INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Formalizing non-termination of recursive programs

Reinhard Kahle and Thomas Studer

to appear in *Journal of Algebraic an Logic Programming*
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-2001-5, March 2001

Formalizing non-termination of recursive programs

Reinhard Kahle and Thomas Studer

Abstract

In applicative theories the recursion theorem provides a term rec which solves recursive equations. However, it is not provable that a solution obtained by rec is minimal. In the present paper we introduce an applicative theory in which it is possible to define a *least* fixed point operator. Still, our theory has a standard recursion theoretic interpretation.

1 Introduction

A recursively defined program is given by a recursion equation

$$f(\vec{x}) = t(f, \vec{x}),$$

where the program f can be called in the body of its definition. Every higher programming languages offers a syntactical construction to define programs recursively. In general, there are several different solutions to such a recursive definition, i.e. there are several functions satisfying the recursion equation. In every introduction to the semantics of programming languages one finds that the intended semantics is given by the *least fixed point* of the recursion equation (with respect to the definedness order), cf. e.g. Manna [21], Schmidt [26] or Jones [16].

Hence we need a powerful principle to prove statements about recursive programs. Probably the most famous such principle is fixed point induction introduced by Scott [27] which is based on a CPO interpretation of terms. For a good overview of Scott's induction principle and its connection to CPO models see for example Mitchell [22]. Looking at the untyped λ calculus we find that in continuous λ-models, e.g. $\mathsf{P}\omega$ or D_∞, fixed point combinators are interpreted by the *least* fixed point operator in the model, cf. e.g. Amadio and Curien [1] or Barendregt [2]. This fact makes it possible to prove semantically many properties of recursively defined programs.

However, if we look at the purely syntactical side of formal frameworks which are used to analyze programming languages, we often do not find any direct account to least fixed points. In particular, the untyped λ calculus allows to define a fixed point combinator, but there is no possibility to

express the leastness of a fixed point, cf. Curry, Hindley, Seldin [4], Hindley, Seldin [14] or Barendregt [2]. Also in the typed λ calculus, we can have fixed point combinators, but the question of leastness, which corresponds to termination, is answered from the outside by the use of normalization proofs. Comparing this with functional programming languages we see that in a type free language, like Scheme, we can define a fixed point operator which "solves" recursive equations; and in typed languages, like ML, such operators are usually built in. However, there is no way to guarantee on the syntactical level that the solution produced by these operators will be the least fixed point. This is only given by the semantical interpretation, cf. e.g. Reade [25].

In this paper we will present an *applicative theory* which allows to define a least fixed point operator. Applicative theories build the first order part of Feferman's systems of explicit mathematics [5, 6], which have originally been designed to formalize Bishop style constructive mathematics. More recently, these systems have been employed for the study of functional and object-oriented programming languages. In particular, they have been shown to provide a unitary axiomatic framework for representing programs, stating properties of programs and proving properties of programs. Important references for the use of explicit mathematics in this context are Feferman [7, 8, 9], Stärk [28, 29], Studer [32, 31, 33] and Turner [35, 36].

Applicative theories are based on type free combinatory logic, cf. Jäger, Kahle and Strahm [15]. So we have the recursion theorem at our disposal which provides a term rec (or Y) to solve recursive equations. However, it is not provable that a solution obtained by rec is minimal. We will make use of the fact that applicative theories are formulated in a *partial logic*, namely Beeson's logic of partial terms [3]. That means, we have an additional predicate expressing the definedness of a term; and quantifiers and variables are ranging over defined objects only. However, the term language is not restricted, i.e. there may be undefined terms. Using the definedness predicate we can introduce a definedness ordering on the terms. With respect to this ordering relation, we can talk about leastness of fixed points. Since not every recursion equation has a least fixed point, we additionally need the notion of *monotonicity* which can be given using the definedness ordering. Moreover, we will introduce the concept of *classes*, which are similar to types in a typed setting, in order to prove that our least fixed point belongs to a certain function space.

Since there are in general total term models for applicative theories we often cannot prove that there exist undefined terms or equivalently that the corresponding programs loop forever. For this reason we strengthen the basic theory by so-called *computability* axioms and we restrict the universe to natural numbers. These additional axioms represent the *recursion-theoretic* view of computations. They are motivated by Kleene's T predicate which

is a ternary primitive recursive relation on the natural numbers so that $\{a\}(\vec{m}) \simeq n$ holds if and only if there exists a computation sequence u with $\mathsf{T}(a, \langle \vec{m} \rangle, u)$ and $(u)_0 = n$. The use of these computability axioms for the definition of a least fixed point operator can be seen as a marriage of convenience of the recursion-theoretic semantics and the least fixed point semantics for computer programs, cf. e.g. Jones [16].

Using the computability axioms we will define the least fixed point combinator as a combinator iterating the functional operator associated with a given recursive equation starting from the totally undefined function. To get the desired properties we have to ensure that the functional operator is monotone with respect to the definedness order. For this reason we will need the notion of monotonicity mentioned above.

The given theory still has a standard recursion-theoretic model; and with respect to the proof-theoretic strength we will not exceed Peano arithmetic. There exists a standard theory to formalize least fixed points, namely the theory ID_1 of non-iterated positive arithmetical inductive definitions, cf. Moschovakis [23] for an introduction to inductive definability or Kahle and Studer [20] for a corresponding theory in the context of explicit mathematics. However, our work essentially concerns Σ^0_1 monotone inductive definitions whereas ID_1 deals with arbitrary arithmetically definable positive operators. Hence ID_1 belongs to a rather different "world" and with respect to its proof-theoretic strength it is much stronger than Peano arithmetic, cf. Pohlers [24].

The structure of the present paper is as follows. In the next section we introduce the theory LFP, an applicative theory including the computability axioms. In Section 3 we define the least fixed point operator and prove the required properties. In particular, we introduce the notion of monotonicity which is needed for a meaningful definition of least fixed points. In the final section we give some concluding remarks.

2 Applicative Theories

In this section we present the basic theory BON of operations and numbers which has been introduced by Feferman and Jäger [11] and extend it with axioms about computability and the statement that everything is a natural number. These two additional principles make the definition of a least fixed point operator possible.

Our applicative theory is formulated in the language $\mathcal{L}$ which contains the individual variables $a, b, c, f, g, h, m, n, x, y, z, \ldots$. The language $\mathcal{L}$ comprises the constants k, s (combinators), $\mathsf{p}, \mathsf{p}_0, \mathsf{p}_1$ (pairing and projections), 0 (zero), s_N (successor), p_N (predecessor) and d_N (definition by numerical cases). Further we have the constant c (computation).

The *terms* $(r, s, t, \ldots)$ of $\mathcal{L}$ are built up from the variables and constants by means of the function symbol $\cdot$ for (partial) application. We use (st) or st as an abbreviation for $(s \cdot t)$ and adopt the convention of association to the left, i.e. $s_1 s_2 \ldots s_n$ stands for $(\ldots(s_1 \cdot s_2)\ldots s_n)$.

The *atomic formulas* of $\mathcal{L}$ are $\mathsf{N}(s)$, $s{\downarrow}$ and $s = t$. Since we work with a logic of partial terms, it is not guaranteed that all terms have values, and $s{\downarrow}$ has to be read as s *is defined* or s *has a value*. Moreover, $\mathsf{N}(s)$ says that s is a natural number.
The formulas $(A, B, C, \ldots)$ of $\mathcal{L}$ are generated from the atomic formulas by closing against the usual propositional connectives and quantifiers. As abbreviations, we use:

$$\begin{aligned} s \simeq t &:= s{\downarrow} \vee t{\downarrow} \to s = t, \\ s \neq t &:= s{\downarrow} \wedge t{\downarrow} \wedge \neg(s = t), \\ s \in \mathsf{N} &:= \mathsf{N}(s), \\ (\exists x \in \mathsf{N})F(x) &:= \exists x(x \in \mathsf{N} \wedge F(x)), \\ (\forall x \in \mathsf{N})F(x) &:= \forall x(x \in \mathsf{N} \to F(x)), \\ f \in (\mathsf{N} \to \mathsf{N}) &:= (\forall x \in \mathsf{N})fx \in \mathsf{N}. \end{aligned}$$

Moreover, we define general n-tupling by induction on $n \geq 2$ as follows: $(s_1, s_2) := \mathsf{p} s_1 s_2$ and $(s_1, \ldots, s_{n+1}) := ((s_1, \ldots, s_n), s_{n+1})$.

The logic for our applicative theories is Beeson's classical *logic of partial terms*, cf. Beeson [3] or Troelstra and van Dalen [34]. The non-logical axioms of BON are

(1) $\mathsf{k}ab = a$,

(2) $\mathsf{s}ab{\downarrow} \wedge \mathsf{s}abc \simeq ac(bc)$,

(3) $\mathsf{p}_0 a{\downarrow} \wedge \mathsf{p}_1 a{\downarrow}$,

(4) $\mathsf{p}_0(a, b) = a \wedge \mathsf{p}_1(a, b) = b$,

(5) $0 \in \mathsf{N} \wedge (\forall x \in \mathsf{N})(\mathsf{s}_\mathsf{N} x \in \mathsf{N})$,

(6) $(\forall x \in \mathsf{N})(\mathsf{s}_\mathsf{N} x \neq 0 \wedge \mathsf{p}_\mathsf{N}(\mathsf{s}_\mathsf{N} x) = x)$,

(7) $(\forall x \in \mathsf{N})(x \neq 0 \to \mathsf{p}_\mathsf{N} x \in \mathsf{N} \wedge \mathsf{s}_\mathsf{N}(\mathsf{p}_\mathsf{N} x) = x)$,

(8) $a \in \mathsf{N} \wedge b \in \mathsf{N} \wedge a = b \to \mathsf{d}_\mathsf{N} xyab = x$,

(9) $a \in \mathsf{N} \wedge b \in \mathsf{N} \wedge a \neq b \to \mathsf{d}_\mathsf{N} xyab = y$.

It is a well-known result that we can introduce λ abstraction and recursion using the combinator axioms (1) and (2).

Theorem 1.

1. *For every variable x and every term t of $\mathcal{L}$, there exists a term $\lambda x.t$ of $\mathcal{L}$ whose free variables are those of t, excluding x, such that*

$$\mathsf{BON} \vdash \lambda x.t{\downarrow} \wedge (\lambda x.t)\, x \simeq t \text{ and } \mathsf{BON} \vdash s{\downarrow} \to (\lambda x.t)\, s \simeq t[s/x].$$

2. *There exists a term rec of $\mathcal{L}$ such that*

$$\mathsf{BON} \vdash \mathsf{rec}\, f{\downarrow} \wedge \forall x(\mathsf{rec}\, f\, x \simeq f\, (\mathsf{rec}\, f)\, x).$$

Proof. The definition of λ terms is standard in the context of partiality, cf. Beeson [3] or Feferman [5]. Also, the definition of the recursion operator is a standard adaptation from the fixed point combinator in type-free λ calculus:

$$\mathsf{rec} := \lambda f.(\lambda y, x.f\, (y\, y)\, x)\, (\lambda y, x.f\, (y\, y)\, x).$$

□

In the sequel we employ full induction on the natural numbers which is given by the following scheme:

$$(\mathcal{L}\text{-}\mathsf{I}_\mathsf{N}) \qquad A(0) \wedge (\forall x \in \mathsf{N})(A(x) \to A(\mathsf{s}_\mathsf{N} x)) \;\to\; (\forall x \in \mathsf{N})A(x),$$

for all formulas A of $\mathcal{L}$.

In $\mathsf{BON} + (\mathcal{L}\text{-}\mathsf{I}_\mathsf{N})$ all the primitive recursive functions and relations are available. Particularly, we will use addition $+$ and multiplication $*$ of natural numbers (both also in infix notation) as well as the usual "less than" $<$ and "less or equal than" $\leq$ relations. Further, we can define a least number operator μ so that the following holds, cf. [3].

Lemma 2. $\mathsf{BON} + (\mathcal{L}\text{-}\mathsf{I}_\mathsf{N})$ *proves:*

1. $f \in (\mathsf{N} \to \mathsf{N}) \to (\mu f \in \mathsf{N} \leftrightarrow (\exists n \in \mathsf{N}) fn = 0)$,
2. $f \in (\mathsf{N} \to \mathsf{N}) \wedge \mu f \in \mathsf{N} \to f(\mu f) = 0$.

Now we introduce non-strict definition by cases (cf. Beeson [3] or Kahle [18]). Observe that if $\mathsf{d}_\mathsf{N} rsuv{\downarrow}$, then $r{\downarrow}$ and $s{\downarrow}$ hold by strictness. However, we often want to define a function by cases so that it is defined if one case holds, even if the value that would have been computed in the other case is undefined. Hence, we let $\mathsf{d}_\mathsf{s} rsuv$ stand for the term $\mathsf{d}_\mathsf{N}(\lambda z.r)(\lambda z.s)uv0$ where the variable z does not occur in the terms r and s. From now on, non-strict definition by cases is denoted by the following notation:

$$\mathsf{d}_\mathsf{s} rsuv \simeq \begin{cases} r & \text{if } u = v, \\ s & \text{otherwise.} \end{cases}$$

Note that it already anticipates the axiom $\forall x \mathsf{N}(x)$, otherwise we should add $\mathsf{N}(u) \wedge \mathsf{N}(v)$ as a premise; and of course, strictness still holds with respect to u and v. We have $\mathsf{d_s} rsuv\downarrow \rightarrow u\downarrow \wedge v\downarrow$. If u or v is undefined, then $\mathsf{d_s} rsuv$ is also undefined. However, if r is a defined term and u and v are defined natural numbers that are equal, then $\mathsf{d_s} rsuv = r$ holds even if s is not defined.

We are interested in the extension of BON with axioms about computability (Comp) and the assertion that everything is a number.

Computability. These axioms are intended to capture the idea that convergent computations should converge in finitely many steps. In the formal statement of the axioms the expression $\mathsf{c}(f, x, n) = 0$ can be read as "the computation fx converges in n steps." The idea of these axioms is due to Friedman (unpublished) and discussed in Beeson [3]. Note that these axioms are satisfied in the usual recursion-theoretic model. The constant c can be interpreted by the characteristic function of Kleene's T predicate.

(Comp.1) $\forall f \forall x (\forall n \in \mathsf{N})(\mathsf{c}(f, x, n) = 0 \vee \mathsf{c}(f, x, n) = 1)$,

(Comp.2) $\forall f \forall x (fx\downarrow \leftrightarrow (\exists n \in \mathsf{N})\mathsf{c}(f, x, n) = 0)$.

In addition, we will restrict the universe to natural numbers. This axiom will be needed to make use of the least number operator μ and the computability term in the definition of the least fixed point operator, see below. Of course, this axiom is absolutely in the spirit of a recursion-theoretic interpretation.

Everything is a number. Formally, this is given by the statement $\forall x \mathsf{N}(x)$.

Now we define the applicative theory for least fixed points LFP as the union of all these axioms:

$$\mathsf{LFP} := \mathsf{BON} + (\mathsf{Comp}) + \forall x \mathsf{N}(x) + (\mathcal{L}\text{-}\mathsf{I_N}).$$

Before we can go on and define the least fixed point operator we have to introduce some auxiliary terms. With some coding provided by pairing and projection, we can easily define a term c^3 which behaves for ternary functions like c does for unary functions, i.e.

1. $\forall f \forall x \forall y \forall z (\forall n \in \mathsf{N})(\mathsf{c}^3(f, x, y, z, n) = 0 \vee \mathsf{c}^3(f, x, y, z, n) = 1)$,
2. $\forall f \forall x \forall y \forall z (fxyz\downarrow \leftrightarrow (\exists n \in \mathsf{N})\mathsf{c}^3(f, x, y, z, n) = 0)$.

The following lemma shows that there exists a function b which is never defined. Later, we will define an order relation on our functions and there b will play the role of the bottom element. Hence, we will be able the define least fixed points of monotonic functionals by recursion starting from b.

Lemma 3. *There exists a closed $\mathcal{L}$ term b so that LFP proves $\forall x(\neg \mathsf{b}x\downarrow)$.*

Proof. We can define $\mathsf{not_N} := \mathsf{rec}\,(\lambda f, x.\mathsf{d_N}\,1\,0\,(f\,x)\,0)\,0$. So it follows that $\neg\mathsf{N}(\mathsf{not_N})$ holds, see Kahle [19]. Since we included $\forall x\mathsf{N}(x)$ to our list of axioms we get $\neg(\mathsf{not_N}{\downarrow})$. Thus, we can set $\mathsf{b} := \lambda x.\mathsf{not_N}$. □

3 Least Fixed Point Operator

In this section we will show how to define a least fixed point operator l in the theory LFP. As usual, in order to find the least fixed point of a monotonic functional g, the operator l will iterate g starting from the bottom element b. The stages of this inductive process are given by the term h, which will be defined first.

Definition 4. We define the term h so that

$$\mathsf{h}gn \simeq \begin{cases} \mathsf{b} & \text{if } n = 0, \\ g(\mathsf{h}g(\mathsf{p_N}n)) & \text{otherwise.} \end{cases}$$

Let q be such that

$$\mathsf{q}gxn \simeq \begin{cases} 0 & \text{if } \mathsf{h}g(\mathsf{p_0}n)x = \mathsf{p_1}n, \\ \mathsf{not_N} & \text{otherwise.} \end{cases}$$

Then the term l is defined by

$$\mathsf{l} \;:=\; \lambda g\lambda x.\mathsf{p_1}(\mathsf{p_0}(\mu(\lambda y.\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p_0}(y), \mathsf{p_1}(y))))).$$

The idea of this definition can be explained roughly as follows. We would like to have that $\mathsf{l}gx = z$ implies that there exists a finite computation of z by iterating the operator g starting from b. Formally, this is expressed by $\exists n(\mathsf{h}gnx = z)$, cf. the third claim of the following lemma. The definition of l is somewhat clumsy because of the several codings. Let g and x be given, then the μ operator is looking for an n so that

$$\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p_0}(n), \mathsf{p_1}(n)) = 0. \tag{1}$$

If there is no such natural number n, then $\mu(\lambda y.\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p_0}(y), \mathsf{p_1}(y)))$ will be undefined. Assume we have found an n so that (1) holds. This means that $\mathsf{q}gx(\mathsf{p_0}(n))$ is terminating in $\mathsf{p_1}(n)$ steps. By the definition of q, we obtain that $\mathsf{q}gx(\mathsf{p_0}(n)){\downarrow}$ implies $\mathsf{h}g(\mathsf{p_0}(\mathsf{p_0}(n)))x = \mathsf{p_1}(\mathsf{p_0}(n))$. Finally, the outer projections are used to extract this value $\mathsf{p_1}(\mathsf{p_0}(n))$.

The behavior of l can be also gathered from (the proof of) the following lemma. Moreover, it also shows why we had to include the axiom $\forall x\mathsf{N}(x)$ to LFP. Without this axiom the sophisticated interplay between the least number operator μ, the computability term c^3, and the coding machinery provided by $\mathsf{p_0}, \mathsf{p_1}$ and p would hardly work. This proof also makes use of the fact that the projection functions are total, see axiom (3) of BON.

Lemma 5. LFP *proves:*

1. $\mathsf{l}g{\downarrow}$,
2. $\mathsf{l}gx{\downarrow} \leftrightarrow \exists n(\mathsf{h}gnx{\downarrow})$,
3. $\mathsf{l}gx = z \rightarrow \exists n(\mathsf{h}gnx = z)$.

Proof. The first claim is a consequence of the theorem about λ abstraction. For the second claim we have:

$$\begin{aligned}
\mathsf{l}gx{\downarrow} &\leftrightarrow \mathsf{p}_1(\mathsf{p}_0(\mu(\lambda y.\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p}_0(y), \mathsf{p}_1(y)))))\downarrow \\
&\leftrightarrow \mu(\lambda y.\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p}_0(y), \mathsf{p}_1(y)))\downarrow \\
&\leftrightarrow \exists n(\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p}_0(n), \mathsf{p}_1(n)) = 0) \\
&\leftrightarrow \exists n(\mathsf{q}gxn{\downarrow}) \\
&\leftrightarrow \exists n(\mathsf{d_s}0\mathsf{not_N}(\mathsf{h}g(\mathsf{p}_0 n)x)(\mathsf{p}_1 n){\downarrow}) \\
&\leftrightarrow \exists n(\mathsf{h}gnx{\downarrow})
\end{aligned}$$

The third claim follows by:

$$\begin{aligned}
\mathsf{l}gx = z &\rightarrow \mathsf{p}_1(\mathsf{p}_0(\mu(\lambda y.\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p}_0(y), \mathsf{p}_1(y))))) = z \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mu(\lambda y.\mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p}_0(y), \mathsf{p}_1(y))) = n) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mathsf{c}^3(\mathsf{q}, g, x, \mathsf{p}_0(n), \mathsf{p}_1(n)) = 0) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mathsf{q}gx(\mathsf{p}_0 n){\downarrow}) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mathsf{d_s}0\mathsf{not_N}(\mathsf{h}g(\mathsf{p}_0(\mathsf{p}_0 n))x)(\mathsf{p}_1(\mathsf{p}_0 n)){\downarrow}) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mathsf{h}g(\mathsf{p}_0(\mathsf{p}_0 n))x = \mathsf{p}_1(\mathsf{p}_0 n)) \\
&\rightarrow \exists n(\mathsf{h}g(\mathsf{p}_0(\mathsf{p}_0 n))x = z) \\
&\rightarrow \exists n(\mathsf{h}gnx = z)
\end{aligned}$$

□

We define the closed term a which will later serve at showing that we can replace the term $g(\mathsf{l}g)$ by a "finite approximation" $g(\mathsf{h}gn)$ (cf. Lemma 12, Claim 7).

Definition 6. Let t be such that

$$\mathsf{t}fx \simeq \begin{cases} \mathsf{c}(\lambda x.xx, f, x) & \text{if } x = 0, \\ \mathsf{t}f(\mathsf{p_N}x) * \mathsf{c}(\lambda x.xx, f, x) & \text{otherwise.} \end{cases}$$

We define the term a using λ abstraction so that

$$\mathsf{a}gfx \simeq \begin{cases} \mathsf{not_N} & \text{if } \mathsf{t}fx = 0, \\ gx & \text{otherwise.} \end{cases}$$

Lemma 7. LFP *proves:*

1. $\forall g \forall f(\neg f f\downarrow \to \forall n(\mathsf{a} g f n \simeq g n))$,
2. $\forall g \forall f(f f\downarrow \to \exists m \forall n(\mathsf{a} g f n\downarrow \to \mathsf{a} g f n = g n \wedge n < m))$.

Proof. From the axioms about computability we obtain by induction:

$$\forall f \forall n(\mathsf{t} f n = 0 \vee \mathsf{t} f n = 1)$$

and

$$\forall f \forall n \forall m(m \leq n \wedge \mathsf{t} f m = 0 \to \mathsf{t} f n = 0).$$

Now, we get the first claim by:

$$\begin{array}{rcl} \neg f f\downarrow & \to & \neg(\lambda x.xx)f\downarrow \\ & \to & \forall n(\mathsf{c}(\lambda x.xx, f, n) = 1) \\ & \to & \forall n(\mathsf{t} f n = 1) \\ & \to & \forall n(\mathsf{d_s not_N}(g n)(\mathsf{t} f n)0 \simeq g n) \\ & \to & \forall n(\mathsf{a} g f n \simeq g n) \end{array}$$

The second claim follows with:

$$\begin{array}{rcl} f f\downarrow & \to & (\lambda x.xx)f\downarrow \\ & \to & \exists m(\mathsf{c}(\lambda x.xx, f, m) = 0) \\ & \to & \exists m(\mathsf{t} f m = 0) \\ & \to & \exists m \forall n(m \leq n \to \mathsf{t} f n = 0) \\ & \to & \exists m \forall n(\mathsf{t} f n \neq 0 \to n < m) \\ & \to & \exists m \forall n(\mathsf{a} g f n\downarrow \to \mathsf{a} g f n = g n \wedge n < m) \end{array}$$

□

Since there is not a least fixed point for every recursion equation, cf. Example 10 below, we can only expect a meaningful solution for functionals satisfying an additional property, namely *monotonicity*. To define this notion, we will first introduce the concept of classes.

An $\mathcal{L}$ formula A containing exactly x as free variable will be called a *class*. Let A and B be classes and let F be an arbitrary formula of $\mathcal{L}$. We will employ the following abbreviations:

$$\begin{array}{rcl} t \in A & := & t\downarrow \wedge A[t/x], \\ A \to B & := & \forall y(y \in A \to xy \in B), \\ A \curvearrowright B & := & \forall y(y \in A \wedge xy\downarrow \to xy \in B), \\ A \cap B & := & x \in A \wedge x \in B, \\ (\forall x \in A)F(x) & := & \forall x(x \in A \to F(x)). \end{array}$$

Note that $t \in A$ has a strictness property built in. We have $t \in A \to t{\downarrow}$. Next we are going to introduce the definedness ordering $\sqsubseteq_T$ with respect to a class T. The meaning of $r \sqsubseteq s$ is that if r has a value, then r equals s; and $f \sqsubseteq_{A \frown B} g$ says that for every $x \in A$ if the computation fx terminates, then gx also terminates and both computations yield the same result.

Definition 8. Let $A_1, \ldots, A_n, B_1, \ldots, B_n$ be classes. Further, let $\mathcal{T}$ be the class $(A_1 \frown B_1) \cap \cdots \cap (A_n \frown B_n)$. Then $\mathcal{T}$ is called an *arrow class*. We define:

$$\begin{array}{rcl} r \sqsubseteq s & := & r{\downarrow} \to r = s, \\ f \sqsubseteq_{\mathcal{T}} g & := & \bigwedge\limits_{1 \le i \le n} (\forall x \in A_i) fx \sqsubseteq gx, \\ f \cong_{\mathcal{T}} g & := & f \sqsubseteq_{\mathcal{T}} g \wedge g \sqsubseteq_{\mathcal{T}} f. \end{array}$$

The formula $r \sqsubseteq s \wedge s \sqsubseteq r$ is equivalent to the standard partial equality relation $r \simeq s$. Hence, our definedness ordering $\sqsubseteq$ is in accordance with the notion of partiality of our applicative theory. Studer [30] employs a least fixed point operator to define a denotational semantics for Featherweight Java. This semantics features an overloading based object model. An overloaded function models type-dependent computations and hence, it belongs to the intersection of several function spaces. Therefore, we define arrow types to be such intersections in order to prepare our setting for this application.

The relations $\sqsubseteq$ and $\sqsubseteq_{\mathcal{T}}$ are transitive.

Lemma 9. *Let $\mathcal{T}$ be an arrow class as given in Definition 8. Then we can prove in* LFP*:*

1. $r \sqsubseteq s \wedge s \sqsubseteq t \to r \sqsubseteq t$,
2. $f \sqsubseteq_{\mathcal{T}} g \wedge g \sqsubseteq_{\mathcal{T}} h \to f \sqsubseteq_{\mathcal{T}} h$.

Proof. We have $r{\downarrow} \to r = s$ as well as $s{\downarrow} \to s = t$. Obviously we get $r{\downarrow} \to r = t$ proving Claim 1. Now we show the second claim. Assume $x \in A_i$ for some i. We have $fx \sqsubseteq gx$ and $gx \sqsubseteq hx$. Therefore, we conclude $fx \sqsubseteq hx$ by the first claim. □

Using the rec term we will find a fixed point for every operation g. But as mentioned before we cannot prove that this is a least fixed point; and of course, there are terms g that do not have a least fixed point.

Example 10. Let f_1 and f_2 be closed terms so that

$$f_1 x \simeq \begin{cases} 1 & \text{if } x = 1, \\ \mathsf{not_N} & \text{otherwise} \end{cases} \qquad \text{and} \qquad f_2 x \simeq \begin{cases} \mathsf{not_N} & \text{if } x = 1, \\ 1 & \text{otherwise.} \end{cases}$$

Now we let g be the operation

$$gx \simeq \begin{cases} f_1 & \text{if } x = f_1, \\ f_2 & \text{otherwise.} \end{cases}$$

Let V be the universal class $x = x$. Then we know $g \in ((\mathsf{V} \curvearrowright \mathsf{V}) \to (\mathsf{V} \curvearrowright \mathsf{V}))$, and if f is a fixed point of g then we have either $f = f_1$ or $\forall x(fx \simeq f_2 x)$. However, g does not have a least fixed point in the sense of $\sqsubseteq_{(\mathsf{V} \curvearrowright \mathsf{V})}$, for we find $\neg f_1 \sqsubseteq_{(\mathsf{V} \curvearrowright \mathsf{V})} f_2 \wedge \neg f_2 \sqsubseteq_{(\mathsf{V} \curvearrowright \mathsf{V})} f_1$. That is f_1 is not comparable with any other fixed point of g and therefore, we do not have a least fixed point.

Only for *monotonic* $g \in (\mathcal{T} \to \mathcal{T})$ we can show that $\mathsf{l}g$ is the least fixed point of g.

Definition 11. Let $\mathcal{T}$ be an arrow class as given in Definition 8. A function $f \in (\mathcal{T} \to \mathcal{T})$ is called $\mathcal{T}$ *monotonic, if*

$$(\forall g \in \mathcal{T})(\forall h \in \mathcal{T})(g \sqsubseteq_{\mathcal{T}} h \to fg \sqsubseteq_{\mathcal{T}} fh).$$

Claims 1–5 of the following lemma correspond to the corollary in the appendix of Feferman [10]. Furthermore, in order to show that l yields a fixed point we need the compactness property stated in the last claim of our lemma.

Lemma 12. *Let $\mathcal{T}$ be the arrow class $(A_1 \curvearrowright B_1) \cap \cdots \cap (A_m \curvearrowright B_m)$. We can prove in* LFP *that if $g \in (\mathcal{T} \to \mathcal{T})$ is $\mathcal{T}$ monotonic, then the following claims hold for all $i \leq m$.*

1. $\forall n(\mathsf{h}gn \in \mathcal{T})$,
2. $\forall n(\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{h}g(n+1))$,
3. $\mathsf{l}g \in \mathcal{T}$,
4. $\forall n(\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{l}g)$,
5. $\mathsf{l}g \sqsubseteq_{\mathcal{T}} g(\mathsf{l}g)$,
6. $\forall m \exists n(\forall x \in A_i)(x \leq m \to \mathsf{l}gx \sqsubseteq \mathsf{h}gnx)$,
7. $(\forall x \in A_i)\exists n(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x)$.

Proof. 1. Proof by induction on the natural numbers. For $n = 0$ we have $\mathsf{h}g0 = \mathsf{b}$. Since $\forall x(\neg \mathsf{b}x\downarrow)$ we obviously get $\mathsf{h}g0 \in \mathcal{T}$. Assume $\mathsf{h}gn \in \mathcal{T}$. Then we have $g(\mathsf{h}gn) \in \mathcal{T}$ and this yields $\mathsf{h}g(n+1) \in \mathcal{T}$.

2. We proceed by induction on the natural numbers. As above we get $\forall x(\neg \mathsf{h}g0x\downarrow)$. Hence we have $(\forall x \in A_i)(\mathsf{h}g0x \sqsubseteq \mathsf{h}g1x)$ for any i. For

the induction step assume $\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{h}g(n+1)$. Since g is $\mathcal{T}$ monotonic and by the previous claim $\forall n(\mathsf{h}gn{\downarrow})$ holds, we get

$$g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} g(\mathsf{h}g(n+1)).$$

This yields $\mathsf{h}g(n+1) \sqsubseteq_{\mathcal{T}} \mathsf{h}g(n+2)$.

3. By Lemma 5 we find $(\forall x \in A_i)(\mathsf{l}gx{\downarrow} \to \exists n(\mathsf{h}gnx = \mathsf{l}gx))$ for any i. Then, by Claim 1 we get $(\forall x \in A_i)(\mathsf{l}gx{\downarrow} \to \mathsf{l}gx \in B_i)$. Hence $\mathsf{l}g \in \mathcal{T}$.

4. We have to show $(\forall x \in A_i)(\mathsf{h}gnx \sqsubseteq \mathsf{l}gx)$ for all i. So assume $x \in A_i$ and $\mathsf{h}gnx{\downarrow}$. We conclude $\mathsf{l}gx{\downarrow}$ by Lemma 5. Hence there exists a natural number m with

$$\mathsf{h}gmx = \mathsf{l}gx. \tag{2}$$

From Claim 2 we get by induction

$$\forall n \forall m(\forall x \in A_i)(\mathsf{h}gnx{\downarrow} \wedge \mathsf{h}gmx{\downarrow} \to hgnx = hgmx).$$

By $x \in A_i$ and (2) we therefore finally obtain $\mathsf{h}gnx \sqsubseteq \mathsf{l}gx$.

5. We have to show $(\forall x \in A_i)\mathsf{l}gx \sqsubseteq g(\mathsf{l}g)x$ for all i. So let $x \in A_i$ and $\mathsf{l}gx{\downarrow}$. Then by Lemma 5 we get $\exists n(\mathsf{l}gx = \mathsf{h}gnx)$. By the definition of h we see $\forall x(\neg \mathsf{h}g0x{\downarrow})$. Hence $\exists n(\mathsf{l}gx = \mathsf{h}g(n+1)x)$. This is

$$\exists n(\mathsf{l}gx = g(\mathsf{h}gn)x). \tag{3}$$

For this natural number n we have by Claim 4 that $\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{l}g$. Because g is $\mathcal{T}$ monotonic we obtain $g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} g(\mathsf{l}g)$ and since $x \in A_i$ this implies $g(\mathsf{h}gn)x \sqsubseteq g(\mathsf{l}g)x$. Finally, we conclude by (3) that $\mathsf{l}gx \sqsubseteq g(\mathsf{l}g)x$.

6. Proof by induction on m. For $m = 0$ the claim follows from Lemma 5. For the induction step assume

$$\exists n_1(\forall x \in A_i)(x \leq m \to \mathsf{l}gx \sqsubseteq \mathsf{h}gn_1x).$$

Employing Lemma 5 we find

$$m+1 \in A_i \wedge \mathsf{l}g(m+1){\downarrow} \to \exists n_2(\mathsf{l}g(m+1) = \mathsf{h}gn_2(m+1)).$$

Taken together this yields

$$\begin{aligned}&\exists n_1 \exists n_2(\forall x \in A_i)\\ &\quad (x \leq m+1 \wedge \mathsf{l}gx{\downarrow} \to (\mathsf{l}gx = \mathsf{h}gn_1x \vee \mathsf{l}gx = \mathsf{h}gn_2x)).\end{aligned}$$

By Claim 2 and Lemma 9 we get

$$\begin{aligned}&\exists n_1 \exists n_2(\forall x \in A_i)\\ &\quad (x \leq m+1 \wedge \mathsf{l}gx{\downarrow} \to \mathsf{l}gx = \mathsf{h}g(n_1+n_2)x).\end{aligned}$$

We finally conclude

$$\exists n(\forall x \in A_i)(x \leq m+1 \to \mathsf{l}gx \sqsubseteq \mathsf{h}gnx).$$

7. Proof by contrapositive: suppose there exists an $x \in A_i$ so that

$$\forall n \neg (g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x). \tag{4}$$

With this $x \in A_i$ we define a term k by

$$k := \lambda f.\mathsf{d_s}0\mathsf{not_N}(g(\mathsf{a}(\mathsf{l}g)f)x)(g(\mathsf{l}g)x).$$

For the so defined k we will show that either assumption $\neg kk\downarrow$ or $kk\downarrow$ leads to a contradiction. As consequence we conclude that there cannot exist an $x \in A_i$ satisfying (4) and hence this claim is proved.

Now suppose $\neg kk\downarrow$. As a direct consequence of Lemma 7 we obtain for any j

$$\forall f(\neg ff\downarrow \to (\forall y \in A_j)\mathsf{a}(\mathsf{l}g)fy \simeq \mathsf{l}gy).$$

Therefore, we get

$$(\forall y \in A_j)\mathsf{a}(\mathsf{l}g)ky \simeq \mathsf{l}gy$$

for any j. The term a is defined by λ abstraction. Hence by Theorem 1 and Claim 3 we find $\mathsf{a}(\mathsf{l}g)k \in \mathcal{T}$. Therefore we obtain by the $\mathcal{T}$ monotonicity of g and $x \in A_i$ that $g(\mathsf{a}(\mathsf{l}g)k)x \simeq g(\mathsf{l}g)x$. By (4) it is the case that $g(\mathsf{l}g)x\downarrow$. Hence $g(\mathsf{a}(\mathsf{l}g)k)x = g(\mathsf{l}g)x$. This implies

$$\mathsf{d_s}0\mathsf{not_N}(g(\mathsf{a}(\mathsf{l}g)k)x)(g(\mathsf{l}g)x)\downarrow,$$

i.e. $(\lambda f.\mathsf{d_s}0\mathsf{not_N}(g(\mathsf{a}(\mathsf{l}g)f)x)(g(\mathsf{l}g)x))k\downarrow$ and $kk\downarrow$. Contradiction.

Suppose $kk\downarrow$. Hence $\mathsf{d_s}0\mathsf{not_N}(g(\mathsf{a}(\mathsf{l}g)k)x)(g(\mathsf{l}g)x)\downarrow$ and

$$g(\mathsf{a}(\mathsf{l}g)k)x = g(\mathsf{l}g)x. \tag{5}$$

By Lemma 7 $kk\downarrow$ implies for any j

$$\exists m(\forall y \in A_j)(\mathsf{a}(\mathsf{l}g)ky\downarrow \to \mathsf{a}(\mathsf{l}g)ky = \mathsf{l}gy \wedge y < m). \tag{6}$$

Using Claim 6 we get $\exists n(\forall y \in A_j)(\mathsf{a}(\mathsf{l}g)ky \sqsubseteq \mathsf{h}gny)$. for any j. Hence $\exists n(\mathsf{a}(\mathsf{l}g)k \sqsubseteq_{\mathcal{T}} \mathsf{h}gn)$. Claim 3 together with (6) yields $\mathsf{a}(\mathsf{l}g)k \in \mathcal{T}$. Since g is $\mathcal{T}$ monotonic we therefore have

$$\exists n(g(\mathsf{a}(\mathsf{l}g)k) \sqsubseteq_{\mathcal{T}} g(\mathsf{h}gn)).$$

Our assumption $x \in A_i$ yields

$$\exists n(g(\mathsf{a}(\mathsf{l}g)k)x \sqsubseteq g(\mathsf{h}gn)x). \tag{7}$$

From (4) we know $\forall n \neg (g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x)$. Using (7) we conclude

$$\neg (g(\mathsf{l}g)x = g(\mathsf{a}(\mathsf{l}g)k)x)$$

which contradicts (5). □

The following theorem states that l indeed yields a fixed point of a monotonic operation g.

Theorem 13. *We can prove in* LFP *that if* $g \in (\mathcal{T} \to \mathcal{T})$ *is* $\mathcal{T}$ *monotonic for* $\mathcal{T}$ *given as in Definition 8, then*

$$\mathsf{l}g \cong_{\mathcal{T}} g(\mathsf{l}g).$$

Proof. By the previous lemma $\mathsf{l}g \sqsubseteq_{\mathcal{T}} g(\mathsf{l}g)$ holds. In order to show the other direction let $x \in A_i$. By the last claim of the previous lemma we obtain

$$\exists n(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x).$$

By the definition of h we get $\exists n(g(\mathsf{l}g)x \sqsubseteq \mathsf{h}g(n+1)x)$. Using Claim 4 of the previous lemma we find $\forall n(\mathsf{h}g(n+1)x \sqsubseteq \mathsf{l}gx)$. Hence, by Lemma 9 we have $g(\mathsf{l}g)x \sqsubseteq \mathsf{l}gx$. Finally, we conclude $g(\mathsf{l}g) \sqsubseteq_{\mathcal{T}} \mathsf{l}g$. □

The next theorem states that $\mathsf{l}g$ is the *least* fixed point of g.

Theorem 14. *We can prove in* LFP *that if* $g \in (\mathcal{T} \to \mathcal{T})$ *is* $\mathcal{T}$ *monotonic for* $\mathcal{T}$ *given as in Definition 8, then*

$$f \in \mathcal{T} \wedge gf \cong_{\mathcal{T}} f \to \mathsf{l}g \sqsubseteq_{\mathcal{T}} f.$$

Proof. Let f be such that $gf \cong_{\mathcal{T}} f$. First, we show by induction on N that

$$\forall n(\mathsf{h}gn \sqsubseteq_{\mathcal{T}} f). \tag{8}$$

We obviously have $\mathsf{h}g0 \sqsubseteq_{\mathcal{T}} f$. Suppose $\mathsf{h}gn \sqsubseteq_{\mathcal{T}} f$ for a natural number n. By the $\mathcal{T}$ monotonicity of g we get $\mathsf{h}g(n+1) = g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} gf \cong_{\mathcal{T}} f$. Therefore, by Lemma 9 we obtain $\mathsf{h}g(n+1) \sqsubseteq_{\mathcal{T}} f$ and (8) is shown. By $g(\mathsf{h}gn) = \mathsf{h}g(n+1)$ this implies

$$\forall n(g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} f). \tag{9}$$

It remains to show $(\forall x \in A_i)(\mathsf{l}gx \sqsubseteq fx)$ for each i. So let $x \in A_i$. By Claim 5 of Lemma 12 we get $\mathsf{l}gx \sqsubseteq g(\mathsf{l}g)x$. By Claim 7 of the same lemma we obtain $\exists n(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x)$. Therefore with (9) and Lemma 9 we conclude $\mathsf{l}gx \sqsubseteq fx$. □

4 Conclusion

For the conclusion let us look at the following recursively defined method written in a Java like language.

```
A m (B x) {
  return m(x);
}
```

Of course, any program calling `m` with some argument s is non-terminating. The semantics of the method `m` is usually given as the least fixed point of the functional $\lambda f \lambda x.fx$. If we model this fixed point by $\mathsf{rec}(\lambda f \lambda x.fx)$, then we cannot prove in BON that $\neg(\mathsf{rec}(\lambda f \lambda x.fx)s\downarrow)$ for any argument s. This is simply because one can build total term models of the theory BON in which every term has a value.

On the other hand, defining the semantics of the method `m` using our least fixed point operator I enables us to prove non-termination in LFP. Let V be the universal class $x = x$ and $\emptyset$ the empty class $x \neq x$. Then the functional $\lambda f \lambda x.fx$ is an element of $(\mathsf{V} \curvearrowright \emptyset) \to (\mathsf{V} \curvearrowright \emptyset)$ and is of course $\mathsf{V} \curvearrowright \emptyset$ monotonic. Therefore, by Lemma 12 we have $\mathsf{I}(\lambda f \lambda x.fx) \in (\mathsf{V} \curvearrowright \emptyset)$ and this implies $\forall y(\neg\mathsf{I}(\lambda f \lambda x.fx)y\downarrow)$. Hence we have proved in LFP that the method `m` loops forever.

The theory LFP can be interpreted in the usual recursion-theoretic way, cf. Beeson [3] or Kahle [17]. This means applications $a \cdot b$ in $\mathcal{L}$ are translated into $\{a\}(b)$, where $\{n\}$ for $n = 0, 1, 2, 3, \ldots$ is a standard enumeration of the partial recursive functions. In fact, the computability axioms are inspired by Kleene's T predicate, which therefore can be used to verify the axioms. So we can reduce LFP to Peano arithmetic. This will be important for obtaining expressively strong but proof-theoretically weak systems for the study of object-oriented programming languages, cf. Studer [30, 33].

The investigation of a least fixed point operator in [17] was motivated by defining an applicative theory with proof-theoretic strength of Peano arithmetic for studying the interactive proof system LAMBDA [12, 13]. This proof system was designed for proving properties of ML programs. Up to Release 3.2 it was based on a partial logic and it was generating minimality rules for recursive function definitions. The theory defined in [17] was capturing a large part of this proof system, in particular, the minimality rules were modeled using the least fixed point operator.

We will finish this paper by addressing two related approaches. First, Feferman [10] develops a form of generalized recursion theory in which computational procedures on domains that are contained in the natural numbers reduce to ordinary computations. There he shows how to obtain a uniform index for the least fixed point operator in the intensional recursion-theoretic model of computation. In fact, the construction of our least fixed point combinator is inspired by this approach.

Secondly, Stärk [29] introduces a *typed* logic of partial terms which incorporates a least fixed point operator and a schema for computational induction. One may question why we do not axiomatize our fixed point operator in a similar way as a primitive operator instead of using the computability axioms. The reason is that formulating Theorem 14 as an axiom would require to introduce the notions of monotonicity and classes before. In our opinion,

this would be a rather inelegant approach since the axioms would already depend on complex, abbreviated notions. Beside the difference of a typed and an untyped approach, there is also a second difference between Stärk's theory and our LFP. He is giving a *domain-theoretic* interpretation of his theory, while LFP allows of a *recursion-theoretic* model.

Acknowledgments. The idea of the given treatment of a least fixed point operator in applicative theories was suggested by Peter Päppinghaus when he supervised the diplom thesis of the first author [17]. We are also grateful to Thomas Strahm and an anonymous referee for valuable comments and suggestions.

References

[1] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Cambrigde University Press, 1998.

[2] Hendrik Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.

[3] Michael J. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer, 1985.

[4] Haskell Curry, James Hindley, and Jonathan Seldin. *Combinatory Logic*, volume II. North-Holland, 1972.

[5] Solomon Feferman. A language and axioms for explicit mathematics. In J.N. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 87–139. Springer, 1975.

[6] Solomon Feferman. Constructive theories of functions and classes. In M. Boffa, D. van Dalen, and K. McAloon, editors, *Logic Colloquium '78*, pages 159–224. North Holland, 1979.

[7] Solomon Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In W. Sieg, editor, *Logic and Computation*, volume 106 of *Contemporary Mathematics*, pages 101–136. American Mathematical Society, 1990.

[8] Solomon Feferman. Logics for termination and correctness of functional programs. In Y. N. Moschovakis, editor, *Logic from Computer Science*, volume 21 of *MSRI Publications*, pages 95–127. Springer, 1991.

[9] Solomon Feferman. Logics for termination and correctness of functional programs II: Logics of strength PRA. In P. Aczel, H. Simmons, and S. S. Wainer, editors, *Proof Theory*, pages 195–225. Cambridge University Press, 1992.

[10] Solomon Feferman. A new approach to abstract data types II: computation on ADTs as ordinary computations. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Computer Science Logic '91*, volume 626 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 1992.

[11] Solomon Feferman and Gerhard Jäger. Systems of explicit mathematics with non-constructive μ-operator. Part I. *Annals of Pure and Applied Logic*, 65(3):243–263, 1993.

[12] Simon Finn and Michael P. Fourman. *Logic Manual for the LAMBDA System 3.2*. Abstract Hardware Ltd., November 1990.

[13] Mick Francis, Simon Finn, and Ellie Mayger. *Reference Manual for the LAMBDA System 3.2*. Abstract Hardware Ltd., 1990.

[14] James Hindley and Jonathan Seldin. *Introduction to Combinators and λ-calculus*. Cambridge University Press, 1986.

[15] Gerhard Jäger, Reinhard Kahle, and Thomas Strahm. On applicative theories. In A. Cantini, E. Casari, and P. Minari, editors, *Logic and Foundations of Mathematics*, pages 83–92. Kluwer, 1999.

[16] Neil Jones. *Computability and Complexity*. MIT Press, 1997.

[17] Reinhard Kahle. Einbettung des Beweissystems Lambda in eine Theorie von Operationen und Zahlen. Diplomarbeit, Mathematisches Institut der Universität München, 1992.

[18] Reinhard Kahle. *Applikative Theorien und Frege-Strukturen*. Dissertation, Institut für Informatik und angewandte Mathematik, Universität Bern, 1997.

[19] Reinhard Kahle. N-strictness in applicative theories. *Archive for Mathematical Logic*, 39(2):125–144, 2000.

[20] Reinhard Kahle and Thomas Studer. A theory of explicit mathematics equivalent to ID_1. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic CSL 2000*, volume 1862 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2000.

[21] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[22] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[23] Yiannis Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[24] Wolfram Pohlers. *Proof Theory*, volume 1407 of *Lecture Notes in Mathematics*. Springer, 1989.

[25] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[26] David Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[27] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. Manuscript, 1969. Later published in *Theoretical Computer Science*, 121:411–440,1993.

[28] Robert Stärk. Call-by-value, call-by-name and the logic of values. In D. van Dalen and M. Bezem, editors, *Computer Science Logic '96*, volume 1258 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 1997.

[29] Robert Stärk. Why the constant 'undefined'? Logics of partial terms for strict and non-strict functional programming languages. *Journal of Functional Programming*, 8(2):97–129, 1998.

[30] Thomas Studer. Constructive foundations for Featherweight Java. Preprint.

[31] Thomas Studer. Impredicative overloading in explicit mathematics. Submitted.

[32] Thomas Studer. A semantics for $\lambda^{\{\}}_{str}$: a calculus with overloading and late-binding. To appear in *Journal of Logic and Computation*.

[33] Thomas Studer. *The Mathematics of Objects*. Dissertation, Institut für Informatik und angewandte Mathematik, Universität Bern, 2001.

[34] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol II*. North Holland, 1988.

[35] Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw Hill, 1991.

[36] Raymond Turner. Weak theories of operations and types. *Journal of Logic and Computation*, 6(1):5–31, 1996.