# Functional Type Assignment for Featherweight Java

**To Rinus Plasmeijer, in honour of his 61st birthday**

S.J. van Bakel and R.N.S. Rowe

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK
s.vanbakel@imperial.ac.uk, r.rowe@doc.ic.ac.uk

**Abstract.** We consider functional type assignment for the class-based object-oriented calculus Featherweight Java. We start with an *intersection type assignment systems* for this calculus for which types are preserved under conversion. We then define a variant for which type assignment is decidable, and define a notion of unification as well as a principal typeing algorithm.
We show the expressivity of both our calculus and our type system by defining an encoding of Combinatory Logic into our calculus and showing that this encoding preserves typeability. We thus demonstrate that the great capabilities of functional types can be applied to the context of class-based object orientated programming.

## Introduction

In this paper we will study a notion of functional type assignment for Featherweight Java (FJ) [15]. We will show its elegance and expressiveness, and advocate its use in type assignment systems for fully fledged Java; of course it would need to be extended in order to fully deal with all the features of that language, but through the system we present here we show that that should be feasible, giving a better notion of types.

Type assignment has more than shown its worth in the context of functional programming, like ML, Clean [11, 17], and Haskell. Not only are types essential for efficient code generation, they provide an excellent means of (an abstract level of) error checking: it is in most programmers' experience that once the type checker has approved of a functional program, said program will be almost error free.[1] But, more importantly, the approach to types in functional programming is that of *type assignment*: programmers have the freedom to not specify types for any part of their code. A type inference algorithm embedded in the compiler guarantees a partial correctness result: the type found for a program is also the type for the result of running the program, a property most commonly known as *subject reduction* or *type soundness*, although that latter term has different meaning as well.

In the context of imperative programming, which contains the Object-Oriented approach (OO) as well, the reality is very different. There it is more common to demand that *all* types are written within the code; then type checking is almost trivial, and mainly concerns checking if special features like inheritance are well typed. The difference between the *functional* approach and the *imperative* one then boils down to the

[1] Almost, yes, not completely; we can but dream of static error checking systems that only approve of error free code, and catch all errors.

difference between *untyped* and *typed* calculi. Often, in the untyped approach, if a term has a type, it has infinitely many, a feature that is exploited when introducing *polymorphism* into a programming language's type system. In the typed approach, each term has (normally) only *one* type. This implies that it is difficult, if not impossible, to express polymorphism in imperative languages.

With that in mind, we set out to investigate if the functional approach is feasible for imperative languages as well. The results presented in this paper are part of the results of that investigation, but in the more concrete setting of functional type assignment for object orientation. In order to be able to concentrate on the essential difficulties, we focus on Featherweight Java [15], a restriction of Java which can be regarded as the minimal core fragment of Java, defined by removing all but the most essential features of the full language; Featherweight Java bears a similar relation to Java as the $\lambda$-calculus (LC) [12, 9] does to languages such as ML and and Haskell.

But rather than defining a notion of type assignment that is implementable, we thought it necessary to first verify that the kernel of our approach *made sense*, *i.e.* accords to some particular kind of abstract semantics.[2] Normally, just *operational semantics* is used: then the only check is that subject reduction is satisfied. This is certainly the minimal requirement for type assignment systems (although variants are proposed that do not even satisfy this), but normally much more can be achieved.

Rather, in [19] we proposed an approach that has strong links with *denotational semantics*, in that it gives a full equational semantics for FJ-programs. The best-known way to achieve that is through setting up a notion of types inspired by Coppo and Dezani's *intersection type discipline* (ITD) [13, 10, 2] and this was the path we followed in [19]. ITD, first defined for LC, is a system that is closed under $\beta$-equality and gives rise to a filter model and semantics; it is defined as an extension of Curry's basic type system for LC by allowing term-variables to have many, potentially non-unifiable, types. This generalisation leads to a very expressive system: for example, strong normalisation of terms can be characterised by assignable types.

Inspired by this expressive power, investigations have taken place of the suitability of intersection type assignment for other computational models: for example, van Bakel and Fernández have studied intersection types in the context of Term Rewriting Systems (TRS) [7, 8] and van Bakel studied them in the context of sequent calculi [3, 5]. In an attempt to bring intersection types to the context of OO, van Bakel and de'Liguoro presented a system for the $\varsigma$-calculus [6]; the main characteristic of that system is that it sees assignable types as an *execution* or *applicability predicate*, rather than as a functional characterisation as is the view in the context of LC and, as a result, recursive calls are typed individually, with different types. This is also the case in our system.

The system we presented there is essentially based on the strict system of [1]; the decidable system we present here, a system with simple record types, is likewise essentially based on Curry types. Our system with intersection types has been shown to give a semantics in [19] of which we will state the main results here; that paper also defined

[2] Too often *ad-hoc* changes to type systems are proposed that only solve a specific problem; the proposer typically gives an example of an untypeable term that in all reasonability should be typeable, and gives a change to the type system in order to make that example typeable.

a notion of *approximation*, inspired by a similar notion defined for the $\lambda$-calculus [20], and showed an approximation result.[3]

Our types are *functional*, contain *field* and *method* information, and characterise how a typeable object can interact with a context in which it is placed. The notion of type assignment we developed can be seen as a notion of 'flow analysis' in that assignable types express how expressions can be approached; as such, the types express run-time behaviour of expressions. Our type system was shown to be closed for *conversion*, *i.e.* closed for both *subject reduction* and *subject expansion*, which implies that types give a complete characterisation of the execution behaviour of programs; as a consequence, type assignment in the full system is undecidable.

That FJ is Turing complete seems to be a well accepted fact; we illustrate the expressive power of our calculus by embedding Combinatory Logic (CL) [14] – and thereby also LC – into it, thus establishing that our calculus is Turing complete as well. To show that our type system provides more than a semantical tool and can be used in practice as well, in this paper we will define a variant of our system by restricting to a notion of Curry type assignment; the variant consists of dealing with recursion differently. We show a principal type property and a type preservation result for this system.

The Curry system we propose here is a first, and certainly not the most expressive, illustrative, or desirable system imaginable. We allow intersection types only in the form of records; for FJ this is natural, since a class should be seen as a combination of all its capabilities. As a special property, our principal typeing algorithm calculates (normally) records as types for classes, and the return type of a method can be a record as well. And thirdly, the way the system types recursive classes could be improved by using recursive types.

## 1 Featherweight Java without casts

In this section, we will define the variant of Featherweight Java we consider in this paper. As in other class-based object-oriented languages, it defines *classes*, which represent abstractions that encapsulate both data (stored in *fields*) and the operations to be performed on that data (encoded as *methods*). Sharing of behaviour is accomplished through the *inheritance* of fields and methods from parent classes. Computation is mediated by *instances* of these classes (called *objects*), which interact with one another by *calling* (also called *invoking*) methods on each other and accessing each other's (or their own) fields. We have removed cast expressions since, as the authors of [15] themselves point out, the presence of *downcasts* is unsound[4], so cannot be modelled semantically; for this reason we call our calculus $\text{FJ}^{\not\Downarrow}$. We also leave constructors as implicit.

**Notation** We use $\underline{n}$ (where $n$ is a natural number) to represent the set $\{1,\ldots,n\}$. A sequence $s$ of $n$ elements $a_1,\ldots,a_n$ is denoted by $\vec{a}_n$; the subscript can be omitted

[3] Although the latter firmly rooted our system semantically, it plays no role when implementing, so we will skip its details here. Suffice that say that, because the language of FJ is first order, we had to define *derivation reduction* and show it strongly normalisable in order to prove the approximation result and the normalisation results that follow as a consequence.

[4] In the sense that typeable expressions can get stuck at runtime by reducing to an expression containing *stupid* casts.

when the exact number of elements in the sequence is not relevant. We write $a \in \vec{a}_n$ whenever there exists some $i \in \underline{n}$ such that $a = a_i$. The empty sequence is denoted by $\epsilon$, and concatenation on sequences by $s_1 \cdot s_2$.

We use familiar meta-variables in our formulation to range over class names (`C` and `D`), field names (`f`), method names (`m`) and variables (`x`).[5] We distinguish the class name `Object` (which denotes the root of the class inheritance hierarchy in all programs) and the self variable `this`, used to refer to the receiver object in method bodies.

**Definition 1 (FJ$^{\not\in}$ Syntax).** An FJ$^{\not\in}$ program $P$ consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* `e` to be run (corresponding to the body of the `main` method in a real Java program). Programs are defined by the grammar:

$$
\begin{array}{rcll}
\texttt{e} & ::= & \texttt{x} \mid \texttt{this} \mid \texttt{new C}(\vec{\texttt{e}}) \mid \texttt{e.f} \mid \texttt{e.m}(\vec{\texttt{e}}) & \\
\texttt{fd} & ::= & \texttt{C f;} & \\
\texttt{md} & ::= & \texttt{D m}(\texttt{C}_1\ \texttt{x}_1, \ldots, \texttt{C}_n\ \texttt{x}_n)\ \{\texttt{return e;}\} & \\
\texttt{cd} & ::= & \texttt{class C extends C'}\ \{\ \overrightarrow{\texttt{fd}}\ \overrightarrow{\texttt{md}}\} & (\texttt{C} \neq \texttt{Object}) \\
\mathcal{CT} & ::= & \overrightarrow{\texttt{cd}} & \\
P & ::= & (\mathcal{CT}; \texttt{e}) &
\end{array}
$$

The remaining concepts that we will define below are dependent (or, more precisely, parametric) on a given class table. For example, the reduction relation we will define uses the class table to look up fields and method bodies in order to direct reduction and our type assignment system will do likewise. Thus, there is a reduction relation and type assignment system *for each program*. However, since the class table is a fixed entity (*i.e.* it is not changed during reduction, or during type assignment), as usual it will be left as an implicit parameter in the definitions that follow.

As we have just mentioned, the sequence of (class) declarations that comprises the class table induces a family of lookup *functions*. In order to ensure that these functions are well defined, we only consider programs which conform to some sensible well-formedness criteria: that there are no cycles in the inheritance hierarchy, that each class is declared only once, that fields and methods in any given branch of the inheritance hierarchy are uniquely named, and that method definitions correspond to closed functions. An exception is made to allow *method override*, i.e. the redeclaration of methods, providing that only the *body* of the method differs from the previous declaration.

We define the following functions to look up elements of class definitions.

**Definition 2 (Lookup Functions).** The following lookup functions are defined to extract the names of fields and bodies of methods belonging to (and inherited by) a class.

1. The following retrieve the name of a class, method, or field from its definition:

$$
\begin{array}{rcl}
\mathcal{CN}(\texttt{class C extends D}\{\overrightarrow{\texttt{fd}}\ \overrightarrow{\texttt{md}}\}) & = & \texttt{C} \\
\mathcal{MN}(\texttt{C m}(\vec{\texttt{x}})\texttt{.e;}) & = & \texttt{m} \\
\mathcal{FN}(\texttt{C f;}) & = & \texttt{f}
\end{array}
$$

2. By abuse of notation, we will treat the *class table*, $\mathcal{CT}$, as a partial map from class names to class definitions:

[5] We use roman teletype font for concrete FJ$^{\not\in}$-code, and italicised teletype font for meta-code.

$$\mathcal{CT}(C) \;=\; cd \qquad (\mathcal{CN}(cd) = C,\; cd \in \mathcal{CT})$$

3. The list of fields belonging to a class $C$ (including those it inherits) is given by:

$$\begin{array}{rcll} \mathcal{F}(\texttt{Object}) & = & \epsilon & \\ \mathcal{F}(C) & = & \mathcal{F}(C') \cdot \vec{f}_n & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}_n\ \overrightarrow{md}\}, \\ & & & \mathcal{FN}(fd_i) = f_i \quad (i \in \underline{n})) \end{array}$$

4. The list of methods belonging to a class $C$ is given by:

$$\begin{array}{rcll} \mathcal{M}(\texttt{Object}) & = & \epsilon & \\ \mathcal{M}(C) & = & \mathcal{M}(C') \cdot \vec{m}_n & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}_n\ \overrightarrow{md}\}, \\ & & & \mathcal{MN}(md_i) = m_i \quad (i \in \underline{n})) \end{array}$$

(notice that method names can appear more than once in $\mathcal{M}(C)$).

5. The function $\mathcal{M}b$, given a class name $C$ and method name $m$, returns a tuple $(\vec{x}, e)$, consisting of a sequence of the method's formal parameters and its body:

$$\begin{array}{rcll} \mathcal{M}b(C,m) & = & (\vec{x}_n, e) & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\} \\ & & & \;\&\; C_0\ m(C_1\ x_1, \ldots, C_n\ x_n)\ \{\texttt{return}\ e;\} \in \overrightarrow{md}) \\ \mathcal{M}b(C,m) & = & \mathcal{M}b(C',m) & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\} \\ & & & \;\&\; m \text{ not in } \overrightarrow{md}) \end{array}$$

6. The function $fv(e)$ returns the set of variables used in $e$.

*Substitution* of expressions for variables is the basic mechanism for reduction in our calculus: when a method is invoked on an object (the *receiver*) the invocation is replaced by the body of the method that is called, and each of the variables is replaced by a corresponding argument, and `this` is replaced by the receiver.

**Definition 3 (Reduction).** 1. A *term substitution*

$$\mathsf{S} \;=\; \langle \texttt{this} \mapsto e', x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \rangle$$

is defined in the standard way as a total function on expressions that systematically replaces all occurrences of the variables $x_i$ and `this` by their corresponding expression. We write $e^{\mathsf{S}}$ for $\mathsf{S}(e)$.

2. The single-step reduction $\rightarrow$ is defined by:

$$\begin{array}{rcll} \texttt{new}\ C(\vec{e}_n).f_i & \rightarrow & e_i & (\mathcal{F}(C) = \vec{f}_n, i \in \underline{n}) \\ \texttt{new}\ C(\vec{e}).m(\vec{e'}_n) & \rightarrow & e^{\mathsf{S}} & (\mathcal{M}b(C,m) = (\vec{x}_n, e), \text{ where } \mathsf{S} = \\ & & & \langle \texttt{this} \mapsto \texttt{new}\ C(\vec{e}),\ x_1 \mapsto e'_1,\ \ldots,\ x_n \mapsto e'_n \rangle) \end{array}$$

We call the left-hand term the *redex* (*red*ucible *ex*pression) and the right hand the *contractum*. As usual, we define $\rightarrow^*$ as the pre-congruence generated by $\rightarrow$.

The nominal system as presented in [15], adapted to our version of Featherweight Java, is defined as follows.

$$(\textsc{new}):\ \frac{\Gamma \vdash e_i : C_i \quad (\forall i \in \underline{n})}{\Gamma \vdash \texttt{new}\ C(\vec{e}) : C}\ (\mathcal{F}(C) = \vec{f}\ \&\ \mathcal{FT}(C, f_i) = D_i\ \&\ C_i <: D_i \quad (\forall i \in \underline{n}))$$

$$(\textsc{invk}):\ \frac{\Gamma \vdash e : E \qquad \Gamma \vdash e_i : C_i \quad (\forall i \in \underline{n})}{\Gamma \vdash e.m(\vec{e}) : C}\ (\mathcal{MT}(E, m) = \vec{D} \to C\ \&\ C_i <: D_i\ (\forall i \in \underline{n}))$$

$$(\textsc{var}):\ \frac{}{\Gamma, x{:}C \vdash x : C} \quad (\textsc{fld}):\ \frac{\Gamma \vdash e : D}{\Gamma \vdash e.f : C}\ (\mathcal{FT}(D, f) = C) \quad (\textsc{u-cast}):\ \frac{\Gamma \vdash e : D}{\Gamma \vdash (C)e : C}\ (D <: C)$$

$$(\textsc{d-cast}):\ \frac{\Gamma \vdash e : D}{\Gamma \vdash (C)e : C}\ (C <: D,\ C \neq D) \qquad (\textsc{s-cast}):\ \frac{\Gamma \vdash e : D}{\Gamma \vdash (C)e : C}\ (C \not<: D,\ D \not<: C)$$

**Fig. 1.** Type assignment rules for the nominal system for FJ$^{\not<:}$

**Definition 4 (Member type lookup).** The *field table* $\mathcal{FT}$ and *method table* $\mathcal{MT}$ are functions which return type information about the elements of a given class in an execution. These functions allow us to retrieve the types of any given field $f$ or method $m$ declared in a particular class $C$: $\mathcal{FT}(C, f) =$

$$\begin{cases} D & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\},\ D\ f \in \overrightarrow{fd}) \\ \mathcal{FT}(C', f) & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\},\ f \text{ not in } \overrightarrow{fd}) \end{cases}$$

$\mathcal{MT}$ is defined similarly: $\mathcal{MT}(C, m) =$

$$\begin{cases} \vec{C}_n \to D & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\}, \\ & \qquad D\ m(\overrightarrow{C\ x})\ \{e\} \in \overrightarrow{md}) \\ \mathcal{MT}(C', m) & (\mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\},\ m \text{ not in } \overrightarrow{md}) \end{cases}$$

Notice both are not defined on `Object`.

Nominal type assignment in FJ$^{\not<:}$ is a relatively easy affair, and more or less guided by the class hierarchy.

**Definition 5 (Nominal type assignment for FJ$^{\not<:}$ [15]).**

1. The sub-typing relation on class types is generated by the `extends` construct, and is defined as the smallest pre-order satisfying: if $\texttt{class}\ C\ \texttt{extends}\ D\ \{\overrightarrow{fd}\ \overrightarrow{md}\} \in \mathcal{CT}$, then $C <: D$.[6]
2. *Statements* are pairs of expression and type, written as $e : C$; *contexts* $\Gamma$ are defined as sets of statements of the shape $x{:}C$, where all variables are distinct, and possibly containing a statement for `this`.
3. Expression type assignment for the nominal system for FJ is defined through the rules given in Figure 1, where (VAR) is applicable to `this` as well.
4. A declaration of method $m$ is well typed in $C$ when the type returned by $\mathcal{MT}(m, C)$ determines a type assignment for the method body.[7]

$$(\textsc{meth}):\ \frac{\overrightarrow{x{:}C}, \texttt{this}{:}C \vdash e_b : D}{E\ m(\overrightarrow{C\ x})\ \{\ \texttt{return}\ e_b;\ \}\ \text{OK IN}\ C} \quad (\mathcal{MT}(m, D) = \vec{C} \to E,\ D <: E,\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\cdots\})$$

5. Classes are well typed when all their methods are and a program is well typed when all the classes are and the expression is typeable.

[6] Notice that this relation depends on the class-table, so the symbol $<:$ should be indexed by $\mathcal{CT}$; as mentioned above, we leave this implicit.

[7] Notice that, by the well-formedness criterion, $e_b$ has no other variables than $\vec{x}$, so all variables are bound in a method declaration, thus avoiding dynamic linking issues.

$$(\textsc{class}):\ \frac{\texttt{md}_i\ \text{OK IN}\ \texttt{C} \quad (\forall i \in \underline{n})}{\texttt{class C extends D}\{\overrightarrow{\texttt{fd}};\ \overrightarrow{\texttt{md}_n}\}\ \text{OK}} \qquad (\textsc{prog}):\ \frac{\overrightarrow{\texttt{cd}\ \text{OK}} \quad \Gamma \vdash \texttt{e} : \texttt{C}}{(\overrightarrow{\texttt{cd}}; \texttt{e})\ \text{OK}}$$

Notice that in the nominal system, classes are typed (or rather type-*checked*) once, and the types declared for their fields and methods are static, unique, and used at invocation.

As mentioned above, we have decided to not consider casts in our work; using a cast is comparable to a promise by the programmer that the casted expression will at run time evaluate to an object having the specified class (or a subclass thereof), and so (for soundness) requires doing a run-time check of the shape

$$\texttt{(C) new D(...)} \rightarrow \texttt{new D(...)} \qquad (\texttt{D <: C})$$

Once this check has been carried out the cast disappears. Of course, for full programming convenience, and to be able to obtain the correct behaviour in overloaded methods, casts are essential.

## 2 Semantic Type Assignment

In [19], we defined a type system for FJ$^{\not\in}$ that is loosely based on the strict intersection type assignment system for the $\lambda$-calculus [1, 2] (see [4] for a survey) and is influenced by the predicate system for the $\varsigma$-calculus [6]; we showed that it satisfies both *subject reduction* and *subject expansion*. Our types can be seen as describing the capabilities of an expression (or rather, the object to which that expression evaluates) in terms of (1) *the operations that may be performed on it (*i.e. *accessing a field or invoking a method),* and *(2) the* outcome *of performing those operations,* where dependencies between the inputs and outputs of methods are tracked using (type) variables. In this way, our types express detailed properties about the contexts in which expressions can safely be used. More intuitively, they capture a certain notion of *observational equivalence*: two expressions with the same set of assignable types will be observationally indistinguishable. Our types thus constitute *semantic predicates*.

**Definition 6 (Semantic Types [19]).** The set of *intersection types* (or *types* for short), ranged over by $\phi$, $\psi$, and its subset of *strict* types, ranged over by $\sigma$, $\tau$ are defined by the following grammar (where $\varphi$ ranges over a denumerable set of *type variables*, $\texttt{C}$ ranges over the set of class names, and $\omega$ is a type constant, the universal type and the top element of the type hierarchy):

$$\begin{array}{rcl} \phi, \psi & ::= & \omega \mid \sigma \mid \phi \cap \psi \\ \sigma & ::= & \varphi \mid \texttt{C} \mid \langle \texttt{f}{:}\sigma \rangle \mid \langle \texttt{m}{:}(\phi_1, \ldots, \phi_n) \rightarrow \sigma \rangle \quad (n \geq 0) \end{array}$$

Notice that our types do not depend on the types that would be assigned in the nominal system; in fact, we could have presented our results for an *untyped* variant of FJ, where all class annotations on parameters and return types are omitted. We have decided not to do so for reasons of compatibility with other work, and to avoid leaving the (incorrect) impression that our results would somehow then depend on the fact that expressions carry no type information.

The key feature of types is that they may group information about many operations together into *intersections* from which any specific one can be selected for an expression

$$(\textsc{newM}):\ \frac{\texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e_b : \sigma \quad \Pi \vdash \texttt{new } C(\vec{e}) : \psi}{\Pi \vdash \texttt{new } C(\vec{e}) : \langle m{:}(\vec{\phi}_n)\to\sigma\rangle}\ (\mathcal{M}b(C,m) = (\vec{x}_n, e_b),\ n \geq 0)$$

$$(\textsc{newF}):\ \frac{\Pi \vdash e_1 : \phi_1 \quad \ldots \quad \Pi \vdash e_n : \phi_n}{\Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f_i{:}\sigma\rangle}\ (\mathcal{F}(C) = \vec{f}_n,\ i \in \underline{n},\ \sigma = \phi_i,\ n \geq 1)$$

$$(\textsc{obj}):\ \frac{\Pi \vdash e_1 : \phi_1 \quad \ldots \quad \Pi \vdash e_n : \phi_n}{\Pi \vdash \texttt{new } C(\vec{e}_n) : C}\ (\mathcal{F}(C) = \vec{f}_n,\ n \geq 0) \qquad (\omega):\ \frac{}{\Pi \vdash e : \omega}$$

$$(\textsc{invk}):\ \frac{\Pi \vdash e : \langle m{:}(\vec{\phi}_n)\to\sigma\rangle \quad \Pi \vdash e_1 : \phi_1 \ \ldots\ \Pi \vdash e_n : \phi_n}{\Pi \vdash e.m(\vec{e}_n) : \sigma} \qquad (\textsc{fld}):\ \frac{\Pi \vdash e : \langle f{:}\sigma\rangle}{\Pi \vdash e.f : \sigma}$$

$$(\textsc{join}):\ \frac{\Pi \vdash e : \sigma_1 \ \ldots\ \Pi \vdash e : \sigma_n}{\Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2) \qquad (\textsc{var}):\ \frac{}{\Pi, x{:}\phi \vdash x : \sigma}\ (\phi \lhd \sigma)$$

**Fig. 2.** Type assignment rules for the semantical system for $\text{FJ}^{\not\in}$

as demanded by the context in which it appears. In particular, an intersection may combine two or more different (even non-unifiable) analyses of the *same* field or method. Types are therefore not *records*: records can be characterised as intersection types of the shape $\langle \ell_1{:}\sigma_1, \cdots, \ell_n{:}\sigma_n\rangle$ where all $\sigma_i$ are intersection-free, and all labels $\ell_i$ are distinct; in other words, records are intersection types, but not vice-versa (see Definition 12).

We include a type constant for each class, which we can use to type objects which therefore always have a type, like for the case when an object does not contain any fields or methods (as is the case for `Object`) or, more generally, because no fields or methods can be safely invoked. The type constant $\omega$ is a *top* (maximal) type, assignable to all expressions and serves typically to type subterms that do not contribute to the normal form of an expression. The following *subtype* relation facilitates the selection of individual behaviours from an intersection.

**Definition 7 (Subtype Relation).** The subtype relation $\lhd$ is induced by the fact that an intersection type is smaller than each of its components, and is defined is the smallest preorder satisfying:

$$\phi \ \lhd\ \omega \qquad \phi \cap \psi \ \lhd\ \phi \qquad \phi \cap \psi \ \lhd\ \psi \qquad \phi \lhd \psi \ \&\ \phi \lhd \psi' \ \Rightarrow\ \phi \lhd \psi \cap \psi'$$

We write $\sim$ for the equivalence relation generated by $\lhd$, extended by

$$\sigma \sim \sigma' \ \Rightarrow\ \langle f{:}\sigma\rangle \ \sim\ \langle f{:}\sigma'\rangle$$
$$\forall i \in \underline{n}\,[\,\phi'_i \sim \phi'_i\,]\ \&\ \sigma \sim \sigma' \ \Rightarrow\ \langle m{:}(\phi_1, \ldots, \phi_n)\to\sigma\rangle \ \sim\ \langle m{:}(\phi'_1, \ldots, \phi'_n)\to\sigma'\rangle$$

We consider types modulo $\sim$; in particular, all types in an intersection are different and $\omega$ does not appear in an intersection. It is easy to show that $\cap$ is associative and commutative with respect to $\sim$, so we will abuse notation slightly and write $\sigma_1 \cap \ldots \cap \sigma_n$ (where $n \geq 2$) to denote a general intersection, where all $\sigma_i$ are distinct and the order is unimportant. In a further abuse of notation, $\phi_1 \cap \ldots \cap \phi_n$ will denote the type $\phi_1$ when $n = 1$, and $\omega$ when $n = 0$.

**Definition 8 (Type Contexts [19]).**
1. A *type statement* is of the form $e : \phi$, with $e$ as *subject*.
2. A context $\Pi$ is a set of type statements with (distinct) variables as subjects; $\Pi, x{:}\phi$ stands for the context $\Pi \cup \{x{:}\phi\}$ (so then either $x$ does not appear in $\Pi$ or $x{:}\phi \in \Pi$) and $x{:}\phi$ for $\emptyset, x{:}\phi$.

3. We extend $\lhd$ to contexts: $\Pi' \lhd \Pi \;\Leftrightarrow\; \forall\, x{:}\phi \in \Pi\; \exists\, \phi' \lhd \phi x{:}\phi' \in \Pi'$.
4. If $\overrightarrow{\Pi_n}$ is a sequence of contexts, then $\bigcap \overrightarrow{\Pi_n}$ is the context defined as follows: $x{:}\phi_1 \cap \ldots \cap \phi_m \in \bigcap \overrightarrow{\Pi_n}$, if and only if $\{x{:}\phi_1, \ldots, x{:}\phi_m\}$ is the non-empty set of all statements in the union of the contexts that have $x$ as subject.

We will now define our notion of type assignment.

**Definition 9 (Semantic Type Assignment [19]).** Semantical type assignment for $\text{FJ}^{\not\in}$ is defined by the natural deduction system of Figure 2.

Notice that new objects like `new C(···)` can be dealt with by both the rules $(\text{OBJ})$ and $(\text{NEWM})$; then the context $\Pi$ can be any. Moreover, we do not need any of the information of the nominal type system here, other than that provided by the rule $(\text{OBJ})$.

We should perhaps emphasise that, as remarked above, we *explicitly do not type classes*; instead, the rules $(\text{NEWF})$ and $(\text{NEWM})$ create a field or method type for an object, essentially stating that this field or method is available, and what its current type is. So method bodies are checked *every time* we need that an object has a specific method type, and the various types for a particular method used throughout a program need not be the same, as is the case for the nominal system (see Definition 5). So, in our system, we would have, in principle, an infinite type for each class, which we cannot establish when typing the class separately; rather, we let the context of each `new D()` decide which type is needed, so the type for an occurrence of `new D()` is 'constructed' by need, and not from an analysis of the class.

The rules of our type assignment system are fairly straightforward generalisations of the rules of the strict intersection type assignment system for LC to OO, whilst making the step from a higher order to a first order language: for example, $(\text{FLD})$ and $(\text{INVK})$ are analogous to $(\to E)$; $(\text{NEWF})$ and $(\text{NEWM})$ are a form of $(\to I)$; and $(\text{OBJ})$ can be seen as a universal $(\omega)$-like rule for *objects* only.

The only non-standard rule from the point of view of similar work for TRS and traditional nominal OO-type systems is $(\text{NEWM})$, which derives a type for an object that presents an analysis of a method that is available in that object. Note that the analysis involves typing the method body and the assumptions (*i.e.* requirements) on the formal parameters are encoded in the derived type (to be checked on invocation). However, a method body may also make requirements on the *receiver*, through the use of the variable `this`. In our system we check that these hold *at the same time* as typing the method body (so-called *early self typing*; with *late self typing*, as used in [6], we would check the type of the receiver at the point of method invocation). This checking of requirements on the object itself is where the expressive power of our system resides. If a method calls itself recursively, this recursive call must be checked, but – crucially – carries a *different* type if a valid derivation is to be found. Thus only recursive calls which terminate at a certain point (*i.e.* which can then be assigned $\omega$ or $c$, and thus ignored) will be typeable in the system.

As is standard for intersection type assignment systems, our system is set up to satisfy both subject reduction *and* subject expansion.

**Theorem 10 (Subject reduction and expansion [19]).** Let $e \to e'$; then $\Pi \vdash e : \phi$ if and only if $\Pi \vdash e' : \phi$.

Notice that, as usual, computational equality between expressions in FJ$^{\not\in}$ is undecidable; as a consequence, through Theorem 10 we obtain that type assignment in our system is undecidable as well.

We have also shown (variants of) the characterisation of normalisation properties:

**Theorem 11 ([19]).** 1. If $e$ is a head-normal form then there exists a strict type $\sigma$ and type context $\Pi$ such that $\Pi \vdash e : \sigma$; moreover, if $e$ is not of the form $\texttt{new } C(\vec{e}_n)$ then for any arbitrary strict type $\sigma$ there is a context such that $\Pi \vdash e : \sigma$.
2. $\Pi \vdash e : \sigma$ if and only if $e$ has a head-normal form.
3. If $\mathcal{D} :: \Pi \vdash e : \sigma$ with $\mathcal{D}$ and $\Pi$ $\omega$-free then $e$ has a normal form.

## 3 Curry type assignment

The nominal type system for Java is so far the accepted standard, but many researchers are looking for more expressive type systems that deal with intricate details of object oriented programming and in particular with side effects. It will be clear that through the system we presented above, we propose a different path, an alternative to the nominal approach. We illustrate the strength of our approach in this section by presenting a basic (decidable) functional system, that allows for us to show a preservation result with respect to a notion of Curry type assignment for CL. This basic system is based on a true restriction of our semantical type system; the restriction consists of removing the type constant $\omega$ as well as intersection types from the type language, but not completely: we will still allow for types to be combined as by rule $(\text{JOIN})$ above, but only if the labels involved are different: the intersection types we allow, thereby, correspond to *records*. The additional change we made was to type classes (and the class table and programs) explicitly, and in that use a different approach when dealing with recursive classes.

It is worthwhile to point out that, above, the fact that we allow more than just record types is crucial for the results: without allowing arbitrary intersections (and $\omega$) we could not show that type assignment is closed under conversion. The undecidability of type inference in our type assignment system follows as a consequence of the conversion result shown in the previous section. Thus, to be of *practical* use for program analysis we must restrict the type assignment system in a decidable fashion.

The notion of the type assignment system we present here is based on Curry's type system for CL (or simply-typed LC), and is inspired by Milner's system for ML [16]. As such, it is not a true restriction of the system we defined above, and therefore different from the Curry system we presented in [19]. However, we can show that our encoding of CL into FJ$^{\not\in}$ (Section 3) preserves typeability; this is mainly because CL is a notion of computation defined without recursion. The basic approach of our restriction is to only assume a *single* behaviour for each element of a program - that is, we remove intersections. So, for example, when deriving a type for a method each argument and each field of the receiver may only have one type. While this may seem overly restrictive, we show that our system is expressive enough to type those programs that correspond to computable functions, $\lambda$-terms typeable in Curry's system of simple types.

We demonstrate the decidability of this restricted form of type assignment by first showing that the system has a *principal typings* property, and then arguing that there is

a terminating algorithm which computes these typings. A principal typings property for a type system states that for each typeable term there is a typing (an context-type pair) which is *most general*, in the sense that all other typings assignable to that term can be generated from it. Typeability is decidable, then, if there is a (terminating) algorithm which computes whether a principal typing exists for any given term. In the latter half of this section, we discuss the implementation of such an algorithm for our restricted type assignment system, which we will now define.

**Definition 12 (Curry types for FJ$^{\not\epsilon}$).** 1. *Curry (object) types* for FJ are defined by:

$$\sigma,\tau \ ::= \ \phi \mid \langle f_1{:}\sigma,\ \ldots,\ f_n{:}\tau,\ m_1{:}(\vec{\alpha})\to\beta,\ \ldots,\ m_k{:}(\vec{\gamma})\to\delta\rangle \quad (n+k\geq 1)$$

2. We will call a type of the shape $\langle f{:}\sigma\rangle$ a *field* type, one of the shape $\langle m{:}(\vec{\alpha})\to\beta\rangle$ a *method* type, and $\langle\cdots\rangle$ with 2 or more components a *record* type and let $\rho$ range over those. We write $\ell$ for arbitrary labels and, by abuse of notation, will also use $\langle\ell{:}\sigma\rangle$ to represent $\langle m{:}\sigma\rangle$, even though in $\langle m{:}(\vec{\alpha})\to\beta\rangle$, the structure $(\vec{\alpha})\to\beta$ is *not* a type. We write $\langle\ell{:}\sigma\rangle\in\rho$ (or $\ell\in\rho$) when $\ell{:}\sigma$ occurs in $\rho$, and assume that all labels are unique in records.
3. We call two types $\rho$ and $\rho'$ *compatible* when: if $\langle\ell{:}\sigma\rangle\in\rho$ and $\langle\ell{:}\sigma'\rangle\in\rho'$, then either: $\sigma$ and $\sigma'$ are compatible types, or $\sigma=\sigma'$.
4. We define the operator $\sqcup$ (*join*) on types by: $\rho_1\sqcup\rho_2$ is the type composed out of the union of two compatible types, defined as: $\langle\ell{:}\sigma\rangle\in\rho_1\sqcup\rho_2$ if and only if either:
   - $\langle\ell{:}\sigma\rangle\in\rho_1$ and $\ell\notin\rho_2$; or
   - $\langle\ell{:}\sigma\rangle\in\rho_2$ and $\ell\notin\rho_1$; or
   - $\sigma=\sigma_1\sqcup\sigma_2$, with $\langle\ell{:}\sigma_1\rangle\in\rho_1$, $\langle\ell{:}\sigma_2\rangle\in\rho_2$ and $\sigma_1$ and $\sigma_2$ are compatible; or
   - $\sigma=\sigma_1$, with $\langle\ell{:}\sigma_1\rangle\in\rho_1$ and $\langle\ell{:}\sigma_2\rangle\in\rho_2$ and $\sigma_1=\sigma_2$.
5. When we write a record type as $\langle\ell{:}\sigma\rangle\sqcup\rho$, then $\ell\notin\rho$.

Notice that compatible records can have different labels, even distinct. Moreover, even when we allow for the notation $\langle m{:}\sigma\rangle$ for a method type, then $\sigma$ is never a record type: we can only derive those for variables, fields, and objects; see Definition 18. As an example of compatible types, consider $\langle f_1{:}\langle f_2{:}\sigma, f_3{:}\tau\rangle\rangle$ and $\langle f_1{:}\langle f_3{:}\tau, f_4{:}\rho\rangle\rangle$; on the other hand, $\langle f_1{:}\langle f_2{:}\sigma, f_3{:}\tau\rangle\rangle$ and $\langle f_1{:}\langle f_3{:}\mu, f_4{:}\rho\rangle\rangle$ with $\tau\neq\mu$ are not compatible.

**Definition 13 (Curry contexts and environments).** 1. A *Curry context* is a mapping from term variables (including `this`) to Curry types.
2. We call two contexts $\Gamma_1$ and $\Gamma_2$ compatible whenever: if $x{:}\sigma\in\Gamma_1$ and $x{:}\tau\in\Gamma_2$, then $\sigma$ and $\tau$ are compatible record types.
3. The operation of $\sqcup$ can be extended to compatible contexts as follows:

$$\begin{array}{lllll} \mathit{Join}\ \Gamma, x{:}\sigma\ \Gamma', x{:}\tau & = & (\mathit{Join}\ \Gamma\ \Gamma'), x{:}\sigma\sqcup\tau & \\ \mathit{Join}\ \Gamma, x{:}\sigma\quad \Gamma' & = & (\mathit{Join}\ \Gamma\ \Gamma'), x{:}\sigma & (x\notin\Gamma') \\ \mathit{Join}\quad \emptyset\quad \Gamma' & = & \Gamma' & \end{array}$$

4. An environment $\mathcal{E}$ is a mapping from class names to types (normally records).

Notice that compatible contexts can have different variables, even distinct; but even if a variable appears in both, the labels in its types can be different as well.

The operation of substitution, which replaces type variables with types (and type variables with simple types), allows one type (or record) to be generated from another.

$$(\text{NEW}_r):\ \frac{\emptyset;\mathcal{E},C{:}\rho\vdash e_i:\sigma_i\quad(i\in\underline{n})}{\Gamma;\mathcal{E},C{:}\rho\vdash \texttt{new}\ C\,(\vec{e})\ :\rho}\ (\overrightarrow{\langle f{:}\sigma_i\rangle}\in\rho)\qquad (\text{VAR}):\ \frac{}{\Gamma,x{:}\sigma;\mathcal{E}\vdash x:\sigma}$$

$$(\text{NEW}_c):\ \frac{\emptyset;\mathcal{E},C{:}\rho\vdash e_i:\sigma_i\quad(i\in\underline{n})}{\Gamma;\mathcal{E},C{:}\rho\vdash \texttt{new}\ C\,(\vec{e})\ :S\rho}\ (\overrightarrow{\langle f{:}\sigma_i\rangle}\in S\rho)\qquad (\text{PROJ}):\ \frac{\Gamma;\mathcal{E}\vdash e:\rho}{\Gamma;\mathcal{E}\vdash e:\langle\ell{:}\sigma\rangle}\ (\langle\ell{:}\sigma\rangle\in\rho)$$

$$(\text{INVK}):\ \frac{\Gamma;\mathcal{E}\vdash e:\langle m{:}(\vec{\sigma}_n)\to\sigma\rangle\quad\Gamma;\mathcal{E}\vdash e_i:\sigma_i\ (i\in\underline{n})}{\Gamma;\mathcal{E}\vdash e.m\,(\vec{e_i})\ :\sigma}\qquad (\text{FLD}):\ \frac{\Gamma;\mathcal{E}\vdash e:\langle f{:}\sigma\rangle}{\Gamma;\mathcal{E}\vdash e.f:\sigma}$$

$$(\mathcal{CT}):\ \frac{\begin{array}{l}\texttt{this}{:}\rho,\overrightarrow{x{:}\sigma}_{n_1};\mathcal{E},C{:}\rho\vdash e_1:\tau_1\qquad(\mathcal{M}b(C,m_1)=(\vec{x}_{n_1},e_1))\quad\cdots\\ \texttt{this}{:}\rho,\overrightarrow{x{:}\sigma}_{n_m};\mathcal{E},C{:}\rho\vdash e_m:\tau_m\qquad(\mathcal{M}b(C,m_m)=(\vec{x}_{n_m},e_m))\end{array}\quad\mathcal{E},C{:}\rho\vdash\mathcal{CT}:\diamond}{\begin{array}{l}\mathcal{E},C{:}\rho\vdash \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\};\mathcal{CT}:\diamond\\ (\rho=\langle\overrightarrow{f{:}\phi}_n,m_1{:}(\vec{\sigma}_{n_1})\to\tau_1,\ldots,m_m{:}(\vec{\sigma}_{n_m})\to\tau_m\rangle,\ \mathcal{F}(C)=\vec{f}_n,\ \mathcal{M}(C)=\vec{m}_m)\end{array}}$$

$$(\varepsilon):\ \frac{}{\Pi\vdash\varepsilon:\diamond}\qquad (\text{PROG}):\ \frac{\mathcal{E}\vdash\mathcal{CT}:\diamond\qquad\Gamma;\mathcal{E}\vdash e:\sigma}{\Gamma;\mathcal{E}\vdash(\mathcal{CT},e):\sigma}$$

**Fig. 3.** Type assignment rules for the Curry type system for FJ$^{\not\in}$

**Definition 14 (Substitution).** 1. The type *substitution* $\langle\varphi\mapsto\sigma\rangle$, which is a function from types to types, is defined as follows:

$$\begin{array}{lcl}(\varphi{\mapsto}\sigma)\,\varphi & = & \sigma\\ (\varphi{\mapsto}\sigma)\,\varphi' & = & \varphi'\quad\varphi\neq\varphi'\\ (\varphi{\mapsto}\sigma)\,\langle f{:}\sigma'\rangle & = & \langle f{:}(\varphi{\mapsto}\sigma)\,\sigma'\rangle\\ (\varphi{\mapsto}\sigma)\,\langle m{:}\vec{\sigma}_n{\to}\sigma'\rangle & = & \langle m{:}((\varphi{\mapsto}\sigma)\,\sigma_1,\ldots,(\varphi{\mapsto}\sigma)\,\sigma_n){\to}(\varphi{\mapsto}\sigma)\,\sigma'\rangle\end{array}$$

The extension to records $(\varphi{\mapsto}\sigma)\,\rho$ is defined as can be expected.

2. If $S_1$ and $S_2$ are substitutions, then so is $S_2\circ S_1$ where $(S_2\circ S_1)\,\sigma=S_2\,(S_1\,\sigma))$; we write $\vec{S}_n$ for $S_n\circ\cdots\circ S_1$.
3. $S\,\Gamma=\{x{:}S\,\sigma\mid x{:}\sigma\in\Gamma\}$.
4. *Id* denotes the *identity* substitution.
5. For two types $\sigma_1$ and $\sigma_2$, if there exists a substitution $S$ such that $S\,\sigma_1=\sigma_2$ then we say that $\sigma_2$ is an *instance* of $\sigma_1$.
6. We say that a type variable is *fresh* if it does not occur in any type we are considering at that moment; we also say that a *fresh instance of* $\sigma$ is the type created out of $\sigma$ by substitution each type variable in $\sigma$ by a fresh one.

Simple type assignment is defined through:

**Definition 15 (Curry type assignment for FJ$^{\not\in}$).** *Curry type assignment* for FJ$^{\not\in}$-expressions is defined through the rules in Figure 3.

Notice that judgements depend not only on contexts, but also on environments; the first six rules type expressions, whereas the others deal with the class table and the program.

Rule $(\mathcal{CT})$ comes in place of the rules deriving OK for the nominal system. It checks the occurrence of $C{:}\rho$ in the environment for class $C$. By using $\texttt{this}{:}\rho$ when typeing the methods, it insists that the types derived for the methods and fields in a class are the same as used for the receivers; also rule $(\text{NEW}_r)$ is used for occurrences of $\texttt{new}\ C\,(\vec{e})$ inside the definition of $C$ which insists that the same record type is used for the 'recursive' instances of $C$ as well. This corresponds to the usual way of dealing with recursion as in

ML's rule $(\texttt{fix})$ (note that OO languages have *two* types of recursion) and corresponds to the nominal rule $(\textsc{class})$ above.

Notice that $\rho$ is not unique; the rule accepts any type that fits. Below, in the algorithm $\mathcal{PT}$, we will calculate the smallest; then when we use a type for $C$, as in rule $(\textsc{new}_c)$ which is used for occurrences of $\texttt{new}\ C(\vec{e})$ *outside* the definition of $C$, this type will be a substitution instance of the one calculated. Thereby, this introduces a notion of *polymorphism* into our system; each instance of class $C$ will be typed differently, but any of its types can be generated from $\mathcal{E}(C)$ by (projection and) substitution.

Also, $(\textsc{new}_r)$ and $(\textsc{new}_c)$, in combination with rule $(\textsc{proj})$, come in place of the rules $(\textsc{newf})$ and $(\textsc{newm})$ of the semantical system. We could have removed the separation of the two $\textsc{new}$ rules and only used rule $(\textsc{new}_c)$; this would give a 'Mycroft'-like way of dealing with recursion, which is only semi-decidable, rather than a 'Milner'-like way as we do now; it could be used for a *type-check* system, however.

Note also that rules $(\varepsilon)$ and $(\mathcal{CT})$ checks the typeability of a class table, and rule $(\textsc{prog})$ specifies how to type a $\textsc{fj}^{\not\iota}$ program.

We can show that type substitution is sound for expressions.

**Theorem 16.** For all substitutions $S$, if $\Gamma;\mathcal{E} \vdash e : \sigma$ then $S(\Gamma;\mathcal{E}) \vdash e : S\sigma$.

We will now show that simple type assignment has a *principal typings* property. At the heart of type inference lies the problem of *unification* - finding a common instance of two types; since we deal with records, we will also need to join those, after we have made them compatible through unification.

**Definition 17 (Type Unification).** The operation of unification is defined on types, and extended to contexts as follows:

1. The function *Unify*, which takes two simple types and returns a substitution, is defined by cases through:

$$
\begin{array}{llll}
\mathit{Unify} & \varphi & \varphi' & = (\varphi \mapsto \varphi') \\
\mathit{Unify} & \varphi & \sigma & = (\varphi \mapsto \sigma) \quad (\varphi \textit{ not in } \sigma) \\
\mathit{Unify} & \sigma & \varphi & = (\varphi \mapsto \sigma) \quad (\varphi \textit{ not in } \sigma) \\
\mathit{Unify} & \langle f{:}\sigma \rangle & \langle f{:}\alpha \rangle & = \mathit{Unify}\ \sigma\ \alpha \\
\mathit{Unify} & \langle m{:}(\vec{\sigma}_n) \to \tau \rangle & \langle m{:}(\vec{\alpha}_n) \to \beta \rangle & = \vec{S}_n \\
\multicolumn{3}{r}{\textit{where}\ \ S_1} & = \mathit{Unify}\ \tau\ \beta \\
\multicolumn{3}{r}{S_{i+1}} & = \mathit{Unify}\ (\vec{S}_i\,\sigma_i)\ (\vec{S}_i\,\alpha_i) \quad (i \in \underline{n{-}1})
\end{array}
$$

   These are the cases where unification is defined, *i,e,* where either a substitution is created, or unification fails (when $\varphi$ occurs in $\sigma$ in the second and third case). This implies that no action is taken when unifying two method types that have different method names; rather, those are joined in a record.
2. On records, unification is defined through:

$$
\begin{array}{llll}
\mathit{Unify}\ \langle \ell{:}\sigma \rangle \sqcup \rho\ \langle \ell{:}\tau \rangle \sqcup \rho' & = & S_2 \circ S_1 & \\
\multicolumn{2}{r}{\textit{where}\ \ S_1} & = \mathit{Unify}\ \sigma\ \tau & \\
\multicolumn{2}{r}{S_2} & = \mathit{Unify}\ (S_1\,\rho)\ (S_1\,\rho') & \\
\mathit{Unify}\ \langle \ell{:}\sigma \rangle \sqcup \rho \quad \rho' & = & \mathit{Unify}\ \rho\ \rho' \quad (\ell \notin \rho') &
\end{array}
$$

3. Unification can be extended to contexts as follows:

$$\begin{array}{lllll} \mathit{Unify}\ \Gamma, x{:}\sigma\ \Gamma', x{:}\tau & = & S' \circ S & & \\ \qquad\qquad\qquad\qquad \mathit{where} & S & = & \mathit{Unify}\ \sigma\ \tau & \\ \qquad\qquad\qquad\qquad\qquad & S' & = & \mathit{Unify}\ (S\,\Gamma)\ (S\,\Gamma') & \\ \mathit{Unify}\ \Gamma, x{:}\sigma \quad \Gamma' & = & \mathit{Unify}\ \Gamma\ \Gamma' & (x \notin \Gamma') & \\ \mathit{Unify} \quad \emptyset \quad \Gamma' & = & \mathit{Id} & & \end{array}$$

4. By abuse of notation, we will allow also the unification of any number of types or contexts '*at one fell swoop*', by defining

$$\begin{array}{rcl} \mathit{Unify}\ \sigma_1\ \sigma_2\ \ldots\ \sigma_n & = & \mathit{Unify}\ (S\,\sigma_2)\ (S\,\sigma_3)\ \ldots\ (S\,\sigma_n) \circ S \\ \mathit{where}\ S & = & \mathit{Unify}\ \sigma_1\ \sigma_2 \end{array}$$

Notice that unification on records (as specified in the last two cases of the first part) recurses on the number of types in the record, ending up with the unification of simple types, which is dealt with in the first five cases.

It is easy to show that unification creates compatible contexts, which it is designed to do; it is, however, not fully satisfactory for use in our principal typing algorithm, since it does not always adequately checks that the 'offered type' of the argument is not less specific that the 'demanded type' of the parameter of a method invocation in terms of labels that occur. For example, as can be seen in the algorithm, for the expression $e.m(e')$, it can be that we infer that the principal type of $e$ is $\langle m{:}\langle f{:}\sigma, f'{:}\sigma'\rangle{\to}\rho\rangle$, and the principal type of $e'$ is $\langle f{:}\tau\rangle$. Assuming $\sigma$ and $\tau$ are unifiable,

$$\mathit{Unify}\ \langle m{:}\langle f{:}\sigma, f'{:}\sigma'\rangle{\to}\rho\rangle\ \langle m{:}(\langle f{:}\tau\rangle){\to}\varphi\rangle$$

would succeed; however, there is no guarantee that $e'$ has type $\langle f'{:}\tau'\rangle$ as well.

We could have amended the unification algorithm, but that would have made it a great deal more intricate. Rather, in the principal typing algorithm, we add an additional test that checks if the offered type matches the demanded type, by checking that the labels in the demanded type all occur in the offered type. This implies, of course, that type assignment can fail not just because unification fails.

Using the concepts of substitution and unification, we can now define what principal typings are for our system. Since our types express information about fields and methods, and since objects may have many different fields and methods (each with their own unique behaviours), our principal types must actually be *records*. This makes defining the principal typings somewhat complicated, as can be seen below.

Also, we in fact calculate the principal typing for a *program*, by traversing the class table, building up the environment that contains the types for the classes, and type the final expression in that environment.

**Definition 18 (Principal Typing).**
1. A *typing* is a pair $\langle \Gamma; \sigma \rangle$ of a context (including `this`) and a type.
2. The function $\mathcal{PT}$ (*principal typing*), from expressions and environments to typings and environments, is defined inductively as follows:

$$
\begin{array}{lcll}
\mathcal{PT}(\mathtt{x};\mathcal{E}) & = & \langle \mathtt{x}{:}\varphi;\varphi\rangle;\mathcal{E} & (\varphi \textit{ fresh})\\
\mathcal{PT}(\mathtt{this};\mathcal{E}) & = & \langle \mathtt{this}{:}\varphi;\varphi\rangle;\mathcal{E} & (\varphi \textit{ fresh})\\
\mathcal{PT}(\mathtt{e}.\mathtt{f};\mathcal{E}) & = & S\langle\Gamma;\varphi\rangle;S\,\mathcal{E}' & \\
\quad \textit{where} & \mathcal{PT}(\mathtt{e};\mathcal{E}) & = \langle\Gamma;\rho\rangle;\mathcal{E}' & \\
& S & = \textit{Unify}\,\langle \mathtt{f}{:}\varphi\rangle\,\rho & (\varphi \textit{ fresh})\\
\multicolumn{4}{l}{\mathcal{PT}(\mathtt{e}.\mathtt{m}(\vec{\mathtt{e}}_n);\mathcal{E}_0) = \vec{S}_2\,\langle\Gamma\sqcup\Gamma_1\sqcup\cdots\sqcup\Gamma_n;\varphi\rangle;\vec{S}_2\,\mathcal{E}_{n+1}}\\
\quad \textit{where} & \mathcal{PT}(\mathtt{e};\mathcal{E}_0) & = \langle\Gamma;\rho\rangle;\mathcal{E}_1 & \\
& \mathcal{PT}(\mathtt{e}_i;\mathcal{E}_i) & = \langle\Gamma_i;\gamma_i\rangle;\mathcal{E}_{i+1} & (i\in\underline{n})\\
& S_1 & = \textit{Unify}\,\langle \mathtt{m}{:}(\vec{\gamma}_n)\to\varphi\rangle\,\rho & (\varphi \textit{ fresh},\\
& & \multicolumn{2}{l}{\quad \textit{all labels in } \sigma_i \textit{ in } \langle \mathtt{m}{:}(\vec{\sigma}_n)\to\tau\rangle\in\rho \textit{ appear in } \gamma_i)}\\
& S_2 & = \textit{Unify}\,(S_1\Gamma)\,(S_1\Gamma_1)\cdots(S_1\Gamma_n) & \\
\multicolumn{4}{l}{\mathcal{PT}(\mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}_n);\mathcal{E}_1) = S'\circ\vec{S}_n\langle\Gamma_1\sqcup\cdots\sqcup\Gamma_n;\rho\rangle;S'\circ\vec{S}_n\mathcal{E}_{n+1}}\\
\quad \textit{where} & \rho & = \begin{cases}\mathcal{E}_1\,\mathtt{C} & (\textit{definition of } \mathtt{C} \textit{ depends on this})\\ \textit{fresh instance of } \mathcal{E}_1\,\mathtt{C} & (\textit{otherwise})\end{cases} & \\
& \multicolumn{3}{l}{\mathcal{PT}(\mathtt{e}_i;\mathcal{E}_i) = \langle\Gamma_i;\sigma_i\rangle;\mathcal{E}_{i+1} \qquad (i\in\underline{n})}\\
& S_i & = \textit{Unify}\,(\vec{S}_{i-1}\langle \mathtt{f}_i{:}\sigma_i\rangle)\,(\vec{S}_{i-1}\rho) & (i\in\underline{n},\ S_0 = \textit{Id},\\
& & & \quad \textit{all labels } \vec{\mathtt{f}}_n \textit{ appear in } \rho)\\
& S' & = \textit{Unify}\,(\vec{S}_n\Gamma)\,(\vec{S}_n\Gamma_1)\cdots(\vec{S}_n\Gamma_n) &
\end{array}
$$

3. We use $\mathcal{PT}$ also for the function that calculates the principal type for a each class definition in the class table, and builds the environment:

$$
\begin{array}{lcll}
\mathcal{PT}(\varepsilon:\mathtt{e};\mathcal{E}) & = & \mathcal{PT}(\mathtt{e};\mathcal{E}) & \\
\multicolumn{4}{l}{\mathcal{PT}(\mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{C}'\ \{\overrightarrow{\mathtt{f}\mathtt{d}_n}\ \overrightarrow{\mathtt{m}\mathtt{d}_m}\},\mathcal{CT}:\mathtt{e};\mathcal{E}) = \mathcal{PT}(\mathcal{CT}:\mathtt{e};\mathcal{E},\mathtt{C}{:}\rho)}\\
\quad \textit{where} & \multicolumn{3}{l}{\mathcal{PT}(\mathtt{e}_{b_j};\mathcal{E}_j) = \langle\overrightarrow{x_i{:}\sigma_i},\mathtt{this}{:}\alpha_j;\tau_j\rangle;\mathcal{E}_{j+1}\ {}^{(*)}}\\
& & \multicolumn{2}{l}{\qquad\qquad \mathcal{MB}(\mathtt{C},\mathtt{m}_j) = (\overrightarrow{x_i},\mathtt{e}_{b_j})\in\overrightarrow{\mathtt{m}\mathtt{d}}}\\
& \mathcal{E}_1 & = \mathcal{E},\mathtt{C}{:}\varphi & (\varphi \textit{ fresh})\\
& S & = \textit{Unify}\,\overrightarrow{\alpha_j}\,(\mathcal{E}_{m+1}\mathtt{C})\,\langle\overrightarrow{\mathtt{f}{:}\varphi},\overrightarrow{\mathtt{m}{:}(\vec{\sigma})\to\tau}\rangle & \\
& \rho & = \overrightarrow{S\,\alpha_j}\sqcup S\,(\mathcal{E}_{m+1}\mathtt{C})\sqcup S\langle\overrightarrow{\mathtt{f}{:}\varphi},\overrightarrow{\mathtt{m}{:}(\vec{\sigma})\to\tau}\rangle & \\
& & \multicolumn{2}{l}{\qquad\qquad (\mathcal{F}(\mathtt{C}) = \vec{\mathtt{f}},\ \vec{\varphi}_n \textit{ fresh})}
\end{array}
$$

${}^{(*)}$ $\sigma_i$ or $\alpha_j$ are fresh variables whenever $x_i \notin fv(\mathtt{e}_{b_j})$ or `this` does not occur in $\mathtt{e}_{b_j}$; of course the assumption is here that all free variables in a method body are mentioned in the parameters list: methods have no free variables. Notice that if $\mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}_n)$ occurs in $\mathtt{e}_{b_j}$, then the type stored in the environment is taken itself, rather than instantiated, so might be affected by the unifications that are calculated.

Notice that, in the case for $\mathcal{PT}(\mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}_n);\mathcal{E})$, we have to take a fresh instance of the type calculated for $\mathtt{C}$. We have stored the principal type for $\mathtt{C}$ into the environment in $\mathcal{PT}(\mathcal{CT};\mathcal{E})$ and need to access it when creating a new object; however, we have to work from a *copy*: otherwise, we would change $\mathcal{E}\,\mathtt{C}$ during the unification process, making it change before a new access. Thereby, this operation introduces a notion of *polymorphism* into our system; each instance of class $\mathtt{C}$ will be typed differently, but any

of its types can be generated from $\mathcal{E}\ C$ by (projection and) substitution; the substitutions will be calculated by $\mathcal{PT}$, as demanded by the context in which the object appears, and depending on exactly what expressions it gets initialised with.

We can show the expected properties for $\mathcal{PT}$:

**Theorem 19 (Soundness of $\mathcal{PT}$).** If $\mathcal{PT}(e;\mathcal{E}) = \langle \Gamma;\sigma\rangle;\mathcal{E}'$, then $\Gamma;\mathcal{E}' \vdash_{\mathtt{C}} e : \sigma$.

**Theorem 20 (Completeness of $\mathcal{PT}$).** If $\Gamma;\mathcal{E} \vdash_{\mathtt{C}} e : \sigma$ (where $e$ is not part of a class definition), then there exists a typing $\langle \Gamma';\sigma'\rangle;\mathcal{E}$ such that $\mathcal{PT}(e;\mathcal{E}) = \langle \Gamma';\sigma'\rangle;\mathcal{E}$ and a substitution $S$ such that $S\,\Gamma' \subseteq \Gamma$, and $\sigma = S\,\sigma'$.

As an example of a program we can type, consider the following which constitutes perhaps the simplest example of a term without head-normal form in OO:

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

This program has a method `m` which simply calls itself recursively, and `new C().m()` *loops*:

$$\texttt{new C().m()} \;\rightarrow\; \texttt{this.m()}[\texttt{new C()/this}] \;=\; \texttt{new C().m()}$$

so, in particular, `new C().m()` has no normal form, not even a head-normal form; this correspond to the ML-program $(\texttt{fix}\,x.x)$. Running $\mathcal{PT}$ returns $C{:}\langle \mathtt{m}{:}()\rightarrow\varphi\rangle$ and therefore running $\mathcal{PT}(\texttt{new C().m()};\mathcal{E})$ gives $(\emptyset;\varphi)$ which is also the typing for $(\texttt{fix}\,x.x)$. Notice that, since `new C().m()` has no head-normal form, it is not typeable in our intersection system, so we cannot show '$\Gamma \vdash_{\mathtt{C}} e : \sigma \Rightarrow \Gamma \vdash e : \sigma$'; the same observation can be made with respect to type assignment for the $\lambda$-calculus and ML.

## 4 OOCL

We will now relate this notion of type assignment to one from the world of functional programming, by defining an encoding of Combinatory Logic [14] (CL) into FJ$^{\not c}$, and showing that assignable types are preserved by this encoding.

**Definition 21.** Combinatory Logic consists of the function symbols $\mathbf{S}, \mathbf{K}$ where terms are defined over the grammar

$$t \;::=\; x \mid \mathbf{S} \mid \mathbf{K} \mid t_1\,t_2$$

and the reduction is defined via the rewrite rules:

$$\begin{array}{lcl} \mathbf{K}\,x\,y & \rightarrow & x \\ \mathbf{S}\,x\,y\,z & \rightarrow & x\,z\,(y\,z) \end{array}$$

```
class Combinator extends Object {
    Combinator app(Combinator x) { return this; }
}
class K extends Combinator {
    Combinator app(Combinator x) { return new K_1(x); }
}
class K_1 extends K {
    Combinator x;
    Combinator app(Combinator y) { return this.x; }
}
class S extends Combinator {
    Combinator app(Combinator x) { return new S_1(x); }
}
class S_1 extends S {
    Combinator x;
    Combinator app(Combinator y) { return new S_2(this.x, y); }
}
class S_2 extends S_1 {
    Combinator y;
    Combinator app(Combinator z) { return this.x.app(z).app(this.y.app(z)); }
}
```

**Fig. 4.** The class table for Object-Oriented Combinatory Logic (OOCL) programs

CL can be seen as a higher-order TRS.

Our encoding of CL in $\text{FJ}^{\not\in}$ is based on a Curryfied first-order version of the system above (see [7] for details), where the rules for **S** and **K** are expanded so that each new rewrite rule has a *single* operand, allowing for the partial application of function symbols. We model application, the basic engine of reduction in TRS, via the invocation of a method named `app`. The reduction rules of Curryfied CL each apply to (or are 'triggered' by) different 'versions' of the **S** and **K** combinators; in our encoding these rules are implemented by the bodies of five different versions of the `app` method which are each attached to different classes representing the different versions of the **S** and **K** combinators. In order to make our encoding a valid (typeable) program in full Java, we have defined a `Combinator` class containing an `app` method from which all the others inherit, essentially acting as an *interface* to which all encoded versions of **S** and **K** must adhere.

**Definition 22 ([19]).** The encoding of Combinatory Logic (CL) into the $\text{FJ}^{\not\in}$ program OOCL (Object-Oriented Combinatory Logic) is defined using the execution context given in Figure 4 and the function $[\![\cdot]\!]$ which translates terms of CL into $\text{FJ}^{\not\in}$ expressions, and is defined as follows:

$$\begin{aligned} [\![x]\!] &= x \\ [\![t_1\ t_2]\!] &= [\![t_1]\!]\texttt{.app(}[\![t_2]\!]\texttt{)} \\ [\![\mathbf{K}]\!] &= \texttt{new K()} \\ [\![\mathbf{S}]\!] &= \texttt{new S()} \end{aligned}$$

We can show that the reduction behaviour of OOCL mirrors that of CL.

**Theorem 23 ([19]).** If $t_1$, $t_2$ are terms of CL and $t_1 \to^* t_2$, then $[\![t_1]\!] \to^* [\![t_2]\!]$ in OOCL.

Through this encoding - and the results we have shown above - we can achieve a type-based characterisation of all (terminating) computable functions in OO. Since CL is a Turing-complete model of computation, as a side effect we show that $\text{FJ}^{\not\in}$ is Turing-complete. Although we are sure this does not come as a surprise, it is a nice formal property for our calculus to have, and comes easily as a consequence of our encoding.

In addition, our type system can perform the same 'functional' analysis as ITD does for LC and CL. This is illustrated by a *type preservation* result. We present Curry's type system for CL and then show we can give equivalent types to OOCL programs.

**Definition 24 (Curry Type Assignment for CL [19]).** 1. The set of *simple types* (also known as Curry types) is defined by the grammar: $A, B \ ::= \ \varphi \mid A \to B$ .
2. A *basis* $\Gamma$ is a mapping from variables to Curry types, written as a set of statements of the form $x{:}A$ in which each of the variables $x$ is distinct.
3. Simple type assignment to CL-terms is defined by the following system:

$$(Ax): \ \frac{}{\Gamma \vdash_{\text{CL}} x : A}\ (x{:}A \in \Gamma) \qquad (\to E): \ \frac{\Gamma \vdash_{\text{CL}} t_1 : A \to B \quad \Gamma \vdash_{\text{CL}} t_2 : A}{\Gamma \vdash_{\text{CL}} t_1 t_2 : B}$$

$$(\mathbf{K}): \ \frac{}{\Gamma \vdash_{\text{CL}} \mathbf{K} : A \to B \to A} \qquad (\mathbf{S}): \ \frac{}{\Gamma \vdash_{\text{CL}} \mathbf{S} : (A \to B \to C) \to (A \to B) \to A \to C}$$

The elegance of our approach is that we can now link types assigned to combinators to types assignable to object-oriented programs. To show this type preservation result, we need to define what the equivalent of Curry's types are in terms of our $\text{FJ}^{\not\in}$ types. To this end, we define the following translation of Curry types.

**Definition 25 (Type Translation [19]).** The function $\lceil\!\lceil \cdot \rfloor\!\rfloor$, which transforms Curry types[8], is defined as follows:

$$\begin{array}{rcl} \lceil\!\lceil \phi \rfloor\!\rfloor & = & \phi \\ \lceil\!\lceil A \to B \rfloor\!\rfloor & = & \langle \texttt{app}{:}(\lceil\!\lceil A \rfloor\!\rfloor) \to \lceil\!\lceil B \rfloor\!\rfloor \rangle \end{array}$$

It is extended to contexts as follows: $\lceil\!\lceil \Gamma \rfloor\!\rfloor = \{x{:}\lceil\!\lceil A \rfloor\!\rfloor \mid x{:}A \in \Gamma\}$.

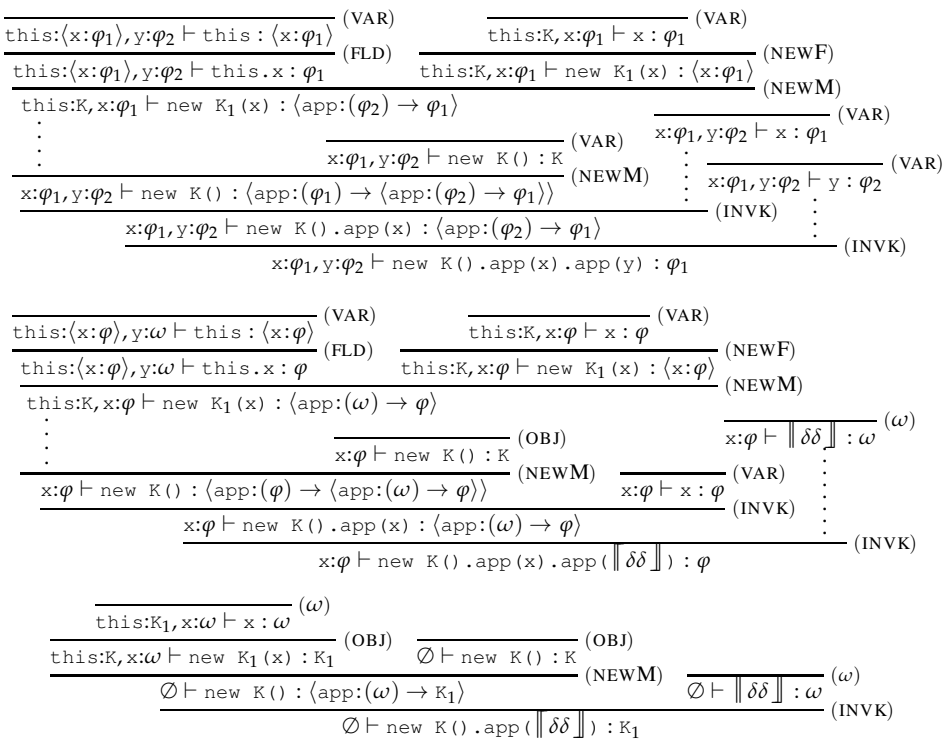
We can now show the type preservation results.

**Theorem 26 (Preservation of Types (cf. [19])).** 1. If $\Gamma \vdash_{\text{CL}} t{:}A$ then $\lceil\!\lceil \Gamma \rfloor\!\rfloor \vdash \llbracket t \rrbracket : \lceil\!\lceil A \rfloor\!\rfloor$
2. Let $\mathcal{E}_{\text{CL}}$ be defined as: $\begin{array}{rcl} \mathcal{E}_{\text{CL}}\,\mathbf{S} & = & \lceil\!\lceil (\varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3 \rfloor\!\rfloor \\ \mathcal{E}_{\text{CL}}\,\mathbf{K} & = & \lceil\!\lceil \varphi_1 \to \varphi_2 \to \varphi_1 \rfloor\!\rfloor \end{array}$
If $\Gamma \vdash_{\text{CL}} t{:}A$ then $\lceil\!\lceil \Gamma \rfloor\!\rfloor ; \mathcal{E}_{\text{CL}} \vdash_{\text{C}} \llbracket t \rrbracket : \lceil\!\lceil A \rfloor\!\rfloor$.

Furthermore, since Curry's well-known translation of the simply typed LC into CL preserves typeability (see [8]), we can also construct a type-preserving encoding of LC into $\text{FJ}^{\not\in}$; it is straightforward to extend this preservation result to full-blown strict intersection types. We stress that this result really demonstrates the validity of our approach.

To conclude this section, we give some example derivations of OOCL programs in our semantic system that illustrate our results; we could, of course, type these expression in our Curry system as well.

[8] Note we have *overloaded* the notation $\lceil\!\lceil \cdot \rfloor\!\rfloor$, which we also use for the translation of CL terms to $\text{FJ}^{\not\in}$ expressions.



where $\delta$ is the CL-term $\mathbf{S}\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})$ – *i.e.* $\delta\delta$ has no head-normal form.

**Fig. 5.** Derivations for Example 27

*Example 27.* Figure 5 shows, respectively, (1) *a derivation typing a strongly normalising expression of* OOCL*; (2) an $\omega$-safe derivation of a normalising (but not strongly normalising) expression of* OOCL*;* and *(3) a non-$\omega$-safe derivation deriving a non-trivial type for a head-normalising (but not normalising)* OOCL *expression,*

## Conclusions & Future Work

We have considered a type-based semantics defined using an intersection type approach for FJ. Our approach constitutes a subtle shift in the philosophy of static analysis for class-based OO: in the traditional (nominal) approach, the programmer specifies the class types that each input to the program (field values and method arguments) should have, on the understanding that the type *checking* system will guarantee that the inputs do indeed have these types.

In the approach suggested by our type system, the programmer is afforded expressive freedom. Thanks the polymorphic character of our types and to type *inference*, which presents the programmer with an 'if-then' input-output analysis of class constructors and method calls, if a programmer wishes to create instances of some particular class (perhaps from a third party) and call its methods in order to utilise some given functionality, then it is then up to them to ensure that they pass appropriate inputs (either field values or method arguments) that guarantee the behaviour they require.

We will reintroduce more features of full Java back into our calculus, to see if our system can accommodate them whilst maintaining the strong theoretical properties that we have shown for the core calculus. For example, similar to $\lambda\mu$ [18], it seems natural to extend our simply typed system to analyse the exception handling features of Java.

## References


1. S. van Bakel. Complete restrictions of the Intersection Type Discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.
2. S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 151(2):385–435, 1995.
3. S. van Bakel. Completeness and Partial Soundness Results for Intersection & Union Typing for $\overline{\lambda}\mu\tilde{\mu}$. *Annals of Pure and Applied Logic*, 161:1400–1430, 2010.
4. S. van Bakel. Strict intersection types for the Lambda Calculus. *ACM Computing Surveys*, 43:20:1–20:49, April 2011.
5. S. van Bakel. Completeness and Soundness results for $\mathcal{X}$ with Intersection and Union Types. *Fundamenta Informaticae*, 121:1–41, 2012.
6. S. van Bakel and U. de'Liguoro. Logical equivalence for subtyping object and recursive types. *Theory of Computing Systems*, 42(3):306–348, 2008.
7. S. van Bakel and M. Fernández. Normalisation Results for Typeable Rewrite Systems. *Information and Computation*, 2(133):73–116, 1997.
8. S. van Bakel and M. Fernández. Normalisation, Approximation, and Semantics for Combinator Systems. *Theoretical Computer Science*, 290:975–1019, 2003.
9. H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.
10. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
11. T. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer. Clean - A Language for Functional Graph Rewriting. In *ICFP'86*, LNCS 274, pp 364–368, 1987.
12. A. Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.
13. M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the $\lambda$-Calculus. *Notre Dame journal of Formal Logic*, 21(4):685–693, 1980.
14. H.B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.
15. A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
16. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
17. E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *PARLE'91*, LNCS 506-II, pp 202–219, 1991.
18. M. Parigot. An algorithmic interpretation of classical natural deduction. *LPAR'92*, LNCS 624, pp. 190-201, 1992.
19. R.N.S. Rowe and S.J. van Bakel. Approximation Semantics and Expressive Predicate Assignment for Object-Oriented Programming. In *TLCA'11*, LNCS 6690, pp 229–244, 2011.
20. C.P. Wadsworth. The relation between computational and denotational properties for Scott's $D_\infty$-models of the lambda-calculus. *SIAM Journal on Computing*, 5:488–521, 1976.