_2 . x_1 < x_2$.

Lemma B.1 *Every VN-reduction sequence without (VNcase)- and (VNbeta)-reduction steps is finite.*

Proof We check that for every possible reduction rule, the measure $mcss(.)$ is strictly decreased: First we prove that the redexes are strictly decreased w.r.t. $css(.)$. As a second step we analyze the possible modifications of the multiset *mcss*.

- 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: $4css(t) + 2\max(r_i) + css(r) > 2css(t) + \max(2css(r_i) + css(r))$. □

Now we argue for the three case-modifying reductions that $mcss$ is strictly reduced. Since for $s \to s'$, we have $css(s) > css(s')$, and thus also $css(s) > css(s') > css(t)$ for the freshly introduced alternatives of the case-expression(s), which implies $mcss(s) >> mcss(s')$.

C Properties of the VN-Standard Reduction

Figure 9 shows the structure of the proof of the Main Theorem 3.13 and the further consequences proved in Section 3.

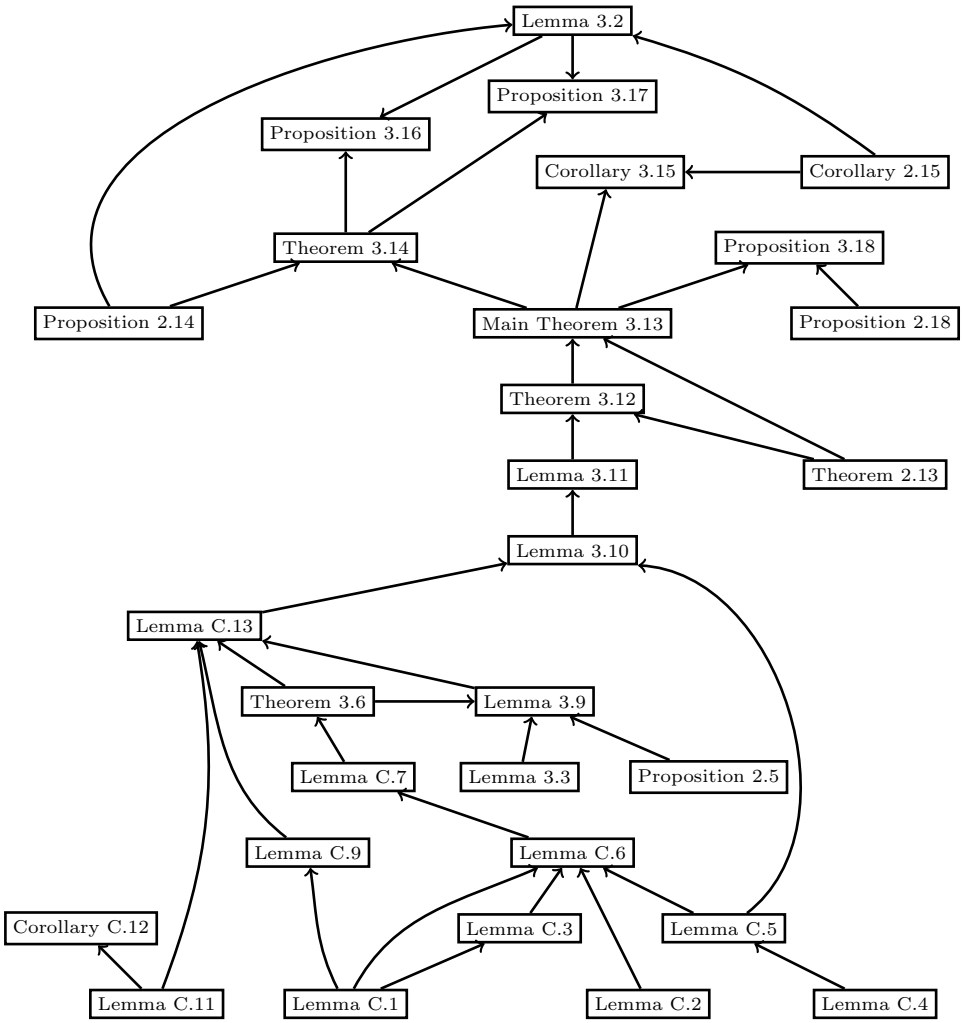


Fig. 9 Structure of the proof of Main Theorem 3.13 and its consequences

C.1 Termination Properties of the VN-Standard Reduction

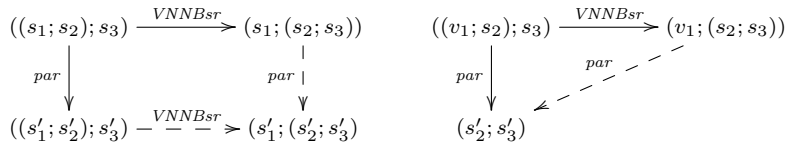
We use a parallel version of reduction which is defined like the 1-reduction in [3], which is denoted as $\xrightarrow{par}$. It may contain sr-reduction, but there may also be non-sr-reductions included. Note that the seq-syntax and hence seq-reductions are permitted.

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.

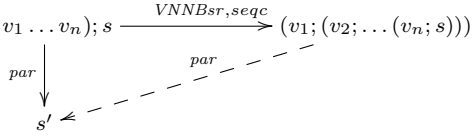
We will use the technique of computing so-called forking diagrams (see e.g. [**?**]) for two reductions. The full arrows indicate given reductions, and the dashed arrows indicate existing reductions. A complete set means to cover all possibilities on the diagram and arrow-level.

Lemma C.1 *The following presents a complete set of forking diagrams for forkings of $\xrightarrow{par}$ and $\xrightarrow{VNNBsr}$-reductions, provided the two given reductions are different.*

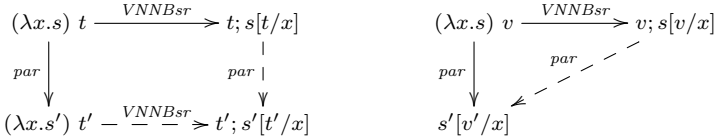
Proof The proof is by inspecting all the possibilities. We explain the typical and the exceptional case. The diagrams for (seqapp) are a typical case:



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



The following diagrams show typical cases for VNbeta.



The cases for VNcase are similar to VNbeta. □

Lemma C.2 *Let s be an (open) F-free expression. If s is sr-irreducible, then the VNNBsr-reduction of s is finite.*

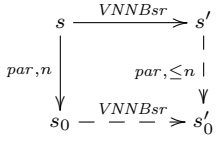
Proof If s is a value, then there is no VNNBsr-reduction. Otherwise, $s = R[x]$ for a reduction context R. Note that the occurrence of x can only be in a subexpression (`case` x *alts*), or $(x\ s)$, since x is a value. Now we use induction on the following depth of the hole of $R[x]$: It is the sum 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 further 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.3 *Let s be an (open) F-free 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 $s' \xrightarrow{par,\leq n} s'_0$:*

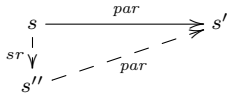


Proof This follows by induction on n: For the base case one has to observe that if s is sr-irreducible, and $s \xrightarrow{VNNBsr} s'$, then s' is sr-irreducible, too. This follows by inspecting the VN-reductions. The induction step $n > 0$ follows by applying one of the diagrams of Lemma C.1 and then using the induction hypothesis. □

For the proof of Lemma C.6 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.4 *Let $s \xrightarrow{par} s'$. Then the reduction can be split into $s \xrightarrow{sr,*} s'' \xrightarrow{par} s'$, where s'' does not contain any sr-redex.*

Proof If the $\xrightarrow{par}$-reduction does not contain an sr-redex, then the claim holds. Otherwise, assume that *par* contains an sr-redex. Then it is easy to verify that the following diagram holds, since the sr-redex is not contained in another redex of the parallel reduction



We have to show that the iterated application (the development) of this diagrams stops with an $\xrightarrow{par}$-reduction that does not contain an sr-redex. We show that the measure ϕ on expressions is strictly reduced during constructing the development, where ϕ is as follows: Let $s \xrightarrow{par} s'$. We assume that all $\xrightarrow{par}$-redexes in s are labeled and that these labels are inherited by the standard reduction (i.e. by applying the above diagram).

- For a labeled subexpression $((\lambda x.s)\ t)$: If it is labeled then $\phi((\lambda x.s)\ t) = 1 + \phi(s) + \phi(t) \cdot \#(x, s)$, where $\#(x, s)$ is the number of occurrences of x in s.
- For a labeled subexpression $(\mathtt{case}_T\ (c\ s_1 \dots s_n) \dots ((c\ x_1 \dots x_k) \to r)$, the ϕ-result is $1 + \phi(r) + \Sigma(\phi(s_i) \cdot \#(x_i, r))$.
- For a labeled subexpression $s; t$: $\phi(s; t) = 1 + \phi(s) + \phi(t)$.
- In all other cases the subexpression is unlabeled, and we define $\phi(x) = 0$ for a variable x, $\phi(\lambda x.s) = \phi(s)$, $\phi(s\ t) = \phi(s) + \phi(t)$, $\phi(s; t) = \phi(s) + \phi(t)$, and $\phi(\mathtt{case}\ s\ (pat_1 \to r_1) \dots (pat_n \to r_n)) = \phi(s) + \sum_{i=1}^{n} \phi(r_i)$

If there is an sr-reduction, and its redex is labeled, then the measure is strictly decreased after a single sr-reduction. If the redex is not labeled, then we are finished, since the $\xrightarrow{par}$-reduction does not contain an sr-reduction. 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. □

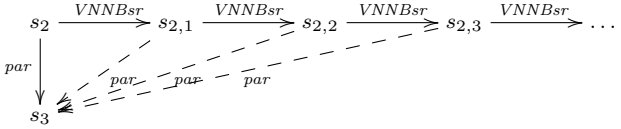
Lemma C.5 *Let $s \xrightarrow{par,*} s'$ where s' is sr-irreducible. Then there is a terminating reduction $s \xrightarrow{sr,*} s''$, where s'' is sr-irreducible.*

Proof This follows from Lemma C.4 and since the internal parallel-reduction can be commuted with sr-reductions, and by induction.

Lemma C.6 *Let s be an (open) F-free 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.2.
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.1:

Lemma C.3 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.5 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.3, the $\xrightarrow{VNNBsr,*}$-reduction using only triangle-diagrams must be finite. □

Lemma C.7 *Let s be an (open) F-free expression. If for some F-free 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 $\sigma(s)$ 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 there is some sr-irreducible s_1 with $s \xrightarrow{sr,*} s_1$. Then Lemma C.6 shows that the *VNNBsr*-reduction of s is finite and ends with an *VNNBsr*-irreducible expression. □

Proof of Theorem 3.6 *The claim is:*

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

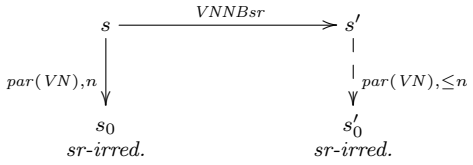
It is a direct consequence of Lemma C.7.

C.2 The $\xrightarrow{par(VN)}$ Reduction and Approximation

We also need more information about the necessary parallel reduction

Definition C.8 The 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.9 *Let s be an F-free (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.1, 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.10 The `Bot`–*cl–depth* of an expression is the minimum of all *cl–depth*s of all occurrences of `Bot`. If there are no `Bot`-occurrences, then it is ∞.

Lemma C.11 *Let s be an F-free (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*. □

Corollary C.12 *Let s be an F-free (open) expression such that $s \xrightarrow{sr} s'$. If n, (or n', resp.) are the* `Bot`*–cl–depths of s (or s'. resp.), then $n - n' \leq 1$.*

Lemma C.13 *Let $\mathcal{P} \sqsubseteq \mathcal{P}'$. Let s be an F-free expression such that $s \xrightarrow{sr,n} v$ where v is a value of $\mathcal{P}$-type. Let s' be an expression constructed from s where (some) subexpressions t are replaced by $ValueConstr_n(t)$. Then $s' \xrightarrow{par(VN),\leq n} v' \leq_{\mathcal{P}',T} v$.*

Proof This follows from Lemma 3.9 and Lemmas C.9 and C.11 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 Theorem 3.6 is required, which shows that the replacement of expressions with infinite *VNNBsr*-reductions are Ω-expressions and thus do not interfere. □

Proof of Lemma 3.10 *The claim is:*

> *Let s be an F-free expression such that $s \xrightarrow{sr,n} v$ where v is a value. Let s' be an expression constructed from s where (some) subexpressions t are replaced by $ValueConstr_n(t)$. Then $s'{\downarrow}$.*

The claim follows from Lemma C.13 and Lemma C.5.

D Proofs for Section 5.1.5 on ATD-formulas

Proof of Proposition 5.9 *The claim is:*

> *Let $\mathcal{P}$ be a program that contains the Boolean type and let $F = \forall x_1 :: T_1 \ldots x_n :: T_n.F'$ be a closed $\mathcal{P}$-formula where F' is quantifier-free, and such that all equations are of DT-type. Let $\mathcal{P}'$ be $\mathcal{P}$ extended by a finite set of functions. Then F is $\mathcal{P}$-valid iff it is $\mathcal{P}'$-valid.*

Proof. If F is $\mathcal{P}'$-valid, then it is also $\mathcal{P}$-valid, since quantification is over less values. Let F be $\mathcal{P}$-valid. We have to show that F is $\mathcal{P}'$-valid. Let v'_i be $\mathcal{P}'$-values and consider $F[v'_1/x_1, \ldots v'_n/x_n]$. Consider the top-expressions in equations $F[v'_1/x_1, \ldots v'_n/x_n]$, which may either be equivalent to $\bot$ or to a value. Let m be the maximum of the lengths of the successful evaluations of all the top-expressions that evaluate to a value. Now consider the values v_i constructed from v'_i by at least $m+1$-times unrolling all, in particular the $\mathcal{P}'$-function definitions by δ-reductions and then replacing the remaining function symbols by $\bot$. Now let s be a top-expression in F. If $s[v'_1/x_1, \ldots v'_n/x_n]{\Uparrow}$, then also $s[v_1/x_1, \ldots v_n/x_n]{\Uparrow}$, since $v_i \leq_{\mathcal{P}',T} v'_i$. If $s[v'_1/x_1, \ldots v'_n/x_n]{\downarrow}$, then also $s[v_1/x_1, \ldots v_n/x_n]{\downarrow}$, due to the length of reductions. Correctness of the reductions implies that $s[v_1/x_1, \ldots v_n/x_n]$ must evaluate to the same value as $s[v'_1/x_1, \ldots v'_n/x_n]$, since $s[v_1/x_1, \ldots v_n/x_n] \leq_{\mathcal{P}',T} s[v'_1/x_1, \ldots v'_n/x_n]$ and since T is a DT-type. Hence the truth value of $F[v'_1/x_1, \ldots v'_n/x_n]$ is the same as $F[v_1/x_1, \ldots v_n/x_n]$, which is `tt` *by assumption. We conclude that F is $\mathcal{P}'$-valid.* □

Proof of Theorem 5.14 *The claim is:*

> *Let $\mathcal{P}$ be a program that contains the Boolean type and sufficiently many $\mathtt{eq}_T$-functions, and at least one LSM $\mathcal{P}^*$ of $\mathcal{P}$ exists, and let F be a (closed) ATD-formula w.r.t. $\mathcal{P}$. Then F is $\mathcal{P}$-valid iff F is loosely $\mathcal{P}$-valid. Consequently, F is a $\mathcal{P}$-tautology iff F is a loose $\mathcal{P}$-tautology.*

Proof. *Let $\mathcal{P}^*$ be a selected LSM and $\mathcal{P}^{**}$ be the union of $\mathcal{P}$ and $\mathcal{P}^*$.*
(A^*): If G is a quantifier-free closed $\mathcal{P}$-formula where every top expression is of DT-type, and G^* the LSM-modified variant, and all top-expressions terminate, then G and G^* have the same logical value. This holds, since for two terminating closed expressions t, t' of DT-type with $t \leq t'$, only $t \sim t'$ is possible.
Let $F = \forall x_1 :: T_1 \ldots x_n :: T_n.F' \implies F''$ be an ATD-formula that is $\mathcal{P}$-valid. Let $F^* = \forall x_1 :: T_1 \ldots x_n :: T_n.F^{*\prime} \implies F^{*\prime\prime}$ be the LSM-variant of F. We have to show two directions.
Let $v_1^*, \ldots, v_n^*$ be closed $\mathcal{P}^*$-values. We have to show that $(F^{*\prime} \implies F^{*\prime\prime})[v_1^*/x_1, \ldots, v_n^*/x_n]$ is $\mathcal{P}^*$-valid. Let $v_1, \ldots, v_n$ be the $\mathcal{P}$-values corresponding to $v_1^*, \ldots, v_n^*$. Then for $\sigma = \{v_1/x_1, \ldots v_n/x_n\}$ and $\sigma^* = \{v_1^*/x_1, \ldots v_n^*/x_n\}$, all expressions $\sigma(s)$ terminate for all top-expressions s of $\sigma(F')$. The same holds for σ^* and the top-expressions $\sigma^*(s^*)$ of $\sigma^*(F^{*\prime})$. Hence (A^*) implies that $\sigma(F')$ and $\sigma(F^{*\prime})$ have the same logical value. If it is ff*, then the proof is finished. If it is* tt*, then $\mathcal{P}$-validity of $\sigma(F' \implies F'')$ shows that the top terms of $\sigma(F'')$ terminate. The LSM-modification now implies that also the top terms of $\sigma^*(F^{*\prime\prime})$ terminate and we can apply (A^*), and see that $\sigma^*(F^{*\prime} \implies F^{*\prime\prime})$ is valid. This holds for all values, hence F^* is $\mathcal{P}^*$-valid.*
For the other direction assume that F^* is $\mathcal{P}$-valid. We have to show that F is $\mathcal{P}$-valid. Let $v_1, \ldots, v_n$ be $\mathcal{P}$-values, and $v_1^*, \ldots, v_n^*$ be the corresponding $\mathcal{P}^*$-values, and σ, σ^* be the respective substitutions. Due to the assumption that F is an ATD-formula, the top expressions of $\sigma(F')$ terminate, hence also the top expressions of $\sigma^*(F^{*\prime})$ terminate, and then (A^*) implies that the logical value of $\sigma(F')$ is the same as of $\sigma^*(F^{*\prime})$. If it is ff*, then the proof is finished. If it is* tt*, then $\mathcal{P}^*$-validity of F^* implies that $\sigma^*(F^{*\prime\prime})$ is $\mathcal{P}^*$-valid. The termination of the top expressions in $\sigma(F'')$ follows from the definition of ATD-formulas, and this implies the termination of the top expressions in $\sigma^*(F^{*\prime\prime})$. Now (A^*) shows that the logical values of $\sigma^*(F^{*\prime\prime})$ and $\sigma(F')$ are both* tt*, and hence that the logical value of $\sigma(F' \implies F'')$ is* tt*. Thus F is $\mathcal{P}$-valid.*

We argue that F is a $\mathcal{P}$-tautology if, and only if F is a loose $\mathcal{P}$-tautology. Let $\mathcal{P}'$ be an extension of $\mathcal{P}$, such that an LSM $(\mathcal{P}')^*$ exists. If the formula F is $\mathcal{P}$-valid, then it is also $\mathcal{P}'$-valid, since ATD-formulas are conservative (Theorem 5.12). Then we can apply the arguments above and obtain that it is also valid in $(\mathcal{P}')^*$.
For the other direction let F is a loose $\mathcal{P}$-tautology. Then the first part of the proof shows that it is also $\mathcal{P}$-valid, and due to conservativity it is also $\mathcal{P}'$-valid. □