Report

Axioms for strict and lazy functional programs

Author(s):
Stärk, Robert F.

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-006714609

Rights / License:
In Copyright - Non-Commercial Use Permitted
Axioms for strict and lazy functional programs

Robert F. Stärk

Computer Science Department, ETH Zürich
ETH Zentrum, CH–8092 Zürich, Switzerland
(e-mail: staerk@inf.ethz.ch)

Abstract

We show the adequacy of axioms and proof rules for strict and lazy functional programs. Our basic logic comprises a huge part of what is common to both styles of functional programming. The logic for call-by-value is obtained by adding the axiom that says that all variables are defined, whereas the logic for call-by-name is obtained by adding the axiom that postulates the existence of undefined object for each type. To show the correctness of the axiomatization we do not use denotational semantics and the adequacy of the evaluation of programs with respect to the semantics. Instead we use the standard term models based on call-by-value and call-by-name evaluation. We introduce a new method to prove on the syntactical level the monotonicity of the evaluation of functional programs with unbounded recursion. The direct method yields result about the proof-theoretic strength of the axiomatization. As a side result we obtain a syntactical proof of the context lemma for simply typed lambda terms with recursion.

1 Introduction

One of the main arguments in favor of functional programming is that it is easier to verify functional programs than imperative programs. Since functional programs have no side effects, the evaluation of a program does not change the global state and reasoning about programs is therefore much easier. The same techniques can be used as in ordinary mathematics. There are, however, differences. Functional programs may not terminate and therefore a logic for functional programs has to deal with termination and partial functions. Functional programs are mainly based on recursion and therefore a logic for functional programs needs appropriate axioms and proof rules for recursion. Moreover, there are two main streams in functional programming. On one side there is ML [13] with a strict (call-by-value) evaluation strategy and on the other side there is...
Haskell [9] with lazy (call-by-name) evaluation. A logic for functional programs should therefore cover both styles of programming.

We think that a reduction calculus for a functional programming language is not enough. In addition, axioms and proof rules are needed in order that it is possible to reason about properties of programs in a mathematical way. We view a reduction calculus ($\beta$-reduction, etc.) only as the basis for a richer formal system that includes quantifiers and induction principles and allows to formulate and prove complex statements about programs using the full power of logic. The reduction calculus is then used to justify the axioms and proof rules. The reduction calculus is simply a bridge between the formal system and the actual implementation of the programming language. In the reduction calculus alone, one cannot prove anything interesting about programs except that they are $\beta$-equivalent.

What are the axioms and proof rules for prominent functional languages like ML and Haskell? We continue the approach started in [18] which is based on a typed logic of partial terms (the system BPT) and is the common ground for call-by-value (or strict) languages like ML and call-by-name (or lazy) languages like Haskell. We give a direct interpretation of the axioms and proof rules of BPT in terms of call-by-value and call-by-name evaluation and avoid the detour over denotational semantics of [18]. This does not mean that denotational semantics is useless. It just means that denotational semantics is in our case not needed to show the correctness of the axioms and proof rules even in the case of higher types.

Properties of programs that can be derived in the basic logic of partial terms (BPT) are true for call-by-value evaluation as well as for call-by-name evaluation. The logic for call-by-value evaluation (VPT) is obtained from the basic system simply by adding axioms that ensure that quantifiers range over defined objects only. The additional axioms say that for each type $\tau$ every variable of type $\tau$ is defined. In the case of call-by-name evaluation (NPT) we simply add axioms that postulate the existence of undefined objects for each type and therefore the quantifiers range also over undefined objects. Hence, our axiomatizations of functional programming show that the world of strict functional programming (ML) and the world of lazy functional programming (Haskell) have more in common than one might think.

We say that a logic $L$ is adequate for call-by-x evaluation, if the following two properties hold for any closed program $t$:

1. (1.1) The formula $t \downarrow$ is derivable in the logic $L$ iff the call-by-x evaluation of the program $t$ terminates.

2. (1.2) If $t$ is of basic type $\iota$ and $v$ is a value of $\iota$, then the equation $t = v$ is derivable in the logic $L$ iff the call-by-x evaluation of the program $t$ terminates with result $v$.

The two conditions are the minimal requirements a logic has to fulfill in order that formulas that are derived in the logic can be interpreted in the real world. Assume, for example, that we can derive in the logic $L$ that the domain of a
program \(s\) is included in the domain of a program \(t\):

\[
\forall x \left( s \downarrow x \rightarrow t \downarrow x \right)
\]

Given that this formula is derivable, how can we conclude that in the real world the program \(t\) terminates on each input on which the program \(s\) terminates?

We can use condition (1.1). If the program \(s\) terminates on the input \(v\), then this fact is derivable in the logic \(L\), i.e., we can derive the formula \(s \downarrow v\). But then we can also derive the formula \(t \downarrow v\) and again by condition (1.1), we can conclude that the program \(t\) terminates on the input \(v\).

For another example, assume that \(s\) and \(t\) are two programs that take an argument of type integer and return an integer. Assume that we can derive in the logic \(L\) that \(s\) and \(t\) have the same input/output behavior:

\[
\forall x \left( s \downarrow x = t \downarrow x \right)
\]

How can we conclude that in the real world the two programs compute the same functions on the integers?

We can use condition (1.2). If the program \(s\) computes for the argument \(i\) the result \(j\), then according to condition (1.2) the equation \(s \downarrow i = j\) is derivable in the logic \(L\). But then we can also derive the equation \(t \downarrow i = j\) in \(L\). Again by condition (1.2), we can conclude that the program \(t\) computes on the argument \(i\) the same result \(j\).

The plan of this paper is as follows. In Sect. 2 we define syntax and evaluation semantics of simply typed functional programs with recursion. Then we summarize in Sect. 3 the axioms and rules of the basic logic of partial terms of [18] and some important derivable principles. In Sect. 4 we show the adequacy of an extension of the basic logic of partial terms for call-by-value evaluation by a direct interpretation of the logic. The main technical point is the monotonicity of the call-by-value evaluation for programs with recursion. In Sect. 5 we do the same for call-by-name evaluation. Sect. 6 contains a syntactical proof of the context lemma. Finally, we conclude with some general remarks on functional programming in Sect. 7.

1.1 Related work

Feferman uses in [4, 5] Beeson’s logic of partial terms [1] to provide a logical foundation for the use of type systems in functional programming and to set up logics for the termination and correctness of programs. His logics are of great expressive power and flexibility while minimal in proof-theoretic strength. Our logics are typed versions of Beeson’s logic of partial terms suitable for typed functional programs with recursion and different evaluation strategies.

Howe introduces in [8] a general method for proving congruence of bisimulation-like equivalences in functional programming languages. His so-called precongruence candidate has similar properties as our auxiliary relations \(\leq^1\), \(\leq^2\), \(\leq^3\). It is not clear how the least fixed point properties that we need in this
paper can be obtained using the general method. Howe’s method is extended by Gordon to typed functional languages in [6].

Kahle and Studer write in [10] that in typed functional programming languages, like ML, fixed point operators are built in, but there is no way to guarantee on the syntactical level that the solution produced by these operators will be the least fixed point. They claim that the least fixed point property is only given by the semantical interpretation. We do not agree. This paper contains a proof on the syntactical level that the built-in recursion operators of strict and lazy functional programming languages are indeed the least fixed points. In addition, Kahle and Studer point out that beside the difference of a typed and untyped approach their theory allows a recursion-theoretic model, whereas the logics of [18] have domain-theoretic interpretations only. In this paper, we show that the domain-theoretic interpretations in [18] are not needed and that the main results of [18] can be obtained on the syntactical level. We give interpretations of the logics directly in terms of call-by-value and call-by-name evaluation. Note also that in our basic logic of partial terms we can derive \( \lnot (\text{rec} f x. f x) v \downarrow \) for any syntactical value \( v \) (see proof of Lemma 3.2). Hence, there are provably non-terminating programs in our theory as they are in the theory LFP of [10].

In [14], Moran and Sands develop an operational theory for the call-by-need \( \lambda \)-calculus with recursive lets, constructors, and case expressions. Their theory is cost-sensitive and reflects the computational distinctions between call-by-need and call-by-name. It would be interesting to extend their theory into a logic in a similar way as we do it in this paper for call-by-value and call-by-name.

Schwichtenberg’s Minlog system [2] is designed as an interactive proof-system for the synthesis of functional programs. The underlying logic is based on first-order formulas over simply typed \( \lambda \)-terms with higher-type primitive recursion. The programs extracted from proofs are always total.

Thompson describes in [21] a logic for the lazy functional programming language Miranda. There are several differences between Thompson’s logic and our logic NPT for call-by-name. First, he follows Scott’s approach in the Logic of Computable Functions (LCF) and uses constants \( \bot \) that denote an undefined value for each type. We use a definedness predicate instead. As explained in [18] the two approaches are not equivalent. Second, Thompson has no axioms which express that recursive function definitions denote the least solution of the corresponding equations. He has just the fixed-point axiom (3.41) but not the minimality axiom (3.20) below. Thompson sketches in [21, Sect. 4.1] a proof of the adequacy of a subsystem of his logic with respect to evaluation of Miranda programs. For the full logic, however, he refers to denotational semantics.

2 Evaluation of functional programs

The fundamental concept in functional programming is the concept of recursively defined functions. A recursively defined function is given by a recursion
The function $f$ can be called in the body of its definition. We use the syntax \texttt{rec } \vec{x} . t(f, \vec{x})\texttt{ for this kind of functions. In ML, it corresponds to the term}

\texttt{let fun } \vec{x} . t(f, \vec{x}) \texttt{ in } f \texttt{ end}

We consider simply typed programs. Types are either basic types $\iota$, or function types $\sigma \rightarrow \tau$. Types are inductively generated as follows:

\[ \sigma, \tau ::= \iota \mid \sigma \rightarrow \tau \]

We assume for each type $\tau$ a countably infinite set of variables $x^\tau, y^\tau, \ldots$ of type $\tau$. Terms (or programs) are denoted by $r, s, t$. We write $t^\tau$ to indicate that the term $t$ is of type $\tau$. Terms are inductively generated. They are of the following kinds:

1. variables: $x^\tau, y^\tau, \ldots, f^{\sigma \rightarrow \tau}, g^{\sigma \rightarrow \tau}, \ldots$
2. constants: $c^\iota$
3. applications: $(s^{\sigma \rightarrow \tau} t^\sigma)^\tau$
4. abstractions: $(\lambda x^\sigma . t^\tau)^{\sigma \rightarrow \tau}$
5. recursively defined functions: $(\texttt{rec } f^{\sigma \rightarrow \tau} x^\sigma . t^\tau)^{\sigma \rightarrow \tau}$

In the following we omit the types unless it is really necessary to indicate them. However, all terms are typed in this article. Omitting the types, terms are of the following form:

\[ r, s, t ::= x \mid f \mid c \mid s t \mid \lambda x . t \mid \texttt{rec } f . t \]

The set of free variables of a term $t$ is denoted by $\text{FV}(t)$. The result of the evaluation of a closed term is a \textit{value}. Values are denoted by $u, v, w$. Values are nothing else than closed terms of the following form:

\[ u, v, w ::= c \mid \lambda x . t \mid \texttt{rec } f . t \]

The relation $t \xrightarrow{\text{cv}} v$ means that the call-by-value evaluation of the closed term $t$ yields the result $v$. Similarly, the relation $t \xrightarrow{\text{cn}} v$ means the call-by-name evaluation of $t$ yields the result $v$. The relation $u v \xrightarrow{\text{cv}} w$ means that $u$ applied to $v$ yields the result $w$ under call-by-value, whereas $u t \xrightarrow{\text{cn}} w$ means that $u$ applied to the unevaluated $t$ yields the result $w$ under call-by-name. The
rules for call-by-value and call-by-name evaluation are:

\[
\begin{align*}
  s \xrightarrow{\text{ev}} v & \quad \frac{u \xrightarrow{\text{ev}} v}{s t \xrightarrow{\text{ev}} v} \\
  t \xrightarrow{\text{ev}} v & \quad \frac{u \xrightarrow{\text{ap}} w}{s t \xrightarrow{\text{ev}} v} \\
  \lambda x. t \xrightarrow{\text{ev}} v & \quad \frac{u \xrightarrow{\text{ap}} v}{s t \xrightarrow{\text{ev}} v} \\
  (\lambda x. t) u \xrightarrow{\text{ev}} v & \quad \frac{u \xrightarrow{\text{ap}} v}{s t \xrightarrow{\text{ev}} v} \\
  (\lambda x. t) u \xrightarrow{\text{ap}} v & \quad \frac{(\lambda x. t) \xrightarrow{\text{ev}} v}{s t \xrightarrow{\text{ev}} v} \\
  \text{If we forget the types and the recursively defined functions, then } t \xrightarrow{\text{ev}} v \text{ is exactly Plotkin's eval}_V(t) = v \text{ of } [16] \text{ and } t \xrightarrow{\text{ev}} v \text{ is eval}_N(t) = v.
\end{align*}
\]

### 3 The basic logic of partial terms

The syntax of the basic logic of partial terms is that of many-sorted first-order predicate calculus, extended by a definedness predicate (written as an arrow that points down) and a binary predicate \( \sqsubseteq_\tau \) for each type \( \tau \). The atomic formulas of BPT are \( t \downarrow \) and \( s \sqsubseteq_\tau t \). The meaning of \( t \downarrow \) is that \( t \) is defined, whereas the meaning of \( s \sqsubseteq_\tau t \) is that \( s \) approximates \( t \). We often omit the type superscripts. Hence, \( s \sqsubseteq_\tau t \) implies that \( s \) and \( t \) are both terms of type \( \tau \).

The formulas of BPT are generated from the atomic formulas by applying the logical connectives and quantifiers and are of the form \( \neg A \), \( A \land B \), \( A \lor B \), \( A \rightarrow B \), \( \forall x \tau A \) and \( \exists x \tau A \). The result of substituting a term \( t \) of type \( \tau \) for a variable \( x \) of the same type in \( A \) is indicated as \( A[t/x] \), or \( A(t) \) when \( A \) is written as \( A(x) \).

The partial equality \( \simeq \) as well as the strict equality \( = \) are defined symbols. For each type \( \tau \) formulas \( s^\tau \simeq t^\tau \) and \( s^\tau = t^\tau \) are defined using the basic predicates. In the following definition, the notion \( A :\equiv B \) means that \( A \) is a syntactic abbreviation for \( B \).

\[
\begin{align*}
  (3.1) \quad s \simeq t & \equiv s \sqsubseteq t \land t \sqsubseteq s \\
  (3.2) \quad s = t & \equiv s \downarrow \land t \downarrow \land s \simeq t
\end{align*}
\]

The intuitive meaning of the partial equation \( s \simeq t \) is that (i) \( s \) is defined iff \( t \) is defined and (ii) if they are both defined, then they are equal. The meaning of the strict equation \( s = t \) is that \( s \) and \( t \) are both defined and equal.\(^1\)

#### 3.1 Axioms and rules of the basic logic of partial terms

It is important to note that most of the axioms below are actually axiom schemes and \( r, s, t \) range over arbitrary terms. Some axioms are restricted to syntactic

---

\(^1\)In [18] the basic symbols of the logic are the definedness predicate \( \downarrow \) and the strict equality \( = \). The approximation relation \( \sqsubseteq_\tau \) is a defined symbol in [18]. Both approaches are equivalent.
values. By that we mean terms generated as follows:

\[ \tilde{u}, \tilde{v} ::= x \mid c \mid \lambda x.t \mid \text{rec } fx.t \]

Note that syntactic values may contain free variables.

I. Propositional axioms: All propositional tautologies.

II. Quantifier axioms: For syntactic values \( \tilde{v} \) of type \( \tau \):

\[ (3.3) \forall x^\tau A(x) \rightarrow A(\tilde{v}) \quad (3.4) A(\tilde{v}) \rightarrow \exists x^\tau A(x) \]

III. Rules of inference:

\[ (3.5) A \rightarrow B \]
\[ (3.6) A(y^\tau) \rightarrow B \] \( \quad (3.7) B \rightarrow A(y^\tau) \) \( \quad (\ast) \]

\( (\ast) \) if the variable \( y \) does not appear free in the conclusion.

IV. Definedness:

\[ (3.8) t \downarrow \rightarrow \exists x (t \simeq x), \text{ for } x \notin \text{FV}(t). \]
\[ (3.9) c \downarrow, \quad (\lambda x.t) \downarrow, \quad (\text{rec } fx.t) \downarrow \]

V. Application and abstraction:

\[ (3.10) (s t) \downarrow \rightarrow s \downarrow \land \exists x (t \simeq x), \text{ for } x \notin \text{FV}(t). \]
\[ (3.11) (\lambda x.t) \tilde{v} \simeq t[\tilde{v}/x], \text{ for syntactic values } \tilde{v}. \]

VI. Approximation:

\[ (3.12) t \sqsubseteq \tau \quad (3.13) r \sqsubseteq \tau s \land s \sqsubseteq \tau t \rightarrow r \sqsubseteq \tau t \]
\[ (3.14) s \sqsubseteq \tau t \land s \downarrow \rightarrow t \sqsubseteq s \]
\[ (3.15) s \lor \downarrow \land \forall x^\sigma (s x \sqsubseteq \tau t x) \rightarrow s \sqsubseteq \sigma \rightarrow \tau t \]

VII. Monotonicity:

\[ (3.16) s \sqsubseteq \tau \quad (3.17) s \downarrow \land t \downarrow \land \forall x^\tau (s x \sqsubseteq \tau t x) \rightarrow s \sqsubseteq \tau t \]

VIII. Least fixed points:

\[ (3.18) s \sqsubseteq \sigma \rightarrow \tau s' \land t \sqsubseteq \sigma t' \rightarrow s t \sqsubseteq \tau s' t' \]

IX. Computational induction: Let \( A(f) := \forall x (f x \downarrow \rightarrow B(x, f x)) \), where \( f \) is of type \( \sigma \rightarrow \iota \) and \( \iota \) is a basic type, or let \( A(f) := \forall x (f x \downarrow \rightarrow B(x)) \), where \( f \) is of type \( \sigma \rightarrow \tau \) and \( \tau \) is arbitrary. Then the induction scheme is:
Remark 3.1 The scheme of computational induction (3.21) is Shankar’s version of the de Bakker-Scott induction principle [17]. The formula $A(f)$ is admissible (or chain complete). This means that $A(\emptyset)$ is obviously true, and if $A(f_n)$ is true for every $f_n$ of an increasing sequence of functions then also $A(\sup_{n \in \mathbb{N}} f_n)$ is true.

3.2 Derivable principles

We summarize several basic principles that are derivable in the basic logic of partial terms. As we will see later, the principles will be valid for call-by-value as well as for call-by-name. First, we observe that the partial equality is an equivalence relation. The strict equality is also an equivalence relation but on defined terms only.

\begin{align*}
(3.22) & \quad t \simeq t \\
(3.23) & \quad s \simeq t \rightarrow t \simeq s \\
(3.24) & \quad r \simeq s \land s \simeq t \rightarrow r \simeq t \\
(3.25) & \quad t \downarrow \rightarrow t = t \\
(3.26) & \quad s = t \rightarrow t = s \\
(3.27) & \quad r = s \land s = t \rightarrow r = t
\end{align*}

Moreover, the application is compatible with the partial equality.

\begin{align*}
(3.28) & \quad s \simeq s' \land t \simeq t' \rightarrow s t \simeq s' t'
\end{align*}

The definedness predicate can be expressed using strict equality.

\begin{align*}
(3.29) & \quad t \downarrow \leftrightarrow \exists x (t = x), \text{ for } x \notin \text{FV}(t).
\end{align*}

The partial equality can also be expressed using the definedness predicate and strict equality.

\begin{align*}
(3.30) & \quad s \simeq t \leftrightarrow (s \downarrow \lor t \downarrow \rightarrow s = t)
\end{align*}

The approximation relations can be characterized for each type.

\begin{align*}
(3.31) & \quad s \sqsubseteq_t t \leftrightarrow (s \downarrow \rightarrow s = t) \\
(3.32) & \quad s \sqsubseteq_{\sigma} t \leftrightarrow (s \downarrow \rightarrow t \downarrow \land \forall x \sigma (s x \sqsubseteq_t t x))
\end{align*}

The strict equality is extensional for defined function terms.

\begin{align*}
(3.33) & \quad s \land t \downarrow \land \forall x (s x \simeq t x) \rightarrow s = t
\end{align*}

Terms are monotonic with respect to substitution.

\begin{align*}
(3.34) & \quad \forall x (s \subseteq t) \rightarrow (\lambda x.s) \subseteq (\lambda x.t) \\
(3.35) & \quad \forall f, x (s \subseteq t) \rightarrow (\text{rec } fx.s) \subseteq (\text{rec } fx.t) \\
(3.36) & \quad s \subseteq t \rightarrow r[s/x] \subseteq r[t/x]
\end{align*}
Terms that are partial equal can be substituted for each other.

\[(3.37)\] \(s \simeq t \land A(s) \rightarrow A(t)\)

The quantifier axioms can be extended from syntactic values to defined terms.

\[(3.38)\] \(\forall x A(x) \land t \downarrow \rightarrow A(t)\)
\[(3.39)\] \(A(t) \land t \downarrow \rightarrow \exists x A(x)\)

The \(\beta\)-axiom can also be extended to defined terms.

\[(3.40)\] \(s \downarrow \rightarrow (\lambda x.t) s \simeq t[s/x]\)

In the fixed point axiom (3.19) the less than or equal to relation can be replaced by equality.

\[(3.41)\] \((\lambda x.t)[\text{rec}\ fx.t/f] = (\text{rec}\ fx.t)\)

Using this equation the following \(\beta\)-axiom for recursive functions is derivable.

\[(3.42)\] \((\text{rec}\ fx.t) \hat{v} \simeq t[\hat{v}/x, \text{rec}\ fx.t/f],\) for syntactic values \(\hat{v}\).

### 3.3 Models of the basic logic of partial terms

The main goal of this article is to provide direct interpretations of the basic logic of partial terms in terms of call-by-value and call-by-name evaluation. Besides these direct interpretations there exist also models of the basic logic of partial terms based on CPOs (complete partial orders where each non-empty, directed set has a least upper bound). There are two different kinds of models which we want to sketch here briefly. The construction of the models and the corresponding adequacy theorems are explained in detail in [18]. One model uses strict functions, whereas the other model uses arbitrary monotonic functions. The functions do not have to be Scott continuous. Monotonicity suffices.

Before we describe the models we summarize some notations. The CPO consisting of all monotonic functions from \(A\) to \(B\) is denoted by \([A \rightarrow B]\). If \(A\) and \(B\) are pointed (contain a least element \(\bot\)), then a function \(f : A \rightarrow B\) is called strict, if \(f(\bot) = \bot\). The space of all strict, monotonic functions from \(A\) to \(B\) is denoted by \([A \circ \rightarrow B]\). The lift \(\hat{A}\) of a CPO \(A\) is obtained by adding a fresh bottom element \(\bot\) to \(A\). Since the space \([A \rightarrow \hat{B}]\) is isomorphic to \([\hat{A} \circ \rightarrow \hat{B}]\), strict monotonic functions from \(\hat{A}\) to \(\hat{B}\) can be identified with points in the space \([A \rightarrow B]\).

To obtain the model with strict functions, to each type \(\tau\) a CPO \(D_\tau\) is constructed. One starts with a discrete set \(D_\iota\) for each basic type \(\iota\). A function type \(\sigma \rightarrow \tau\) is interpreted as the space \(D_{\sigma \rightarrow \tau} = [D_\sigma \rightarrow (D_\tau)_\bot]\). Variables of type \(\tau\) range over elements of \(D_\tau\) and denote therefore defined elements. The denotation of a term \(t\) of type \(\tau\) is an element of the space \((D_\tau)_\bot\). The term is considered to be defined, if its denotation is different from \(\bot\). A recursive function \(\text{rec}\ fx.t\) of type \(\sigma \rightarrow \tau\) is interpreted as the least fixed point of the monotonic operator that maps a function \(a \in D_{\sigma \rightarrow \tau}\) to the function from \(D_\sigma\).
into \((D_\tau)_\perp\) that assigns to an element \(b \in D_\sigma\) the value of the term \(t\) in \((D_\tau)_\perp\) when the variable \(f\) is bound to the function \(a\) and \(x\) is bound to \(b\).

In a similar way one can construct the model with arbitrary (not necessarily strict) monotonic functions. To each basic type \(\iota\) a flat CPO \(E_\iota\) with bottom element \(\perp\) is assigned. A function type \(\sigma \rightarrow \tau\) is interpreted as the lifted CPO \(E_\sigma \rightarrow E_\tau = [E_\sigma \rightarrow E_\tau]_\perp\). Hence, each space \(E_\tau\) contains a bottom element \(\perp\).

Variables of type \(\tau\) range over elements in \(E_\tau\) and the denotation of a term \(t\) of type \(\tau\) is in \(E_\tau\). Variables can be bound to the bottom element \(\perp\), and a function applied to \(\perp\) is allowed to have a result different from \(\perp\).

### 3.4 Extensions for call-by-value and call-by-name

Two extensions of the basic logic of partial terms are introduced in [18]. The first extension \(\text{VPT}\) is adequate for call-by-value evaluation, whereas the other extension \(\text{NPT}\) is adequate for call-by-name evaluation.

\[
\text{VPT} := \text{BPT} + \forall x : \tau \downarrow \quad \text{NPT} := \text{BPT} + \exists x : \tau \neg \downarrow
\]

In \(\text{VPT}\) we postulate that variables are defined for every type. As a consequence, the formula \(\exists x : (t \simeq x)\) is equivalent to \(t \downarrow\) in \(\text{VPT}\) (if \(x\) is not free in \(t\)). In \(\text{VPT}\) one can derive from Axiom (3.10) the formula \((s t) \downarrow \rightarrow t \downarrow\). The formula means that, if an application is defined, the argument is defined, too. This is certainly true for call-by-value evaluation.

In \(\text{NPT}\) we postulate that for each type there exists undefined elements. As a consequence, the formula \(\exists x : (t \simeq x)\) is derivable in \(\text{NPT}\) (if \(x\) is not free in \(t\)). Moreover, the unrestricted quantifier axioms are derivable in \(\text{NPT}\) as well as the unrestricted \(\beta\)-Axiom \((\lambda x.t)s \simeq t[s/x]\) for arbitrary terms \(s\).

By the unrestricted quantifier axioms we mean the following formulas for arbitrary terms \(t\):

\[
\forall x : \tau A(x) \rightarrow A(t^\tau) \quad \text{and} \quad A(t^\tau) \rightarrow \exists x : \tau A(x)
\]

**Lemma 3.2** Over \(\text{BPT}\) the unrestricted quantifier axioms are equivalent to the axioms \(\exists x : \tau \neg \downarrow\) for each type \(\tau\).

**Proof.** Assume \(\exists x : \tau \neg \downarrow\) for each type \(\tau\). We show how to derive the axiom \(\forall x : \tau A(x) \rightarrow A(t)\). Assume \(\forall x : \tau A(x)\).

- Case \(t \downarrow\): We obtain \(A(t)\) by (3.38).
- Case \(\neg t \downarrow\): Let \(y^\tau\) be a variable not free in \(t\) such that \(\neg y \downarrow\). Then we have \(t \simeq y\) and obtain \(A(t)\) by the substitution principle (3.37).

The axiom \(A(t) \rightarrow \exists x : \tau A(x)\) for the existential quantifier can be derived in a similar way.

For the converse direction assume the unrestricted quantifier axioms. Consider the term \(t \equiv (\text{rec } f^{\tau \rightarrow \tau} x^{\tau}.fx)y^\tau\) of type \(\tau\). Using computational induction we can derive \(\neg t \downarrow\) as follows. Take \(A(f) \equiv \forall x : (fx \downarrow \rightarrow \bot)\). Then we have \(A(f) \rightarrow A(\lambda x.fx)\) and, by (3.21), we obtain \(A(\text{rec } fx.fx)\) and therefore \(\neg t \downarrow\).
We can apply the unrestricted axiom for the existential quantifier and obtain
\[ \exists y \tau \neg y \downarrow \]. □

Remark 3.3 If we forget the types and just consider the fragment of the logic that deals with the definedness predicate and strict/partial equality, then VPT is exactly Beeson’s logic of partial terms [1] extended by the extensionality axiom.

4 Adequacy of the logic for call-by-value

We extend the call-by-value evaluation function to a total function which returns the value ⊥ in case the evaluation of the program does no terminate. The introduction of the symbol ⊥ is not essential but it simplifies the presentation. Let \( V^\tau \) be the set of closed values of type \( \tau \) and \( \hat{V}^\tau \) the same set extended by the symbol ⊥, i.e., \( \hat{V}^\tau := V^\tau \cup \{\perp\} \). Elements of \( \hat{V}^\tau \) are denoted by \( \hat{u}, \hat{v}, \hat{w} \).

The call-by-value application function and the call-by-value evaluation function are defined as follows:

\[
\begin{align*}
\text{apply}(\hat{u}, \hat{v}) & := \begin{cases} w, & \text{if } \hat{u} \hat{v} \rightarrow_{\hat{\text{ap}}} w; \\
\perp, & \text{if there is no } w \text{ with } \hat{u} \hat{v} \rightarrow_{\text{ap}} w. \end{cases} \\
\text{eval}(t) & := \begin{cases} v, & \text{if } t \rightarrow_{\text{ev}} v; \\
\perp, & \text{if there is no } v \text{ with } t \rightarrow_{\text{ev}} v. \end{cases}
\end{align*}
\]

The functions have the following properties:

(4.1) apply(⊥, ∅) = ∅ = apply(∅, ⊥)
(4.2) apply(λx.t, v) = eval(t[v/x])
(4.3) apply(rec fx.t, v) = eval(t[v/x, rec fx.t/f])
(4.4) eval(∅) = ∅
(4.5) eval(s t) = apply(eval(s), eval(t))

Definition 4.1 The call-by-value approximation relation \( \leq_\tau \) on the set \( \hat{V}^\tau \) is defined by induction on the type \( \tau \):

\[
\begin{align*}
\hat{u} \leq_\tau \hat{v} & :\iff \hat{u} = \perp \text{ or } \hat{u} = \hat{v} \\
\hat{u} \leq_{\sigma \rightarrow \tau} \hat{v} & :\iff \hat{u} = \perp \text{ or } \\
& \hat{v} \neq \perp \text{ and } \text{apply}(\hat{u}, w) \leq_\tau \text{apply}(\hat{v}, w) \text{ for every } w \in V^\sigma
\end{align*}
\]

By an induction on types one can see that the so defined relations are pre-orderings and that \( \perp \) is the least element in \( \hat{V}^\tau \). We have:

(4.6) \( \hat{v} \leq_\tau \hat{v} \)
(4.7) If $\hat{u} \leq_{\tau} \hat{v}$ and $\hat{v} \leq_{\tau} \hat{w}$, then $\hat{u} \leq_{\tau} \hat{w}$.

(4.8) If $\hat{v} \leq_{\tau} \bot$, then $\hat{v} = \bot$.

A call-by-value environment is a finite function that assigns to variables closed values (different from $\bot$) of the same type:

$$E = \{ x_1^{\tau_1} \mapsto v_1^{\tau_1}, \ldots, x_n^{\tau_n} \mapsto v_n^{\tau_n} \}$$

An environment $E$ can be applied to a term $t$ (written $tE$) and is then understood as a simultaneous substitution. By $E[x \mapsto v]$ we denote the environment obtained from $E$ by adding the binding $x \mapsto v$ and thereby overriding an old binding for $x$ in $E$ if there exists one. If an environment $E$ covers the free variables of a formula $A$, then the call-by-value interpretation $[A]_E \in \{t, f\}$ is defined in the following way:

$$[t.]_E := \begin{cases} t, & \text{if } \text{eval}(tE) \neq \bot; \\ f, & \text{otherwise}. \end{cases}$$

$$[s \sqsubseteq t]_E := \begin{cases} t, & \text{if } \text{eval}(sE) \leq \text{eval}(tE); \\ f, & \text{otherwise}. \end{cases}$$

$$[\forall x^{\tau} A]_E := \begin{cases} t, & \text{if } [A]_{E[x \mapsto v]} = t \text{ for each } v \in V^{\tau}; \\ f, & \text{otherwise}. \end{cases}$$

$$[\exists x^{\tau} A]_E := \begin{cases} t, & \text{if there exists a } v \in V^{\tau} \text{ with } [A]_{E[x \mapsto v]} = t; \\ f, & \text{otherwise}. \end{cases}$$

For the boolean connectives the truth-value is defined using the standard truth tables of classical propositional logic.

The main goal is now to show that the axioms and rules of VPT are valid under this interpretation. Except for the monotonicity axiom (3.18), the minimality axiom (3.20) and the scheme for computational induction (3.21) this is an easy task. We first observe that the following substitution principle holds.

**Lemma 4.2 (Substitution for call-by-value)**

If $\tilde{v}$ is a syntactic value, then $[A[\tilde{v}/x]]_E = [A]_{E[x \mapsto \tilde{v}]_E}$.

**Proof.** Consider a term $t$. Then $t[\tilde{v}/x]E = t(E[x \mapsto \tilde{v}])$. Hence we have

$$\text{eval}(t[\tilde{v}/x]E) = \text{eval}(t(E[x \mapsto \tilde{v}])).$$

Thus for atomic formulas we immediately obtain

$$[t[\tilde{v}/x]]_E = [t.]_{E[x \mapsto \tilde{v}]_E} \text{ and } [(s \sqsubseteq t)[\tilde{v}/x]]_E = [(s \sqsubseteq t)]_{E[x \mapsto \tilde{v}]_E}. $$

The extension to arbitrary formulas is shown by induction on the length of a formula. □
That the quantifier axioms (3.3) and (3.4) as well as the quantifier rules (3.6) and (3.7) of BPT are valid under the call-by-value interpretation follows immediately from the substitution principle.

The additional axiom \( \forall x \tau x \downarrow \) of VPT is true, since variables are bound to values in a call-by-value environment and values evaluate to themselves and are different from \( \bot \).

For the definedness axiom \( t \downarrow \rightarrow \exists x (t \simeq x) \) (3.8), where \( x \) is not free in \( t \), assume that \( \llbracket t \rrbracket_E = t \). This means that there exists a value \( v \) such that eval(\( tE \)) = \( v \). We can take \( v \) and see that \( \llbracket t \simeq x \rrbracket_{E[v \mapsto v]} = t \). Hence, \( \exists x (t \simeq x) \) is true under \( E \).

The other definedness axioms \( c \downarrow \), \( (\lambda x.t) \downarrow \) (3.9) are true, since constants, lambda abstractions and recursively defined functions are values.

For the axiom \( (s t) \downarrow \rightarrow s \downarrow \wedge t \downarrow \) (3.10) assume that \( (s t) \downarrow \) is true under \( E \). This means that eval\((s t)E\) \( \neq \bot \). By (4.5), we have

\[
\text{eval}((s t)E) = \text{apply}(\text{eval}(sE), \text{eval}(tE)).
\]

Hence, by (4.1), it must be the case that eval\((sE)\) \( \neq \bot \) and eval\((tE)\) \( \neq \bot \). Hence, both formulas \( s \downarrow \) and \( t \downarrow \) are true under \( E \).

The validity of the \( \beta \)-axiom \( (\lambda x.t) \hat{v} \simeq t[\hat{v}/x] \) (3.11) for syntactic values \( \hat{v} \) follows from the following calculation:

\[
\text{eval}((\lambda x.t) \hat{v})E = \text{apply}(\text{eval}((\lambda x.t)E), \text{eval}(\hat{v}E)) \quad (4.5)
= \text{apply}((\lambda x.t)E, \hat{v}E) \quad (4.4)
= \text{eval}(tE[x \mapsto \hat{v}E]) \quad (4.2)
\]

\[
= \text{eval}(t[\hat{v}/x]E)
\]

The validity of the reflexivity axiom (3.12) and the transitivity axiom (3.13) for the formal symbol \( \sqsubseteq \) follows directly from the corresponding properties of the call-by-value approximation relation \( \leq \) in (4.6) and (4.7).

For the axiom \( s \sqsubseteq t \wedge s \downarrow \rightarrow t \downarrow \) (3.14) assume that eval\((sE)\) \( \leq \) eval\((tE)\) and eval\((sE)\) \( \neq \bot \). By (4.8), it follows that eval\((tE)\) \( \neq \bot \), hence the axiom is true.

For the axiom \( s \downarrow \vee s \sqsubseteq t \downarrow \) (3.15) there are two cases. If eval\((sE)\) = \( \bot \), then the formula \( s \sqsubseteq t \) is true under \( E \). If eval\((sE)\) \( \neq \bot \), then the formula \( s \downarrow \) is true under \( E \).

For the axiom \( s \sqsubseteq t \wedge t \downarrow \rightarrow s \sqsubseteq \tau \) \( s \downarrow \) (3.16) assume that eval\((sE)\) \( \leq \), eval\((tE)\) and eval\((sE)\) \( \neq \bot \). Then eval\((sE)\) = eval\((tE)\) (Def. 4.1) and hence the formula \( t \sqsubseteq s \) is true under \( E \).

For the axiom \( s \downarrow \wedge t \downarrow \rightarrow \forall x \tau (s x \sqsubseteq \tau t x) \) (3.17), where \( x \) is not free in \( s \) or \( t \), assume that eval\((sE)\) \( \neq \bot \), eval\((tE)\) \( \neq \bot \) and \( \forall x \tau (s x \sqsubseteq \tau t x) \) is true under \( E \). We have to show that eval\((sE)\) \( \leq \) eval\((tE)\). Let \( w \) be a closed
value of type $\sigma$. Then we have:

\[
\begin{align*}
\text{apply}(\text{eval}(sE), w) &= \text{apply}(\text{eval}(sE), \text{eval}(w)) \quad (4.4) \\
&= \text{eval}(sE \ w) \quad (4.5) \\
&= \text{eval}((s \ x)E[x \mapsto \ w]) \quad (x \text{ not free in } s) \\
&\leq \text{eval}((t \ x)E[x \mapsto \ w]) \\
&= \text{apply}(\text{eval}(tE), w)
\end{align*}
\]

Hence, by the definition of $\leq$ in Def. 4.1, we have $\text{eval}(sE) \leq \text{eval}(tE)$.

For the closure axiom $(\lambda x.t)[\text{rec } f x.t/f] \sqsubseteq (\text{rec } f x.t)$ (3.19) let $E$ be an environment that covers all free variables of $\text{rec } f x.t$ and let $u \equiv (\text{rec } f x.t)E$. We have to show that $(\lambda x.t)E[f \mapsto u] \leq u$. Let $w$ be a closed value. Then we have:

\[
\text{apply}((\lambda x.t)E[f \mapsto u], w) \overset{(4.2)}{=} \text{eval}(tE[f \mapsto u, x \mapsto w]) \overset{(4.3)}{=} \text{apply}(u, w)
\]

By the definition of $\leq$ in Def. 4.1, it follows that the closure axiom is true.

How can we show the validity of the monotonicity axiom (3.18)? We need to prove that the call-by-value application is monotonic in the following sense:

(4.9) If $v \leq w$, then $\text{apply}(u, v) \leq \text{apply}(u, w)$.

What about the minimality axiom (3.20) for recursively defined functions? We have to show the following property:

(4.10) If $(\lambda x.t)[v/f] \leq v$, then $(\text{rec } f x.t) \leq v$.

 Attempts to prove properties (4.9) or (4.10) by induction on the type or by induction on $\frac{ev}{v}$, $\frac{ap}{v}$ fail very soon. Therefore we inductively define two new relations $u \leq^1 v$ and $u \leq^2 v$. We will see in Lemma 4.6 below that the relation $\leq^2$ is the same as the relation $\leq$. Hence, $\leq^2$ is just another characterization of the approximation relation $\leq$. The clauses of the following definition are exactly what is needed to prove the monotonicity of the application with respect to $\leq^2$ by a simple induction on the length of the call-by-value evaluation and to prove the minimality property (4.10).

**Definition 4.3** The relations $u \leq^1 v$ and $u \leq^2 v$ are inductively generated by the following clauses:

(4.11) $c \leq^1 c$.

(4.12) If $s = (\lambda x.t)$ or $s = (\text{rec } f x.t)$, and $E(y) \leq^2 F(y)$ for each $y \in \text{FV}(s)$, then $sE \leq^1 sF$.

(4.13) If $(\lambda x.t)[v/f] \leq v$, then $(\text{rec } f x.t) \leq^1 v$.

(4.14) If $u \leq^1 v$ and $v \leq w$, then $u \leq^2 w$. 

14
The first three clauses are for the relation $\leq^1$ and the fourth clause is for the relation $\leq^2$. The relation $\leq^2$ is extended to $V^\tau$ in the obvious way such that $\bot$ is the least element. It is easy to see that both relations are reflexive. Moreover, the relation $\leq$ implies $\leq^2$ and the relation $\leq^1$ implies $\leq^2$. The goal is to show that $\leq^2$ is the same as $\leq$. We have:

(4.15) $v \leq^1 v, v \leq^2 v$.

(4.16) If $u \leq^2 v$ and $v \leq w$, then $u \leq^2 w$.

(4.17) If $u \leq v$, then $u \leq^2 v$.

(4.18) If $u \leq^1 v$, then $u \leq^2 v$.

Why do we introduce these relations and not others? Since the following technical lemma for $\leq^2$ can be proved by induction on the length of the call-by-value evaluation of a term.

**Lemma 4.4** For terms $t$, environments $E, F$ and values $u, u', v, v'$ we have:

(a) If $E(x) \leq^2 F(x)$ for each $x \in \text{FV}(t)$, then $\text{eval}(tE) \leq^2 \text{eval}(tF)$.

(b) If $u \leq^2 u'$ and $v \leq^2 v'$, then $\text{apply}(u, v) \leq^2 \text{apply}(u', v')$.

**Proof.** By induction on $t \xrightarrow{E}{v} w$ and $u \xrightarrow{\text{ap}} v w$ we simultaneously prove:

(a) If $tE \xrightarrow{E}{v} w$ and $E(x) \leq^2 F(x)$ for each $x \in \text{FV}(t)$, then $w \leq^2 \text{eval}(tF)$.

(b) If $u \xrightarrow{\text{ap}} v w$ and $u \leq^2 u'$ and $v \leq^2 v'$, then $w \leq^2 \text{apply}(u', v')$.

(a) Assume that $tE \xrightarrow{E}{v} w$ and $E(x) \leq^2 F(x)$ for each $x \in \text{FV}(t)$.

Case 1: $t = y$. Since $yE \xrightarrow{E}{v} E(y)$, we obtain that $w = E(y) \leq^2 F(y) = \text{eval}(yF)$.

Case 2: $t = c$.

Since $cE \xrightarrow{E}{v} c$, we obtain that $w = c \leq^2 c = \text{eval}(cF)$.

Case 3: $t = (\lambda x.s)$ or $t = (\text{rec} f x.s)$.

By clause (4.12) of Def. 4.3, we have $tE \leq^1 tF$. Since $tE \xrightarrow{E}{v} tE$, we have $w = tE \leq^2 tF = \text{eval}(tF)$.

Case 4: $t = (r s), rE \xrightarrow{E}{v} u, sE \xrightarrow{E}{v} v, u \xrightarrow{\text{ap}} w$.

By the induction hypothesis (a), we obtain $u \leq^2 \text{eval}(rF)$ and $v \leq^2 \text{eval}(sF)$.

By the induction hypothesis (b), we obtain

$$w \leq^2 \text{apply}([\text{eval}(rF), \text{eval}(sF)]) = \text{eval}((r s)F).$$

(b) Assume that $u \xrightarrow{\text{ap}} v w$ and $u \leq^2 u'$ and $v \leq^2 v'$. Since $u \leq^2 u'$, by clause (4.14) of Def. 4.3, there exists a value $u''$ such that $u \leq^1 u''$ and $u'' \leq u'$. Since $u \leq^1 u''$, there are two cases according to the clauses (4.12) and (4.13) in Def. 4.3.
Case 1: $u = sE$, $u'' = sF$ and $E(y) \leq^2 F(y)$ for all variables free in $s$, where $s = (\lambda x.t)$ or $s = (\text{rec } f.x.t)$. Consider the case of $s = (\text{rec } f.x.t)$:

Since $u \overset{ap}{{\rightarrow}} v$, we have $t(E[x \mapsto v, f \mapsto u]) \overset{\alpha v}{\rightarrow} w$.

The induction hypothesis (a) yields that

$$w \leq^2 \text{eval}(t(F[x \mapsto v', f \mapsto u''])) = \text{apply}(u'', v') \leq \text{apply}(u', v').$$

By Property (4.16), we obtain that $w \leq^2 \text{apply}(u', v')$.

Case 2: $u = (\text{rec } f.x.t)$ and $(\lambda x.t)[u''/f] \leq u''$.

Since $u \overset{ap}{{\rightarrow}} w$, we have $t[v/x, u/f] \overset{\alpha v}{\rightarrow} w$.

The induction hypothesis (a) yields that

$$w \leq^2 \text{eval}(t[v'/x, u''/f]) = \text{apply}((\lambda x.t)[u''/f], v') \leq \text{apply}(u', v').$$

In the last step we use the fact that $(\lambda x.t)[u''/f] \leq u'' \leq u'$. □

The following lemma is proved by induction on the type of the values $u$ and $v$.

**Lemma 4.5** If $u \leq^2 v$, then $u \leq v$.

**Proof.** By induction on the type of $u$. Assume that $u \leq^2 v$. By clause (4.14) of Def. 4.3, there exists a value $u'$ such that $u \leq^1 u'$ and $u' \leq v$.

Assume that $u$ is of basic type. Then $u$ is a constant and $u = u'$. Thus, $u \leq v$.

Assume that $u$ is of type $\sigma \rightarrow \tau$. Let $w$ be a closed value of type $\sigma$. By the previous Lemma 4.4 it follows that $\text{apply}(u, w) \leq^2 \text{apply}(v, w)$. Since $\text{apply}(u, w)$ is of type $\tau$, by the induction hypothesis, it follows that $\text{apply}(u, w) \leq \text{apply}(v, w)$. Since the term $w$ has been chosen arbitrarily, by the definition of $\leq_{\sigma \rightarrow \tau}$, we obtain $u \leq v$. □

We have shown that the relation $\leq^2$ is the same as the original relation $\leq$.

**Lemma 4.6** $\hat{u} \leq^2 \hat{v} \iff \hat{u} \leq \hat{v}$

Now we can show that the monotonicity axiom (3.18) is valid. Assume that $\text{eval}(sE) \leq \text{eval}(s'E)$ and $\text{eval}(tE) \leq \text{eval}(t'E)$. Then we have

$$\text{eval}((s \ t)E) = \text{apply}(\text{eval}(sE), \text{eval}(tE)) \leq \text{apply}(\text{eval}(s'E), \text{eval}(t'E)) \leq \text{eval}((s' \ t')E) \quad (4.1), \ \text{Lemmas 4.4 and 4.6}$$

Hence, the axiom $s \subseteq s' \wedge t \subseteq t' \rightarrow s \ t \subseteq s' \ t'$ (3.18) is valid.

In order to show that the minimality axiom (3.20) is valid assume that $(\lambda x.t)[\hat{v}/f] \subseteq \hat{v}$ is true under $E$. Hence, $(\lambda x.t)E[f \mapsto \hat{v}] \leq \hat{v} E$. By (4.13) of Def. 4.3, we have $(\text{rec } f.x.t)E \leq^1 \hat{v} E$. By (4.18) and Lemma 4.5, it follows that $(\text{rec } f.x.t)E \leq \hat{v} E$ and thus $(\text{rec } f.x.t) \subseteq \hat{v}$ is true under $E$. Hence, the minimality axiom (3.20) is valid.
The scheme of computational induction (3.21),
\[ \forall f \,(A \to A[\lambda x.t/f]) \to A[\text{rec} \,f \cdot x \cdot t/f], \]
has two versions. In the first version, \( A \) is of the form \( \forall x \,(f \,x \downarrow \to B[f \,x/y]) \)
where \( f \) is not free in \( B \) and \( f \) is of type \( \sigma \to \iota \) with a basic type \( \iota \). In the second version, \( A \) is of the form \( \forall x \,(f \,x \downarrow \to B) \) where \( f \) is not free in \( B \) and there is no restriction on the type of \( f \).

Take an instance of the scheme and assume that \( \forall f \,(A \to A[\lambda x.t/f]) \) is true under \( E \). Define the terms \( \text{rec}_n \) as follows:
\[ \text{rec}_0 : = (\text{rec} \,f \cdot x \cdot x) \quad \text{and} \quad \text{rec}_{n+1} : = (\lambda x.t) E[f \mapsto \text{rec}_n] \quad \text{for } n \in \mathbb{N} \]
We show that the formula \( A \) is true under \( E[f \mapsto \text{rec}_n] \) by induction on \( n \). The base case follows from the fact that the function \( \text{rec}_0 \) is everywhere undefined and therefore \( \text{eval}(\text{rec}_0 \,v) = \bot \) for each value \( v \). Hence, \( A \) is true under \( E[f \mapsto \text{rec}_0] \).

In the induction step, we assume that \( A \) is true under \( E[f \mapsto \text{rec}_n] \). Since \( \forall f \,(A \to A[\lambda x.t/f]) \) is true under \( E \), it follows that \( A[\lambda x.t/f] \) is true under \( E[f \mapsto \text{rec}_n] \). By the Substitution Lemma 4.2, it follows that \( A \) is true under \( E[f \mapsto (\lambda x.t) E[f \mapsto \text{rec}_n]] \), which means that \( A \) is true under \( E[f \mapsto \text{rec}_{n+1}] \).

We have to show that \( A[\text{rec} \,f \cdot x \cdot t/f] \) is true under \( E \). We first consider the case where \( A \) is the formula \( \forall x \,(f \,x \downarrow \to B[f \,x/y]) \), \( f \) is not free in \( B \) and the type of \( f \) is \( \sigma \to \iota \). Assume that \( v \) is a value such that \( \text{eval}((\text{rec} \,f \cdot x \cdot t) E \,v) \neq \bot \). Then there exists a value \( w \) such that \( (\text{rec} \,f \cdot x \cdot t) E \,v \xrightarrow{\sigma} w \). Since the value \( w \) is of basic type \( \iota \), it must be a constant. By Lemma 4.7 below, there exists an \( n \in \mathbb{N} \) such that \( \text{rec}_n \,v \xrightarrow{\sigma} w \). Since \( A \) is true under \( E[f \mapsto \text{rec}_n] \) and \( \text{eval}(\text{rec}_n \,v) \neq \bot \), the formula \( B[f \,x/y] \) is true under \( E[f \mapsto \text{rec}_n, x \mapsto v] \). Since \( f \,x \simeq (\text{rec} \,f \cdot x \cdot t) \,x \) is true under \( E[f \mapsto \text{rec}_n, x \mapsto v] \), we can apply the substitution principle (3.37) (which is derivable in BPT without computational induction) and obtain that the formula \( B[(\text{rec} \,f \cdot x \cdot t) \,x/y] \) is true under \( E[x \mapsto v] \). Since \( v \) has been chosen arbitrarily, it follows that \( A[\text{rec} \,f \cdot x \cdot t/f] \) is true under \( E \).

In the second case, \( A \) is the formula \( \forall x \,(f \,x \downarrow \to B) \), where \( f \) is not free in \( B \). Let \( v \) be a value such that \( \text{eval}((\text{rec} \,f \cdot x \cdot t) E \,v) \neq \bot \). We have to show that \( B \) is true under \( E[x \mapsto v] \). By Lemma 4.7 below, there exists an \( n \in \mathbb{N} \) such that \( \text{eval}(\text{rec}_n \,v) \neq \bot \). Since \( A \) is true under \( E[f \mapsto \text{rec}_n] \), it follows that \( B \) is true under \( E[x \mapsto v] \). Hence, \( A[\text{rec} \,f \cdot x \cdot t/f] \) is true under \( E \).

It remains to show that recursively defined functions can be approximated by finite unfoldings of the definition. Let \( \text{rec} \,g \cdot z \cdot x \) be a closed term. We define the finite unfoldings \( \text{rec}_n \) of \( \text{rec} \,g \cdot z \cdot x \) as follows:
\[ \text{rec}_0 : = \text{rec} \,g \cdot z \cdot g \quad \text{and} \quad \text{rec}_{n+1} : = (\lambda z.t) \,[\text{rec}_n / g] \quad \text{for } n \in \mathbb{N} \]
Hence, \( \text{rec}_0 \) is an everywhere undefined function and \( \text{rec}_{n+1} \) is obtained by substituting the term \( \text{rec}_n \) in \( \lambda z.t \) for the variable \( g \). By Lemma 4.4 (a) and Lemma 4.6 we obtain:
\[ \text{rec}_0 \leq \text{rec}_1 \leq \ldots \leq \text{rec}_n \leq \ldots \leq \text{rec} \,g \cdot z \cdot x \]
Let \( t/n \) be the result of replacing in \( t \) all occurrences of \( \text{rec} \,g \cdot z \cdot x \) by \( \text{rec}_n \). For ground terms \( t \) we obtain by Lemma 4.4 (a) and Lemma 4.6:
If $m \leq n$, then $\text{eval}(t|m) \leq \text{eval}(t|n)$.

(4.20) $\text{eval}(t|n) \leq \text{eval}(t)$ for every $n \in \mathbb{N}$.

If a closed term $t$ has value $v$, then in the computation from $t$ to $v$ only finitely many unfoldings of $\text{rec gz.r}$ are used. These unfoldings can already be made at the beginning as the following lemma shows.

**Lemma 4.7 (Unfolding)** Let $t$ be a closed term.

(a) If $t \xrightarrow{\text{ev}} v$, then $\forall m \in \mathbb{N} \exists n \in \mathbb{N} \ (v|m \leq \text{eval}(t|n))$.

(b) If $u v \xrightarrow{\text{ap}} w$, then $\forall m \in \mathbb{N} \exists n \in \mathbb{N} \ (w|m \leq \text{apply}(u|n, v|n))$.

**Proof.** By simultaneous induction on the definition of $t \xrightarrow{\text{ev}} v$ and $u v \xrightarrow{\text{ap}} w$.

**Case 1:** $v \xrightarrow{\text{ev}} v$. We have $v|m = \text{eval}(v|m)$

**Case 2:** $s \xrightarrow{\text{ev}} u$, $t \xrightarrow{\text{ev}} v$, $u v \xrightarrow{\text{ap}} w$, $s t \xrightarrow{\text{ev}} w$. By the induction hypothesis there exists an $i$ such that $w|m \leq \text{apply}(u|i, v|i)$. Again, by the induction hypothesis, there exist $j$ and $k$ such that $u|i \leq \text{eval}(s|j)$ and $v|i \leq \text{eval}(t|k)$. Let $n = \max(j, k)$. Then we have:

\[
\begin{align*}
v|m & \leq \text{apply}(u|i, v|i) \\
& \leq \text{apply}(\text{eval}(s|j), \text{eval}(t|k)) \quad \text{Lemma 4.4 (b)} \\
& \leq \text{apply}(\text{eval}(s|n), \text{eval}(t|n)) \quad (4.19) \\
& = \text{eval}((s t)|n) \quad (4.5)
\end{align*}
\]

**Case 3:** $r[\text{rec gz.r}/g, u/z] \xrightarrow{\text{ev}} v$, $u v \xrightarrow{\text{ap}} w$. By the induction hypothesis there exists an $n$ such that $v|m \leq \text{eval}(r[\text{rec}_n/g, (u|n)/z])$. We obtain

\[
\begin{align*}
v|m & \leq \text{eval}(r[\text{rec}_n/g, (u|n)/z]) \\
& = \text{apply}((\lambda z. r)[\text{rec}_n/g], u|n) \quad (4.2) \\
& = \text{apply}((\text{rec}_{n+1}, u|n) \quad \text{Def. of rec}_{n+1} \\
& \leq \text{apply}(\text{rec}_{n+1}, u|n + 1) \quad (4.19) \text{ and Lemma 4.4 (b)}
\end{align*}
\]

**Case 4:** $t[\text{rec fx.t}/f, u/x] \xrightarrow{\text{ev}} v$, $u v \xrightarrow{\text{ap}} v$ and the term $\text{rec fx.t}$ is different from $\text{rec gz.r}$. By the induction hypothesis there exists an $n$ such that $v|m \leq \text{eval}(t|n[\text{rec fx.t}/f, (u|n)/x]) = \text{apply}((\text{rec fx.t}|n, u|n))$. □

The preceding lemma concludes the proof that the axioms and rules of VPT are valid under the call-by-value interpretation. We have shown the following theorem.

**Theorem 4.8** If a formula $A$ is derivable in VPT, then $[A]_E = t$ for every call-by-value environment $E$ that covers the free variables of $E$.  

18
Once we know that the system VPT is sound with respect to the call-by-value interpretation, we can easily show that VPT is adequate for call-by-value evaluation. By adequacy we mean the two properties (1.1) and (1.2) mentioned in the introduction.

**Theorem 4.9** The logic VPT is adequate for call-by-value evaluation.

**Proof.** Let \( t \) be a closed program.

Assume that the formula \( t \downarrow \) is derivable in VPT. Then \( t \downarrow \) is true under the call-by-value interpretation which means that \( \text{eval}(t) \neq \perp \). Hence, the evaluation of \( t \) terminates.

Assume that \( t \) is of basic type and that the equation \( t = v \) is derivable in VPT for a value \( v \). Then the formula \( v \sqsubseteq t \) is derivable in VPT as well. Hence, the formula \( v \sqsubseteq t \) is true under the call-by-value interpretation and \( v \leq \text{eval}(t) \).

Since \( v \) is of a basic type, this means that \( v = \text{eval}(t) \). Hence, the evaluation of \( t \) terminates with result \( v \).

For the converse directions the following statements can be shown by induction on \( t \xrightarrow{ev} v \) and \( u \xrightarrow{ap} w \):

(a) If \( t \xrightarrow{ev} v \), then \( t = v \) is derivable in BPT.

(b) If \( u \xrightarrow{ap} w \), then \( u \xrightarrow{ap} w \) is derivable in BPT.

The principles (3.11), (3.28) and (3.42) are used in the derivations. \( \Box \)

### 5 Adequacy of the logic for call-by-name

The call-by-name evaluation function is extended to a total function. We use the same names and symbols as in the previous section. The names and symbols, however, now refer to call-by-name evaluation.

\[
\text{eval}(t) := \begin{cases} v, & \text{if } t \xrightarrow{ev} v; \\ \perp, & \text{if there is no } v \text{ with } t \xrightarrow{ev} v. \end{cases}
\]

\[
\text{apply}(\hat{u}, t) := \begin{cases} v, & \text{if } \hat{u} t \xrightarrow{ap} v; \\ \perp, & \text{if there is no } v \text{ with } \hat{u} t \xrightarrow{ap} v. \end{cases}
\]

The call-by-name evaluation function and the call-by-name application function have the following properties:

\begin{align}
(5.1) \quad & \text{apply}(\perp, t) = \perp \\
(5.2) \quad & \text{apply}(\lambda x.t, s) = \text{eval}(t[s/x]) \\
(5.3) \quad & \text{apply}(\text{rec } f x.t, s) = \text{eval}(t[s/x, \text{rec } f x.t/f]) \\
(5.4) \quad & \text{eval}(v) = v
\end{align}
(5.5) \( \text{eval}(s \ t) = \text{apply}(\text{eval}(s), t) \)

The difference between the call-by-name approximation relation in the following definition and the call-by-value approximation relation in Def. 4.1 is that for functions one has to take into account arbitrary closed terms as arguments and not only values.

**Definition 5.1** The call-by-name approximation relation \( \leq \tau \) is defined on the set \( \hat{V}^\tau \) by induction on the type \( \tau \):

\[
\hat{u} \leq_e \hat{v} \iff \hat{u} = \bot \text{ or } \hat{u} = \hat{v}
\]

\[
\hat{u} \leq_{\sigma \rightarrow \tau} \hat{v} \iff \hat{u} = \bot \text{ or } \hat{v} \neq \bot \text{ and } \text{apply}(\hat{u}, t) \leq \tau \text{ apply}(\hat{v}, t) \text{ for every closed term } t^\sigma
\]

An induction on types yields that the so defined relations are pre-orderings and that \( \bot \) is the least element in \( \hat{V}^\tau \).

A call-by-name environment is a finite function that assigns closed terms (and not just values) to variables.

\[ E = \{ x_1^{\tau_1} \mapsto s_1^{\tau_1}, \ldots, x_n^{\tau_n} \mapsto s_n^{\tau_n} \} \]

Since the evaluation of a closed term may diverge, a call-by-name environment can assign undefined terms to variables. The truth value \([A]_{E} \in \{t, f\}\) is defined as in the call-by-value case except that the quantifiers now range over arbitrary closed terms and not only over values:

\[
[\forall x^\tau A]_{E} := \begin{cases} t, & \text{if } [A]_{E[x^\tau \mapsto t]} = t \text{ for each closed term } t^\tau; \\ f, & \text{otherwise}. \end{cases}
\]

\[
[\exists x^\tau A]_{E} := \begin{cases} t, & \text{if there exists a closed term } t^\tau \text{ with } [A]_{E[x^\tau \mapsto t]} = t; \\ f, & \text{otherwise}. \end{cases}
\]

The main goal is to show that all axioms and rules of NPT are valid under this interpretation. The following substitution principle for the call-by-name interpretation is more general than the corresponding principle of the call-by-value interpretation (Lemma 4.2). The substitution principle is not restricted to syntactic values.

**Lemma 5.2 (Substitution for call-by-name)**

If \( t \) is a term, then \([A[t/x]]_{E} = [A]_{E[x^\tau \mapsto t]} \).

The proof that the axioms and rules of NPT are valid under the call-by-name interpretation goes similar to the call-by-value case. To show that the axiom \( \exists x^\tau \neg x \) of NPT is valid we consider the following closed term of type \( \tau \):

\[
(\text{rec} \ f^{(x^\tau \mapsto t, x^\tau \mapsto t)} \ 0 \ t \ x) (\lambda y^\tau \cdot y)
\]
The evaluation of this term does not terminate. Hence, if we bind the variable \( x \) to this term, then the formula \( x \downarrow \) becomes false and therefore \( \exists x^r \neg x \downarrow \) is true.

For the monotonicity axiom (3.18) and the minimality axiom (3.20) we introduce the auxiliary relations \( u \leq_1 v, u \leq_2 v \) and \( s \leq_3 t \). We will show in (5.18) below that the relation \( \leq_2 \) is the same as the call-by-name approximation relation \( \leq \). Moreover, we will see in (5.19) that the relation \( s \leq_3 t \) holds exactly if \( \text{eval}(s) \leq \text{eval}(t) \).

**Definition 5.3** The relations \( u \leq_1 v, u \leq_2 v \) and \( s \leq_3 t \) are inductively generated by the following clauses:

(5.6) \( c \leq_1 c \).

(5.7) If \( s = (\lambda x. t) \) or \( s = (\text{rec } f x. t) \) and \( E(y) \leq_3 F(y) \) for each \( y \in \text{FV}(s) \), then \( sE \leq_1 sF \).

(5.8) If \( (\lambda x. t)[v/f] \leq_1 v \), then \( \text{rec } f x. t \leq_1 v \).

(5.9) If \( u \leq_1 v \) and \( v \leq w \), then \( u \leq_2 w \).

(5.10) If \( \text{eval}(s) \leq_2 \text{eval}(t) \), then \( s \leq_3 t \).

(5.11) If \( s \leq_3 s' \) and \( t \leq_3 t' \), then \( s \leq_3 t' \).

The first three clauses are for the relation \( \leq_1 \), the fourth clause is for the relation \( \leq_2 \) and the last two clauses are for \( \leq_3 \). It is easy to see that all three relations are reflexive. Moreover, the relation \( \leq \) implies \( \leq_2 \), the relation \( \leq_1 \) implies \( \leq_2 \), and the relation \( \leq_2 \) implies \( \leq_3 \).

(5.12) \( v \leq_1 v, v \leq_2 v, t \leq_3 t \)

(5.13) If \( u \leq_2 v \) and \( v \leq w \), then \( u \leq_2 w \).

(5.14) If \( u \leq v \), then \( u \leq_2 v \).

(5.15) If \( u \leq_1 v \), then \( u \leq_2 v \).

(5.16) If \( u \leq_2 v \), then \( u \leq_3 v \).

The following substitution property can be shown by induction on the length of a term \( t \):

(5.17) If \( E(x) \leq_3 F(x) \) for each \( x \in \text{FV}(t) \), then \( tE \leq_3 tF \).

The main technical lemma is proved by induction on the length of the call-by-name evaluation of a term.

**Lemma 5.4** For closed terms \( s, t \) and closed values \( u, v \) we have:

(a) If \( s \leq_3 t \), then \( \text{eval}(s) \leq_2 \text{eval}(t) \).

(b) If \( u \leq_2 v \) and \( s \leq_3 t \), then \( \text{apply}(u, s) \leq_2 \text{apply}(v, t) \).
Proof. By induction on $s \xrightarrow{\text{cv}} w$ and $u \xrightarrow{\text{ap}} w$ we prove:

(a) If $t \xrightarrow{\text{cv}} w$ and $t \leq^3 t'$, then $w \leq^2 \text{eval}(t')$.

(b) If $u \xrightarrow{\text{ap}} w$ and $u \leq^2 u'$ and $t \leq^3 t'$, then $w \leq^2 \text{apply}(u', t')$.

(a) Assume that $t \xrightarrow{\text{cv}} w$ and $t \leq^3 t'$. Since $t \leq^3 t'$, there are two cases according to the clauses $(5.10)$ and $(5.11)$ in Def. 5.3.

Case 1: $\text{eval}(t) \leq^2 \text{eval}(t')$.

Since $w = \text{eval}(t)$, we obtain $w \leq^2 \text{eval}(t')$.

Case 2: $t = (r \ s), t' = (r' \ s')$, $r \leq^3 r'$, $s \leq^3 s'$.

Since $t \xrightarrow{\text{cv}} w$ there exists a value $u$ such that $r \xrightarrow{\text{cv}} u$ and $u \xrightarrow{\text{ap}} w$. By the induction hypothesis, we obtain that $u \leq^2 \text{eval}(r')$ and again, by the induction hypothesis,

$$w \leq^2 \text{apply}(\text{eval}(r'), s') \overset{(5.5)}{=} \text{eval}(r' \ s') = \text{eval}(t').$$

(b) Assume that $u \xrightarrow{\text{ap}} w$ and $u \leq^2 u'$ and $t \leq^3 t'$. Since $u \leq^2 u'$, by clause $(5.9)$ of Def. 5.3, there exists a value $v$ such that $u \leq^1 v$ and $v \leq u'$. Since $u \leq^1 v$, there are two cases according to the clauses $(5.7)$ and $(5.8)$ in Def. 5.3.

Case 1: $u = sE, v = sF$ and $E(y) \leq^3 F(y)$ for all variables free in $s$, where $s = (\lambda x. t)$ or $s = (\text{rec } f x. t)$. Consider the case of $s = (\text{rec } f x. t)$:

Since $u \xrightarrow{\text{ap}} w$, we have $r[E[x \mapsto t, f \mapsto u]) \xrightarrow{\text{cv}} w$.

By the substitution property $(5.17)$, we obtain that

$$r(E[x \mapsto t, f \mapsto u]) \leq^3 r(F[x \mapsto t', f \mapsto u']).$$

The induction hypothesis yields that

$$w \leq^2 \text{eval}(r(F[x \mapsto t', f \mapsto u'])) \overset{(5.3)}{=} \text{apply}(v, t') \leq \text{apply}(u', t').$$

In the last step we use Def. 5.1 and $v \leq u'$.

Case 2: $u = (\text{rec } f x. s)$ and $(\lambda x. s)[v/f] \leq v$:

Since $u \xrightarrow{\text{ap}} w$, we have $s[t/x, u/f] \xrightarrow{\text{cv}} w$.

By the substitution property $(5.17)$, we obtain that

$$s[t/x, u/f] \leq^3 s[t'/x, v/f].$$

The induction hypothesis (a) yields that

$$w \leq^2 \text{eval}(s[t'/x, v/f]) \overset{(5.2)}{=} \text{apply}((\lambda x. s)[v/f], t') \leq \text{apply}(u', t').$$

In the last step we use Def. 5.1 and the fact that $(\lambda x. s)[v/f] \leq v \leq u'$. □
The following lemma is proved by induction on the type of the value $u$.

**Lemma 5.5** If $u \leq^2 v$, then $u \leq v$.

**Proof.** Assume that $u \leq^2 v$. By clause (5.9) of Def. 5.3, there exists a value $u'$ such that $u \leq^1 u'$ and $u' \leq v$.

Assume that $u$ is of basic type. Then $u$ is a constant and $u = u'$. Thus, $u \leq v$.

Assume that $u$ is of type $\sigma \rightarrow \tau$. Let $t$ be a closed term of type $\sigma$. By Lemma 5.4 it follows, that $\text{apply}(u, t) \leq^2 \text{apply}(v, t)$. Since $\text{apply}(u, t)$ is of type $\tau$, by the induction hypothesis, it follows that $\text{apply}(u, t) \leq \text{apply}(v, t)$. Since the term $t$ has been chosen arbitrarily, by the definition of $\leq_{\sigma \rightarrow \tau}$ in Def. 5.1, we obtain $u \leq v$. $\Box$

We have shown that the relation $\leq^2$ is the same as the original relation $\leq$:

\[(5.18) \quad \hat{u} \leq^2 \hat{v} \iff \hat{u} \leq \hat{v}\]

We can use this fact together with Lemma 5.4 to obtain the following characterization of the relation $\leq^3$:

\[(5.19) \quad s \leq^3 t \iff \text{eval}(s) \leq \text{eval}(t)\]

Now we can show that the monotonicity axiom (3.18) is valid. Assume that $\text{eval}(sE) \leq \text{eval}(s'E)$ and $\text{eval}(tE) \leq \text{eval}(t'E)$. Then we have

\[
\text{eval}((s \ t)E) = \text{apply}(\text{eval}(sE), tE) \quad (5.5)
\]

\[
\leq \text{apply}(\text{eval}(s'E), t'E) \quad (5.18), (5.19), \text{Lemma } 5.4
\]

\[
= \text{eval}((s' \ t')E) \quad (5.5)
\]

Hence, the monotonicity axiom $s \subseteq s' \land t \subseteq t' \rightarrow s \ t \subseteq s' \ t'$ is valid.

For the minimality axiom (3.20) assume that $(\lambda x.t)[\hat{v}/f] \subseteq \hat{v}$ is true under $E$. Hence, $(\lambda x.t)E[f \mapsto \hat{v}] \leq \text{eval}(\hat{v}E)$. Let $v = \text{eval}(\hat{v}E)$. By (5.19) and Lemma 5.4, it follows that

\[
(\lambda x.t)E[f \mapsto v] \leq (\lambda x.t)E[f \mapsto \hat{v}] \leq v.
\]

By (5.8), we have $(\text{rec } fx.t)E \leq^1 v$ and, by (5.15) and (5.18) $(\text{rec } fx.t)E \leq v$.

Hence, $(\text{rec } fx.t) \subseteq \hat{v}$ is true under $E$.

That the scheme of computational induction (3.21) is true under the call-by-name interpretation can be shown by a similar argument as in the call-by-value case using a call-by-name version of Lemma 4.7 about the unfolding of recursively defined functions. Thus, we can conclude with the main theorem.

**Theorem 5.6** If a formula $A$ is derivable in NPT, then $[A]_E = t$ for every call-by-name environment $E$ that covers the free variables of $E$. 

---

23
For the embedding of call-by-name computations into NPT we use the unrestricted version of the \(\beta\)-Axiom (3.11)

\[(\lambda x.t) s \simeq t[s/x]\]

and the unrestricted version of (3.42)

\[(\text{rec} fx.t) s \simeq t[s/x, \text{rec}\, fx.t/f]\]

for arbitrary terms \(s\) and obtain the following theorem.

**Theorem 5.7** The logic NPT is adequate for call-by-name evaluation.

As a corollary we obtain the following embedding of call-by-value evaluation into call-by-name evaluation: If a closed term \(t\) terminates with result \(v\) in call-by-value evaluation, then \(t\) terminates also in call-by-name evaluation with a result \(w\) such that \(v \leq w\) and \(w \leq v\) (with the relation \(\leq\) of Def. 5.1). This can be seen as follows. If \(t\) terminates with result \(v\) in call-by-value evaluation, then the equation \(t = v\) is derivable in the basic system BPT (see proof of Theorem 4.9). Hence, the equation \(t = v\) is true under the call-by-name interpretation which means that \(\text{eval}(t) \leq v\) and \(v \leq \text{eval}(t)\), where \(\text{eval}(t)\) is the result of the call-by-name evaluation of the program \(t\).

### 5.1 A comparison of the two cases

In both cases the adequacy of the axioms is shown by interpreting the formulas in the standard term models for call-by-value and call-by-name evaluation. In the case of call-by-value the environment \(E\) in \([A]_{E}\) assigns closed values to the free variables of the formula \(A\), whereas in the case of call-by-name, the environment \(E\) assigns arbitrary closed terms (including application terms which may diverge) to the variables.

In the definition of the approximation relation \(\leq\) for call-by-value in Def. 4.1 a term \(u\) of a function type approximates a term \(v\) of the same type if for each closed value \(w\) of the argument type, \(u\) applied to \(w\) approximates \(u\) applied to \(w\). In the case of call-by-name in Def. 5.1, one has to quantify over arbitrary closed terms instead.

In both cases the truth of most of the axioms in the standard term model can be established easily. The difficulties arise for the monotonicity axiom (3.18) of function application and the second fixed-point axiom (3.20) that says that functions defined by recursion are minimal. The problem is that in order to show that a recursive function \(\text{rec}\, fx.t\) is monotonic, one has already to know that it is monotonic, since the same term \(\text{rec}\, fx.t\) is substituted for \(f\) when the function is applied to an argument.

To break out of this vicious circle, auxiliary relations \(\leq^1\) and \(\leq^2\) (resp. \(\leq^3\)) are inductively defined in Def. 4.3 and 5.3. The auxiliary relations are defined in such a way that the monotonicity of the application with respect to \(\leq^2\) can be proved by a simple induction on the length of the evaluation. In a second lemma,
one shows that the relation \( \leq^2 \) is the same as the approximation relation \( \leq \) by a simple induction on types, hence the application is also monotonic with respect to \( \leq \).

6 Observational equivalence

The basic logic of partial terms BPT and its extensions VPT and NPT are extensional. Two programs are considered to be equal, if they have the same domain and produce equal outputs when they are applied to arguments, i.e.

\[
\downarrow s \land \downarrow t \to (s = t \iff \forall x (s x \simeq t x)).
\]

Thereby, the strict equality (=) and partial equality (\( \simeq \)) of programs is defined in terms of a more basic approximation relation (\( \sqsubseteq \)) which is extensional, too:

\[
\downarrow s \land \downarrow t \land \forall x (s x \sqsubseteq t x) \to s \sqsubseteq t
\]

In the call-by-value and call-by-name interpretations of the logic, the approximation relation \( \sqsubseteq \) is interpreted by an extensional relation \( \leq \). The relation \( \leq \) is defined by induction on the types of the programs (Def. 4.1 for call-by-value evaluation and Def. 5.1 for call-by-name evaluation).

Another possibility is to consider two programs to be equal, if one program can be replaced by the other in any context without changing the functional behavior of the overall program. In other words, two programs are equal, if no difference can be observed from outside when one subprogram is replaced by the other. This kind of equality is called observational equality.

For simply typed programs, the extensional equality and the observational equality are the same. This is known as the context lemma and has been proved for the first time by R. Milner in [12] for simply typed combinators and call-by-name evaluation using an analysis of the reductions of terms. In the following we give a proof of the context lemma for simply typed programs that include also recursion. We use the call-by-value interpretation and the call-by-name interpretation of the previous sections and the technical lemmas about the monotonicity of the relation \( \leq \). We focus on the call-by-value case. The call-by-name version of the context lemma can be proved in the same way.

A context \( C \) is a typed term that contains exactly one occurrence of the (typed) star symbol. The term \( C[t] \) is obtained from \( C \) by replacing the star symbol by the term \( t \). Free variables of \( t \) can thereby become bound by the context. A context \( C \) closes a term \( t \), if \( C[t] \) is closed.

**Definition 6.1 (Observational approximation)** A term \( s \) observationally approximates \( t \) (written \( s \preceq_{\text{obs}} t \)), if for each closing context \( C \) the following two conditions hold:

(6.1) If eval\((C[s])\) \(\neq \perp\), then eval\((C[t])\) \(\neq \perp\).

(6.2) If \( C[s] \) is of ground type and eval\((C[s])\) \(\neq \perp\), then eval\((C[s]) = \text{eval}(C[t])\).
Two terms $s$ and $t$ are observationally equivalent, if $s \preceq_{\text{obs}} t$ and $t \preceq_{\text{obs}} s$.

Several properties of the observationally approximation relation are easy to prove. For example, the relation $\preceq_{\text{obs}}$ is preserved by contexts.

(6.3) If $s \preceq_{\text{obs}} t$, then $C[s] \preceq_{\text{obs}} C[t]$.

For an environment $E$ a canonical context $C_E$ is defined as follows:

(6.4) If $E = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, then $C_E = (\lambda x_1 \cdots \lambda x_n. *) v_1 \cdots v_n$.

Obviously, the evaluation of a term $t$ in an environment $E$ yields the same result as the evaluation of $t$ in the canonical context $C_E$.

(6.5) $\text{eval}(tE) = \text{eval}(C_E[t])$ for every closing environment $E$ of $t$

The following context lemma says that a program $s$ observationally approximates $t$ iff the formula $s \sqsubseteq t$ is true under the call-by-value interpretation.

Lemma 6.2 (Context) Let $s$ and $t$ be two terms of the same type. Then the following statements are equivalent:

(a) $s \preceq_{\text{obs}} t$

(b) For each closing environment $E$, $\text{eval}(sE) \leq \text{eval}(tE)$.

Proof. The implication (a) $\Rightarrow$ (b) is shown by induction on types. Assume that $s \preceq_{\text{obs}} t$. Let $E$ be a closing environment for $s$ and $t$. We have to show that $\text{eval}(sE) \leq \text{eval}(tE)$.

Case 1. The terms $s$ and $t$ are of basic type: Then $C_E[s]$ and $C_E[t]$ are of basic type, too. Assume that $\text{eval}(sE) \neq \bot$. Then $\text{eval}(C_E[s]) \neq \bot$ and, by the definition of the relation $\preceq_{\text{obs}}$, $\text{eval}(C_E[s]) = \text{eval}(C_E[t])$. Hence, by Def. 4.1, we obtain $\text{eval}(sE) \leq \text{eval}(tE)$.

Case 2. The terms $s$ and $t$ are of type $\sigma \rightarrow \tau$: If $\text{eval}(sE) = \bot$, there is nothing to show. Otherwise, $\text{eval}(C_E[s]) = \text{eval}(sE) \neq \bot$ and, since $s \preceq_{\text{obs}} t$, $\text{eval}(tE) = \text{eval}(C_E[t]) \neq \bot$. Let $u = \text{eval}(sE)$ and $v = \text{eval}(tE)$. We have to show that $\text{apply}(u, w) \leq \text{apply}(v, w)$ for each value $w$ of type $\sigma$. By (6.3), it follows that $C_E[s][w] \preceq_{\text{obs}} C_E[t][w]$. Since $C_E[s][w]$ and $C_E[t][w]$ are both of type $\tau$, it follows, by the induction hypothesis, that $\text{eval}(C_E[s][w]) \leq \text{eval}(C_E[t][w])$. We obtain:

$$\text{apply}(u, w) = \text{apply}(\text{eval}(sE), w)$$
$$= \text{apply}(\text{eval}(C_E[s]), w) \quad (6.5)$$
$$= \text{eval}(C_E[s][w]) \quad (4.5)$$
$$\leq \text{eval}(C_E[t][w]) \quad \text{(induction hypothesis)}$$
$$= \text{apply}(\text{eval}(tE), w) \quad (4.5), (6.5)$$
$$= \text{apply}(v, w)$$

26
Since $w$ has been chosen arbitrarily, it follows that $u \leq v$.

For the direction (b) $\Rightarrow$ (a) assume that $\text{eval}(sE) \leq \text{eval}(tE)$ for every closing environment $E$. This means that the formula $\forall \vec{x}(s \subseteq t)$ is true, where $\vec{x}$ is the list of free variables of $s$ and $t$. Using the properties (3.18), (3.34) and (3.35) one can derive in BPT the formula

$$\forall \vec{x}(s \subseteq t) \rightarrow C[s] \subseteq C[t].$$

for every context $C$. By Theorem 4.8, it follows that $C[s] \subseteq C[t]$ is true under the call-by-value interpretation for every context $C$. By the definition of the interpretation of $\subseteq$, this means that $\text{eval}(C[s]) \leq \text{eval}(C[t])$ for every context $C$. Hence, $s \preceq_{\text{obs}} t$. □

Note that the main technical ingredients of the above proof are Lemma 4.4 in the call-by-value case and Lemma 5.4 in the call-by-name case which are the basis of the Theorems 4.8 and 5.6.

7 Conclusion

The direct method does not simplify the correctness proof. The direct method, however, yields also results about the proof-theoretic strength of the formal systems. The term models under consideration can be encoded into weak systems like Peano Arithmetic (PA) or subsystems of PA. This means that although our logics contain functionals of higher types they are first-order systems (in the proof-theoretic sense).

Of course, one could add principles that postulate the existence of higher-order objects. An example of such a principle is the comprehension scheme for monotonic functions of arbitrary types. By that we mean the collection of formulas of the form

$$\text{mon}(A, x^\sigma, y^\tau) \rightarrow \exists f^{\sigma \rightarrow \tau} \forall x, y (f x \simeq y \leftrightarrow A(x, y))$$

where $f$ is not free in $A(x, y)$ and $\text{mon}(A, x^\sigma, y^\tau)$ is an abbreviation for the formula

$$\forall x^\sigma \exists y^\tau A(x, y) \land \forall x, y, x', y' (A(x, y) \land A(x', y') \land x \subseteq x' \rightarrow y \subseteq y')$$

which expresses the fact that $A(x, y)$ is the graph of a monotonic function from $\sigma$ to $\tau$. The comprehension scheme for monotonic functions is not valid under the call-by-value or the call-by-name interpretations given in this paper. To validate the comprehension scheme one has to take the way of denotational semantics sketched in Sect. 3.3.

In this article we focus on a small fragment of functional programming languages. Our approach, however, can be extended to product types and tuples, inductive algebraic types with data constructors and pattern matching, simultaneous recursive definitions and generic programs with type parameters. Hence,
we believe that our axioms and proof-rules already cover a large subset of existing functional programming languages.

One advantage of functional programming is that the programs cannot update the global state. Functional programs can therefore be viewed as mathematical functions and reasoning about functional programs is possible in a way we are familiar with from mathematics. The absence of a global state, however, has been the main obstacle for a wide-spread use of functional programming languages in practice. In the much bigger world of object-oriented programming the universe consists of objects that have a state and each function call updates the state of one or more of the objects. In other words, object-oriented programming is state-based.

One possibility to overcome the drawbacks of functional programming is to add imperative features like, for example, the pointers of ML. On the logical side, however, this requires rather complicated and unintuitive axioms and proof rules (see [11] for an axiomatization of a Lisp-like language with destructive updates). Another possibility proposed by Odersky in [15] is to add variable functions that can be updated.

From the logical point of view, however, the difference between pointers and variable functions is minimal, since pointers are nothing else than arguments of a global memory function. For example, the assignment x := 2 of ML can be viewed as an update of the memory function ‘!’ at the argument x to the value 2. From the practical point of view there might be a difference, since variable functions are local and cannot be updated by every part of the program. In both approaches the character of a high-level programming language disappears and the resulting languages are on the abstraction level of their lowest ingredient (pointers or array updates).

Another solution is the combination (and not the mingling) of functional programming languages with Gurevich Abstract State Machines [3, 7]. The idea is to use a pure applicative functional programming language for the specification (and implementation) of the static part of the system (static functions in ASM terminology) and ASMs for the dynamic part (dynamic functions). The result is an executable high-level system specification. The high-level character of the functional programming language is kept in this way. Also the freedom of ASMs in choosing the abstraction level is not restricted.

This approach has been used in the specification and verification of Java, the Java Virtual Machine, the Java Bytecode Verifier and the Java-to-JVM compiler in [20]. We plan to implement the axiomatization of functional programming described in this paper together with the logic for Abstract State Machines of [19]. We hope to obtain a system for doing high-level correctness proofs of security critical components like, for example, the bytecode verifier in [20].

References


Appendix: Derivable principles of BPT

We show how the basic principles of Sect. 3.2 are derivable in the basic logic of partial terms. Remember that \( s \approx t \) is an abbreviation for \( s \subseteq t \land t \subseteq s \) and that \( s = t \) is an abbreviation for \( s \downarrow \land t \downarrow \land s \approx t \).

The formula \( t \equiv t \) (3.22) is derivable from \( t \equiv t \) (3.12) using the definition of ‘\( \equiv \)’.

The formula \( s \equiv t \rightarrow t \equiv s \) (3.23) is a tautology.

The formula \( r \equiv s \land s \equiv t \rightarrow r \equiv t \) (3.24) is derivable from \( r \equiv s \land s \equiv t \rightarrow r \equiv t \) (3.13).

The formula \( t \downarrow \rightarrow t = t \) (3.25) is derivable from \( t \equiv t \) (3.12).

The formula \( s = t \rightarrow t = s \) (3.26) is a tautology.

The formula \( r = s \land s = t \rightarrow r = t \) (3.27) is derivable from (3.24).

The formula \( s \equiv s' \land t \equiv t' \rightarrow s \equiv s' \land t \equiv t' \) (3.28) is derivable from \( s \equiv s' \land t \equiv t' \rightarrow s \equiv s' \land t \equiv t' \) (3.18).

The formula \( t \downarrow \leftrightarrow \exists x (t = x) \) (3.29) is derivable from \( t \downarrow \rightarrow \exists x (t \approx x) \) (3.8) and \( t \equiv x \land t \downarrow \rightarrow x \downarrow \) (3.14).

The implication \( s \equiv t \rightarrow (s \downarrow \land t \downarrow \rightarrow s = t) \) of (3.30) is derivable from \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.14).

The implication \( (s \downarrow \lor t \downarrow \rightarrow s = t) \rightarrow s \equiv t \) of (3.30) is derivable from \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.15) and \( t \downarrow \lor t \subseteq s \) (3.15).

The implication \( s \subseteq t \rightarrow (s \downarrow \rightarrow s = t) \) of (3.31) is derivable from \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.14) and \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.14).

The implication \( (s \downarrow \rightarrow s = t) \rightarrow s \subseteq t \) of (3.31) is derivable from \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.15) and \( t \downarrow \lor t \subseteq s \) (3.15).

The implication \( s \subseteq s' \rightarrow t \rightarrow (s \downarrow \rightarrow s = t) (s \downarrow \rightarrow s = t) \) of (3.32) is derivable from \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.14), \( x \subseteq x \) (3.12) and \( s \subseteq t \land x \subseteq x \rightarrow s \subseteq t \) (3.18).

The implication \( (s \downarrow \rightarrow t \downarrow \land \forall x (s x \equiv t x)) \rightarrow s \subseteq s_{\sigma \rightarrow t} \) of (3.32) is derivable from \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.15) and \( s \subseteq t \land s \downarrow \rightarrow t \downarrow \) (3.14) and \( s \subseteq s_{\sigma \rightarrow t} \) (3.17).

The formula \( s \downarrow \land t \downarrow \land \forall x (s x \equiv t x) \rightarrow s = t \) (3.33) is derivable from \( s \subseteq t \land t \downarrow \land \forall x (s x \equiv t x) \rightarrow s \subseteq t \) (3.17).

For the formula \( \forall x (s \subseteq t) \rightarrow (\lambda x. s) \subseteq (\lambda x. t) \) (3.34) we assume \( \forall x (s \subseteq t) \).

By (3.11) we have \( (\lambda x. s) x \approx s \) and \( (\lambda x. t) x \approx t \). Hence we obtain \((\lambda x. s) x \approx s \subseteq t \approx (\lambda x. t) x \) and therefore \((\lambda x. s) x \subseteq (\lambda x. t) x \).

Since \((\lambda x. s) (\lambda x. t) \) and \((\lambda x. t) \downarrow \) (3.9), we can apply \((\lambda x. s) \downarrow \land (\lambda x. t) \downarrow \land \forall x ((\lambda x. s) x \subseteq (\lambda x. t) x) \rightarrow (\lambda x. s) \subseteq (\lambda x. t) \) (3.17) and obtain \((\lambda x. s) \subseteq (\lambda x. t)\).

For the formula \( \forall f, x (s \subseteq t) \rightarrow (\text{rec } f x . s) \subseteq (\text{rec } f x . t) \) (3.35) we assume \( \forall f, x (s \subseteq t) \).

By (3.34) we obtain \( \forall f, x (\lambda x. s) \subseteq (\lambda x . t) \).
Since \( \text{rec } f \cdot x \cdot t \) is a syntactic value, we can instantiate the quantifier and obtain \( (\lambda x.s)[\text{rec } f \cdot x \cdot t] = (\lambda x.t)[\text{rec } f \cdot x \cdot t] \).

Using the transitivity of \( \sqsubseteq \) (3.13) and \( (\lambda x.t)[\text{rec } f \cdot x \cdot t] \sqsubseteq (\text{rec } f \cdot x \cdot t) \) (3.19) we obtain \( (\lambda x.s)[\text{rec } f \cdot x \cdot t] \sqsubseteq (\text{rec } f \cdot x \cdot t) \). Hence we can apply \( (\lambda x.s)[\text{rec } f \cdot x \cdot t] \sqsubseteq (\text{rec } f \cdot x \cdot t) \rightarrow (\text{rec } f \cdot x.s) \sqsubseteq (\text{rec } f \cdot x.t) \) (3.20) and obtain \( (\text{rec } f \cdot x.s) \sqsubseteq (\text{rec } f \cdot x.t) \).

The principle \( s \sqsubseteq t \rightarrow r[s/x] \sqsubseteq r[t/x] \) (3.36) can be derived by induction on the size of the term \( r \). Assume \( s \sqsubseteq t \).

In the case of a variable \( y \) use \( y \sqsubseteq y \) (3.12) or the assumption \( s \sqsubseteq t \).

In the case of an abstraction \( \lambda y.r \) we use

\[
\lambda y.r \sqsubseteq \lambda y.s \rightarrow \lambda x.r \sqsubseteq \lambda x.s \forall x t \Rightarrow \lambda x.r \sqsubseteq \lambda x.s
\]

In the case of a recursive function \( \text{rec } f \cdot y.r \) we can assume that \( f \) and \( y \) are neither free in \( s \) nor in \( t \) and use

\[
\forall y (r[s/x] \sqsubseteq r[t/x]) \rightarrow (\lambda y.r)[s/x] \sqsubseteq (\lambda y.r)[t/x] \quad (3.34).
\]

In the case of a recursive function \( \text{rec } f \cdot y.r \) we can assume that \( f \) and \( y \) are neither free in \( s \) nor in \( t \) and use

\[
\forall y (r[s/x] \sqsubseteq r[t/x]) \rightarrow (\text{rec } f \cdot y.r)[s/x] \sqsubseteq (\text{rec } f \cdot y.r)[t/x] \quad (3.35).
\]

The substitution principle \( s \simeq t \land A(s) \rightarrow A(t) \) (3.37) can be derived by induction on the size of the formula \( A \). Assume \( s \simeq t \). We consider the cases of atomic formulas. Assume \( r[s/x] \downarrow \). Since \( s \sqsubseteq t \), by (3.36), we obtain \( r[s/x] \sqsubseteq r[t/x] \).

Using \( r[s/x] \sqsubseteq r[t/x] \land r[s/x] \downarrow \rightarrow r[t/x] \downarrow \) (3.14), we obtain \( r[t/x] \downarrow \).

Assume \( r[s/x] \sqsubseteq r[s/x] \). From \( r_1[t/x] \sqsubseteq r_1[s/x] \) (3.36) and \( r_2[s/x] \sqsubseteq r_2[t/x] \) we can derive \( r_1[t/x] \sqsubseteq r_2[t/x] \) using the transitivity of \( \sqsubseteq \).

The quantifier axiom \( \forall x A(x) \land t \downarrow \rightarrow A(t) \) (3.38) can be derived as follows. Assume \( \forall x A(x) \land t \downarrow \). We can assume that \( x \) is not free in \( t \). By (3.8), we have \( \exists x (t \simeq x) \). Let \( x \) with \( t \simeq x \). Since \( x \) is a syntactic value, we can apply \( \forall x A(x) \rightarrow A(x) \) (3.3) and obtain \( A(x) \). Using \( x \simeq t \land A(x) \rightarrow A(t) \) (3.37) we can derive \( A(t) \).

The quantifier axiom \( A(t) \land t \downarrow \rightarrow \exists x A(x) \) (3.39) can be derived in a similar way from the restricted quantifier axiom (3.4).

For the \( \beta \)-axiom \( s \downarrow \rightarrow (\lambda x.t) s \simeq t[s/x] \) (3.40) we assume \( s \downarrow \). We can further assume that \( x \) is not free in \( s \). By (3.8), we have \( \exists x (s \simeq x) \). Let \( x \) with \( s \simeq x \). Since \( x \) is a syntactic value, we can use \( (\lambda x.t) x \simeq t \) (3.11). Using the substitution principle (3.37) we obtain \( (\lambda x.t) s \simeq t[s/x] \).

For the equation \( (\lambda x.t)[\text{rec } f \cdot x \cdot t] = (\text{rec } f \cdot x \cdot t) \) (3.41) we start with

\[
(\lambda x.t)[\text{rec } f \cdot x \cdot t] \sqsubseteq (\text{rec } f \cdot x \cdot t) \quad (3.19).
\]

Then we apply the monotonicity principle for terms (3.36) to \( \lambda x.t \) and obtain

\[
(\lambda x.t)[\text{rec } f \cdot x \cdot t]/f \sqsubseteq (\text{rec } f \cdot x \cdot t)/f \]

Using the minimality principle (3.20) we obtain \( (\text{rec } f \cdot x \cdot t) \sqsubseteq (\lambda x.t)[\text{rec } f \cdot x \cdot t] \).

The \( \beta \)-axiom \( (\text{rec } f \cdot x \cdot t) \tilde{v} \simeq t[\tilde{v}/x, \text{rec } f \cdot x \cdot t] \) (3.42) can be derived as follows:

\[
(\text{rec } f \cdot x \cdot t) \tilde{v} \simeq (\lambda x.t)[\text{rec } f \cdot x \cdot t] \tilde{v} \quad (3.41) \text{ and } (3.28)
\]

\[
\simeq t[\tilde{v}/x, \text{rec } f \cdot x \cdot t] \quad (3.11)
\]