Constructive Foundations for Featherweight Java

Thomas Studer *

Institut für Informatik und angewandte Mathematik,
Universität Bern
Neubrückstrasse 10, CH-3012 Bern, Switzerland
tstuder@iam.unibe.ch

Abstract. In this paper we present a recursion-theoretic denotational semantics for Featherweight Java. Our interpretation is based on a formalization of the object model of Castagna, Ghelli and Longo in a predicative theory of types and names. Although this theory is proof-theoretically weak, it allows to prove many properties of programs written in Featherweight Java. This underpins Feferman's thesis that impredicative assumptions are not needed for computational practice.

Keywords: object-oriented programming, denotational semantics, reasoning about programs, Featherweight Java, explicit mathematics

1 Introduction

The question of the mathematical meaning of a program is usually asked to gain more insight into the language the program is written in. This may be to bring out subtle issues in language design, to derive new reasoning principles or to develop an intuitive abstract model of the programming language under consideration so as to aid program development. Moreover, a precise semantics is also needed for establishing certain properties of programs (often related to some aspects concerning security) with mathematical rigor.

As far as the Java language is concerned, most of the research on its semantics is focused on the operational approach (cf. e.g. Börger, Schmid, Schulte and Stärk [7], Cenciarelli, Knapp, Reus and Wirsing [12], Drossopoulou, Eisenbach and Khurshid [13], Nipkow and Oheimb [31], and Syme [40]). Notable exceptions are Oheimb [32] who introduces a Hoare-style calculus for Java as well as Alves-Foss and Lam [2] who present a denotational semantics which is, as usual, based on *domain-theoretic* notions, cf. e.g. Fiore, Jung, Moggi, O'Hearn, Riecke, Rosolini and Stark [20] for a recent survey on domains and denotational semantics. Also, the projects aiming at a verification of Java programs using modern CASE tools and theorem provers have to make use of a formalization of the Java language (cf. e.g. the KeY approach by Ahrendt, Baar, Beckert, Giese, Habermalz, Hähnle,

* Research supported by the Swiss National Science Foundation. This paper is a part of the author's dissertation thesis [38].

Menzel and Schmitt [1] as well as the LOOP project by Jacobs, van den Berg, Huisman, van Berkum, Hensel and Tews [26]).

The main purpose of the present paper is the study of a *recursion-theoretic* denotational semantics for Featherweight Java, called FJ. Igarashi, Pierce and Wadler [25, 24] have proposed this system as a minimal core calculus for Java, making it easier to understand the consequences of extensions and variations. For example, they employ it to prove type safety of an extension with generic classes as well as to obtain a precise understanding of inner classes. Ancona and Zucca [4] present a module calculus where the module components are class declarations written in Featherweight Java; and the introductory text by Felleisen and Friedman [19] shows that many useful object-oriented programs can be written in a purely functional style à la Featherweight Java.

In order to give a denotational semantics for FJ, we need to formalize an object model. Often, models for statically typed object-oriented programming languages are based on a highly *impredicative* type theory. Bruce, Cardelli and Pierce [8] for example use $\mathsf{F}^{\omega}_{<:}$ as common basis to compare different object encodings. A new approach to the foundations of object-oriented programming has been proposed by Castagna, Ghelli and Longo [9, 11, 22] who take overloading and subtyping as basic rather than encapsulation and subtyping. Although in its full generality this approach leads to a new form of impredicativity, see Castagna [9] and Studer [39], only a *predicative* variant of their system is needed in order to model object-oriented programming. This predicative object model will be our starting point for constructing a denotational semantics for Featherweight Java in a theory of types and names.

Theories of types and names, or explicit mathematics, have originally been introduced by Feferman [14, 15] to formalize Bishop style constructive mathematics. In the sequel, these systems have gained considerable importance in proof theory, particularly for the proof-theoretic analysis of subsystems of second order arithmetic and set theory. More recently, theories of types and names 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 [16–18], Kahle and Studer [30], Stärk [35, 36], Studer [37, 39] as well as Turner [43, 44]. Beeson [6] and Tatsuta [41] make use of realizability interpretations for systems of explicit mathematics to prove theorems about program extraction.

Feferman [16] claims that impredicative comprehension principles are not needed for applications in computational practice. Further evidence for this is also given by Turner [44] who presents computationally weak but highly expressive theories, which suffice for constructive functional programming. In our paper we provide constructive foundations for Featherweight Java in the sense that our denotational semantics for FJ will be formalized in a constructive theory of types and names using the predicative object model of Castagna, Ghelli and Longo [11]. This supports Feferman's thesis that impredicative assumptions

are not needed. Although our theory is proof-theoretically weak we can prove soundness of our semantics with respect to subtyping, typing and reductions. Moreover, the theory of types and names we use has a recursion-theoretic interpretation. Hence, computations in FJ will be modeled by ordinary computations. For example, a non-terminating computation is not interpreted by a function, which yields $\bot$ as result, but by a *partial* function which does not terminate, either.

The plan of the present paper is as follows. In the next section we introduce the general framework of theories of types and names and we show how to construct fixed point types in it. Further we recall a theorem about a least fixed point operator in explicit mathematics, which will be the crucial ingredient of our construction. The presentation of Featherweight Java in Section 3 is included in order to make this paper self-contained. The overloading based object model we employ for our interpretation is introduced in Section 4. Section 5 is concerned with the study of some examples written in FJ which will motivate our denotational semantics as presented in Section 6. Section 7 contains the soundness proofs of our semantics with respect to subtyping, typing and reductions. A conclusion sums up what we have achieved and suggest further work, in particular the extension to the dynamic definition of new classes.

2 Theories of Types and Names

Explicit Mathematics has been introduced by Feferman [14] for the study of constructive mathematics. In the present paper, we will not work with Feferman's original formalization of these systems; instead we treat them as theories of types and names as developed in Jäger [27]. First we will present the base theory EETJ. Then we will extend it with the principle of dependent choice and show that this implies the existence of certain fixed points. Last but not least we are going to add 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.

2.1 Basic notions

The theory of types and names which we will consider in the sequel is formulated in the two sorted language $\mathcal{L}$ about individuals and types. It comprises *individual variables* $a, b, c, f, g, h, x, y, z, \ldots$ as well as *type variables* $A, B, C, X, Y, Z, \ldots$ (both possibly with subscripts).

The language $\mathcal{L}$ includes the *individual constants* k, s (combinators), $\mathsf{p}, \mathsf{p}_0, \mathsf{p}_1$ (pairing and projections), 0 (zero), s_N (successor), p_N (predecessor), d_N (definition by numerical cases) and the constant c (computation). There are additional individual constants, called *generators*, which will be used for the uniform representation of types. Namely, we have a constant c_e (elementary comprehension) for every natural number e, as well as the constants j (join) and dc (dependent choice).

The *individual terms* $(r, s, t, r_1, s_1, t_1, \dots)$ 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, this means $s_1 s_2 \dots s_n$ stands for $(\dots(s_1 \cdot s_2)\dots s_n)$. Finally, we define general n tupling by induction on $n \geq 2$ as follows:

$$(s_1, s_2) := \mathsf{p} s_1 s_2 \quad \text{and} \quad (s_1, s_2, \dots, s_{n+1}) := (s_1, (s_2, \dots, s_{n+1})).$$

The *atomic formulas* of $\mathcal{L}$ are $s\!\downarrow$, $\mathsf{N}(s)$, $s = t$, $s \in U$ and $\Re(s, U)$. Since we work with a logic of partial terms, it is not guaranteed that all terms have values, and $s\!\downarrow$ is read as *s is defined* or *s has a value*. Moreover, $\mathsf{N}(s)$ says that s is a natural number, and the formula $\Re(s, U)$ is used to express that the individual s *represents* the type U or is a *name* of U.

The *formulas* of $\mathcal{L}$ are generated from the atomic formulas by closing against the usual propositional connectives as well as quantification in both sorts. A formula is called *elementary* if it contains neither the relation symbol $\Re$ nor bound type variables. The following table contains a list of useful abbreviations, where F is an arbitrary formula of $\mathcal{L}$:

$$\begin{array}{rcl}
s \simeq t & := & s\!\downarrow \vee t\!\downarrow \;\to\; s = t, \\
s \neq t & := & s\!\downarrow \wedge t\!\downarrow \wedge \neg(s = t), \\
s \in \mathsf{N} & := & \mathsf{N}(s), \\
(\exists x \in A) F(x) & := & \exists x (x \in A \wedge F(x)), \\
(\forall x \in A) F(x) & := & \forall x (x \in A \to F(x)), \\
f \in (A \to B) & := & (\forall x \in A) fx \in B, \\
A \subset B & := & \forall x (x \in A \to x \in B), \\
A = B & := & A \subset B \wedge A \subset B, \\
f \in (A \rightharpoonup B) & := & \forall x (x \in A \wedge fx\!\downarrow \to fx \in B), \\
x \in A \cap B & := & x \in A \wedge x \in B, \\
s \mathrel{\dot\in} t & := & \exists X (\Re(t, X) \wedge s \in X), \\
s \mathrel{\dot\subset} t & := & (\forall x \mathrel{\dot\in} s) x \mathrel{\dot\in} t, \\
s \doteq t & := & s \mathrel{\dot\subset} t \wedge t \mathrel{\dot\subset} s, \\
(\exists x \mathrel{\dot\in} s) F(x) & := & \exists x (x \mathrel{\dot\in} s \wedge F(x)), \\
(\forall x \mathrel{\dot\in} s) F(x) & := & \forall x (x \mathrel{\dot\in} s \to F(x)), \\
\Re(s) & := & \exists X \Re(s, X), \\
f \in (\Re \to \Re) & := & \forall x (\Re(x) \to \Re(fx)).
\end{array}$$

The vector notation $\vec{\mathcal{Z}}$ is sometimes used to denote finite sequences $\mathcal{Z}_1, \dots \mathcal{Z}_n$ of expressions. The length of such a sequence $\vec{\mathcal{Z}}$ is then either given by the context or irrelevant. For example, for $\vec{U} = U_1, \dots, U_n$ and $\vec{s} = s_1, \dots, s_n$ we write

$$\begin{aligned}
\Re(\vec{s}, \vec{U}) &:= \Re(s_1, U_1) \wedge \dots \wedge \Re(s_n, U_n), \\
\Re(\vec{s}) &:= \Re(s_1) \wedge \dots \wedge \Re(s_n).
\end{aligned}$$

Now we introduce the theory EETJ which provides a framework for explicit elementary types with join. Its logic is Beeson's [5] classical *logic of partial terms* for individuals and classical logic for types. The logic of partial terms takes into account the possibility of undefined terms, i.e. terms which represent non-terminating computations. Scott [34] has given a logic similar to the logic of partial terms, but he treats existence like an ordinary predicate. Troelstra and van Dalen [42] give a discussion about the different approaches to partial terms.

Among the main features of the logic of partial terms are its *strictness axioms* stating that if a term has a value, then all its subterms must be defined, too. This corresponds to a call-by-value evaluation strategy, where all arguments of a function must first be fully evaluated before the final result will be computed. Stärk [35, 36] examines variants of the logic of partial terms which also allow of call-by-name evaluation.

The nonlogical axioms of EETJ can be divided into the following three groups.

I. Applicative axioms. These axioms formalize that the individuals form a partial combinatory algebra, that we have paring and projections and the usual closure conditions on the natural numbers as well as definition by numerical cases.

(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) $(\forall x \in \mathsf{N})(x \neq 0 \rightarrow \mathsf{p}_\mathsf{N} x \in \mathsf{N} \wedge \mathsf{s}_\mathsf{N}(\mathsf{p}_\mathsf{N} x) = x)$,
(8) $a \in \mathsf{N} \wedge b \in \mathsf{N} \wedge a = b \rightarrow \mathsf{d}_\mathsf{N} xyab = x$,
(9) $a \in \mathsf{N} \wedge b \in \mathsf{N} \wedge a \neq b \rightarrow \mathsf{d}_\mathsf{N} xyab = y$.

II. Explicit representation and extensionality. The following are the usual ontological axioms for systems of explicit mathematics. They state that each type has a name, that there are no homonyms and that $\Re$ respects the extensional equality of types. Note that the representation of types by their names is intensional, while the types themselves are extensional in the usual set-theoretic sense.

(10) $\exists x \Re(x, A)$,
(11) $\Re(s, A) \wedge \Re(s, B) \rightarrow A = B$,
(12) $A = B \wedge \Re(s, A) \rightarrow \Re(s, B)$.

III. Basic type existence axioms.

Elementary Comprehension. Let $F(x, \vec{y}, \vec{Z})$ be an elementary formula of $\mathcal{L}$ with at most the indicated free variables and with Gödelnumber e for any fixed Gödelnumbering, then we have the following axioms:

(13) $\Re(\vec{b}) \rightarrow \Re(\mathsf{c}_e(\vec{a}, \vec{b}))$,
(14) $\Re(\vec{b}, \vec{T}) \rightarrow \forall x(x \mathrel{\dot\in} \mathsf{c}_e(\vec{a}, \vec{b}) \leftrightarrow F(x, \vec{a}, \vec{T}))$.

With elementary comprehension we get a universal type V containing every individual. Simply let F be the elementary formula $x = x$ and apply the above axioms.

Join

$$(15)\ \ \Re(a) \wedge (\forall x \mathrel{\dot\in} a)\Re(fx) \ \rightarrow\ \Re(\mathsf{j}(a,f)) \wedge \Sigma(a,f,\mathsf{j}(a,f)).$$

In this axiom the formula $\Sigma(a,f,b)$ means that b names the disjoint union of f over a, i.e.

$$\Sigma(a,f,b) \ :=\ \forall x(x \mathrel{\dot\in} b \leftrightarrow \exists y \exists z(x = (y,z) \wedge y \mathrel{\dot\in} a \wedge z \mathrel{\dot\in} fy)).$$

It is a well-known result that we can introduce λ abstraction and recursion using the combinator axioms for k and s, cf. Beeson [5] or Feferman [14].

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*

$$\mathsf{EETJ} \vdash \lambda x.t{\downarrow} \wedge (\lambda x.t)\, x \simeq t \ \textit{and}\ \mathsf{EETJ} \vdash s{\downarrow} \rightarrow (\lambda x.t)\, s \simeq t[s/x].$$

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

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

Now we introduce non-strict definition by cases (cf. e.g. Beeson [5]). Observe that if $\mathsf{d_N}abcd{\downarrow}$, then $a{\downarrow}$ and $b{\downarrow}$ 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 $\mathsf{d_s}abcd$ stand for the term $\mathsf{d_N}(\lambda z.a)(\lambda z.b)cd0$ where the variable z does not occur in the terms a and b. We will use the following notation for non-strict definition by cases

$$\mathsf{d_s}abcd \simeq \begin{cases} a & \text{if } c = d, \\ b & \text{else.} \end{cases}$$

This notation already anticipates the axiom $\forall x \mathsf{N}(x)$, otherwise we should add $\mathsf{N}(c) \wedge \mathsf{N}(d)$ 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. In the sequel we employ type induction on the natural numbers which is given by the following axiom ($\mathsf{T\text{-}I_N}$):

$$\forall X(0 \in X \wedge (\forall x \in \mathsf{N})(x \in X \rightarrow \mathsf{s_N}x \in X) \ \rightarrow\ (\forall x \in \mathsf{N})x \in X).$$

2.2 Fixed Point Types

The classes of Java will be modeled by types in explicit mathematics. Since the Java classes may be defined by mutual recursion, i.e. class `A` may contain an attribute of class `B` and vice versa, their interpretations have to be given as fixed point types in our theory of types and names. They can be constructed by the principle of *dependent choice* (dc). These axioms have been proposed by Jäger and their proof-theoretic analysis has been carried out by Probst [33].

Dependent choice.

($\mathsf{dc}.1$) $\Re(a) \wedge f \in (\Re \to \Re) \to \mathsf{dc}(a,f) \in (\mathsf{N} \to \Re)$,
($\mathsf{dc}.2$) $\Re(a) \wedge f \in (\Re \to \Re) \to$
$\mathsf{dc}(a,f)0 \simeq a \wedge (\forall n \in \mathsf{N})\mathsf{dc}(a,f)(n+1) \simeq f(\mathsf{dc}(a,f)n)$.

First, let us introduce some notation. By primitive recursion we can define in $\mathsf{EETJ} + (\mathsf{T}\text{-}\mathsf{I}_\mathsf{N})$ the usual relations $<$ and $\leq$ on the natural numbers. The i^{th} section of U is defined by $(U)_i := \{y \mid (i,y) \in U\}$. If s is a name for U, then $(s)_i$ represents the type $(U)_i$. By abuse of notation, we let the formula $(s)_i \in U$ stand for $\mathsf{p}_0 s = i \wedge \mathsf{p}_1 s \in U$. The context will ensure that it is clear how to read $(s)_i$. Product types are defined according to the definition of n tupling by $S_1 \times S_2 := \{(x,y) \mid x \in S_1 \wedge y \in S_2\}$ and

$$S_1 \times S_2 \times \cdots \times S_{n+1} := S_1 \times (S_2 \times \cdots \times S_{n+1}).$$

We define projection functions π_i^k for $k \geq 2$ and $1 \leq i \leq k$ so that

$$\pi_i^k(s_1, \ldots, s_k) \simeq s_i.$$

A *fixed point specification* is a system of formulas of the form

$$\begin{aligned} (X)_1 &= Y_{11} \times \cdots \times Y_{1m_1} \\ &\vdots \\ (X)_n &= Y_{n1} \times \cdots \times Y_{nm_n} \end{aligned}$$

where each Y_{ij} may be any type variable other than X or of the form

$$\{x \in X \mid \mathsf{p}_0 x = k_1\} \cup \cdots \cup \{x \in X \mid \mathsf{p}_0 x = k_l\}$$

for $k_i \leq n$. Those Y_{ij} which are just a type variable other than X are called *parameters* of the specification.

Our aim is to show that for every fixed point specification there exists a fixed point satisfying it and this fixed point can be named uniformly in the parameters of its specification.

Assume we are given a fixed point specification as above with parameters $\vec{Y}$. Then we find by elementary comprehension that there exists a closed individual term t of $\mathcal{L}$ so that EETJ proves for all $\vec{a}$ whose length is equal to the number of parameters of the specification:

1. $\Re(\vec{a}) \wedge \Re(b) \rightarrow \Re(t(\vec{a}, b))$,
2. $\Re(\vec{a}, \vec{Y}) \wedge \Re(b, X) \rightarrow$
$\forall x(x \mathrel{\dot\in} t(\vec{a}, b) \leftrightarrow (x)_1 \in Y_{11} \times \cdots \times Y_{1m_1} \vee \ldots \vee (x)_n \in Y_{n1} \times \cdots \times Y_{nm_n})$.

In the following we assume $\Re(\vec{a})$ and let a_{ij} denote that element of $\vec{a}$ which represents Y_{ij}. The term $\lambda x.t(\vec{a}, x)$ is an operator form mapping names to names. Note that it is monotonic, i.e.

$$b \mathrel{\dot\subset} c \rightarrow t(\vec{a}, b) \mathrel{\dot\subset} t(\vec{a}, c). \tag{1}$$

Starting from the empty type, represented by $\emptyset$, this operation can be iterated in order to define the stages of the inductive definition of our fixed point. To do so, we define a function f by:

$$f(\vec{a}, n) \simeq \mathsf{dc}(\emptyset, \lambda x.t(\vec{a}, x))n.$$

As a direct consequence of ($\mathsf{dc}.1$) we find $(\forall n \in \mathsf{N})\Re(f(\vec{a}, n))$. Hence, we let J be the type represented by $\mathsf{j}(\mathsf{nat}, \lambda x.f(\vec{a}, x))$. Making use of ($\mathsf{T\text{-}I_N}$) we can prove

$$(\forall n \in \mathsf{N})\forall x((n, x) \in J \rightarrow (n + 1, x) \in J)$$

and therefore

$$(\forall m \in \mathsf{N})(\forall n \in \mathsf{N})(m \leq n \rightarrow f(\vec{a}, m) \mathrel{\dot\subset} f(\vec{a}, n)). \tag{2}$$

We define the fixed point $\mathsf{FP} := \{x \mid (\exists n \in \mathsf{N})(n, x) \in J\}$. By the uniformity of elementary comprehension and join there exists a closed individual term fp so that $\mathsf{fp}(\vec{a})$ is a name for FP, i.e. the fixed point can be represented uniformly in its parameters. A trivial corollary of this definition is

$$(\exists n \in \mathsf{N})(x \mathrel{\dot\in} f(\vec{a}, n)) \leftrightarrow x \mathrel{\dot\in} \mathsf{fp}(\vec{a}). \tag{3}$$

The following theorem states that FP is indeed a fixed point of t. We employ $(s)_{ij} \in U$ as abbreviation for $\mathsf{p_0}s = i \wedge \pi_j^{m_i}(\mathsf{p_1}s) \in U$.

Theorem 2. *It is provable in* $\mathsf{EETJ} + (\mathsf{dc}) + (\mathsf{T\text{-}I_N})$ *that* FP *is a fixed point satisfying the fixed point specification, i.e.*

$$\Re(\vec{a}) \rightarrow \forall x(x \mathrel{\dot\in} \mathsf{fp}(\vec{a}) \leftrightarrow x \mathrel{\dot\in} t(\vec{a}, \mathsf{fp}(\vec{a}))).$$

Proof. Assume $x \mathrel{\dot\in} \mathsf{fp}(\vec{a})$. By (3) there exists a natural number n so that $x \mathrel{\dot\in} f(\vec{a}, n)$. By (2) we find $x \in f(\vec{a}, n+1)$ and by the definition of f we get $f(\vec{a}, n+1) = t(\vec{a}, f(\vec{a}, n))$. By (3) we obtain $f(\vec{a}, n) \mathrel{\dot\subset} \mathsf{fp}(\vec{a})$ and with (1) we conclude $x \in t(\vec{a}, \mathsf{fp}(\vec{a}))$. Next, we show $\forall x(x \mathrel{\dot\in} t(\vec{a}, \mathsf{fp}(\vec{a})) \rightarrow x \mathrel{\dot\in} \mathsf{fp}(\vec{a}))$. Let $x \mathrel{\dot\in} t(\vec{a}, \mathsf{fp}(\vec{a}))$, i.e. we have for all i, j with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ either $(x)_{ij} \mathrel{\dot\in} a_{ij}$ or by (3)

$$(x)_{ij} \in \{y \mid (\exists n \in \mathsf{N})y \mathrel{\dot\in} f(\vec{a}, n) \wedge \mathsf{p_0}y = k_1\} \cup \cdots \cup \{y \mid (\exists n \in \mathsf{N})y \mathrel{\dot\in} f(\vec{a}, n) \wedge \mathsf{p_0}y = k_l\}$$

depending on the specification. Since f is monotonic there exists a natural number n so that for all i, j with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ we have either $(x)_{ij} \mathrel{\dot\in} a_{ij}$ or $(x)_{ij}$ is an element of

$$\{y \mid y \mathrel{\dot\in} f(\vec{a}, n) \wedge \mathsf{p}_0 y = k_1\} \cup \cdots \cup \{y \mid y \mathrel{\dot\in} f(\vec{a}, n) \wedge \mathsf{p}_0 y = k_l\}$$

depending on the specification and this implies $x \mathrel{\dot\in} f(\vec{a}, n{+}1)$. Hence we conclude by (3) that $x \mathrel{\dot\in} \mathsf{fp}(\vec{a})$ holds. □

2.3 Least Fixed Point Operator

As shown in Theorem 1 the combinatory axioms of EETJ provide a term rec which solves recursive equations. However, it is not provable that the solution obtained by rec is minimal. Here we are going to extend $\mathsf{EETJ} + (\mathsf{T}\text{-}\mathsf{I}_\mathsf{N})$ with axioms about computability (Comp) and the statement that everything is a natural number. The resulting system allows to define a *least* fixed point operator and therefore it will be possible to show that recursively defined methods belong to a certain function space, cf. Kahle and Studer [30].

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 [5]. Note that these axioms can be satisfied in the usual recursion-theoretic model. The constant c can be interpreted by the characteristic function of Kleene's T predicate.

($\mathsf{Comp}.1$) $\forall f \forall x (\forall n \in \mathsf{N})(\mathsf{c}(f, x, n) = 0 \vee \mathsf{c}(f, x, n) = 1)$,
($\mathsf{Comp}.2$) $\forall f \forall x (fx{\downarrow} \leftrightarrow (\exists n \in \mathsf{N})\mathsf{c}(f, x, n) = 0)$.

In addition we will restrict the universe to natural numbers. This axiom is needed to construct the least fixed point operator. Of course, it is absolutely in the spirit of a recursion-theoretic interpretation.

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

For the rest of this section LFP will be the theory

$$\mathsf{EETJ} + (\mathsf{Comp}) + \forall x \mathsf{N}(x) + (\mathsf{T}\text{-}\mathsf{I}_\mathsf{N}).$$

We are going to define ordering relations $\sqsubseteq_T$ for certain types T. The meaning of $f \sqsubseteq_T g$ is that f is smaller than g with respect to the usual pointwise ordering of functions, e.g. we have

$$f \sqsubseteq_{A \rightharpoonup B} g \to (\forall x \in A)(fx{\downarrow} \to fx = gx).$$

Definition 1. *Let $A_1, \ldots, A_n, B_1, \ldots, B_n$ be types Further, let $\mathcal{T}$ be the type $(A_1 \rightharpoonup B_1) \cap \cdots \cap (A_n \rightharpoonup B_n)$. We define:*

$$\begin{aligned} f \sqsubseteq g &:= f{\downarrow} \to f = g, \\ f \sqsubseteq_{\mathcal{T}} g &:= \bigwedge_{1 \leq i \leq n} (\forall x \in A_i) fx \sqsubseteq gx), \\ f \cong_{\mathcal{T}} g &:= f \sqsubseteq_{\mathcal{T}} g \wedge g \sqsubseteq_{\mathcal{T}} f. \end{aligned}$$

Definition 2. *Let $\mathcal{T}$ be as given in Definition 1. A function $f \in (\mathcal{T} \to \mathcal{T})$ is called* $\mathcal{T}$ monotonic, if

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

Using the rec term we will find a fixed point for every operation g. But as we have mentioned above we cannot prove in EETJ that this is a least fixed point, and of course, there are terms g that do not have a least fixed point. However, there exists a closed individual term I of $\mathcal{L}$ so that LFP proves that $\mathsf{I}g$ is the least fixed point of a monotonic functional $g \in (\mathcal{T} \to \mathcal{T})$. A proof of the following theorem can be found in Kahle and Studer [30].

Theorem 3. *There exists a closed individual term I of $\mathcal{L}$ such that 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 1, then*

1. $\mathsf{I}g \in \mathcal{T}$,
2. $\mathsf{I}g \cong_{\mathcal{T}} g(\mathsf{I}g)$,
3. $f \in \mathcal{T} \wedge gf \cong_{\mathcal{T}} f \to \mathsf{I}g \sqsubseteq_{\mathcal{T}} f$.

Now we define the theory PTN about programming with types and names as the union of all these axioms:

$$\mathsf{PTN} := \mathsf{EETJ} + (\mathsf{dc}) + (\mathsf{Comp}) + \forall x \mathsf{N}(x) + (\mathsf{T}\text{-}\mathsf{I}_{\mathsf{N}}).$$

The axioms about computability can be interpreted in the usual recursion-theoretic model, see Beeson [5] or Kahle [29]. This means that 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 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 $(u)_0 = n$. Hence, it can be used to verify the axioms in a recursion-theoretic interpretation; and of course, $\forall x \mathsf{N}(x)$ will also be satisfied in such a model.

Probst [33] presents a recursion-theoretic model for the system $\mathsf{EETJ} + (\mathsf{dc}) + (\mathsf{T}\text{-}\mathsf{I}_{\mathsf{N}})$ and shows that the proof-theoretic ordinal of this theory is $\varphi\omega 0$, i.e. it is slightly stronger than Peano arithmetic but weaker than Martin-Löf type theory with one universe ML_1 or the system $\mathsf{EETJ} + (\mathcal{L}\text{-}\mathsf{I}_{\mathsf{N}})$ of explicit mathematics with elementary comprehension, join and full induction on the natural numbers.

These two constructions can be combined in order to get a recursion-theoretic model for PTN so that computations in PTN are modeled by ordinary recursion-theoretic functions. We obtain that PTN is proof-theoretically still equivalent to $\mathsf{EETJ} + (\mathsf{dc}) + (\mathsf{T}\text{-}\mathsf{I}_{\mathsf{N}})$. Hence, PTN is a predicative theory which is proof-theoretically much weaker than the systems that are usually used to talk about object-oriented programming, for most of these calculi are extensions of system F that already contains full analysis, cf. e.g. Bruce, Cardelli and Pierce [8]. Nevertheless, PTN is sufficiently strong to model Featherweight Java and to prove many properties of the represented programs. In PTN we can also prove soundness of our interpretation with respect to subtyping, typing and reductions.

3 Featherweight Java

Featherweight Java is a minimal core calculus for Java proposed by Igarashi, Pierce and Wadler [25] for the formal study of an extension of Java with parameterized classes. Igarashi and Pierce [24] employed Featherweight Java also to obtain a precise understanding of inner classes. FJ is a minimal core calculus in the sense that as many features of Java as possible are omitted, while maintaining the essential flavor of the language and its type system. Nonetheless, this fragment is large enough to include many useful programs. In particular, most of the examples in Felleisen and Friedman's text [19] are written in the purely functional style of Featherweight Java. In this section we will present the formulation of Featherweight Java given in [24].

Syntax. The abstract syntax of FJ class declarations, constructor declarations, method declarations and expressions is given by:

$$
\begin{array}{rcl}
\texttt{CL} & ::= & \texttt{class C extends C } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\
\texttt{K} & ::= & \texttt{C } (\overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{f}});\ \texttt{this.}\overline{\texttt{f}}\ =\ \overline{\texttt{f}};\} \\
\texttt{M} & ::= & \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\texttt{return e};\} \\
\texttt{e} & ::= & \texttt{x} \\
 & | & \texttt{e.f} \\
 & | & \texttt{e.m}(\overline{\texttt{e}}) \\
 & | & \texttt{new C}(\overline{\texttt{e}}) \\
 & | & \texttt{(C)e}
\end{array}
$$

The meta-variables `A,B,C,D,E` range over class names, `f` and `g` range over field names, `m` ranges over method names, `x` ranges over variable names and `d,e` range over expressions (all possibly with subscripts). `CL` ranges over class declarations, `K` ranges over constructor declarations and `M` ranges over method declarations. We assume that the set of variables includes the special variable `this`, but that `this` is never used as the name of an argument to a method.

We write $\overline{\texttt{f}}$ as shorthand for $\texttt{f}_1, \ldots, \texttt{f}_n$ (and similarly for $\overline{\texttt{C}}, \overline{\texttt{x}}, \overline{\texttt{e}}$, etc.) and we use $\overline{\texttt{M}}$ for $\texttt{M}_1 \ldots \texttt{M}_n$ (without commas). The empty sequence is written as $\bullet$ and $\sharp(\overline{\texttt{x}})$ denotes the length of the sequence $\overline{\texttt{x}}$. Operations on pairs of sequences are abbreviated in the obvious way, e.g. "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" stands for "$\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$" and "$\overline{\texttt{C}}\ \overline{\texttt{f}};$" is a shorthand for "$\texttt{C}_1\ \texttt{f}_1; \ldots; \texttt{C}_n\ \texttt{f}_n;$" and similarly "$\texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};$" abbreviates "$\texttt{this.f}_1 = \texttt{f}_1; \ldots; \texttt{this.f}_n = \texttt{f}_n;$". We assume that sequences of field declarations, parameter names and method declarations contain no duplicate names.

A *class table* CT is a mapping from class names `C` to class declarations `CL`. A program is a pair $(CT, \texttt{e})$ of a class table and an expression. In the following we always assume that we have a *fixed* class table CT which satisfies the following sanity conditions:

1. $CT(\texttt{C}) = \texttt{class C} \ldots$ for every `C` in the domain of CT, i.e. the class name `C` is mapped to the declaration of the class `C`,

2. `Object` is not an element of the domain of CT,
3. every class `C` (except `Object`) appearing anywhere in CT belongs to the domain of CT,
4. there are no cycles in the subtype relation induced by CT, i.e. the $<:$ relation is antisymmetric.

Subtyping. The following rules define the subtyping relation $<:$ which is induced by the class table. Note that every class defined in the class table has a super class, declared with `extends`.

$$\frac{}{\mathtt{C} <: \mathtt{C}}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}$$

Computation. These rules define the reduction relation $\longrightarrow$ which models field accesses, method calls and casts. In order to look up fields and method declarations in the class table we use some auxiliary functions that will be defined later on. We write $\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{this}]$ for the result of simultaneously replacing $\mathtt{x}_1$ by $\mathtt{d}_1, \ldots, \mathtt{x}_n$ by $\mathtt{d}_n$ and `this` by `e` in the expression $\mathtt{e}_0$.

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{f_i} \longrightarrow \mathtt{e_i}}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow \mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}]}$$

$$\frac{\mathtt{C} <: \mathtt{D}}{(\mathtt{D})\mathtt{new\ C}(\overline{\mathtt{e}}) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{e}})}$$

We say that an expression `e` is in *normal form* if there is no expression `d` so that $\mathtt{e} \longrightarrow \mathtt{d}$.

Now we present the typing rules for expressions, method declarations and class declarations. An environment Γ is a finite mapping from variables to class names, written $\overline{\mathtt{x}} : \overline{\mathtt{C}}$. Again, we employ some auxiliary functions which will be given later. Stupid casts (the last of the expression typing rules) are included only for technical reasons, cf. Igarashi, Pierce and Wadler [25]. The Java compiler will reject expressions containing stupid casts as ill typed. This is expressed by the hypothesis *stupid warning* in the typing rule for stupid casts.

Expression typing.

$$\frac{}{\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f_i} \in \mathtt{C_i}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e_o} \in \mathtt{C}_0 \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{C}}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \text{stupid warning}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}$$

Method typing.

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 \in \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\} \\ \text{if } \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0, \text{ then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ e}_0;\}\ \text{OK in } \mathtt{C}}$$

Class typing.

$$\frac{\begin{array}{c}\mathtt{K = C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}}); \mathtt{this.}\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}} \text{ OK in } \mathtt{C}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \text{OK}}$$

We define the auxiliary function which are used in the rules for computation and typing.

Field lookup.

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

Method type lookup.

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ e};\} \text{ belongs to } \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}(\mathtt{m}, \mathtt{D})}$$

Method body lookup.

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ e};\} \text{ belongs to } \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{C}) = mbody(\mathtt{m}, \mathtt{D})}$$

We call a Featherweight Java expression $\mathtt{e}$ *well-typed* if $\Gamma \vdash \mathtt{g} \in \mathtt{C}$ can be derived for some environment Γ and some class $\mathtt{C}$.

Igarashi, Pierce and Wadler [25] prove that if an FJ program is well-typed, then the only way it can get stuck is if it reaches a point where it cannot perform a downcast. This is stated in the following theorem about progress.

Theorem 4. *Suppose* $\mathtt{e}$ *is a well-typed expression.*

1. *If the expression* $\mathtt{e}$ *is of the form* $\mathtt{new\ C_0}(\overline{\mathtt{e}}).\mathtt{f}$ *or contains such a subexpression, then* $fields(\mathtt{C_0}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} \in \overline{\mathtt{f}}$.
2. *If* $\mathtt{e}$ *is of the form* $\mathtt{new\ C_0}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$ *or contains such a subexpression, then* $mbody(\mathtt{m}, \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{e_0})$ *and* $\sharp(\overline{\mathtt{x}}) = \sharp(\overline{\mathtt{d}})$.

4 The Object Model

In this section we present the object model of Castagna, Ghelli and Longo [9, 11] which will be employed to interpret Featherweight Java in explicit mathematics. In his book [9], Castagna introduces a kernel object-oriented language KOOL and defines its semantics via an interpretation in a meta-language λ_object for which an operational semantics is given. Our construction will be very similar in spirit to Castagna's interpretation, although we will have to solve many new problems since we start with an already existing language and will end up with a recursion-theoretic model for it.

In the object model we use, the state of an object is separated from its methods. Only the fields of an object are bundled together as one unit, whereas the methods of an object are *not* encapsulated inside it. Indeed, methods are implemented as branches of global *overloaded* functions. If a message is sent to an object, then this message determines a function and this function will be applied to the receiving object. However, messages are not ordinary functions. For if the same message is sent to objects of different classes, then different methods may be retrieved, i.e. different code may be executed. Hence, messages represent overloaded functions: depending on the type of their argument (the object the message is passed to), a different method is chosen. Since this selection of the method is based on the dynamic type of the object, i.e. its type at run-time, we also have to deal with *late-binding*.

In our semantics of FJ, objects are modeled as pairs (type, value). This encoding of objects was proposed by Castagna [9] in order to interpret late-binding. The value component will be a record consisting of all the fields of the object and the type component codes the run-time type of the object. All the methods with the same name, although maybe defined in different classes, are combined to one overloaded function which takes the receiving object as an additional argument. Hence, the type information contained in the interpretation of the receiving object can be used to resolve this overloaded application, i.e. to select

the code to be executed. Since methods may be defined recursively, the interpreting function has to be given by recursion, too. As we will see later, in order to obtain a sound model with respect to typing, we even need a *least* fixed point operator for the definition of the semantics of methods. Since several methods may call each other by mutual recursion, we have to give their interpretation using one recursive definition treating all methods in one go.

As Castagna [9] notices, even λ calculi with overloading and late-binding do not posses enough structure to reason about object-oriented programs. Hence he introduces the meta-language λ_object which is still based on overloading and late-binding but which also is enriched with new features (e.g. commands to define new types or to handle the subtyping hierarchy) that are necessary to reproduce the constructs of a programming language. Therefore λ_object is an appropriate tool for representing object-oriented programs. However, in λ_object it is not possible to state (or even prove) properties of these programs.

Our theories of types and names have much more expressive power. They provide not only an axiomatic framework for representing programs, but also for stating and proving properties of programs. Nevertheless, from a proof-theoretic point of view they are quite weak (only a bit stronger than Peano Arithmetic) as demanded by Feferman [16–18] or Turner [44]. In its general form overloading and late-binding seem to be proof-theoretically very strong, cf. Studer [37]. However, our work shows that in order to model real object-oriented programming languages we do not need the full power of these principles; as already mentioned by Castagna, Ghelli and Longo [9, 11] a predicative variant suffices for our practical purposes.

Last but not least, the theory of types and names we use in this paper has a standard recursion-theoretic model. Hence, our interpretation of FJ in explicit mathematics shows how computations in Featherweight Java can be seen as ordinary mathematical functions and the interpretation of the classes will be given by sets in the usual mathematical sense.

5 Evaluation Strategy and Typing

Now we study some examples written in Featherweight Java which will motivate our semantics for FJ as presented in the next section. We will focus on Java's evaluation strategy and on typing issues. In the last example the interplay of free variables, static types and late-binding is investigated.

Java features a call-by-value evaluation strategy, cf. the Java language specification by Gosling, Joy and Steele [23]. This corresponds to the strictness axioms of the logic of partial terms upon which explicit mathematics is built. They imply that an application only has a value, i.e. is terminating, if all its arguments have a value.

In Java we not only have non-terminating programs we also have run-time exceptions, for example when an illegal down cast should be performed. With respect to these features, Featherweight Java is much more coarse grained. There is no possibility to state that a program terminates and exceptions are completely

ignored. For good reasons, as we should say, since it is intended as a minimal core calculus for modeling Java's type system. However, this lack of expressiveness has some important consequences which will be studied in the sequel. Let us first look at the following example.

Example 1.

```
class A extends Object {
  A () { super(); }
  C m() {
    return this.m();
  }
}

class C extends Object {
    int x;
    A y;
    C (int a,A b) {
        super();
        this.x = a;
        this.y = b;
    }
}
```

Of course, `new A().m()` is a non-terminating loop. Although, if it is evaluated on an actual Java implementation, then we get after a short while a `java.lang.StackOverflowError` because of too many recursive method calls. In Featherweight Java `new A().m()` has no normal form which reflects the fact that it loops forever.

Now let `e` be the expression `new C(5,new A().m()).x`. Due to Java's call-by-value evaluation strategy, the computation of this expression will not terminate either, for `new A().m()` is a subexpression of `e` and has therefore to be evaluated first.

The operational semantics of Featherweight Java uses a non-deterministic small-step reduction relation which does not enforce a call-by-value strategy. Hence we have two different possibilities for reduction paths starting from `e`. If we adopt a call-by-value strategy, then we have to evaluate `new A().m()` first and we obtain an infinite reduction path starting from `e`. Since FJ's reduction relation is non-deterministic we also have the possibility to apply the computation rule for field access. If we decide to do so, then `e` reduces to 5 which is in normal form.

In theories of types and names we have the possibility to state that a computation terminates. The formula $t\!\downarrow$ expresses that t has a value, meaning the computation represented by t is terminating. Let $[\![\mathtt{e}]\!]$ be the interpretation of the expression `e`. In our mathematical model Java's call-by-value strategy is implemented by the strictness axioms, hence $\neg[\![\mathtt{e}]\!]\!\downarrow$ will be provable. Since 5 surely has a value we obtain $[\![\mathtt{e}]\!] \not\simeq 5$ although 5 is the normal form of `e`. This means that

in our interpretation we cannot model the non-deterministic reduction relation of Featherweight Java but we will implement a call-by-value strategy.

Non-terminating programs are not the only problem in modeling computations of Java. A second problem is the lack of a notion of run-time exception in Featherweight Java. For example, if a term is in normal form, then we cannot tell, whether this is the case because the computation finished properly, or because an illegal down-cast should be performed. It may even be the case that the final expression does not contain any down-casts at all, but earlier during the computation an exception should have been thrown. Let us illustrate this fact with the following example, where the class `C` is as in Example 1.

Example 2.

```
class main extends Object{
  public static void main (String arg[]) {
      System.out.println(new C(5,(A)(new Object())).x);
  }
}
```

If we run this `main` method, then Java throws the following exception:

```
java.lang.ClassCastException: java.lang.Object
        at main.main(main.java:4)
```

Whereas in Featherweight Java the expression

$$\texttt{new C(5,(A)(new Object())).x}$$

reduces to 5. This is due to the fact that the term $(\texttt{C})(\texttt{new Object}())$, which causes the exception in Java, is treated as final value in Featherweight Java and therefore it can be used as argument in further method calls.

In our model we will introduce a special value ex to denote the result of a computation which throws an exception. An illegal down cast produces $(\mathsf{ex}, 0)$ as result and we can check every time an expression is used as argument in a method invocation or in a constructor call whether its value is $(\mathsf{ex}, 0)$ or not. If it is not, then the computation can continue; but if an argument value represents an exception, then the result of the computation is this exception value. Therefore in our model we can distinguish whether an exception occurred or not. For example, the above expression evaluates to $(\mathsf{ex}, 0)$. We will have

$$\begin{aligned}[\![\texttt{(A)(new Object())}]\!] &= \mathsf{cast}\,\texttt{A}^*\,[\![\texttt{new Object()}]\!] \\ &= \mathsf{cast}\,\texttt{A}^*\,(\texttt{Object}^*, 0) \\ &= (\mathsf{ex}, 0)\end{aligned}$$

since $\mathsf{sub}(\texttt{Object}^*, \texttt{A}^*) \neq 1$, i.e. `Object` is not a subclass of `A`. Later, we will define the operation $*$ so that if `A` is a name for a class, then $\texttt{A}^*$ is a numeral in $\mathcal{L}$. Then we define a term `sub` which decides the subclass relation. That is for two class names `A` and `B`, we have $\mathsf{sub}(\texttt{A}^*, \texttt{B}^*) = 1$ if and only if `A` is a subclass of `B`. The term `cast` will be used to model casts. If o is the interpretation of an

object and A is a class to which this object should be casted, then $\mathsf{cast}\,\mathtt{A}^*\,o$ is the result of this cast. The term cast uses sub to decide whether it is a legal cast. If it is not, as in the above example, the cast will return $(\mathsf{ex}, 0)$.

From these considerations it follows that we cannot prove soundness of our model construction with respect to reductions as formalized in Featherweight Java. However, we are going to equip FJ with a restricted reduction relation $\longrightarrow'$ which enforces a call-by-value evaluation strategy as it is used in the Java language and which also respects illegal down casts. With respect to this new notion of reduction we will be able to prove that our semantics adequately models FJ computations.

In Featherweight Java we cannot talk about the termination of programs. As usual in type systems for programming languages the statement "expression e has type T" has to be read as "if the computation of e terminates, then its result is of type T". Let A be the class of Example 1. Then in FJ `new A().m()` $\in$ C is derivable, although the expression `new A().m()` denotes a non-terminating loop. Hence in our model we will have to interpret $\mathtt{e} \in \mathtt{C}$ as $[\![\mathtt{e}]\!]\downarrow \to [\![\mathtt{e}]\!] \in [\![\mathtt{C}]\!]$.

As we have seen before, the computation of e may result in an exception. In this case we have $[\![\mathtt{e}]\!] = (\mathsf{ex}, 0)$ which is a defined value. Hence, by our interpretation of the typing relation, we have to include $(\mathsf{ex}, 0)$ to the interpretation of every type.

In the following we consider a class B which is the same as A except that the result type of the method m is changed to D.

```
class A extends Object {
  A () { super(); }
  C m() {
    return this.m();
  }
}

class B extends Object {
  B () { super(); }
  D m() {
    return this.m();
  }
}
```

Since the method bodies for m are the same in both classes A and B we can assume that the interpretations of `(new A()).m()` and `(new B()).m()` will be the same, that is

$$[\![\texttt{new A().m()}]\!] \simeq [\![\texttt{new B().m()}]\!].$$

In this example the classes C and D may be chosen arbitrarily. In particular, they may be disjoint, meaning that maybe, there is no object belonging to both of them. Hence, if our modeling of the typing relation is sound, it follows that we are in the position to prove that the computation of `new B().m()` is non-terminating, that is $\neg[\![\texttt{(new B()).m()}]\!]\downarrow$.

Usually, in lambda calculi such recursive functions are modeled using a fixed point combinator. In continuous λ-models, such as $\mathsf{P}\omega$ or D_∞, these fixed point combinators are interpreted by *least* fixed point operators and hence one can show that certain functions do not terminate. In applicative theories on the other hand, recursive equations are solved with the rec term provided by the recursion theorem. Unfortunately, one cannot prove that this operator yields a least fixed point; and hence, it is not provable that certain recursive functions do not terminate. Therefore we have to employ the special term I to define the semantics of FJ expressions. Since this term provides a least solution to certain fixed point equations, it will be possible to show $\neg[\![\texttt{(new B()).m()}]\!]{\downarrow}$ which is necessary for proving soundness of our interpretation with respect to typing.

Now we are going to examine the role of free variables in the context of static types and late-binding. In the following example let `C` be an arbitrary class with no fields.

Example 3.

```
class A extends Object{
  A () { super(); }
  C m() {
    return this.m();
  }
}

class B extends A{
  B () { super(); }
  C m() {
    return new C();
  }
}
```

As in Example 1 class `A` defines the method `m` which does not terminate. Class `B` extends `A`, hence it is a subclass of `A`, and it overrides method `m`. Here `m` creates a new object of type `C` and returns it to the calling object.

Let `x` be a free variable with (static) type `A`. The rules for method typing guarantee that the return type of `m` cannot be changed by the overriding method; and by the typing rules of FJ we can derive $\texttt{x:A} \vdash \texttt{x.m()} \in \texttt{C}$. As we have seen before this means "if `x.m()` yields a result, then it belongs to `C`." Indeed, as a consequence of Java's late-binding evaluation strategy, knowing only the static type `A` of `x` we cannot tell whether in `x.m()` the method `m` defined in class `A` or the one of class `B` will be executed. Hence we do not know whether this computation terminates or not. Only if we know the object which is referenced by `x` we can look at its dynamic type and then say, by the rules of method body lookup, which method actually gets called.

This behavior has the consequence that there are FJ expression in normal form whose interpretation will not have a value. For example, `x.m()` is in normal form, but maybe `x` references an object of type `A` and in this case the

interpretation of x.m() will not have a value. Therefore we conclude that only the interpretation of a *closed* term in normal form will always be defined.

6 Interpreting Featherweight Java

Assume we are given a program written in Featherweight Java. This consist of a fixed class table CT and an expression e. In this section we will show how to translate such a program into the language of types and names. This allows us to state and prove properties of FJ programs. In the sequel we will work with the theory PTN of explicit mathematics.

We generally assume that all classes and methods occurring in our fixed class table CT are well-typed. This means for every class C of CT we can derive class C...OK by the rules for class typing and for every method m defined in this class we can derive ...m...OK IN C by the rules for method typing.

The *basic types* of Java such as boolean, int, ... are not included in FJ. However, EETJ provides a rich type structure which is well-suited to model these basic data types, cf. e.g. Feferman [16] or Jäger [28]. Hence we will include them in our modeling of Featherweight Java.

Let $*$ be an injective mapping from all the names for classes, basic types, fields and methods occurring in the class table CT into the numerals of $\mathcal{L}$. This mapping will be employed to handle the run-time type information of FJ terms as well as to model field access and method selection.

First we show how objects will be encoded as sequences in our theory of types and names. Let C be a class of our class table CT with $fields(\mathtt{C}) = \mathtt{D_1}\ \mathtt{g_1}, \ldots, \mathtt{D_n}\ \mathtt{g_n}$ and let $m_\mathtt{C}$ be the least natural number such that for all field names $\mathtt{g_j}$ occurring in $fields(\mathtt{C})$ we have $\mathtt{g_j}^* < m_\mathtt{C}$. An object of type C will be interpreted by a sequence $(\mathtt{C}^*, (s_1, \ldots, s_{m_\mathtt{C}}))$, where s_i is the interpretation of the field $\mathtt{g_j}$ if $i = \mathtt{g_j}^*$ and $s_i = 0$ if there is no corresponding field. In particular, we always have $s_{m_\mathtt{C}} = 0$. Note that in this model the type of an object is encoded in the interpretation of the object.

We have to find a way for dealing with invalid down casts. What should be the value of (A) new Object() in our model, when A is a class different from Object? In FJ the computation simply gets stuck, no more reductions will be performed. In our model we choose a natural number ex which is not in the range of * and set the interpretation of illegal down casts to $(\mathtt{ex}, 0)$. This allows us to distinguish them from other expressions using definition by cases on the natural numbers. Hence, every time when an expression gets evaluated we can check whether one of its arguments is the result of an illegal cast.

This is the reason why we will have to add run-time type information to elements of basic types, too. Let us look for example at the constant 17 of Java which surely is of type int. If it is simply modeled by the $\mathcal{L}$ term 17, then it might happen that $17 = (\mathtt{ex}, 0)$ and we could not decide whether this $\mathcal{L}$ term indicates that an illegal down cast occurred or whether it denotes the constant 17 of Java. On the other hand, if the Java constant 17 is modeled by $(\mathtt{int}^*, 17)$,

i.e. with run-time type information, then it is provably different from $(\mathsf{ex}, 0)$. The next example illustrates the coding of objects.

Example 4. Assume we have the following class modeling points.

```
class Point extends Object{
  int x;
  int y;

  Point (int a, int b) {
    super();
    this.x = a;
    this.y = b;
  }
}
```

Assume $*$ is such that $\mathtt{x}^* = 1$ and $\mathtt{y}^* = 3$. Hence, we have $m_{\mathtt{Point}} = 4$ An object of the class `Point` where `x=5` and `y=6` is now modeled by by the sequence $(\mathtt{Point}^*, ((\mathtt{int}^*, 5), 0, (\mathtt{int}^*, 6), 0))$.

In order to deal with the subtype hierarchy of FJ, we define a term sub modeling the subtype relation.

Definition 3 (of the term sub). *Let the term sub be so that for all $a, b \in \mathsf{N}$ we have:*

1. *If a or b codes a basic type, e.g. $a = \mathtt{A}^*$ for a basic type `A`, and $a = b$, then $\mathsf{sub}(a, b) = 1$.*
2. *If $\mathtt{C} <: \mathtt{D}$ can be derived for two classes `C` and `D` and $\mathtt{C}^* = a$ as well as $\mathtt{D}^* = b$ hold, then $\mathsf{sub}(a, b) = 1$.*
3. *Otherwise we set $\mathsf{sub}(a, b) = 0$.*

Since CT is finite, sub can be defined using definition by cases on the natural numbers; recursion is not needed.

In the following, we will define a semantics for expression of Featherweight Java. In a first step, this semantics will be given only relative to a term invk which is used to model method calls. As we have shown in our discussion of the object model, we can define invk in a second step as the least fixed point of a recursive equation involving all methods occurring in our fixed class table.

Of course, if at some stage of a computation an invalid down cast occurs and we obtain $(\mathsf{ex}, 0)$ as intermediate result, then we have to propagate it to the end of the computation. Therefore all of the following terms are defined by distinguishing two cases: if none of the arguments equals $(\mathsf{ex}, 0)$, then the application will be evaluated; if one of the arguments is $(\mathsf{ex}, 0)$, then the result is also $(\mathsf{ex}, 0)$.

We we define a term proj in order to model field access.

Definition 4 (of the term proj). *Let the term proj be so that*

$$\mathsf{proj}\, i\, x \simeq \begin{cases} x & \mathsf{p}_0 x = \mathsf{ex}, \\ \mathsf{p}_0(\mathsf{tail}\, i\, (\mathsf{p}_1 x)) & \textit{otherwise}, \end{cases}$$

where tail *is defined by primitive recursion such that*

$$\mathsf{tail}\,1\,s \simeq s \qquad \mathsf{tail}\,(n+1)\,s \simeq \mathsf{p_1}(\mathsf{tail}\,n\,s)$$

for all natural numbers $n \geq 1$.

Hence for $i \leq n$ and $t \neq \mathsf{ex}$ we have

$$\mathsf{proj}\,i\,(t,(s_1,\ldots,s_n,s_{n+1})) \simeq s_i.$$

Next, we show how to define the interpretation of the keyword `new`.

Definition 5 (of the term new**).** *For every class* `C` *of our class table* CT *with*

$$fields(\mathtt{C}) = \mathtt{D_1}\ \mathtt{g_1},\ldots,\mathtt{D_n}\ \mathtt{g_n}$$

we find a closed $\mathcal{L}$ *term* $t_{\mathtt{C}}$ *such that:*

1. *If* $a_i{\downarrow}$ *holds for all* a_i $(i \leq n)$ *and if there is a natural number* j *such that* $\mathsf{p_0}a_j = \mathsf{ex}$*, then we get* $t_{\mathtt{C}}(a_1,\ldots,a_n) = a_j$ *where* j *is the least number satisfying* $\mathsf{p_0}a_j = \mathsf{ex}$.
2. *Else we find* $t_{\mathtt{C}}(a_1,\ldots,a_n) \simeq (\mathtt{C}^*,(b_1,\ldots,b_{m_{\mathtt{C}}}))$*, where* $b_i \simeq a_j$ *if there exists* $j \leq n$ *with* $i = \mathtt{g_j}^*$ *and* $b_i = 0$ *otherwise.*

Using definition by cases on the natural numbers we can build a term new *so that* $\mathsf{new}\,\mathtt{C}^*\,(\vec{s}) \simeq t_{\mathtt{C}}(\vec{s})$ *for every class* `C` *in* CT.

The next example shows how the terms $t_{\mathtt{C}}$ work.

Example 5. Consider the class `Point` of Example 4. We get

$$t_{\mathtt{Point}}((\mathtt{int}^*,5),(\mathtt{int}^*,6)) \simeq (\mathtt{Point}^*,((\mathtt{int}^*,5),0,(\mathtt{int}^*,6),0))$$

and

$$t_{\mathtt{Point}}((\mathtt{int}^*,5),(\mathsf{ex},0)) \simeq (\mathsf{ex},0).$$

If b is a term so that $\neg b{\downarrow}$ holds, then we get $\neg t_{\mathtt{Point}}(a,b){\downarrow}$ by strictness even if $a = (\mathsf{ex},0)$. The strictness principle of explicit mathematics implies Java's call-by-value strategy.

Again, using definition by cases we build a term cast.

Definition 6 (of the term cast**).** *Let the term* cast *be so that*

$$\mathsf{cast}\,a\,b \simeq \begin{cases} (\mathsf{ex},0) & \mathsf{sub}(\mathsf{p_0}b,a) = 0, \\ b & \textit{otherwise.} \end{cases}$$

Now, we give the translation $[\![\mathtt{e}]\!]_{\mathsf{invk}}$ of a Featherweight Java expression `e` into an $\mathcal{L}$ term relative to a term invk.

Definition 7 (of the interpretation $[\![\mathsf{e}]\!]_{\mathsf{invk}}$ relative to invk). *For a sequence $\overline{\mathsf{e}} = \mathsf{e}_1, \ldots, \mathsf{e}_n$ we write $[\![\overline{\mathsf{e}}]\!]_{\mathsf{invk}}$ for $[\![\mathsf{e}_1]\!]_{\mathsf{invk}}, \ldots, [\![\mathsf{e}_n]\!]_{\mathsf{invk}}$. We assume that for every variable $\boldsymbol{x}$ of Featherweight Java there exists a corresponding variable x of $\mathcal{L}$ such that two different variables of FJ are mapped to different variables of $\mathcal{L}$. In particular, we suppose that our language $\mathcal{L}$ of types and names includes a variable this so that $[\![\boldsymbol{this}]\!]_{\mathsf{invk}} = this$.*

$$\begin{aligned}
[\![\mathtt{x}]\!]_{\mathsf{invk}} &:= x \\
[\![\mathtt{e.f}]\!]_{\mathsf{invk}} &:= \mathsf{proj}\,\mathtt{f}^*\,[\![\mathtt{e}]\!]_{\mathsf{invk}} \\
[\![\mathtt{e.m}(\overline{\mathtt{f}})]\!]_{\mathsf{invk}} &:= \mathsf{invk}(\mathtt{m}^*, [\![\mathtt{e}]\!]_{\mathsf{invk}}, [\![\overline{\mathtt{f}}]\!]_{\mathsf{invk}}) \\
[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{invk}} &:= \mathsf{new}\,\mathtt{C}^*([\![\overline{\mathtt{e}}]\!]_{\mathsf{invk}}) \\
[\![(\mathtt{C})\mathtt{e}]\!]_{\mathsf{invk}} &:= \mathsf{cast}\,\mathtt{C}^*\,[\![\mathtt{e}]\!]_{\mathsf{invk}}
\end{aligned}$$

In the following we are going to define the term invk which models method calls. To this aim, we have to deal with overloading and late-binding in explicit mathematics, cf. Studer [37, 39].

Definition 8 (of overloaded functions). *Assume we are given n natural numbers $s_1, \ldots, s_n$. Using sub we build for each $j \leq n$ a term $\mathsf{min}^j_{s_1,\ldots,s_n}$ such that for all natural numbers s we have $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 0 \vee \mathsf{min}^j_{s_1,\ldots,s_n}(s) = 1$ and $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 1$ if and only if*

$$\mathsf{sub}(s, s_j) = 1 \wedge \bigwedge_{\substack{1 \leq l \leq n \\ l \neq j}} (\mathsf{sub}(s, s_l) = 1 \rightarrow \mathsf{sub}(s_l, s_j) = 0).$$

Hence, $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 1$ holds if s_j is a minimal element (with respect to sub) of the set $\{s_i \,|\, \mathsf{sub}(s, s_i) = 1 \wedge 1 \leq i \leq n\}$; and otherwise we have $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 0$.

We can define a term $\mathsf{over}_{s_1,\ldots,s_n}$ which combines several functions $f_1, \ldots, f_n$ to one overloaded function $\mathsf{over}_{s_1,\ldots,s_n}(f_1, \ldots, f_n)$ such that

$$\mathsf{over}_{s_1,\ldots,s_n}(f_1, \ldots, f_n)\,(x, \vec{y}) \simeq \begin{cases} f_1(x, \vec{y}) & \mathsf{min}^1_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1, \\ \vdots & \\ f_n(x, \vec{y}) & \mathsf{min}^n_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1 \\ & \bigwedge\limits_{i<n} \mathsf{min}^i_{s_1,\ldots,s_n}(\mathsf{p}_0 x) \neq 1, \\ (\mathsf{ex}, 0) & \mathsf{p}_0 x = \mathsf{ex}. \end{cases}$$

Next, we define the term r which gives the recursive equation which will be solved by invk.

Definition 9 (of the term r). *Assume the method $\mathtt{m}$ is defined exactly in the classes $\mathtt{C}_1, \ldots, \mathtt{C}_n$ and $mbody(\mathtt{m}, \mathtt{C}_i) = (\overline{\mathtt{x}}_i, \mathtt{e}_i)$ for all $i \leq n$. Assume further that $\overline{\mathtt{x}}_i$ is $\mathtt{x}_1, \ldots, \mathtt{x}_z$, then we can define an $\mathcal{L}$ term $g^{\mathsf{invk}}_{\mathtt{e}_i}$ so that we have for $\vec{b} = b_1, \ldots, b_z$:*

1. *If* $a\downarrow$ *and* $\vec{b}\downarrow$ *hold and if there is a natural number* j *such that* $\mathsf{p}_0 b_j = \mathsf{ex}$*, then we get* $g^{\mathsf{invk}}_{\mathsf{e}_i}(a,\vec{b}) = b_j$ *where* j *is the least number satisfying* $\mathsf{p}_0 b_j = \mathsf{ex}$*.*
2. *Else we find* $g^{\mathsf{invk}}_{\mathsf{e}_i}(a,\vec{b}) \simeq (\lambda this.\lambda[\![\overline{\mathtt{x}}_i]\!]_{\mathsf{invk}}.[\![\mathsf{e}_i]\!]_{\mathsf{invk}})a\vec{b}$*.*

We see that the terms $g^{\mathsf{invk}}_{\mathsf{e}_i}$ *depend on* invk*. Now we let the* $\mathcal{L}$ *term* r *be such that for every method* m *in our class table* CT *we have*

$$\mathsf{r}\,\mathsf{invk}(\mathtt{m}^*, x, \vec{y}) \simeq \mathsf{over}_{\mathtt{C}_1^*,\dots,\mathtt{C}_n^*}(g^{\mathsf{invk}}_{\mathsf{e}_1}, \dots, g^{\mathsf{invk}}_{\mathsf{e}_n})(x,\vec{y}). \tag{4}$$

We define the term invk to be the least fixed point of r.

Definition 10 (of the term invk**).** *We set* $\mathsf{invk} := \mathsf{I}\,\mathsf{r}$*.*

In the following we will write only $[\![\mathtt{e}]\!]$ for the translation of an expression e relative to the term r invk defined as above.

It remains to define the interpretation of Featherweight Java classes. Let us begin with the basic types. The example of the type boolean will show how we can use the type structure of explicit mathematics to model the basic types of Java. If we let 0 and 1 denote "false" and "true", respectively, then the interpretation $[\![\mathtt{boolean}]\!]$ of the basic type boolean is given by

$$\{(\mathtt{boolean}^*, b) \mid b = 0 \vee b = 1\}.$$

Here we see that an element $x \in [\![\mathtt{boolean}]\!]$ is pair whose first component carries the run-time type information of x, namely $\mathtt{boolean}^*$, and whose second component is the actual truth value. For the Java expressions false and true we can set

$$[\![\mathtt{false}]\!] = (\mathtt{boolean}^*, 0) \qquad [\![\mathtt{true}]\!] = (\mathtt{boolean}^*, 1).$$

We will interpret the classes of FJ as fixed point types in explicit mathematics satisfying the following fixed point specification.

Definition 11 (of the fixed point FP**).** *If the class table* CT *contains a class named* C *with* $\mathtt{C}^* = i$*, then the following formula is included in our specification:*

$$(X)_i = Y_{i1} \times \cdots \times Y_{im_{\mathtt{C}}},$$

where $m_{\mathtt{C}}$ *is again the least natural number such that for all field names* f *occurring in fields*(C) *we have* $\mathtt{f}^* < m_{\mathtt{C}}$*.* Y_{ij} *is defined according to the following three clauses:*

1. *If there is a basic type* D *and a field name* f *such that* D f *belongs to fields*(C)*, then* Y_{if^*} *is equal to the interpretation of* D*.*
2. *If there is a class* D *and a field name* f *such that* D f *belongs to fields*(C) *and if* $E_1, \dots, E_n$ *is the list of all classes* E_j *in* CT *for which* $E_j <: D$ *is derivable, then*

$$Y_{if^*} = \{(E_1^*, x) \mid x \in (X)_{E_1^*}\} \cup \cdots \cup \{(E_n^*, x) \mid x \in (X)_{E_n^*}\} \cup \{(\mathsf{ex}, 0)\}.$$

3. *If there is no field name* $\mathtt{f}$ *occurring in fields*($\mathtt{C}$) *so that* $\mathtt{f}^* = j$*, then* Y_{ij} *is the universal type* V*, in particular we find* $Y_{im_\mathtt{C}} = \mathsf{V}$*.*

As we have shown before, in PTN *there provably exists a fixed point* FP *satisfying the above specification.*

Since our fixed class table CT contains only finitely many classes we can set up the following definition for the interpretation $[\![\mathtt{C}]\!]$ of a class $\mathtt{C}$.

Definition 12 (of the interpretation $[\![\mathtt{C}]\!]$ of a class $\mathtt{C}$). *If* $\mathtt{E}_1, \ldots, \mathtt{E}_n$ *is the list of all classes* $\mathtt{E}_i$ *in FJ for which* $\mathtt{E}_i <: \mathtt{C}$ *is derivable, then*

$$[\![\mathtt{C}]\!] = \{(\mathtt{E}_1^*, x) \mid x \in (\mathsf{FP})_{\mathtt{E}_1^*}\} \cup \cdots \cup \{(\mathtt{E}_n^*, x) \mid x \in (\mathsf{FP})_{\mathtt{E}_n^*}\} \cup \{(\mathsf{ex}, 0)\}.$$

We include the value $(\mathsf{ex}, 0)$ to the interpretation of all classes because this simplifies the presentation of the proofs about soundness with respect to typing. Of course we could exclude $(\mathsf{ex}, 0)$ from the above types, which would be more natural, but then we had to treat it as special case in all the proofs.

Now we will present to examples for the definition of classes.

Example 6. We take the class `Point` given in Example 4 and extend it to a class `ColorPoint`.

```
class ColorPoint extends Point{
  String color;

  ColorPoint (int a, int b, String c) {
    super(a,b);
    this.color = c;
  }
}
```

Assume $*$ is such that $\mathtt{color}^* = 4$. Hence, we have $m_{\mathtt{ColorPoint}} = 5$ Consider an object of the class `ColorPoint` with `x=5`, `y=6` and `color="black"` where the string `"black"` is modeled by say 256. This object is now interpreted by the sequence

$$(\mathtt{Point}^*, ((\mathtt{int}^*, 5), 0, (\mathtt{int}^*, 6), (\mathtt{String}, 256), 0)).$$

Since the `Point` belongs to our class table, the fixed point specification contains a line

$$(X)_{\mathtt{Point}^*} = [\![\mathtt{int}]\!] \times (\mathsf{V} \times ([\![\mathtt{int}]\!] \times \mathsf{V})).$$

We obtain that the value of our *colored* point object

$$((\mathtt{int}^*, 5), (0, ((\mathtt{int}^*, 6), ((\mathtt{String}, 256), 0))))$$

belongs to the interpretation of `Point` since $((\mathtt{String}, 256), 0) \in \mathsf{V}$. This example illustrates why we take the product with V in the interpretation of objects. It guarantees that the model is sound with respect to subtyping, see Theorem 6 below. This encoding of objects in records is due to Cardelli.

The next example shows why we have to employ fixed points to model the classes.

Example 7. Consider a class `Node` pointing to a next `Node`.

```
class Node extends Object{
   Node next;

   Node(Node x){
     super();
     this.next=x;
   }
}
```

This class will be interpreted as fixed point of

$$X = (\{(\mathtt{Node}^*, x) \mid x \in X\} \cup \{(\mathsf{ex}, 0)\}) \times \mathsf{V}.$$

Iterating this operator form will yield a fixed point only after ω many steps. Therefore, one has to employ dependent choice in the construction of the fixed points modeling classes.

7 Soundness results

In this section we will prove that our model for Featherweight Java is sound with respect to subtyping, typing and reductions. We start with a theorem about soundness with respect to subtyping which is a trivial consequence of the interpretation of classes.

Theorem 5. *For all classes* $\mathtt{C}$ *and* $\mathtt{D}$ *of the class table with* $\mathtt{C} <: \mathtt{D}$ *it is provable in* PTN *that* $[\![\mathtt{C}]\!] \subset [\![\mathtt{D}]\!]$.

The soundness of the semantics of the subtype relation does not depend on the fact that a type is interpreted as the union of all its subtypes. As the next theorem states, our model is sound with respect to subtyping if we allow to coerce the run-time type of an object into a super type. Coercions are operations which change the type of an object. In models of object-oriented programming these constructs can be used to give a semantics for features which are based on early-binding, cf. Castagna, Ghelli and Longo [9, 10]. This is achieved by coercing the type of an object into its static type before selecting the best matching branch. In Java for example, the resolution of overloaded methods is based on static types, i.e. the choice of the function to be applied is based on early-binding.

Theorem 6. *For all classes* $\mathtt{C}$ *and* $\mathtt{D}$ *with* $\mathtt{C} <: \mathtt{D}$ *the following is provable in* PTN*:*

$$x \in [\![\mathtt{C}]\!] \wedge \mathsf{p}_0 x \neq \mathsf{ex} \rightarrow (\mathtt{D}^*, \mathsf{p}_1 x) \in [\![\mathtt{D}]\!].$$

Proof. By induction on the length of the derivation of $\mathtt{C} <: \mathtt{D}$ we show that $(\mathsf{FP})_{\mathtt{C}^*} \subset (\mathsf{FP})_{\mathtt{D}^*}$. The only non-trivial case is, when the following rule has been applied

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}.$$

So we assume $CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}$. By our definition of FP we obtain

$$(\mathsf{FP})_{\mathtt{C}^*} = Y_{\mathtt{C}^*1} \times \cdots \times Y_{\mathtt{C}^* m_{\mathtt{C}}}$$

and $(\mathsf{FP})_{\mathtt{D}^*}$ is of the form $Y_{\mathtt{D}^*1} \times \cdots \times Y_{\mathtt{D}^* m_{\mathtt{D}}}$. By the rules for field lookup we know that if *fields*(D) contains $\mathtt{E\ g}$, then $\mathtt{E\ g}$ also belongs to *fields*(C). Therefore we have $m_{\mathtt{D}} \leq m_{\mathtt{C}}$ and for all $i < m_{\mathtt{D}}$ we get $Y_{\mathtt{C}^* i} \subset Y_{\mathtt{D}^* i}$ by the fixed point specification for FP and our general assumption that class typing is ok. Moreover, we obviously have

$$Y_{\mathtt{C}^* m_{\mathtt{D}}} \times \cdots \times Y_{\mathtt{C}^* m_{\mathtt{C}}} \subset Y_{\mathtt{D}^* m_{\mathtt{D}}} = \mathsf{V}.$$

Therefore, we conclude that the claim holds. □

Before proving soundness with respect to typing we have to show some preparatory lemmas.

Definition 13. *If* $\overline{\mathtt{D}}$ *is the list* $\mathtt{D}_1, \ldots, \mathtt{D}_n$*, then* $[\![\overline{\mathtt{D}}]\!]$ *stands for* $[\![\mathtt{D}_1]\!] \times \cdots \times [\![\mathtt{D}_n]\!]$*; and if* $\overline{\mathtt{e}} = \mathtt{e}_1, \ldots \mathtt{e}_n$*, then* $[\![\overline{\mathtt{e}}]\!]_{\mathsf{invk}} \in [\![\overline{\mathtt{D}}]\!]$ *means*

$$([\![\mathtt{e}_1]\!]_{\mathsf{invk}}, \ldots, [\![\mathtt{e}_n]\!]_{\mathsf{invk}}) \in [\![\mathtt{D}_1]\!] \times \cdots \times [\![\mathtt{D}_n]\!].$$

For $\Gamma = \mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n$ *we set*

$$[\![\Gamma]\!]_{\mathsf{invk}} := [\![\mathtt{x}_1]\!]_{\mathsf{invk}} \in [\![\mathtt{D}_1]\!] \wedge \cdots \wedge [\![\mathtt{x}_n]\!]_{\mathsf{invk}} \in [\![\mathtt{D}_n]\!].$$

Definition 14. *We define the type* $\mathcal{T}$ *to be the intersection of all the types*

$$(\{\mathtt{m}^*\} \times [\![\mathtt{C}]\!] \times [\![\overline{\mathtt{D}}]\!]) \curvearrowright [\![\mathtt{B}]\!]$$

for all methods m *and all classes* C *occurring in* CT *with* $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}} \to \mathtt{B}$.

The next lemma states that if we have an interpretation relative to a function h belonging to $\mathcal{T}$, then this interpretation is sound with respect to typing.

Lemma 1. *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ *is derivable in FJ, then we can prove in* PTN *that* $h \in \mathcal{T}$ *implies*

$$[\![\Gamma]\!]_h \wedge [\![\mathtt{e}]\!]_h\!\downarrow \to [\![\mathtt{e}]\!]_h \in [\![\mathtt{C}]\!].$$

Proof. Proof by induction on the derivation length of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$. We assume $[\![\Gamma]\!]_h \wedge [\![\mathtt{e}]\!]_h\!\downarrow$ and distinguish the different cases for the last rule in the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$:

1. $\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$: trivial.

2. $\Gamma \vdash \mathtt{e.f}_i \in \mathtt{C}_i$: $[\![\mathtt{e.f}_i]\!]_h\downarrow$ implies $[\![\mathtt{e}]\!]_h\downarrow$ by strictness. Hence, we get by the induction hypothesis $[\![\mathtt{e}]\!]_h \in [\![\mathtt{C}_0]\!]$ and $\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. By the definition of $[\![\mathtt{C}_0]\!]$ this yields $\mathsf{proj}\,\mathtt{f}_i^*\,[\![\mathtt{e}]\!]_h \in [\![\mathtt{C}_i]\!]$. Finally we conclude by $\mathsf{proj}\,\mathtt{f}_i^*\,[\![\mathtt{e}]\!]_h \simeq [\![\mathtt{e.f}_i]\!]_h$ that the claim holds.
3. $\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$: by the induction hypothesis and Theorem 5 we obtain $[\![\overline{\mathtt{e}}]\!]_h \in [\![\overline{\mathtt{C}}]\!] \subset [\![\overline{\mathtt{D}}]\!]$ and $[\![\mathtt{e}_0]\!]_h \in [\![\mathtt{C}_0]\!]$. Moreover, we have

$$\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}.$$

   Hence we conclude by $h \in \mathcal{T}$ and $[\![\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}]\!]_h \simeq h(\mathtt{m}^*, [\![\mathtt{e}_0]\!]_h, [\![\overline{\mathtt{e}}]\!]_h)$ that the claim holds.
4. $\Gamma \vdash \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$: by the induction hypothesis and Theorem 5 we have $[\![\overline{\mathtt{e}}]\!]_h \in [\![\overline{\mathtt{C}}]\!] \subset [\![\overline{\mathtt{D}}]\!]$. Further we know $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$. Therefore the claim holds by the definition of new.
5. If the last rule was an upcast, then the claim follows immediately from the induction hypothesis, the definition of the term cast and Theorem 5.
6. Assume the last rule was a downcast or a stupid cast. By the induction hypothesis we get $[\![\mathtt{e}]\!]_h \in [\![\mathtt{D}]\!]$. Then $\mathtt{D} \not<: \mathtt{C}$ implies $\mathsf{sub}(\mathsf{p}_0[\![\mathtt{e}]\!]_h, \mathtt{C}^*) = 0$. We get $[\![\mathtt{(C)e}]\!]_h \simeq (\mathsf{ex}, 0)$ by the definition of cast. Hence the claim holds. □

The following lemma says that our interpretation of FJ expressions is in accordance with the definedness ordering $\sqsubseteq_{\mathcal{T}}$.

Lemma 2. *If $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ is derivable in FJ, then we can prove in* PTN *that $g, h \in \mathcal{T}$ and $g \sqsubseteq_{\mathcal{T}} h$ imply $[\![\Gamma]\!]_g \wedge [\![\mathtt{e}]\!]_g\downarrow \to [\![\mathtt{e}]\!]_g = [\![\mathtt{e}]\!]_h$.*

Proof. Proof by induction on the derivation length of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$. Assume $[\![\Gamma]\!]_g$ and $[\![\mathtt{e}]\!]_g\downarrow$ hold. We distinguish the following cases:

1. $\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$: trivial.
2. $\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{C}_i$: we know $\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0$. Hence we get by the induction hypothesis $[\![\mathtt{e}_0]\!]_g = [\![\mathtt{e}_0]\!]_h$ and therefore the claim holds.
3. $\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$: we get $\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0$, $\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}}$ and

$$\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \quad \text{as well as} \quad \overline{\mathtt{C}} <: \overline{\mathtt{D}}. \tag{5}$$

   Because of $[\![\mathtt{e}]\!]_g\downarrow$ we obtain $[\![\mathtt{e}_0]\!]_g\downarrow$ and $[\![\overline{\mathtt{e}}]\!]_g\downarrow$. Hence the induction hypothesis yields $[\![\mathtt{e}_0]\!]_g = [\![\mathtt{e}_0]\!]_h$ as well as $[\![\overline{\mathtt{e}}]\!]_g = [\![\overline{\mathtt{e}}]\!]_h$. Using Lemma 1, we get $[\![\Gamma]\!]_g \vdash [\![\mathtt{e}_0]\!]_g \in [\![\mathtt{C}_0]\!]$, $[\![\Gamma]\!]_g \vdash [\![\overline{\mathtt{e}}]\!]_g \in [\![\overline{\mathtt{C}}]\!]$ as well as $[\![\Gamma]\!]_g \vdash [\![\mathtt{e}]\!]_g \in [\![\mathtt{C}]\!]$. With (5), $g \in \mathcal{T}$, $h \in \mathcal{T}$ and $g \sqsubseteq_{\mathcal{T}} h$ we conclude

$$\begin{aligned} [\![\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}]\!]_g &= g(\mathtt{m}^*, [\![\mathtt{e}_0]\!]_g, [\![\overline{\mathtt{e}}]\!]_g) \\ &= h(\mathtt{m}^*, [\![\mathtt{e}_0]\!]_h, [\![\overline{\mathtt{e}}]\!]_h) \\ &= [\![\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}]\!]_h \end{aligned}$$

4. $\Gamma \vdash \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$: as in the second case, the claim follows immediately from the induction hypothesis.

5. If the last rule was a cast, then again the claim is a direct consequence of the induction hypothesis. □

Remark 1. In the logic of partial terms it is provable that

$$\forall x F \wedge t{\downarrow} \to F[t/x]$$

for all formulas F of $\mathcal{L}$, cf. e.g. Beeson [5].

Now, we want to show that $\mathsf{invk} \in \mathcal{T}$. First we prove $\mathsf{r} \in (\mathcal{T} \to \mathcal{T})$.

Lemma 3. *In* PTN *it is provable that* $\mathsf{r} \in (\mathcal{T} \to \mathcal{T})$.

Proof. Assume $h \in \mathcal{T}$ and let $(\mathtt{m}^*, c, \vec{d}) \in (\{\mathtt{m}^*\} \times [\![\mathtt{C}_0]\!] \times [\![\overline{\mathtt{D}}]\!])$ for a method $\mathtt{m}$ and classes $\mathtt{C}_0, \overline{\mathtt{D}}, \mathtt{C}$ with $mtype(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}$. We have to show

$$\mathsf{r}h(\mathtt{m}^*, c, \vec{d}){\downarrow} \to \mathsf{r}h(\mathtt{m}^*, c, \vec{d}) \in [\![\mathtt{C}]\!].$$

So assume $\mathsf{r}h(\mathtt{m}^*, c, \vec{d}){\downarrow}$. By (4) we find

$$\mathsf{r}h(\mathtt{m}^*, c, \vec{d}) = \mathsf{over}_{\mathtt{C}_1^*,\dots,\mathtt{C}_n^*}(g^h_{\mathsf{e}_1}, \dots, g^h_{\mathsf{e}_n})(c, \vec{d}).$$

Hence, if $c = (\mathsf{ex}, 0)$ then we obtain $\mathsf{r}h(\mathtt{m}^*, c, \vec{d}) = (\mathsf{ex}, 0)$ and the claim holds. If $c \neq (\mathsf{ex}, 0)$ then we get $\mathsf{sub}(\mathsf{p}_0 c, \mathtt{C}_0) = 1$. By our interpretation of classes, there exists a class $\mathtt{B}$ such that $\mathtt{B} <: \mathtt{C}_0$, $\mathsf{p}_0 c = \mathtt{B}^*$ as well as $c \in [\![\mathtt{B}]\!]$. Let $mbody(\mathtt{m}, \mathtt{B}) = (\overline{\mathtt{x}}_i, \mathsf{e}_i)$. Hence we have

$$\mathsf{r}\,h\,(\mathtt{m}^*, c, \vec{d}) = g^h_{\mathsf{e}_i}(c, \vec{d}) = [\![\mathsf{e}_i]\!]_h[c/this, \vec{d}/\vec{x}_i]. \tag{6}$$

By the rules for method body lookup there exists a class $\mathtt{A}$ such that $\mathtt{B} <: \mathtt{A} <: \mathtt{C}_0$ and the method $\mathtt{m}$ is defined in $\mathtt{A}$ by the expression e_i. By our general assumption that method typing is ok we obtain

$$\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{A} \vdash \mathsf{e}_i \in \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}. \tag{7}$$

By Theorem 5 we get $c \in [\![A]\!]$ and therefore we conclude by $h \in \mathcal{T}$, (6), Lemma 1 and Remark 1 that $\mathsf{r}h(\mathtt{m}^*, c, \vec{d}) \in [\![\mathtt{C}]\!]$ holds. □

Now, we prove that r is $\mathcal{T}$ monotonic which implies $\mathsf{invk} \in \mathcal{T}$.

Lemma 4. *In* PTN *it is provable that* $\mathsf{invk} \in \mathcal{T}$.

Proof. We have defined invk as $\mathsf{l}\,\mathsf{r}$. Lemma 3 states $\mathsf{r} \in (\mathcal{T} \to \mathcal{T})$. Therefore it remains to show that r is $\mathcal{T}$ monotonic. Let $g, h \in \mathcal{T}$ such that $g \sqsubseteq_{\mathcal{T}} h$. We have to show $\mathsf{r}\,g \sqsubseteq_{\mathcal{T}} \mathsf{r}\,h$. So assume $\mathsf{r}g \in \mathcal{T}$. Now we have to show

$$\mathsf{r}h \in \mathcal{T}. \tag{8}$$

Moreover, we have to show for all methods $\mathtt{m}$ and classes $\mathtt{C}_0$, $\overline{\mathtt{D}}$ and $\mathtt{C}$ with $mtype(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}$ that

$$(\forall x \in \{\mathtt{m}^*\} \times [\![\mathtt{C}_0]\!] \times [\![\overline{\mathtt{D}}]\!])\mathsf{r}\,g\,x \sqsubseteq_{\mathtt{C}} \mathsf{r}\,h\,x. \tag{9}$$

(8) is a direct consequence of Lemma 3. In order to show (9) we let $(\mathtt{m}^*, c, \vec{d}) \in \{\mathtt{m}^*\} \times [\![\mathtt{C}_0]\!] \times [\![\overline{\mathtt{D}}]\!]$ and $\mathsf{r}\, g\, (\mathtt{m}^*, c, \vec{d}) \in [\![\mathtt{C}]\!]$. It remains to show

$$\mathsf{r}\, g(\mathtt{m}^*, c, \vec{d}) = \mathsf{r}\, h\, (\mathtt{m}^*, c, \vec{d}). \tag{10}$$

As in Lemma 3 we find $\mathsf{r}\, g\, (\mathtt{m}^*, c, \vec{d}) = [\![\mathtt{e}_i]\!]_g[c/this, \vec{d}/\vec{x}_i]$ for some i and we have to show that this is equal to $[\![\mathtt{e}_i]\!]_h[c/this, \vec{d}/\vec{x}_i]$. By (7), which was a consequence of our general assumption that method typing is OK, and Lemma 2 we find

$$x \in [\![\overline{\mathtt{D}}]\!] \wedge this \in [\![\mathtt{C}_0]\!] \rightarrow [\![\mathtt{e}_i]\!]_g = [\![\mathtt{e}_i]\!]_h.$$

Hence by Remark 1 we obtain

$$[\![\mathtt{e}_i]\!]_g[c/this, \vec{d}/\vec{x}_i] = [\![\mathtt{e}_i]\!]_h[c/this, \vec{d}/\vec{x}_i]$$

and we finally conclude that (10) holds. □

The next theorem states that our model is sound with respect to typing.

Theorem 7. *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ *is derivable in FJ, then in* PTN *it is provable that*

$$[\![\Gamma]\!] \wedge [\![\mathtt{e}]\!]{\downarrow} \rightarrow [\![\mathtt{e}]\!] \in [\![\mathtt{C}]\!].$$

Proof. By the previous lemma we obtain $\mathsf{invk} \in \mathcal{T}$ and therefore $\mathsf{r}\,\mathsf{invk} \in \mathcal{T}$ by Lemma 3. Then we apply Lemma 1 in order to verify our claim. □

As we have seen in Example 1 we cannot prove soundness with respect to reductions of our model construction for the original formulation of reductions in Featherweight Java. The reason is that FJ does not enforce a call-by-value evaluation strategy whereas theories of types and names adopt call-by-value evaluation via their strictness axioms. Moreover, Examples 2 and 3 show that we also have to take care of exceptions and the role of late-binding. Let $\longrightarrow'$ be the variant of the reduction relation $\longrightarrow$ with a call-by-value evaluation strategy which respects exceptions.

Definition 15. *Let* $\mathtt{a}$ *and* $\mathtt{b}$ *be two FJ expressions. We define the reduction relation* $\longrightarrow$ *by induction on the structure of* $\mathtt{a}$*:* $\mathtt{a} \longrightarrow' \mathtt{b}$ *if and only if* $\mathtt{a} \longrightarrow \mathtt{b}$*, where all subexpressions of* $\mathtt{a}$ *are in closed normal form with respect to* $\longrightarrow'$ *and* $\mathtt{a}$ *does not contain subexpressions like* $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ *with* $\mathtt{C} \not<: \mathtt{D}$*.*

As shown in Example 3 the following lemma can only be established for *closed* expressions in normal form.

Lemma 5. *Let* $\mathtt{e}$ *be a well-typed Featherweight Java expression in closed normal form with respect to* $\longrightarrow'$*. Then in* PTN *it is provable that* $[\![\mathtt{e}]\!]{\downarrow}$*. Moreover, if* $\mathtt{e}$ *is not of the form* $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ *with* $C \not<: D$ *and does not contain subexpressions of this form, then it is provable in* PTN *that* $\mathsf{p}_0[\![\mathtt{e}]\!] \neq \mathsf{ex}$*.*

Proof. Let $\mathtt{e}$ be a well-typed closed FJ expression in normal form. First, we prove by induction on the structure of $\mathtt{e}$ that one of the following holds: $\mathtt{e}$ itself is of the form $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$ or it contains a subexpression of this form or $\mathtt{e}$ does not contain such subexpressions and $\mathtt{e}$ is $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ for a class $\mathtt{C}$ and expressions $\overline{\mathtt{e}}$. We distinguish the five cases for the built up of $\mathtt{e}$ as given by the syntax for expressions.

1. $\mathtt{x}$: is not possible since $\mathtt{e}$ is a closed term.
2. $\mathtt{e}_0.\mathtt{f}$: the induction hypothesis applies to $\mathtt{e}_0$. In the first two cases we obtain that $\mathtt{e}$ contains a subexpression of the form $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$. In the last case we get by Theorem 4 about progress that $\mathtt{e}$ cannot be in normal form.
3. $\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$: similar to the previous case.
4. $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$: the induction hypothesis applies to $\overline{\mathtt{e}}$. Again, we obtain in the first two cases that $\mathtt{e}$ contains a subexpression of the form $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$. In the last case we see that $\mathtt{e}$ also fulfills the conditions of the last case.
5. $(\mathtt{C})\mathtt{e}_0$: we apply the induction hypothesis to infer that $\mathtt{e}$ must satisfy condition one or two since it is in normal form.

Now, we know that $\mathtt{e}$ satisfies one of the three conditions above. In the first two cases we obtain by induction on the structure of $\mathtt{e}$ that $[\![\mathtt{e}]\!] = (\mathsf{ex}, 0)$. If the first two cases do not apply, then $\mathtt{e}$ is built up of $\mathtt{new}$ expressions only and we can prove by induction on the structure of $\mathtt{e}$ that $[\![\mathtt{e}]\!]\downarrow$ and $\mathsf{p}_0[\![\mathtt{e}]\!] \neq \mathsf{ex}$. □

Our interpretation of FJ expressions respects substitutions.

Lemma 6. *For all FJ expressions* $\mathtt{e}, \overline{\mathtt{d}}$ *and variables* $\overline{\mathtt{x}}$ *it is provable in* PTN *that* $[\![\mathtt{e}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq [\![\mathtt{e}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$.

Proof. We proceed by induction on the term structure of $\mathtt{e}$. The following cases have to be distinguished.

1. $\mathtt{e}$ is a variable, then the claim obviously holds.
2. $\mathtt{e}$ is of the form $\mathtt{e}_0.\mathtt{g}$. We have

$$[\![\mathtt{e}_0.\mathtt{g}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq (\mathsf{proj}\,\mathtt{g}^*\,[\![\mathtt{e}_0]\!])[[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]].$$

   Since none of the variables of $[\![\overline{\mathtt{x}}]\!]$ occurs freely in proj or $\mathtt{g}^*$, this is equal to $\mathsf{proj}\,\mathtt{g}^*\,([\![\mathtt{e}_0][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]])$, which equals $\mathsf{proj}\,\mathtt{g}^*\,([\![\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!])$ by the induction hypothesis. Finally, we obtain $[\![\mathtt{e}_0.\mathtt{g}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$.
3. $\mathtt{e}$ is of the form $\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$. We have

$$[\![\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq (\mathsf{r}\,\mathsf{invk}\,(\mathtt{m}^*, [\![\mathtt{e}_0]\!], [\![\overline{\mathtt{e}}]\!]))[[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]].$$

   Again, since none of the variables of $[\![\overline{\mathtt{x}}]\!]$ occurs freely in $\mathsf{r}\,\mathsf{invk}$ or in $\mathtt{m}^*$ this is equal to $\mathsf{r}\,\mathsf{invk}\,(\mathtt{m}^*, [\![\mathtt{e}_0][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]], [\![\overline{\mathtt{e}}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]])$. By the induction hypothesis we obtain $\mathsf{r}\,\mathsf{invk}\,(\mathtt{m}^*, [\![\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!], [\![\overline{\mathtt{e}}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!])$, and finally we get $[\![\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$.

4. $\mathtt{e}$ is of the form $\mathtt{new\ C}(\overline{\mathtt{e}})$. We have

$$[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq (\mathsf{new}\,\mathtt{C}^*([\![\overline{\mathtt{e}}]\!]))[[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]].$$

   Again this is equal to $\mathsf{new}\,\mathtt{C}^*([\![\overline{\mathtt{e}}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]])$ which is by the induction hypothesis $\mathsf{new}\,\mathtt{C}^*([\![\overline{\mathtt{e}}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!])$. This is $[\![\mathtt{new\ C}(\overline{\mathtt{e}})[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$ by the interpretation of $\mathtt{new}$.
5. $\mathtt{e}$ is of the form $(\mathtt{C})\mathtt{e}_0$. We obtain

$$[\![(\mathtt{C})\mathtt{e}_0]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq \mathsf{cast}\,\mathtt{C}^*([\![\mathtt{e}_0]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]]).$$

   By the induction hypothesis this is equal to

$$\mathsf{cast}\,\mathtt{C}^*\,[\![\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!] \simeq [\![(\mathtt{C})\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!].$$

□

Now we prove soundness with respect to call-by-value reductions.

Theorem 8. *Let* $\mathtt{g}, \mathtt{h}$ *be two FJ expressions so that* $\mathtt{g}$ *is well-typed and* $\mathtt{g} \longrightarrow' \mathtt{h}$ *is derivable in FJ, then in* PTN *it is provable that* $[\![\mathtt{g}]\!] \simeq [\![\mathtt{h}]\!]$.

Proof. We distinguish the three different rules for computations.

1. $\mathtt{g}$ is of the form $(\mathtt{new\ C}(\overline{\mathtt{e}}))\mathtt{.f_i}$ and $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. We obtain

$$[\![(\mathtt{new\ C}(\overline{\mathtt{e}}))\mathtt{.f_i}]\!] \simeq \mathsf{proj}\,\mathtt{f}_i^*\,[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!].$$

   By the definition of new this is equal to $\mathsf{proj}\,\mathtt{f}_i^*\,(t_{\mathtt{C}}[\![\overline{\mathtt{e}}]\!])$. For $\mathtt{g} \longrightarrow' \mathtt{h}$, we know that all subexpressions of $\mathtt{g}$ are closed, fully evaluated and not of the form $(\mathtt{D})\mathtt{new\ C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$. Hence we obtain by the Lemma 5 that $\mathsf{p}_0[\![\overline{\mathtt{e}}]\!] \neq \mathsf{ex}$ holds and therefore $\mathsf{proj}\,\mathtt{f}_i^*\,(t_{\mathtt{C}}[\![\overline{\mathtt{e}}]\!]) \simeq [\![\mathtt{e}_i]\!]$.
2. $\mathtt{g}$ is of the form $(\mathtt{new\ C}(\overline{\mathtt{e}}))\mathtt{.m}(\overline{\mathtt{d}})$ and $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$. Assume $\mathtt{m}$ is defined exactly in the classes $\mathtt{C_1}, \ldots, \mathtt{C_n}$. Now we show by induction on the length of the derivation of $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ that there exists k so that $1 \leq k \leq n$,

$$\mathsf{min}^k_{\mathtt{C}_1^*,\ldots,\mathtt{C}_n^*}(\mathtt{C}^*) = 1 \bigwedge_{l<k} \mathsf{min}^l_{\mathtt{C}_1^*,\ldots,\mathtt{C}_n^*}(\mathtt{C}^*) \neq 1 \tag{11}$$

   and

$$\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \mathit{mbody}(\mathtt{m}, \mathtt{C_k}). \tag{12}$$

   If $\mathtt{m}$ is defined in $\mathtt{C}$, then there exists a k in $1, \ldots, n$ so that $\mathtt{C} = \mathtt{C_k}$. Hence (11) and (12) trivially hold. If $\mathtt{m}$ is not defined in $\mathtt{C}$, then $\mathtt{C}$ extends a class $\mathtt{B}$ with $\mathit{mbody}(\mathtt{m}, \mathtt{B}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$. In this case we have $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \mathit{mbody}(\mathtt{m}, \mathtt{B})$. Therefore (11) and (12) follow by the induction hypothesis.

   We have assumed that $(\mathtt{new\ C}(\overline{\mathtt{e}}))\mathtt{.m}(\overline{\mathtt{d}})$ is well-typed. Therefore we get that $\mathtt{new\ C}(\overline{\mathtt{e}})$ and $\overline{\mathtt{d}}$ are well-typed. Let $\mathtt{B}$ be the type satisfying $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \mathit{mbody}(\mathtt{m}, \mathtt{B})$ so that $\mathtt{m}$ is defined in $\mathtt{B}$. We find $\mathtt{C} <: \mathtt{B}$. Furthermore, let $\overline{\mathtt{D}}$ be

the types with $\overline{\mathtt{d}} \in \overline{\mathtt{D}}$. The expressions $\overline{\mathtt{e}}, \overline{\mathtt{d}}$ are in closed normal form and hence we obtain by Lemma 5, Theorem 7 and Theorem 5

$$[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}} \in [\![\mathtt{B}]\!] \qquad [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}} \in [\![\overline{\mathtt{D}}]\!]. \tag{13}$$

By our general assumption that method typing is ok we obtain

$$\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{B} \vdash \mathtt{e}_0 \in \mathtt{E}_0.$$

Hence applying Lemma 2 with $\mathsf{r\,invk} \cong_{\mathcal{T}} \mathsf{invk}$ yields

$$[\![\overline{\mathtt{x}}]\!]_{\mathsf{invk}} \in [\![\overline{\mathtt{D}}]\!] \wedge [\![\mathtt{this}]\!]_{\mathsf{invk}} \in [\![\mathtt{B}]\!] \to [\![\mathtt{e}_0]\!]_{\mathsf{invk}} \simeq [\![\mathtt{e}_0]\!]_{\mathsf{r\,invk}}. \tag{14}$$

Summing up, we get by (13), (14) and Remark 1

$$\begin{aligned}[\![(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}})]\!]_{\mathsf{r\,invk}} &\simeq \mathsf{r\,invk}\,(\mathtt{m}^*, [\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}}, [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}}) \\ &\simeq [\![\mathtt{e}_0]\!]_{\mathsf{invk}}([\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}}/\mathit{this}, [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}}/[\![\overline{\mathtt{x}}]\!]) \\ &\simeq [\![\mathtt{e}_0]\!]_{\mathsf{r\,invk}}([\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}}/\mathit{this}, [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}}/[\![\overline{\mathtt{x}}]\!]).\end{aligned}$$

In view of Lemma 6 this is partially equal to

$$[\![\mathtt{e}_0[\mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}, \overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]_{\mathsf{r\,invk}}.$$

3. $\mathtt{g}$ is of the form $(\mathtt{D})(\mathtt{new\ C}(\overline{\mathtt{e}}))$ and $\mathtt{C} <: \mathtt{D}$. We have $\mathsf{sub}(\mathtt{C}^*, \mathtt{D}^*) = 1$ and therefore

$$[\![(\mathtt{D})(\mathtt{new\ C}(\overline{\mathtt{e}}))]\!] \simeq [\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!].$$

□

8 Conclusion

Usually, the research on Java's semantics takes an *operational* approach. And if a denotational semantics for object-oriented principles is presented, then it is often given in *domain-theoretic* notions. In contrast to this work, we investigate a *denotational* semantics for Featherweight Java which is based on *recursion-theoretic* concepts.

Our interpretation of Featherweight Java is based upon a formalization of the object model of Castagna, Ghelli and Longo [11]. Its underlying type theory can be given using predicative notions only, whereas most other object encodings are impredicative, cf. e.g. Bruce, Cardelli and Pierce [8]. We have formalized the object model in a predicative theory of types and names which shows that this model is really simple from a proof-theoretic perspective. Hence, our formalization provides constructive foundations for object-oriented programming. Moreover, this gives further evidence for Feferman's claim that impredicative assumptions are not needed for computational practice. A claim which has, up to now, only been verified for polymorphic functional programs. Our work yields

first positive results about its status in the context of object-oriented programming.

We have a proof-theoretically weak but highly expressive theory for representing object-oriented programs and for stating and proving many properties of them similar to the systems provided by Feferman [16–18] and Turner [43, 44] for functional programming. Due to the fact that a least fixed point operator is definable in our theory, we also can prove that certain programs will not terminate. This is not possible in the systems of Feferman and Turner.

Since Featherweight Java is the functional core of the Java language and since the object model we employ provides a unified foundation for both Simula's and CLOS's style of programming, our work also contributes to the study of the relationship between object-oriented and functional programming. It shows that these two paradigms of programming fit well together and that their combination has a sound mathematical model.

Usually, denotational semantics are given in domain-theoretic notions. In such a semantics one has to include to each type an element $\perp$ which denotes the result of a non-terminating computation of this type, cf. e.g. Alves-Foss and Lam [2]; whereas our recursion-theoretic model has the advantage that computations are interpreted as ordinary computations. This means we work with *partial* functions which possibly do not yield a result for certain arguments, i.e. computations may really not terminate. In our opinion this model is very natural and captures well our intuition about non-termination.

As already pointed out by Castagna, Ghelli and Longo [11] the dynamic definition of new classes is one of the main problems when overloaded functions are used to define methods. Indeed, in our semantics we assumed a fixed class table, i.e. the classes are given from the beginning and they will not change. This fact makes our semantics non-compositional. If we add new classes to our class table, then we get a new interpretation for our objects. An important goal would be to investigate an overloading based semantics for object-oriented programs with dynamic class definitions. Our work has also shown that theories of types and names are a powerful tool for analyzing concepts of object-oriented programming languages. Therefore, we think it would be worthwhile to employ such theories for exploring further principles, i.e. the combination of overloading and parametric polymorphism, cf. Castagna [9], or the addition of mixins to class-based object-oriented languages, cf. e.g. Ancona, Lagorio and Zucca [3] or Flatt, Krishnamurthi and Felleisen [21]. Last but not least it would be very interesting to have a semantics for concurrent and distributed computations in explicit mathematics.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: integrating object oriented design and formal verification. In Gerhard Brewka and Luís Moniz Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA)*, Lecture Notes in Artificial Intelligence. Springer, 2000.

2. Jim Alves-Foss and Fong Shing Lam. Dynamic denotational semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 201–240. Springer, 1999.
3. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP 2000 - 14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*. Springer, 2000.
4. Davide Ancona and Elena Zucca. A module calculus for Featherweight Java. Submitted.
5. Michael J. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer, 1985.
6. Michael J. Beeson. Proving programs and programming proofs. In R. Barcan Marcus, G.J.W. Dorn, and P. Weingartner, editors, *Logic, Methodology and Philosophy of Science VII*, pages 51–82. North-Holland, 1986.
7. Egon Börger, Joachim Schmid, Wolfram Schulte, and Robert Stärk. *Java and the Java Virtual Machine*. Lecture Notes in Computer Science. Springer. To appear.
8. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.
9. Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhäuser, 1997.
10. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A semantics for $\lambda\&$-early: a calculus with overloading and early binding. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 1993.
11. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.
12. Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer, 1999.
13. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and practice of object systems*, 5(1):3–24, 1999.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. Matthias Felleisen and Daniel P. Friedman. *A Little Java, A Few Patterns*. MIT Press, 1998.

20. Marcello Fiore, Achim Jung, Eugenio Moggi, Peter O'Hearn, Jon Riecke, Giuseppe Rosolini, and Ian Stark. Domains and denotational semantics: history, accomplishments and open problems. *Bulletin of the European Association for Theoretical Computer Science*, 59:227–256, 1996.
21. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical report, Rice University, 1999. Corrected Version, original in J. Alvess-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269, Springer, 1999.
22. Giorgio Ghelli. A static type system for late binding overloading. In A. Paepcke, editor, *Proc. of the Sixth International ACM Conference on Object-Oriented Programming Systems and Applications*, pages 129–145. Addison-Wesley, 1991.
23. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996. Also available via `http://java.sun.com/docs/`.
24. Atsushi Igarashi and Benjamin Pierce. On inner classes. In *Informal Proceedings of the Seventh Internatianal Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2000.
25. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA '99)*, volume 34 of *ACM SIGPLAN Notices*, pages 132–146, 1999.
26. Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA '98)*, volume 33 of *ACM SIGPLAN Notices*, pages 329–340, 1998.
27. Gerhard Jäger. Induction in the elementary theory of types and names. In E. Börger, H. Kleine Büning, and M.M. Richter, editors, *Computer Science Logic '87*, volume 329 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 1988.
28. Gerhard Jäger. Type theory and explicit mathematics. In H.-D. Ebbinghaus, J. Fernandez-Prida, M. Garrido, M. Lascar, and M. Rodriguez Artalejo, editors, *Logic Colloquium '87*, pages 117–135. North-Holland, 1989.
29. Reinhard Kahle. Einbettung des Beweissystems Lambda in eine Theorie von Operationen und Zahlen. Diploma thesis, Mathematisches Institut der Universität München, 1992.
30. Reinhard Kahle and Thomas Studer. Formalizing non-termination of recursive programs. To appear in *Journal of Logic and Algebraic Programming*.
31. Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, New York, 1998.
32. David von Oheimb. Axiomatic semantics for Java$_{\ell ight}$. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Fernuniversität Hagen, 2000.
33. Dieter Probst. Dependent choice in explicit mathematics. Diploma thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 1999.
34. Dana S. Scott. Identity and existence in intuitionistic logic. In M. Fourman, C. Mulvey, and D. Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer, 1979.
35. 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.

36. 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.
37. Thomas Studer. A semantics for $\lambda^{\{\}}_{str}$: a calculus with overloading and late-binding. To appear in *Journal of Logic and Computation*.
38. Thomas Studer. *Object-Oriented Programming in Explicit Mathematics: Towards the Mathematics of Objects*. PhD thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 2001.
39. Thomas Studer. Impredicative overloading in explicit mathematics. Submitted.
40. Don Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer, 1999.
41. Makoto Tatsuta. Realizability for constructive theory of functions and classes and its application to program synthesis. In *Proceedings of Thirteenth Annual IEEE Symposium on Logic in Computer Science, LICS '98*, pages 358–367, 1998.
42. Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol II*. North Holland, 1988.
43. Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw Hill, 1991.
44. Raymond Turner. Weak theories of operations and types. *Journal of Logic and Computation*, 6(1):5–31, 1996.