Semantic Types and Approximation for Featherweight Java

(Theoretical Computer Science, 517:34-74, 2014)

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

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK
rnr07@doc.ic.ac.uk svb@doc.ic.ac.uk

Abstract

We consider semantics for the class-based object-oriented calculus Featherweight Java (without casts) based upon *approximation*. We also define an *intersection type assignment system* for this calculus and show that it satisfies *subject reduction* and *expansion*, *i.e.* types are preserved under reduction and its converse. We establish a link between type assignment and the approximation semantics by showing an approximation result, which leads to a sufficient condition for the characterisation of head-normalisation and termination.
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 also show that our system characterises the normalising and strongly normalising terms for this encoding. We thus demonstrate that the great analytic capabilities of intersection types can be applied to the context of class-based object orientation.

keywords: Featherweight Java, Intersection Types, Approximation Semantics, Derivation Reduction, Strong Normalisation

Introduction

In this paper we will study semantics for Featherweight Java (FJ) [48] through both a notion of *intersection type assignment* [31, 32, 21, 7] and of *approximation* [67]. Our types are *functional* (expressing the types of methods, in particular, as functions, and assigned to untyped expressions, as common in functional programming), contain field and method information, and characterise how a typeable object can be accessed by a context in which it is placed. Our type system will be 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 is undecidable.

The notion of type assignment we develop can be seen as a notion of 'flow analysis' in that assignable types express how expressions can interact with a context; as such, the types express run-time behaviour of expressions. On the other hand, our notion of approximation is defined similarly to Wadsworth's notion [67, 68] for the λ-calculus (LC) [28, 20]: masking out computationally active subterms on a *reduction sequence* (all the terms created by the execution of a term) creates a notion of approximation for terms that induces a semantics. We will show that these two approaches lead essentially to the same model by establishing a strong link between typeable terms and their approximants: we will show that every type that can be assigned to a term can be assigned to one of its approximants, and vice versa. We will then explore these results further and fully characterise normalisation and termination of terms through assignable types.

Semantics for object-oriented programming The *object-oriented* (OO) programming paradigm, as exemplified by languages such as C++ [63], Java [45], C# [1], Ruby [43], ECMAscript (or Javascript) [2] and Python [61], has been the subject of extensive theoretical study over the last two decades. OO-languages come in two broad flavours: the *object* (or prototype) based, and the *class* based. A number of formal models has been developed [25, 55, 26, 41, 42, 4, 48] which attempt to distill the many features of OO into a core set of primitive operations. Of these, the ς-calculus [4] and Featherweight Java (FJ) have been well received as elementary models for object based and class-based OO, respectively.

Most of the previous work on semantics for OO dates from quite some time back, but there is some more recent work on denotational semantics for Java. Two major contributions are Abadi and Cardelli's denotational PER model for the ς-calculus [3] and Bruce's semantics mapping his OO-languages to F-bounded second order λ-models [22]. Since both consider the language *explicitly* typed, programs and their types are strongly linked; Abadi and Cardelli used their semantics to show that the type system for the ς-calculus is sound. Bruce used his for the same purpose: he relates the interpretation of programs to that of types by making sure that the interpretation of a term is an element of the interpretation of its type, and also Abadi and Cardelli consider an interpretation of types as well as terms. However, neither of these papers state a result relating the semantic model to reduction. The subtyping relation is also proven to be semantic under this interpretation - *i.e.* if $\sigma \leq \tau$, then $[\![\sigma]\!] \subseteq [\![\tau]\!]$, and this is used to show that well-typed expressions do not correspond to the Error value in the semantic domain.

We believe our work to be the first to define a semantic model for OO that gets related to the model induced by the reduction relation (*i.e.* conversion) - it is certainly the first to study an approximation model of OO.

Other related work includes Cook and Palsberg's denotational treatment of inheritance and method lookup [29, 30]. Reddy [59] also gives a denotational semantics to object-oriented concepts, in which objects are viewed as closures (*i.e.* `let`-bound functions). The main point of this work is to give a more fundamental view of what objects really are, rather than to consider their reduction behaviour - the paper does not consider reduction and its relationship to the semantics at all. A similar semantics is defined for the language SmallTalk by Kamin [51], but differs in that the interpretation of an object is simply a record of its field values; Reddy and Kamin together compared their two semantics and proving them equivalent [52]. Additionally, Castagna [27] has done work on defining an OO-calculus and a denotational PER semantics for it.

Using an alternative approach, semantics for OO has been studied by encoding OO-calculi in various typed λ-calculi. Cardelli, Bruce and Pierce [23] gave a survey of some of the main approaches in this direction, and compare four different encodings. Glew [44] builds on this, and presents a different typed encoding and gives a very comprehensive overview of previous and related encodings. Viswanathan [66] uses an encoding of OO into a λ-calculus in order to study the observational equivalence/full abstraction issue.

More recently, and more immediately relevant to our work, some papers were published that consider denotational semantics for (Featherweight) Java. Studer [64] defined a semantics for Featherweight Java using a model based on Feferman's Explicit Mathematics formalism [40]. Studer mentions that his model is theoretically weaker than other models that have previously been considered (as mentioned above), and his result is that his semantics is adequate with respect to the Java nominal class type system. Alves-Foss [5] has done work on giving a denotational semantics to the full Java language; his system is impressively comprehensive but, as far as we can see, is not used for any kind of analysis - at least not in [5]. Finally, Burt

in his PhD thesis [24] builds a denotational model for a featherweight model of Java with state based on game semantics, via a translation to a PCF-like language.

Intersection types Over the years, many expressive type systems have been defined and investigated for a variety of calculi. Amongst those, the *intersection type discipline* (ITD), first defined for LC, stands out as a powerful system, closed under β-equality and giving 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, termination (*i.e.* strong normalisation) of terms can be characterised by assignable types. Furthermore, intersection-type-based filter models and approximation results show that intersection types describe the semantical behaviour of typeable terms in full. Intersection type systems have also been employed successfully in analyses for dead code elimination [35], strictness analysis [50], and control-flow analysis [19], proving them a versatile framework for reasoning about programs.

Inspired by this expressive power, investigations have taken place into the suitability of intersection type assignment for other computational models: for example, van Bakel and Fernández [14, 15, 16] have studied intersection types in the context of Term Rewriting Systems (TRS) [53, 36] and van Bakel studied them [10, 12] in the context of sequent calculi [33, 17]. In addition, van Bakel and de'Liguoro [13] have developed a system for the ς-calculus, bringing intersection types to the context of OO; the main characteristic of that system is that it sees assignable types as an *execution predicate*, 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.

In this paper we aim to develop denotational semantics for class-based OO; in order to be able to concentrate on the essential difficulties, we focus on Featherweight Java [48], a restriction of Java defined by removing all but the most essential features of the full language; Featherweight Java bears a similar relation to Java as LC does to languages such as ML [54] and Haskell [47]. We will use two approaches, by defining both an approximation based and type-based semantics for FJ; to achieve the latter, we introduce a notion of intersection type assignment. For that notion, we will show that the expected properties of a system based on intersection hold, *i.e.*:

(*subject reduction*): if e has type σ and e reduces to e', then also e' has type σ, and
(*subject expansion*): if e' has type σ and e reduces to e', then also e has type σ.

Approximation The notions of *approximant* and *approximation* were first introduced by Wadsworth in [67] for LC, where they are used in order to better express the relation between equivalence of meaning in Scott's models and the usual notions of conversion and reduction. Wadsworth defines approximation of terms through the replacement of any parts of a term remaining to be evaluated (*i.e.* β-redexes) by $\bot$. Repeatedly applying this process over a reduction sequence starting with M gives a set of approximants, each giving some - in general incomplete - information about the reduction behaviour of M. Once this reduction produces $\lambda x.yN_1\cdots N_n$, all remaining redexes occur in $N_1,\ldots,N_n$, which then in turn will be approximated. Following this approach, [67] defines $\mathcal{A}(M)$ (similar to Definition 2.7 below) as the set of approximants of the λ-term M, which forms a meet semi-lattice. In [68], the connection is established between approximation and semantics, by showing

$$[\![M]\!]_{D_\infty} p \;=\; \bigsqcup\{\, [\![A]\!]_{D_\infty} p \mid A \in \mathcal{A}(M)\,\}.$$

So, essentially, approximants are partially evaluated expressions in which the locations of incomplete evaluation (*i.e.* where reduction *may* still take place) are explicitly marked by the

element $\bot$; thus, they *approximate* the result of computations. Intuitively, an approximant can be seen as a 'snapshot' of a computation, where we focus on that part of the resulting program which will no longer change, which corresponds to the (observable) *output*.

A notion of *approximants* for FJ-programs is defined similarly. This is used to show an *approximation result* which states that, for every intersection type assignable to a term in our system, an approximant of that term exists which can be assigned the same type; for LC, this result was shown by Ronchi della Rocca [60] (see also [7]). Interpreting a FJ-program by its set of approximants gives an *approximation semantics* and the approximation result then relates the approximation and the type-based semantics; it demonstrates that our type system is sound and complete with respect to the approximation semantics, allowing a type-based analysis of termination. As is also the case for LC and TRS, in our system this result is shown using a notion of computability; since the notion of reduction we consider is *weak* (in the sense that methods have a fixed arity, and all arguments need to be present before they can be invoked and are all 'consumed' in one go[1]), the traditional approach to the proof of the approximation result does not work and, as in [16], we need to resort to a proof of the much stronger property that reduction on type derivations is strongly normalising, from which the approximation result follows.

Expressivity 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) [34] – and thereby also LC – into it, thus confirming explicitly that (our variant of) FJ is Turing-complete. To show that our type system provides more than a semantical tool and can be used in practice as well, we define a restriction of our system by restricting to a notion of Curry type assignment and show a type preservation result: types assignable to CL-terms in the Curry system correspond to types in our system that can be assigned to the interpreted CL-terms. This could then easily be extended to the strict intersection type assignment system for LC [6]; combined with the results we show in this paper, this then implies that the collection of typeable OO-expressions correspond to the terms that are typeable using intersection types, *i.e.* all λ-terms that are semantically meaningful (terms having a head-normal form).

Contents of this paper In Section 1, we present the calculus $\mathbf{FJ}^{\not c}$, Featherweight Java without casts, for which in Section 2 we define an approximation semantics. In Section 3, we define our notion of intersection type assignment, and show subject reduction and expansion. In Section 4 we define a notion of reduction on derivations that follows reduction on $\mathbf{FJ}^{\not c}$-expressions, and show that this notion is strongly normalisable. The two approaches of approximation and intersection types are linked in Section 5, where we show the approximation result and show that this is a direct consequence of the strong normalisability of derivation reduction; we also show some characterisation results for head-normalisation and strong normalisation. In Section 6 we present a restriction using Curry types and show how to encode Combinatory Logic into $\mathbf{FJ}^{\not c}$, whilst preserving assignable Curry types. In Section 7, we give some detailed examples and observations, followed by our conclusions.

An extended abstract of this paper has appeared as [62]. In [18] we presented a similar system which here has been simplified. In particular, we have removed the (functional) *field update* feature (which can be modelled using method calls,[2]) which gives a more straightforward presentation of system and proofs. We have also decoupled our intersection type system

[1] This is also the case for reduction in combinator systems, and TRS in general. This differs from, for example, the notion of reduction in calculi like LC, where arguments are 'consumed' one-at-the-time. Also, it differs from the notion of weak reduction in LC, which prohibits reduction under an abstraction.

[2] One possible solution is to add to every class C, for each field f_i belonging to the class, a method `C update_`f_i `(x) { return new C(this.`f_1`,...,x,...,this.`f_n`); }`.

from the existing nominal type system, as was used in [18, 13], which shows that the approximation result does not depend on the class type system in any way. Moreover, we moved away from *late self typing* (where the type for the receiver is checked when invoking the method), which was making the proofs of our results unnecessarily complex, towards *early self typing* (where the type for the receiver is checked when assigning a method type to an object).

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 encapsulating 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 they introduce the possibility of certain run-time errors meaning that they are, in a certain sense, 'unsafe'; for this reason we call our calculus $\text{FJ}^{\not\Leftarrow}$. We discuss the motivations behind this decision more fully in Section 7.3. We also leave constructors[3] as implicit, as they plays no role in the reduction semantics.

Before defining the calculus itself, we introduce some notational conventions that we will use in the remainder of this paper.

Definition 1.1 (Notation) *i*) We use $\overline{n}$ (where n is a natural number) to represent the set $\{1,\ldots,n\}$.

ii) A sequence s of n elements $a_1,\ldots,a_n$ is denoted by $\vec{a}_n$; the subscript can be omitted when the exact number of elements in the sequence is not relevant.

iii) We write $a \in \vec{a}_n$ whenever there exists some $i \in \overline{n}$ such that $a = a_i$.

iv) The empty sequence is denoted by ϵ, and concatenation on sequences by $s_1 \cdot s_2$.

v) 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`).

vi) We use roman teletype font for concrete $\text{FJ}^{\not\Leftarrow}$-code, and italicised teletype font for meta-code.

We distinguish the class name `Object` (which denotes the root of the class inheritance hierarchy in all programs) and treat the self reference `this` (used to refer to the receiver object in method bodies) as a separate syntactic entity rather than a variable[4].

Definition 1.2 ($\text{FJ}^{\not\Leftarrow}$ Syntax) *i*) Assuming countably infinite sets of class, field, method, and variable names (not necessarily disjoint), expressions are defined by the following grammar:

[3] In [48], each class has an explicit constructor which has as many parameters as the fields of the class and explicitly assigns the passed parameters $\vec{e}$ in `new C(`$\vec{e}$`)` to the fields.

[4] Note that `this` is not a variable in the traditional sense, since it is not used to mark the position in the method's body where a parameter can be passed, nor for the position in a term that can be replaced by another term. Were we to define an interpretation of expressions into an appropriate domain, via $\lceil \texttt{e} \rfloor_\xi$, using the valuation ξ that maps variables to arbitrary terms, then the fact that `this` can only be mapped to the *receiver* would need to be treated directly in the definition of $\lceil \texttt{e} \rfloor_\xi$, and cannot be dealt with by ξ; so `this`, formally, is not a variable. However, whenever convenient, we will treat `this` as a variable, so will normally not mention it separately when replacing variables in an expression. Formally, there is no need to stipulate that there is no variable called `this`, although for parsing purposes this may be useful.

$$e ::= x \mid \texttt{this} \mid \texttt{new } C(\vec{e}) \mid e.f \mid e.m(\vec{e})$$

ii) The function VARS returns the set of variables used in an expression (notice that this set does not include `this` even if it occurs in the method body, since in our formalism `this` is not a variable).

iii) An FJ$^{\not\in}$ program P consists of a *class table* $\mathcal{CT}$, 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 following grammar:

$$\begin{aligned} fd &::= C\ f; \\ md &::= D\ m(C_1\ x_1, \ldots, C_n\ x_n)\ \{\texttt{return } e;\} \\ cd &::= \texttt{class } C \texttt{ extends } C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\} \qquad (C \neq \texttt{Object}) \\ \mathcal{CT} &::= \overrightarrow{cd} \\ P &::= (\mathcal{CT}, e) \end{aligned}$$

Thus, class tables are comprised of a number of *class declarations* cd, which themselves contain *field declarations* fd, and *method declarations* md. For a method declaration

$$D\ m(C_1\ x_1, \ldots, C_n\ x_n)\ \{\texttt{return } e;\},$$

we call $D\ m(C_1\ x_1, \ldots, C_n\ x_n)$ the *signature* of the method, and e the method *body*. The variables $x_1, \ldots, x_n$ are called the *formal parameters* of the method.

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), it will be left as an implicit parameter in the definitions that follow. This is done in the interests of readability, and is a standard simplification in the literature (see, *e.g.*, [48]).

As mentioned above, 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 the following well-formedness criteria, which are standard for class-based OO: that there are no cycles in the inheritance hierarchy; that each class is declared only once; that fields in any given branch of the inheritance hierarchy are uniquely named; and that each formal parameter in a method declaration must be unique in that declaration. Two further well-formedness criteria deserve more detailed explanation. Firstly, if there are multiple method declarations containing the same method name in any given branch of the inheritance hierarchy, then each of those declarations must have the same *signature* (modulo renaming of formal parameters). Each such method re-declaration is permitted to have a different method *body*, however. This is known in the parlance of class-based OO as *method override*. Secondly, the formal parameters of a method must constitute a superset of the variables used in the method body, so method definitions correspond to closed functions, thus avoiding dynamic linking issues.

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

Definition 1.3 (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.

i) The following functions retrieve the name of a class or field from its definition:

$$\begin{aligned}\mathcal{CN}(\texttt{class } C \texttt{ extends } D\{\overrightarrow{fd}\ \overrightarrow{md}\}) &= C\\ \mathcal{FN}(C\ f;) &= f\\ \mathcal{MN}(D\ m(C_1\ x_1,\ \ldots,\ C_n\ x_n)\ \{\texttt{return } e;\}) &= m\end{aligned}$$

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

$$\mathcal{CT}(C) = cd \text{ if } \mathcal{CN}(cd) = C \text{ and } cd \in \mathcal{CT}$$

iii) The list of fields belonging to a class C (including those it inherits) is given by the function $\mathcal{F}$, which is defined as follows:

$$\begin{aligned}\mathcal{F}(\texttt{Object}) &= \epsilon\\ \mathcal{F}(C) &= \mathcal{F}(C')\cdot\vec{f}_n \text{ if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C'\ \{\overrightarrow{fd}_n\ \overrightarrow{md}\}\\ &\qquad\qquad \text{and } \mathcal{FN}(fd_i) = f_i \text{ for all } i \in \overline{n}\end{aligned}$$

iv) 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:

$$\begin{aligned}\mathcal{M}b(C,m) &= (\vec{x}_n,e) \quad \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\} \text{ and there exist } C_0,\vec{C}_n\\ &\qquad\qquad \text{such that } C_0\ m(C_1\ x_1,\ldots,C_n\ x_n)\ \{\texttt{return } e;\} \in \overrightarrow{md}\\ \mathcal{M}b(C,m) &= \mathcal{M}b(C',m) \text{ if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\}\\ &\qquad\qquad \text{and } m \neq \mathcal{MN}(md) \text{ for all } md \in \overrightarrow{md}\end{aligned}$$

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, each of the variables is replaced by the corresponding argument, and `this` is replaced by the receiver.

Definition 1.4 (Reduction) *i*) 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^{S} for $\mathsf{S}(e)$.

ii) The reduction relation $\rightarrow$ is the smallest contextually closed relation on expressions satisfying:

$$\begin{aligned}\texttt{new } C(\vec{e}_n).f_i &\rightarrow e_i \quad \text{for class name } C \text{ with } \mathcal{F}(C) = \vec{f}_n \text{ and } i \in \overline{n}.\\ \texttt{new } C(\vec{e}).m(\overrightarrow{e'}_n) &\rightarrow e^{\mathsf{S}} \quad \text{for class name } C \text{ and method } m \text{ with } \mathcal{M}b(C,m) = (\vec{x}_n,e),\\ &\qquad\qquad \text{where } \mathsf{S} = \langle \texttt{this} \mapsto \texttt{new } C(\vec{e}),\ x_1 \mapsto e'_1,\ \ldots,\ x_n \mapsto e'_n \rangle\end{aligned}$$

We call the left-hand term the *redex* (*red*ucible *ex*pression) and the right hand the *contractum*. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$.

This notion of reduction is *confluent*, which is easily shown by a standard 'colouring' argument (as is done in [20] for LC).

The LC view is that all normal forms are meaningful (in a semantic sense). However, note that in our system there are some normal forms which are clearly problematic for this point of view. Take, for example, `new C().m()` with method `m` not existing in class `C`. It seems obvious that this is not an expression which we should treat as meaningful. Indeed, in real Java running such a program would result in a `NoSuchMethodError`. One approach we could have taken would have been to model runtime errors explicitly. Although it would be straightforward to extend the system in this way, for simplicity we chose not to take this approach. Instead, we will consider such normal forms to be not well-formed (see Definition 5.5), and

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

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

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

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

Figure 1: Type assignment rules for the Nominal Type Assignment system.

ensure that they are mapped to the bottom element of our semantic domain in Section 2.

The nominal[5] type system as presented in [48], adapted to our version of Featherweight Java, is defined as follows.

Definition 1.5 (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 to retrieve the types of any given field f or method m declared in a particular class C:

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

$\mathcal{MT}$ is defined similarly:

$$\mathcal{MT}(C, m) = \begin{cases} \vec{E} \to D & \text{if } \mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\} \text{ and } D\ m(\overrightarrow{E\ x})\ \{e\} \in \overrightarrow{md} \\ \mathcal{MT}(C', m) & \text{if } \mathcal{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{fd}\ \overrightarrow{md}\} \\ & \quad \text{and } m \neq \mathcal{MN}(md) \text{ for all } md \in \overrightarrow{md} \end{cases}$$

Notice both are not defined on `Object`.

Nominal type assignment in FJ is a relatively easy affair, and more or less guided by the class hierarchy.

Definition 1.6 (Nominal type assignment for FJ) *i*) The set of expressions of FJ is defined as in Definition 1.2, but adding the alternative `(C)e` (*cast*).

ii) The sub-typing relation[6] $<:$ 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 $C <: D$.

iii) *Statements* are pairs of expression and type, written as $e : \phi$; *contexts* Γ are defined as sets of statements of the shape $x{:}\phi$, where all variables are distinct, and possibly containing a statement for `this`.

iv) Expression type assignment for the nominal system for FJ is defined in [48] through the rules of Figure 1, where (VAR) is applicable to `this` as well.

[5] This notion is called *nominal* since the set of types is taken to be the set of class names in the class table, and compatibility and equivalence of types is determined based on identity of names only; in particular, two class types with different names are incompatible, even if they have identical field and method declarations.

[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.

```
class IntList extends Object {
    IntList square() { return new IntList(); }
    IntList removeMultiplesOf(int n) { return new IntList(); }
    IntList sieve() { return new IntList(); }
    IntList listFrom(int n) { return new NonEmpty(n, this.listFrom(n+1)); }
    IntList primes() { return this.listFrom(2).sieve(); }
}

class NonEmpty extends IntList {
    int val;
    IntList next;
    IntList square() { return new NonEmpty(this.val * this.val, this.next.square()); }
    IntList removeMultiplesOf(int n) {
        if (this.val % n == 0) {
            return this.next.removeMultiplesOf(n);
        } else {
            return new NonEmpty(this.val, this.next.removeMultiplesOf(n));
        }
    }
    IntList sieve() {
        return new NonEmpty(this.val, this.next.removeMultiplesOf(this.val).sieve(); );
    }
}
```

Figure 2: The class table for the Sieve of Eratosthenes in $\mathbf{FJ}^{\not\in}$

v) 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.

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

vi) 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.

$$(\textsc{class}): \frac{md_i\ \text{OK IN}\ C \quad (\forall i \in \overline{n})}{\texttt{class}\ C\ \texttt{extends}\ D\{\overrightarrow{fd};\ \overrightarrow{md_n}\}\ \text{OK}} \qquad (\textsc{prog}): \frac{\overrightarrow{cd\ \text{OK}} \quad \Gamma \vdash e : C}{(\overrightarrow{cd}, e)\ \text{OK}}$$

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. We will see below (Definition 3.4) that this is *not* the case for our notion of intersection type assignment; rather than typing classes, it has two rules (newF) and (newM) that create a field or method type for an object (essentially stating that this field or method is available, and what its current type is). Using that approach, method bodies are typed *every time* the context requires that an object has a specific method type, and the various types constructed for a method that are used throughout a program need not be the same.

As mentioned above, we have decided to not consider casts in this paper, since they create run-time problems, as already observed in [48].

2 An Approximation Semantics for $\mathbf{FJ}^{\not\in}$

In this section, we define a notion of *approximation* for $\mathbf{FJ}^{\not\in}$, as a generalisation of a similar notion first introduced by Wadsworth in [67] for LC, which we will use to define an *approximation semantics* for $\mathbf{FJ}^{\not\in}$. Essentially, approximants are partially evaluated expressions in which the locations of incomplete evaluation (*i.e.* where reduction *may* still take place) are explicitly marked by the element $\bot$; thus, they *approximate* the result of computations; intuitively, an approximant can be seen as a 'snapshot' of a computation, where we focus on that part of the resulting program which will no longer change.

We first illustrate this concept.

Example 2.1 Consider $\mathbf{FJ}^{\not\in}$ extended with numerals, arithmetic operators, and an if-then-else construct, and take the class table given in Figure 2. Let the notation $n_1{:}n_2{:}\ldots{:}n_k{:}[]$ be shorthand for the $\mathbf{FJ}^{\not\in}$ expression:

$$\texttt{new NonEmpty(}n_1\texttt{, new NonEmpty(}n_2\texttt{,}\ldots\texttt{ new NonEmpty(}n_k\texttt{, new IntList())...))}$$

Then	*which has the approximant*
$\texttt{(1:2:3:[]).square()}$	$\bot$
$\rightarrow^* \texttt{1:(2:3:[]).square()}$	$\texttt{1:}\bot$
$\rightarrow^* \texttt{1:4:(3:[]).square()}$	$\texttt{1:4:}\bot$
$\rightarrow^* \texttt{1:4:9:([]).square()}$	$\texttt{1:4:9:}\bot$
$\rightarrow^* \texttt{1:4:9:[]}$	$\texttt{1:4:9:[]}$

In this case, the output is finite, and the final approximant is the end-result itself. The class table in Figure 2 is also able to calculate the (infinite) list of prime numbers using the well known 'sieve of Eratosthenes'.

Then (where we abbreviate `removeMultiplesOf` *by* `rMO`*)*	*which has the approximant*
$\texttt{new IntList().primes()}$	$\bot$
$\rightarrow^* \texttt{(2:3:4:5:6:7:8:...).sieve()}$	$\bot$
$\rightarrow^* \texttt{2:(3:(4:5:6:7:8:...).rMO(2)).sieve()}$	$\texttt{2:}\bot$
$\rightarrow^* \texttt{2:3:(((5:6:7:8:...).rMO(2)).rMO(3)).sieve()}$	$\texttt{2:3:}\bot$
$\rightarrow^* \texttt{2:3:5:((((7:8:...).rMO(2)).rMO(3)).rMO(5)).sieve()}$	$\texttt{2:3:5:}\bot$
$\vdots$	$\vdots$

In this case, the computation is infinite, and so is the output - there is no final approximant since the 'result' is never reached and thus $\bot$ is in every approximant.

Notice that, under reduction, more and more information about the structure of the end result of the computation is revealed.

Approximate expressions and approximate normal forms for $\mathbf{FJ}^{\not\in}$ are defined below.

Definition 2.2 (Approximate Expressions) *i*) The set $\mathcal{A}$ of *approximate* $\mathbf{FJ}^{\not\in}$ expressions is defined, essentially adding $\bot$ as an expression, by the grammar:

$$a \;::=\; \bot \mid x \mid \texttt{this} \mid a.f \mid a.m(\vec{a}_n) \mid \texttt{new } C(\vec{a}_n) \quad (n \geq 0)$$

ii) The set of *approximate normal forms* (*apn* for short), $\mathcal{A}$, ranged over by A, is a strict subset of the set of approximate expressions and is defined by the following grammar:

$$\begin{aligned} A \;::=\; & \bot \mid x \mid \texttt{this} \mid \texttt{new } C(\vec{A}_n) \\ & \mid A.f \mid A.m(\vec{A}_n) \qquad (A \neq \bot, A \neq \texttt{new } C(\vec{A}_n)) \end{aligned}$$

The notion of approximation is formalised through an approximation relation on approximate expressions.

Definition 2.3 (Approximation Relation) The *approximation relation* $\sqsubseteq \subseteq \mathcal{A}^2$ is defined as the smallest preorder satisfying:

$$\bot \;\sqsubseteq\; a$$

$$a \sqsubseteq a' \;\&\; \forall i \in \overline{n}\ [a_i \sqsubseteq a'_i] \;\Rightarrow\; \begin{cases} a.f & \sqsubseteq\; a'.f \\ \texttt{new } C(\vec{a}_n) & \sqsubseteq\; \texttt{new } C(\vec{a'}_n) \\ a.m(\vec{a}_n) & \sqsubseteq\; a'.m(\vec{a'}_n) \end{cases}$$

If $a \sqsubseteq e$, we call a a *direct approximant* of e.

As mentioned above, the idea behind approximation is to cover up incomplete evaluation with the element $\bot$. Thus, for example, if the expression `new C(e)` can reduce to `new C(e′)` via a reduction in the subexpression e, then we may cover this reduction with $\bot$, obtaining `new C(`$\bot$`)` $\sqsubseteq$ `new C(e)`.

The other crucial aspect that we require of approximants is that they represent information about the result of a computation that *cannot change* through further reduction. It is for this purpose that we have defined approximate normal forms. Notice that we do *not* consider $\bot.f$ or $\bot.m(\vec{A}_n)$ to be *apn*s: for such expressions it can be that $\bot$ hides an expression that reduces to an object `new C(`$\vec{A}_n$`)`, in which case the field or method invocation can run and thereby disappears. Moreover, if in the *apn* $A[\bot]$ the bottom gets replaced by e, an expression is created that can possibly reduce but *only* inside the subexpression e, creating $A[e']$, thus maintaining the outer shape $A[\cdot]$.

This is expressed by the following result, which characterises the relationship between the approximation relation and reduction.

Lemma 2.4 If $A \sqsubseteq e$ *and* $e \to^* e'$, *then* $A \sqsubseteq e'$.

Proof: By induction on the length of reduction sequences; we only show the base case, which gets shown by induction on the structure of *apn*s, of which we show only one illuminating case.

$(A = A'.m(\vec{A}_n))$: Then $e = e_0.m(\vec{e}_n)$ with $A' \sqsubseteq e_0$ and $A_i \sqsubseteq e_i$ for each $i \in \overline{n}$. Since $A' \neq$ `new C(`$\vec{A}$`)` it follows that $e_0 \neq$ `new C(`$\vec{e'}$`)`. Since e is not a redex, there are only two possibilities for the reduction step:

a) $e_0 \to e'_0$ and $e' = e'_0.m(\vec{e}_n)$. Then by induction $A' \sqsubseteq e'_0$ and so also $A'.m(\vec{A}_n) \sqsubseteq e'_0.m(\vec{e}_n)$.

b) $e_j \to e'_j$ for some $j \in \overline{n}$ and $e' = e_0.m(\vec{e'}_n)$ with $e'_k = e_k$ for each $k \in \overline{n}$ such that $k \neq j$. Then, clearly $A_k \sqsubseteq e'_k$ for each $k \in \overline{n}$ such that $k \neq j$. Also, by induction $A_j \sqsubseteq e'_j$. Thus $A'.m(\vec{A}_n) \sqsubseteq e_0.m(\vec{e'}_n)$. □

As desired, this property expresses that the observable behaviour of a program can only *increase* (in terms of $\sqsubseteq$) through reduction, corresponding to the idea that while running a program we discover more about its result. For $A \sqsubseteq e$, the *apn* A corresponds to that part of the result that will no longer change during reduction.

Notice that while we have called $\mathcal{A}$ the set of approximate *normal forms*, as per the discussion of the previous section they do not correspond exactly to the set of normal forms with respect to reduction. As pointed out above, the expression `new C().m()`, with method `m` not existing in class `C`, is a normal form but is not a well-formed one; thus, we exclude it as an *apn*. Despite this, we have chosen to name the members of $\mathcal{A}$ approximate normal forms in order to draw an explicit parallel between our notion of approximants, and that of other systems (namely LC and TRS). In the LC for example, the reduction relation can be extended with the rules $\bot M \to \bot$ and $\lambda x.\bot \to \bot$. With respect to this extended reduction relation, the syntactically defined approximate normal forms are precisely the terms which cannot be further reduced.

We also define a *join* operation on approximate expressions, which will be needed to prove the approximation result of Section 5.

Definition 2.5 (JOIN OPERATION) The *join* operation $\sqcup$ on approximate expressions is a partial operator defined as the reflexive and contextual closure of: $\bot \sqcup a = a \sqcup \bot = a$. We extend the join operation to sequences of approximate expressions by: $\sqcup \epsilon = \bot$ and $\sqcup\, a \cdot \vec{a}_n = a \sqcup (\sqcup\, \vec{a}_n)$.

Notice that the join of two approximate expressions is not always defined.

The following lemma shows that $\sqcup$, if defined, acts as an upper bound on approximate

expressions, and that it is closed over *apn*s in that the join of two *apn*s, if defined, is itself an *apn*.

Lemma 2.6 *i) Let* a_1, a_2 *and* a_3 *be approximate expressions, then*

$$\begin{array}{rcl} a_1 \sqsubseteq a_3 \;\&\; a_2 \sqsubseteq a_3 & \Rightarrow & a_1 \sqcup a_2 \sqsubseteq a_3 \;\&\; a_1 \sqsubseteq a_1 \sqcup a_2 \;\&\; a_2 \sqsubseteq a_1 \sqcup a_2 \\ (a_1 \sqcup a_2) \sqcup a_3 & = & a_1 \sqcup (a_2 \sqcup a_3) \\ a_1 \sqcup a_2 & = & a_2 \sqcup a_1 \end{array}$$

ii) $A_1 \sqcup A_2 \in \mathcal{A}$ *(when defined).*

Proof: *i*) By induction on the structure of approximate expressions; we show a more illustrating case.

$(a_1 = a'_1.f,\ a_2 = a'_2.f,\ a'_1 \sqsubseteq a',\ a'_2 \sqsubseteq a')$: By induction, $a'_1 \sqcup a'_2 \sqsubseteq a'$, $a'_1 \sqsubseteq a'_1 \sqcup a'_2$, and $a'_2 \sqsubseteq a'_1 \sqcup a'_2$. Then, by Definition 2.3, $(a'_1 \sqcup a'_2).f \sqsubseteq a'.f$, $a'_1.f \sqsubseteq (a'_1 \sqcup a'_2).f$, and $a'_2.f \sqsubseteq (a'_1 \sqcup a'_2).f$. Then, by Definition 2.5, $a_1 \sqcup a_2 = (a'_1 \sqcup a'_2).f$.

ii) By induction on the structure of *apn*s; again, we only show one case.

$(A_1 = A'_1.f,\ A_2 = A'_2.f,\ A'_1 \sqsubseteq A',\ A'_2 \sqsubseteq A')$: By definition $A'_1 \in \mathcal{A}$ and $A'_2 \in \mathcal{A}$, with both A'_1 and A'_2 being neither $\bot$, nor of the form $\texttt{new}\ C(\overrightarrow{A''})$. Then by induction $A'_1 \sqcup A'_2 \in \mathcal{A}$, and by Definition 2.5 the join is neither equal to $\bot$ nor of the form $\texttt{new}\ C(\overrightarrow{A''}_n)$. Thus, by Definition 2.3, $(A'_1 \sqcup A'_2).f = A_1 \sqcup A_2 \in \mathcal{A}$. □

Notice that, in particular, the first part shows that if $a_1 \sqsubseteq e \;\&\; a_2 \sqsubseteq e$, then $a_1 \sqcup a_2 \sqsubseteq e$.

We now define the set of approximants of a term.

Definition 2.7 (Approximants) The symbol $\mathcal{A}$ also is used for a function that returns the set of *approximants* of an expression e and is defined by:

$$\mathcal{A}(e) \;=\; \{A \mid \exists e' \,[e \to^* e' \;\&\; A \sqsubseteq e']\}$$

Thus, an approximant of some expression e is an *apn* that approximates some (intermediate) stage of execution of e.

We will now show that $\mathcal{A}(\cdot)$ induces an *approximation semantics* in that it equates pairs of expressions that are in the reduction relation, as shown by the following theorem.

Theorem 2.8 *Let* $e_1 \to^* e_2$*; then* $\mathcal{A}(e_1) = \mathcal{A}(e_2)$.

Proof: $(\supseteq)$:

$$\begin{array}{lll} e_1 \to^* e_2 \;\&\; A \in \mathcal{A}(e_2) & \Rightarrow & \text{(Definition 2.7)} \\ e_1 \to^* e_2 \;\&\; \exists e_3\,[e_2 \to^* e_3 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \\ \exists e_3\,[e_1 \to^* e_3 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \text{(Definition 2.7)} \\ A \in \mathcal{A}(e_1) & & \end{array}$$

$(\subseteq)$:

$$\begin{array}{lll} e_1 \to^* e_2 \;\&\; A \in \mathcal{A}(e_1) & \Rightarrow & \text{(Definition 2.7)} \\ e_1 \to^* e_2 \;\&\; \exists e_3\,[e_1 \to^* e_3 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \text{(Church-Rosser)} \\ \exists e_3, e_4\,[e_1 \to^* e_2 \;\&\; e_2 \to^* e_4 \;\&\; e_1 \to^* e_3 \;\&\; e_3 \to^* e_4 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \text{(Lemma 2.4)} \\ \exists e_4\,[e_2 \to^* e_4 \;\&\; A \sqsubseteq e_4] & \Rightarrow & \text{(Definition 2.7)} \\ A \in \mathcal{A}(e_2) & & \end{array}$$

□

Since this result states that terms that are related through reduction have the same interpretation, we can even reverse the reduction order; this allows us to define a semantics for $\text{FJ}^{\not\in}$ by interpreting expressions by the set of their approximants:

Definition 2.9 (Approximation Semantics) The *approximation model* for $\text{FJ}^{\not\in}$ expressions (given a class table) is a structure $\langle \wp(\mathcal{A}), [\![\cdot]\!]_{\mathcal{A}} \rangle$, where $[\![e]\!]_{\mathcal{A}} = \mathcal{A}(e)$.

That this indeed gives a semantics follows from Theorem 2.8; notice that an abstract notion of model for $\mathbf{FJ}^{\not\in}$ does not exist (as it does for LC), so we have no other means to verify that $\langle \wp(\mathcal{A}), [\![\cdot]\!]_{\mathcal{A}} \rangle$ does indeed give a model.

Before moving on to describe our type assignment system and its relationship to the semantics we have just defined, we will make one final point concerning our treatment non-well-formed normal forms such as `new C().m()`, where method `m` does not exist in class `C`. We have explained above why we consider such normal forms to be meaningless, even though we have chosen not to reflect this in the reduction system. Notice that the only *apn* which approximates this expression is $\bot$ and thus its semantic denotation is the set $\{\bot\}$, the bottom element of the semantic domain. Of course, it is exactly these kinds of results that the nominal type system of Definition 1.6 rejects. This might give the impression that we will implicitly only be considering those expressions which are nominally well-typed, however this is not the case. The type system which we consider in the remainder of this paper assigns types to *all* expressions. Note that there are programs which are rejected by the nominal type system but which nevertheless have meaningful results and thus *are* typeable in our semantic system. We examine in detail an example of such a program in Section 7.3.

3 Semantic Type Assignment

Having defined a semantics for $\mathbf{FJ}^{\not\in}$, we continue by considering a type system for $\mathbf{FJ}^{\not\in}$ which is sound and complete with respect to this semantics in the sense that every type assignable to an expression is also assignable to an approximant of that expression and vice-versa. Notice that, since in approximants redexes are replaced by $\bot$, this result is not an immediate consequence of a subject reduction result; moreover, as we will see in the next section, it is the type derivation itself which determines the approximant in question.

The type assignment system defined below follows in the *intersection type discipline*; it is influenced by the predicate system for the ς-calculus [13], and is ultimately based upon the strict intersection type system for LC [6, 7] (see [11] for a survey). Our types can be seen as describing the capabilities of an expression (or rather, the object to which that expression evaluates) in terms of *i) the operations that may be performed on it (*i.e. *accessing a field or invoking a method),* and *ii) 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 3.1 (FUNCTIONAL TYPES) The set of *functional intersection types* (or *types* for short), ranged over by ϕ, ψ, and its subset of *strict* types, ranged over by σ,τ are defined by the following grammar (where φ ranges over a denumerable set of *type variables*, C ranges over the set of class names, and ω is a type constant):

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

We call $\langle f{:}\sigma \rangle$ a *field type* and $\langle m{:}(\phi_1,\ldots,\phi_n) \to \sigma \rangle$ a *method type*, and, in these, f and m are *labels*; labels are ranged over by ℓ.

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

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

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

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

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

$$(\textsc{join}):\ \frac{\Pi \vdash e : \sigma_1 \ \ldots\ \Pi \vdash e : \sigma_n}{\Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n}\ (n \geq 2) \qquad (\omega):\ \frac{}{\Pi \vdash e : \omega}$$

Figure 3: Type assignment rules for the Functional Type Assignment system.

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 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 σ_i are intersection free, and all labels ℓ_i are distinct; in other words, records are intersection types, but not vice-versa; see also Definition 6.1.

In the language of intersection type systems, our types are *strict* in the sense of [7], since they must describe the outcome of performing an operation in terms of a(nother) *single* operation rather than an intersection. 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 ω 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 3.2 (Subtype Relation) The subtype relation $\leqslant$ is induced by the fact that an intersection type is smaller than each of its components, and is defined is the smallest preorder satisfying:

$$\begin{array}{rcll} \phi & \leqslant & \omega & \text{for all } \phi \\ \phi\cap\psi & \leqslant & \phi & \\ \phi\cap\psi & \leqslant & \psi & \\ \phi \leqslant \psi \ \&\ \phi \leqslant \psi' & \Rightarrow & \phi \leqslant \psi\cap\psi' & \end{array}$$

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

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

Note that $\phi\cap\omega \sim \phi$.

We will consider types modulo $\sim$; in particular, all types in an intersection are different and ω 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 σ_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 ϕ_1 when $n = 1$, and ω when $n = 0$.

Definition 3.3 (Type Environments) *i*) A *type statement* is of the form $e : \phi$, where e is called the *subject* of the statement.

ii) An environment Π is a set of type statements with variables (and possibly `this`) as subjects, and with subjects pairwise distinct; for ease of notation, we will let x range over `this` as well as variables in type statements of the form $x{:}\phi$. $\Pi, x{:}\phi$ stands for the environment $\Pi \cup \{x{:}\phi\}$ (so then either x does not appear in Π or $x{:}\phi \in \Pi$) and $x{:}\phi$ stands for $\emptyset, x{:}\phi$.

iii) We extend $\leqslant$ to environments by: $\Pi' \leqslant \Pi \;\Leftrightarrow\; \forall x{:}\phi \in \Pi \; \exists \phi' \leqslant \phi \, [x{:}\phi' \in \Pi']$.

iv) If $\overrightarrow{\Pi}_n$ is a sequence of environments, then $\bigcap \overrightarrow{\Pi}_n$ is the environment 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 environments that have x as subject.

We will now define our notion of type assignment, which is a slight variant of the system defined in [18].

Definition 3.4 (Functional Type Assignment) Functional type assignment for $\mathbf{FJ}^{\not\in}$ is defined by the natural deduction system of Figure 3.

We will give extended examples for our system in Section 7. For now, we can make the following observations on the type assignment rules:

- Rule (newM) expresses that we consider `new C(`$\vec{e}$`)` typeable with $\langle m{:}(\vec{\phi}_n) \to \sigma \rangle$ only if m's method body e_b (in `C`) can be typed with σ, where the type used for each variable x_i is exactly ϕ_i, and assuming that the expression `new C(`$\vec{e}$`)` *itself* is typeable with the type ψ needed for `this` when typing e_b. Notice that this is required in order to be able to show subject reduction; moreover, it introduces a kind of 'recursion' into our notion of type assignment: in order to type `new C(`$\vec{e}$`)`, we need first to type `new C(`$\vec{e}$`)`, a fact we will investigate in Section 7.2. Notice that, for typeable method bodies, this means that, eventually, we end up not needing a type for `this` (for example, when it does not occur, or occurs in a subexpression typed using rule (ω)), or we only need to know that it has type `C`.
- Rule (newF) expresses that the same expression `new C(`$\vec{e}$`)` can be typed with $\langle f_i{:}\sigma \rangle$, provided we can type the expression e_i with type σ; we demand that all other expressions are typeable as well (their types are not relevant) mainly to be able to prove Theorem 5.9.
- Rule (obj) states that `C` is a type for `new C(`$\vec{e}$`)` as well. Crucially, these three rules ensure that the correct number of arguments are provided for the constructor.
- Rule (invk) expresses that, if an expression `e` has a method type, then that method can be invoked on `e`, provided the arguments have the correct demanded types. Similar for rule (fld).
- Rule (join) allows us to group several types in an intersection, and rule (ω) says that every expression has type ω; this rule is used whenever the type of an expression is not relevant and can be ignored as far as type assignment is concerned.

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, (fld) and (invk) are analogous to

$(\to E)$; (newF) and (newM) are a form of $(\to I)$; and (obj) can be seen as a universal (ω)-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 (newM), which derives a type for an object that presents an analysis of a method that is invokable on that object. Note that the analysis involves typing the body of the method, 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* as well as the formal method parameters, 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*, whereas with *late self typing* (as used in [13]) 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 ω or C, and thus ignored) will be typeable in the system.

We will accept

$$(\text{newM}') : \frac{x_1{:}\phi_1,\ \ldots,\ x_n{:}\phi_n \vdash e_b : \sigma \quad \Pi \vdash e_1 : \phi'_1 \ \ldots\ \Pi \vdash e_n : \phi'_n}{\Pi \vdash \texttt{new}\ C(\vec{e}) : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle}\ (\texttt{this not in } e_b, \mathcal{M}b(C,m) = (\vec{x}_n, e_b),\ n \geq 0)$$

as a variant of rule (newM), since this rule is admissible:

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

The type assignment rules in fact operate on the larger set of approximate expressions, but we abuse notation slightly and use the meta-variable e for expressions rather than a. Note that there is no special rule for typing $\bot$, meaning that if $\bot$ appears in a term, then some part of that term, containing that $\bot$, is typed with ω.

We should perhaps emphasise that, as remarked above, we *explicitly do not type classes*; instead, the rules (newF) and (newM) create a field or method type for an object. This entails that 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; they have to be in the nominal system.

Example 3.5 Take the FJ$^{\not\in}$ program

```
class List
{
    List cons(Object o) { return new NonEmptyList(o, this); }
    List append(Object o) { return new NonEmptyList(o, new EmptyList()); }
}

class EmptyList extends List { }

class NonEmptyList extends List
{
    Object head;
    List tail;
    List append(Object o) {
        return new NonEmptyList(this.head, this.tail.append(o)); }
}
```

We can assign `new NonEmptyList()` any of the type schemes

- ω,
- `NonEmptyList`,
- $\langle \texttt{cons} : \phi \to \texttt{NonEmptyList} \rangle$,
- $\langle \texttt{cons} : \phi_1 \to \langle \texttt{cons} : \phi_2 \to \texttt{NonEmptyList} \rangle \rangle$, ...
- $\langle \texttt{append} : \phi \to \texttt{NonEmptyList} \rangle$,
- $\langle \texttt{append} : \phi_1 \to \langle \texttt{append} : \phi_2 \to \texttt{NonEmptyList} \rangle \rangle$, etc.

We can even assign it any 'combination' of these types, like for example

$$\langle \texttt{cons} : \phi_1 \to \langle \texttt{append} : \phi_2 \to \langle \texttt{cons} : \phi_3 \to \texttt{NonEmptyList} \rangle \rangle \rangle$$

So, in our system, we would have, in principle, an infinite type for each class,[7] which we cannot establish when typing the class separately; rather, we let the *context* of each object declaration (of the shape `new C(`$\vec{e}$`)`) decide which type is needed, so the type for an occurrence of `new C(`$\vec{e}$`)` is 'constructed' by need, and not from a complete analysis of the class.

As is standard for intersection type assignment systems, our system is set up to satisfy both subject reduction *and* subject expansion, which we will show below. First we show:

Lemma 3.6 (Weakening) *Let $\Pi' \trianglelefteq \Pi$ and $\phi \trianglelefteq \psi$; then $\Pi \vdash e : \phi \Rightarrow \Pi' \vdash e : \psi$.*

Proof: By easy induction on the structure of derivations. The base case of (ω) follows immediately, and for (var) it follows by transitivity of the subtype relation. □

The next result forms the basis for the proof of Theorem 3.8; notice that, for brevity, we treat `this` as a variable here, which need not appear amongst the $\vec{x}$.

Lemma 3.7 (Replacement and Extraction) *i) If $\overrightarrow{x{:}\phi}_n \vdash e : \phi$ and there exists Π and $\vec{e}_n$ such that $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$, then $\Pi \vdash e^{\mathsf{S}} : \phi$ where $\mathsf{S} = \langle \overrightarrow{x \mapsto e}_n \rangle$.*

ii) For an expression e and term substitution $\mathsf{S} = \langle \overrightarrow{x \mapsto e}_n \rangle$ with $\textsc{vars}(e) \subseteq \{\vec{x}\}$, if $\Pi \vdash e^{\mathsf{S}} : \phi$, then there are $\vec{\phi}_n$ such that $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$ and $\overrightarrow{x{:}\phi}_n \vdash e : \phi$.

Proof: By induction on the structure of derivations; we show only one case for the second part:

$((\textsc{newM}))$: Then $e^{\mathsf{S}} = \texttt{new C}(\overrightarrow{e'}_{n'})$ and $\phi = \langle m{:}(\overrightarrow{\phi'}_{n'}) \to \sigma \rangle$ for some m, $\overrightarrow{\phi'}_{n'}$ and σ; also, there are e_b and $\overrightarrow{x'}_{n'}$ such that $\mathcal{M}b(C,m) = (\overrightarrow{x'}_{n'}, e_b)$. Without loss of generality, assume that `this` appears in e_b, then there exists some ψ such that $\texttt{this}{:}\psi, x'_1{:}\phi'_1, \ldots, x'_{n'}{:}\phi'_{n'} \vdash e_b : \sigma$ and $\Pi \vdash \texttt{new C}(\overrightarrow{e'}_{n'}) : \psi$ - that is $\Pi \vdash e^{\mathsf{S}} : \psi$. Then by induction, there exists some $\vec{\phi}_n$ such that $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$, and $x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e : \psi$. Now, there are two cases to consider for e:

- $(e = \texttt{new C}(\overrightarrow{e''}_{n'}))$: then we have $x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash \texttt{new C}(\overrightarrow{e''}_{n'}) : \psi$ and by rule $(\textsc{newM})$ it follows that $x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash \texttt{new C}(\overrightarrow{e''}_{n'}) : \langle m{:}(\overrightarrow{\phi'}_{n'}) \to \sigma \rangle$; that is $x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e : \phi$.
- $(e = x_j$ for some $j \in \overline{n})$: then $e_j = \texttt{new C}(\overrightarrow{e'}_{n'})$, and so we have $x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash x_j : \psi$. From rules $(\textsc{join})$ and (var) it follows that $\phi_j \trianglelefteq \psi$. Since $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$, it follows that $\Pi \vdash \texttt{new C}(\overrightarrow{e'}_{n'}) : \phi_j$ and then by Lemma 3.6 that $\Pi \vdash \texttt{new C}(\overrightarrow{e'}_{n'}) : \psi$. From this and rule $(\textsc{newM})$ we then have that $\Pi \vdash \texttt{new C}(\overrightarrow{e'}_{n'}) : \langle m{:}(\overrightarrow{\phi'}_{n'}) \to \sigma \rangle$; that

[7] This has the flavour of polymorphism, but is in fact more general: it is, for example, not possible to define a finite principal pair for each typeable term.

is $\Pi \vdash e_j : \langle m{:}(\overrightarrow{\phi'}_{n'}) \to \sigma \rangle$. Now take $\overrightarrow{\phi''}_n$ such that $\phi''_j = \langle m{:}(\overrightarrow{\phi'}_{n'}) \to \sigma \rangle$ and $\phi''_k = \phi_k$ for each $k \in \overline{n}$ such that $k \neq j$. Notice that by rule (VAR) we have $x_1{:}\phi''_1, \ldots, x_n{:}\phi''_n \vdash x_j : \langle m{:}(\overrightarrow{\phi'}_{n'}) \to \sigma \rangle$; that is $x_1{:}\phi''_1, \ldots, x_n{:}\phi''_n \vdash e : \phi$. □

We can now show that type assignment is closed under reduction as well as under expansion.

Theorem 3.8 (SUBJECT REDUCTION AND EXPANSION) *Let $e \to e'$; then $\Pi \vdash e : \phi$ if, and only if, $\Pi \vdash e' : \phi$.*

Proof: By induction on the definition of reduction. We show the cases for the two kinds of redex (the inductive cases are easy) and only for ϕ is strict; when $\phi = \omega$ the result follows immediately since we can always type both e and e' using the (ω) rule, and when ϕ is an intersection we can reason that the result holds for each strict type in the intersection, and then apply the (JOIN) rule.

$(\mathcal{F}(C) = \vec{f}_n \Rightarrow \texttt{new } C(\vec{e}_n).f_j \to e_j,\ j \in \overline{n})$:

- (*if*): Assume $\Pi \vdash \texttt{new } C(\vec{e}_n).f_j : \sigma$. The last rule applied must be (FLD) so $\Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f_j{:}\sigma \rangle$. This in turn must have been derived using the (NEWF) rule and so there are $\phi_1, \ldots, \phi_n$ such that $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. Furthermore, $\sigma \leqslant \phi_j$ and so it must be that $\phi_j = \sigma$. Therefore $\Pi \vdash e_j : \sigma$.
- (*only if*): Assume $\Pi \vdash e_j : \sigma$. Notice that using (ω) we can derive $\Pi \vdash e_i : \omega$ for each $i \in \overline{n}$ such that $i \neq j$. Then, using the (NEWF) rule, we can derive $\Pi \vdash \texttt{new } C(\vec{e}_n) : \langle f_j{:}\sigma \rangle$ and by (FLD) also $\Pi \vdash \texttt{new } C(\vec{e}_n).f_j : \sigma$.

$(\mathcal{M}b(C,m) = (\vec{x}_n, e_b) \Rightarrow \texttt{new}\, C(\overrightarrow{e'}).m(\vec{e}_n) \to e_b{}^{\mathsf{S}},\ \mathsf{S} = \langle \texttt{this} \mapsto \texttt{new}\, C(\overrightarrow{e'}), \overrightarrow{x_i \mapsto e_i} \rangle)$:

- (*if*): Assume $\Pi \vdash \texttt{new } C(\overrightarrow{e'}).m(\vec{e}_n) : \sigma$. The last rule applied must be (INVK), so there is $\vec{\phi}_n$ such that $\Pi \vdash \texttt{new } C(\overrightarrow{e'}) : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle$ and $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. Furthermore, the last rule applied in the derivation of $\Pi \vdash \texttt{new } C(\overrightarrow{e'}) : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle$ must be (NEWM) and so there is some type ψ such that $\Pi \vdash \texttt{new } C(\overrightarrow{e'}) : \psi$ and $\Pi' \vdash e_b : \sigma$ where $\Pi' = \texttt{this}{:}\psi, x_1{:}\phi_i, \ldots, x_n{:}\phi_n$. Then $\Pi \vdash e_b{}^{\mathsf{S}} : \sigma$ by Lemma 3.7((*i*)).
- (*only if*): Assume that $\Pi \vdash e_b{}^{\mathsf{S}} : \sigma$. Then by Lemma 3.7((*ii*)) there is ψ, $\vec{\phi}_n$ such that $\Pi' \vdash e_b : \sigma$ where $\Pi' = \texttt{this}{:}\psi, x_1{:}\phi_i, \ldots, x_n{:}\phi_n$ with $\Pi \vdash \texttt{new } C(\overrightarrow{e'}) : \psi$ and $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. By the (NEWM) rule we can then derive $\Pi \vdash \texttt{new } C(\overrightarrow{e'}) : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle$, and by applying (INVK) rule that $\Pi \vdash \texttt{new } C(\overrightarrow{e'}).m(\vec{e}_n) : \sigma$. □

Notice that, as usual, computational equality between expressions in $\text{FJ}^{\not\in}$ is undecidable; as a consequence, through Theorem 3.8 we obtain that type assignment in our system is undecidable as well. In fact, we can use our types to build a semantics for $\text{FJ}^{\not\in}$ programs: following [21], we can define a *filter* d as a set of types that contains ω, and is closed for $\cap$ and $\leqslant$ (so if $\phi, \psi \in d$, then also $\phi \cap \psi \in d$, and if $\phi \in d$, and $\phi \leqslant \psi$, then also $\psi \in d$). It is then straightforward to show that, for every e, the set $\{\phi \mid \exists \Pi\, [\Pi \vdash e : \phi]\}$ is a filter; we could use Theorem 3.8 to define a filter semantics for $\text{FJ}^{\not\in}$, defining $\llceil e \rrparenthesis = \{\phi \mid \exists \Pi\, [\Pi \vdash e : \phi]\}$. In Section 5 we will show, essentially, that this semantics would coincide with our approximation semantics, so we will not develop the line of filter semantics in this paper any further.

4 Strong Normalisation of Derivation Reduction

The approximation result we show in the next section is, as in other systems [8, 16], a direct consequence of the strong normalisability of derivation reduction which we will define in this section. As in [16], we need to consider derivation reduction to achieve the approximation

result; since reduction on expressions is *weak* (the language is first order, methods have an arity, and equality between expressions is non-extensional), the 'normal' approach (as used, for example, in [60, 7]) to show the approximation result does not work. The traditional computability approach is not expressive enough, since, as argued in [16], it depends strongly on the presence of abstraction which FJ lacks. Also, as can be seen in [6], that approach is inherently *extensional* (so closed for η-reduction), a property our system lacks; that is why also for the strict system of [6], also non-extensional, the characterisation of strong normalisation has to be shown using the derivation reduction technique; see [8, 11] for details of this result.

In [16] an approximation result is shown for combinator systems (that have weak reduction), for which an *encompassment* relation on terms is used; this technique is standard in the context of term rewriting, and was also used in [14, 15]. Since our notion of reduction is weak as well, and one might think that a similar approach would be necessary for $\mathbf{FJ}^{\not\in}$. This is not the case however, since our approach differs in that method bodies are typed for *each* individual invocation, and are part of the overall derivation. Thus, there will be sub-derivations for the constituents of each redex that will appear during reduction. The consequence of this is that we are able to prove our main result by straightforward induction on the structure of derivations.

Definition 4.1 (Notation for Derivations) The meta-variable $\mathcal{D}$ ranges over derivations. We will use the notation $\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: \Pi \vdash e : \phi$ to represent the derivation concluding with the judgement $\Pi \vdash e : \phi$ where the last rule applied is (r) and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are the (sub) derivations for each of that rule's premises. By abuse of notation, we may sometimes write $\mathcal{D} :: \Pi \vdash e : \phi$ for $\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: \Pi \vdash e : \phi$ when the structure of the derivation is not relevant, and simply write $\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle$ when the conclusion of the derivation is not relevant or is implied by the context.

The notion of *derivation reduction* is essentially a form of cut-elimination on type derivations, diagrammatically defined through the following two basic 'cut' rules:

$$\dfrac{\dfrac{\boxed{\mathcal{D}_1} \quad \cdots \quad \boxed{\mathcal{D}_n}}{\Pi \vdash e_1 : \phi_1 \quad \cdots \quad \Pi \vdash e_n : \phi_n}\,(\textsc{newF})}{\Pi \vdash \texttt{new}\ C(\vec{e}_n) : \langle f_i : \sigma \rangle}\,(\textsc{fld}) \quad \rightarrow_{\mathfrak{D}} \quad \dfrac{\boxed{\mathcal{D}_i}}{\Pi \vdash e_i : \sigma}$$

and

$$\dfrac{\dfrac{\boxed{\mathcal{D}_b} \quad \boxed{\mathcal{D}_{\text{self}}}}{\texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e_{\mathsf{b}} : \sigma \quad \Pi \vdash \texttt{new}\ C(\vec{e'}) : \psi}\,(\textsc{newM})}{\Pi \vdash \texttt{new}\ C(\vec{e'}) : \langle m : (\vec{\phi}_n) \to \sigma \rangle} \quad \dfrac{\boxed{\mathcal{D}_1} \quad \boxed{\mathcal{D}_n}}{\Pi \vdash e_1 : \phi_1 \quad \cdots \quad \Pi \vdash e_n : \phi_n}\,(\textsc{invk}) \quad \rightarrow_{\mathfrak{D}} \quad \dfrac{\boxed{\mathcal{D}_b{}^{\mathcal{S}}}}{\Pi \vdash e_{\mathsf{b}}{}^{\mathsf{S}} : \sigma}$$

$$\Pi \vdash \texttt{new}\ C(\vec{e'}).m(\vec{e}_n) : \sigma$$

(so (newF) followed by (fld), or (newM) followed by (invk)); here $\mathcal{D}_b{}^{\mathcal{S}}$ is the derivation obtained from $\mathcal{D}_b$ by replacing all sub-derivations of the form $\langle \mathrm{Q} \rangle var :: \Pi, x_i{:}\phi_i \vdash x_i : \sigma$ by a derivation constructed out of sub-derivations of $\mathcal{D}_i$, and replacing sub-derivations of the form $\langle \mathrm{Q} \rangle var :: \Pi, \texttt{this}{:}\psi \vdash \texttt{this} : \sigma$ by a derivation constructed out of sub-derivations of $\mathcal{D}_{\text{self}}$. This induces $e_{\mathsf{b}}{}^{\mathsf{S}}$, obtained from e_{b} by replacing each variable x_i by the expression e_i, and `this` by $\texttt{new}\ C(\vec{e'})$. This reduction creates exactly the derivation for a contractum as suggested by the proof of the subject reduction, but is explicit in all its details, which gives the expressive power to show the approximation result. An important feature of derivation reduction is that sub-derivations of the form $\langle \mathrm{Q} \rangle \omega :: \Pi \vdash e : \omega$ do *not* reduce, since they are already in normal

form; however, notice that the expression involved, e, need not be in normal form. This is crucial for the strong normalisability of derivation reduction, since it decouples the reduction of a derivation from the possibly infinite reduction sequence of the expression which it types.

We now introduce some further notational concepts to aid us in describing and reasoning about the structure and reduction of derivations. The first of these is the notion of *position* in an expression or derivation. We then extend expressions and derivations with a notion of placeholder, so that we can refer to and reason about specific subexpressions and subderivations.

Definition 4.2 (Position) The *position* pq of one (sub) expression – similarly of one (sub) derivation – in another, denoted by $pos\,(e, e')$ – or $pos\,(\mathcal{D}, \mathcal{D}')$ – is a partial function on a pair of expressions or derivations, and returns, if defined, a non-empty sequence of integers:

i) *Positions in expressions* are defined inductively as follows:

$$
\begin{array}{rcl}
pos\,(e, e) & = & 0 \\
pos\,(e', e) = p \;\Rightarrow & & \begin{cases} pos\,(e', e.f) & = & 0 \cdot p \\ pos\,(e', e.m(\vec{e})) & = & 0 \cdot p \end{cases} \\
pos\,(e', e_j) = p \text{ with } j \in \overline{n} \;\Rightarrow & & \begin{cases} pos\,(e', e.m(\vec{e}_n)) & = & j \cdot p \\ pos\,(e', \texttt{new } C(\vec{e}_n)) & = & j \cdot p \end{cases}
\end{array}
$$

ii) *Positions in derivations* are defined inductively as follows:

$$
\begin{array}{rcl}
pos\,(\mathcal{D}, \mathcal{D}) & = & 0 \\
pos\,(\mathcal{D}, \mathcal{D}') & = & pos\,(\mathcal{D}, \langle \mathcal{D}_b, \mathcal{D}', \textsc{newm} \rangle) \\
pos\,(\mathcal{D}, \mathcal{D}_j) = p \text{ with } j \in \overline{n} & \Rightarrow & pos\,(\mathcal{D}, \langle \vec{\mathcal{D}}_n, \textsc{join} \rangle) = p \\
pos\,(\mathcal{D}, \mathcal{D}') = p \;\Rightarrow & & \begin{cases} pos\,(\mathcal{D}, \langle \mathcal{D}', \textsc{fld} \rangle) & = & 0 \cdot p \\ pos\,(\mathcal{D}, \langle \mathcal{D}', \vec{\mathcal{D}}_n, \textsc{invk} \rangle) & = & 0 \cdot p \end{cases} \\
pos\,(\mathcal{D}, \mathcal{D}_j) = p \text{ with } j \in \overline{n} \;\Rightarrow & & \begin{cases} pos\,(\mathcal{D}, \langle \mathcal{D}', \vec{\mathcal{D}}_n, \textsc{invk} \rangle) & = & j \cdot p \\ pos\,(\mathcal{D}, \langle \vec{\mathcal{D}}_n, \textsc{obj} \rangle) & = & j \cdot p \\ pos\,(\mathcal{D}, \langle \vec{\mathcal{D}}_n, \textsc{newf} \rangle) & = & j \cdot p \end{cases}
\end{array}
$$

Notice that due to the (join) rule, sub-derivations indicated by positions in derivations are not necessarily unique.

iii) We define the following terminology:

- We say that e' (or $\mathcal{D}'$) *appears at position* p *in* e ($\mathcal{D}$) if $pos\,(e', e) = p$ ($pos\,(\mathcal{D}', \mathcal{D}) = p$).
- We say that position p *exists in* e ($\mathcal{D}$) if there exists some e' ($\mathcal{D}'$) that appears at position p in e ($\mathcal{D}$).

Notice that different occurrences of a sub-expression have different positions.

Definition 4.3 (Expression Contexts) *i*) An *expression context* $\mathfrak{C}$ is an expression containing a unique 'hole' (denoted by $[\,]$) defined by the following grammar:

$$\mathfrak{C} \;::=\; [\,] \mid \mathfrak{C}.f \mid \mathfrak{C}.m(\vec{e}) \mid e.m(\ldots, e_{i-1}, \mathfrak{C}, e_{i+1}, \ldots) \mid \texttt{new } C(\ldots, e_{i-1}, \mathfrak{C}, e_{i+1}, \ldots)$$

ii) $\mathfrak{C}[e]$ denotes the expression obtained by replacing the hole in $\mathfrak{C}$ with e.

iii) We write $\mathfrak{C}_p$ to indicate that the hole in $\mathfrak{C}$ appears at position p.

iv) Contexts $\mathfrak{C}_p$ where $p = \vec{0}_n$, for some $n \geq 1$, are called *neutral*.

v) Expressions of the form $\mathfrak{C}[x]$ where $\mathfrak{C}$ is neutral are also called neutral.

Neutral expressions are simply those expressions consisting of a (possibly empty) sequence

of successive method invocations and field accesses on a variable. Neutral expressions, along with the following property which is easy to show, are a crucial element to the computability technique that we use to prove our strong normalisation result for derivation reduction, the details of which can be seen in the appendix.

Proposition 4.4 Approximate normal forms of the form $\mathtt{A.f}$ *and* $\mathtt{A.m}(\vec{A})$ *are neutral.*

We also use the notion of *derivation context* that is like a derivation, but concluding with a statement assigning a strict type to a neutral context. We need to extend our notion of type assignment for that:

Definition 4.5 (Derivation Contexts) *i*) We add the inference rule:

$$\frac{}{\Pi \vdash [] : \sigma}\ ([])$$

ii) A *derivation context* $\mathfrak{D}_{(p,\sigma)}$ (where with p we mark at which position the hole appears and which strict type σ it gets assigned) is straightforwardly defined as a generalisation over derivations.

iii) For a derivation $\mathcal{D} :: \Pi \vdash e : \sigma$ and derivation context $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C} : \sigma'$, we write $\mathfrak{D}_{(p,\sigma)}[\mathcal{D}] :: \Pi \vdash \mathfrak{C}[e] : \sigma'$ to denote the derivation obtained by replacing the hole in $\mathfrak{D}$ by $\mathcal{D}$.

We now define an explicit *derivation weakening* operation on derivations, which is straightforwardly extended to derivation contexts. This will be crucial in defining our notion of *computability* which we will use to show that derivation reduction is strongly normalising.

Definition 4.6 (Weakening) A *weakening*, written $[\Pi' \trianglelefteq \Pi]$ where $\Pi' \trianglelefteq \Pi$, is an operation on derivations that replaces environments by smaller environments (with respect to $\trianglelefteq$).

We now define two sets of derivations: strong and ω-safe derivations. The idea behind these kinds of derivation is to restrict the use of the (ω) rule in order to preclude non-termination (*i.e.* guarantee normalisation). In strong derivations, we do not allow the (ω) rule to be used at all. This restriction is relaxed slightly for ω-safe derivations in that ω may be used to type the arguments to a method call. The idea behind this is that when those arguments disappear during reduction it is 'safe' to type them with ω since non-termination at these locations can be ignored. We will show later that our definitions do indeed entail the desired properties, since expressions typeable using strong derivations are strongly normalising, and expressions which can be typed with ω-safe derivations using an ω-safe environment, while not necessarily being strongly normalising, have a normal form.

Definition 4.7 (Strong and ω-safe Derivations) *i*) *Strong derivations* are defined as in Definition 3.4, but by excluding rule (ω).

ii) ω-safe derivations are defined inductively as follows:

- $\langle \mathrm{Q} \rangle var :: x{:}\phi \vdash x : \sigma$ is ω-safe for any ϕ and σ.
- $\langle \overrightarrow{\mathcal{D}}_n, \textsc{join} \rangle$, $\langle \overrightarrow{\mathcal{D}}_n, \textsc{obj} \rangle$ and $\langle \overrightarrow{\mathcal{D}}_n, \textsc{newf} \rangle$ are ω-safe, if each derivation $\mathcal{D}_i$ is ω-safe.
- $\langle \mathcal{D}, \textsc{fld} \rangle$ is ω-safe, if $\mathcal{D}$ is ω-safe.
- $\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk} \rangle$ is ω-safe, if $\mathcal{D}$ is ω-safe and for each $\mathcal{D}_i$ either $\mathcal{D}_i$ is ω-safe or $\mathcal{D}_i$ is of the form $\langle \mathrm{Q} \rangle \omega :: \Pi \vdash e : \omega$.
- $\langle \mathcal{D}, \mathcal{D}', \textsc{newm} \rangle$ is ω-safe, if both $\mathcal{D}$ and $\mathcal{D}'$ are ω-safe.

iii) We call a type ϕ *strong* if it does not contain ω. We call a type environment Π strong if for all $x{:}\phi \in \Pi$, ϕ is strong. Similarly we call Π ω-safe if, for all $x{:}\phi \in \Pi$, either ϕ is strong or $\phi = \omega$.

Notice that ω can appear in ω-safe derivations, but can never be the derived type, and that an ω-safe derivation can have subderivations that are not ω-safe. In Section 6 below we give examples of each kind of derivation (strong, ω-safe and non-ω-safe).

The following lemma is used in the proof of Theorem 5.9.

Lemma 4.8 If $\mathcal{D} :: \Pi \vdash \mathtt{A} : \phi$ with ω-safe $\mathcal{D}$ and Π, then $\mathtt{A}$ does not contain $\bot$; moreover, if $\mathtt{A}$ is neutral, then ϕ does not contain ω.

Proof: By induction on the structure of derivations; we only show one interesting case.

$(\langle \mathcal{D}', \overrightarrow{\mathcal{D}}_n, \textsc{invk} \rangle)$: Then $\mathtt{A} = \mathtt{A}'.m(\vec{\mathtt{A}}_n)$ and ϕ is strict, hereafter called σ. Also $\mathcal{D}' :: \Pi \vdash \mathtt{A}' : \langle m : (\vec{\phi}_n) \to \sigma \rangle$ with $\mathcal{D}'$ ω-safe, and $\mathcal{D}_i :: \Pi \vdash \mathtt{A}_i : \phi_i$ for each $i \in \overline{n}$. By induction, $\mathtt{A}'$ does not contain $\bot$. Also, notice that $\mathtt{A}$ must be neutral, and therefore so must $\mathtt{A}'$. Then it also follows by induction that $\langle m : (\vec{\phi}_n) \to \sigma \rangle$ does not contain ω. This means that no ϕ_i is equal to ω, and so it must be that each $\mathcal{D}_i$ is ω-safe; thus by induction, no $\mathtt{A}_i$ contains $\bot$ either. Consequently, $\mathtt{A}'.m(\vec{\mathtt{A}}_n)$ does not contain $\bot$ and σ does not contain ω. □

Continuing with the definition of derivation reduction, we point out that, just as term substitution is the main engine for reduction on expressions, a notion of substitution for derivations, in which instances of the (VAR) rule are replaced by derivations, will form the basis of derivation reduction. It is formally defined as follows:

Definition 4.9 (DERIVATION SUBSTITUTION) Let $\mathcal{D}_1 :: \Pi' \vdash e_1 : \phi_1$, …, $\mathcal{D}_n :: \Pi' \vdash e_n : \phi_n$ be derivations, then $\mathcal{S} = \langle x_1 : \phi_1 \mapsto \mathcal{D}_1, \ldots, x_n : \phi_n \mapsto \mathcal{D}_n \rangle$ is a *derivation substitution* (*based on Π'*; when each $\mathcal{D}_i$ is strong (ω-safe) then we say that $\mathcal{S}$ is also strong (ω-safe)), a partial function from derivations to derivations, characterised by its effect on subderivations of $\langle \mathrm{Q} \rangle \mathrm{var}$, and is defined by:

i) If $\mathcal{D} :: \Pi \vdash e : \phi$, and $\Pi \subseteq dom(\mathcal{S})$, then $\mathcal{S}$ is *applicable* to $\mathcal{D}$.

ii) If $\mathcal{D} :: \Pi \vdash e : \phi$, $\mathcal{S}$ is applicable to $\mathcal{D}$ and based on Π', then $\mathcal{S}(\mathcal{D})$ (we normally write $\mathcal{D}^{\mathcal{S}}$) is defined inductively as follows (where S is the term substitution induced by $\mathcal{S}$, *i.e.* $\mathsf{S} = \langle x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \rangle$):

$(\mathcal{D} = \langle \mathrm{Q} \rangle var :: \Pi \vdash x : \sigma)$: Then there are two cases to consider:

1) either $x : \sigma \in \Pi$ and so $x = x_i$ for some $i \in \overline{n}$ with $\mathcal{D}_i :: \Pi' \vdash e_i : \sigma$: then $\mathcal{D}^{\mathcal{S}} = \mathcal{D}_i$; or
2) $x : \phi \in \Pi$ with $\phi = \sigma_1 \cap \ldots \cap \sigma_{n'}$ and $\sigma = \sigma_j$ for some $j \in \overline{n'}$. Also in this case, $x = x_i$ for some $i \in \overline{n}$, so then $\mathcal{D}_i = \langle \mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}, \textsc{join} \rangle :: \Pi' \vdash e_i : \phi$ and $\mathcal{D}^{\mathcal{S}} = \mathcal{D}'_j :: \Pi' \vdash e_i : \sigma_j$.

$(\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}', \textsc{newM} \rangle :: \Pi \vdash \mathtt{new}\ C(\vec{e}) : \langle m : (\vec{\phi}) \to \sigma \rangle)$: Then

$$\mathcal{D}^{\mathcal{S}} = \langle \mathcal{D}_b, \mathcal{D}'^{\mathcal{S}}, \textsc{newM} \rangle :: \Pi \vdash \mathtt{new}\ C(\vec{e})^{\mathsf{S}} : \langle m : (\vec{\phi}) \to \sigma \rangle$$

$(\mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: e : \phi, r \notin \{ (\textsc{var}), (\textsc{newM}) \})$: Then $\mathcal{D}^{\mathcal{S}} = \langle \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, r \rangle :: \Pi' \vdash e^{\mathsf{S}} : \phi$.

Notice that the last case includes the base case of derivations of the form $\langle \mathrm{Q} \rangle \omega :: \Pi \vdash e : \omega$ as a special case.

iii) We extend the weakening operation to derivation substitutions as follows: for a derivation substitution $\mathcal{S} = \langle \overrightarrow{x : \psi \mapsto \mathcal{D} :: \Pi \vdash e : \phi} \rangle$, we write $\mathcal{S}[\Pi' \trianglelefteq \Pi]$ for the derivation substitution $\langle \overrightarrow{x : \psi \mapsto \mathcal{D}[\Pi' \trianglelefteq \Pi]} \rangle$.

Example 4.10 Consider the derivations below for two expressions e_1 and e_2:

$$\frac{\boxed{\mathcal{D}_1}}{\Pi \vdash e_1 : \langle m : (\varphi_1 \cap \varphi_2) \to \sigma \rangle} \qquad \mathcal{D}_2 :: \frac{\boxed{\mathcal{D}'_2} \quad \boxed{\mathcal{D}''_2}}{\Pi \vdash e_2 : \varphi_1 \qquad \Pi \vdash e_2 : \varphi_2}\,(\textsc{join})}{\Pi \vdash e_2 : \varphi_1 \cap \varphi_2}$$

and also the following derivation of $x.m(y)$, where $\Pi' = x : \langle m : (\varphi_1 \cap \varphi_2) \to \sigma \rangle, y : \varphi_1 \cap \varphi_2$:

$$\mathcal{D} :: \quad \dfrac{\dfrac{}{\Pi' \vdash x : \langle m{:}(\varphi_1 \cap \varphi_2) \to \sigma\rangle}\,(\textsc{var}) \qquad \dfrac{\dfrac{}{\Pi' \vdash y : \varphi_1}\,(\textsc{var}) \quad \dfrac{}{\Pi' \vdash y : \varphi_2}\,(\textsc{var})}{\Pi \vdash y : \varphi_1 \cap \varphi_2}\,(\textsc{join})}{\Pi' \vdash x.m\,(y) : \sigma}\,(\textsc{invk})$$

Take $\mathcal{S} = \langle x{:}\langle m{:}(\varphi_1 \cap \varphi_2) \to \sigma\rangle \mapsto \mathcal{D}_1, y{:}\varphi_1 \cap \varphi_2 \mapsto \mathcal{D}_2\rangle$; then the result of applying the substitution to $\mathcal{D}$ is the following derivation, where instances of the $(\textsc{var})$ rule in $\mathcal{D}$ have been replaced by the appropriate (sub) derivations in $\mathcal{D}_1$ and $\mathcal{D}_2$:

$$\mathcal{D}^{\mathcal{S}} :: \quad \dfrac{\dfrac{\boxed{\mathcal{D}_1}}{\Pi \vdash e_1 : \langle m{:}(\varphi_1 \cap \varphi_2) \to \sigma\rangle} \qquad \dfrac{\boxed{\mathcal{D}'_2}}{\Pi \vdash e_2 : \varphi_1} \quad \dfrac{\boxed{\mathcal{D}''_2}}{\Pi \vdash e_2 : \varphi_2}\,(\textsc{join})}{\Pi \vdash e_2 : \varphi_1 \cap \varphi_2}\,(\textsc{invk})}{\Pi \vdash e_1.m\,(e_2) : \sigma}$$

Notice that the collection of derivations used in the $(\textsc{join})$ of derivation $\mathcal{D}_2$ 'distributes.'

Derivation substitution is sound, preserves strong and ω-safe derivations, and the operations of weakening and derivation substitution are commutative.

Lemma 4.11 (Soundness of Derivation Substitution) *i) Let $\mathcal{D} :: \Pi \vdash e : \phi$ and $\mathcal{S}$ be based on Π' and applicable to $\mathcal{D}$; then $\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash e^{\mathsf{S}} : \phi$, where S is the term substitution induced by $\mathcal{S}$.*

ii) If $\mathcal{D}$ is strong (ω-safe) then, for any strong (ω-safe) derivation substitution $\mathcal{S}$ applicable to $\mathcal{D}$, $\mathcal{D}^{\mathcal{S}}$ is also strong (ω-safe).

iii) Let $\mathcal{D} :: \Pi'' \vdash e : \phi$ be a derivation and $\mathcal{S}$ be a derivation substitution based on Π and applicable to $\mathcal{D}$, and let $[\Pi' \trianglelefteq \Pi]$ be a weakening. Then $\mathcal{D}^{\mathcal{S}}[\Pi' \trianglelefteq \Pi] = \mathcal{D}^{\mathcal{S}[\Pi' \trianglelefteq \Pi]}$.

Proof: By easy induction on the structure of derivations. □

Definition 4.12 (Identity Substitutions) Each environment Π induces a derivation substitution Id_Π which is called the *identity substitution* for Π. Let $\Pi = \overrightarrow{x{:}\phi_n}$; then $Id_\Pi \triangleq \langle \overrightarrow{x{:}\phi \mapsto \mathcal{D}_n}\rangle$ where for each $i \in \overline{n}$:

- If $\phi_i = \omega$ then $\mathcal{D}_i = \langle \mathrm{Q}\rangle\omega :: \Pi \vdash x_i : \omega$;
- If ϕ_i is a strict type σ then $\mathcal{D}_i = \langle \mathrm{Q}\rangle var :: \Pi \vdash x_i : \sigma$;
- If $\phi_i = \sigma_1 \cap \ldots \cap \sigma_{m_i}$ for some $m_i \geq 2$ then $\mathcal{D}_i = \langle \overrightarrow{\mathcal{D}'}_{m_i}, \textsc{join}\rangle :: \Pi \vdash x_i : \sigma_1 \cap \ldots \cap \sigma_{m_i}$, with $\mathcal{D}'_j = \langle \mathrm{Q}\rangle var :: \Pi \vdash x_i : \sigma_j$ for each $j \in \overline{m_i}$.

Notice that for every environment Π, the identity substitution Id_Π is also *based on* Π.

We can of course show that Id_Π is indeed the identity for the substitution operation on derivations using Π.

Proposition 4.13 Let $\mathcal{D} :: \Pi \vdash e : \phi$, then $\mathcal{D}^{Id_\Pi} = \mathcal{D}$.

Before defining the notion of derivation reduction itself, we first define the auxiliary notion of *advancing* a derivation. This is an operation which contracts redexes at some given position in expressions covered by ω in derivations. This operation will be used to reduce derivations which introduce intersections.

Definition 4.14 (Advancing) *i*) The *advance* operation $\rightsquigarrow$ on expressions contracts the redex at a given position p in e if it exists, and is undefined otherwise. It is defined as the smallest relation on tuples (p, e) and expressions satisfying the following properties (where we write $e \overset{p}{\rightsquigarrow} e'$ to mean $((p, e), e') \in \rightsquigarrow$):

$$e \overset{p}{\rightsquigarrow} e' \;\Rightarrow\; \mathcal{D} :: \Pi \vdash e : \omega \overset{p}{\rightsquigarrow} \langle \mathrm{Q} \rangle \text{`w} :: \Pi \vdash e' : \omega$$

$$\mathcal{D} :: \Pi \vdash e : \langle f{:}\sigma \rangle \overset{p}{\rightsquigarrow} \mathcal{D}' :: \Pi \vdash e' : \langle f{:}\sigma \rangle \;\Rightarrow\; \langle \mathcal{D}, \textsc{fld} \rangle \overset{0 \cdot p}{\rightsquigarrow} \langle \mathcal{D}', \textsc{fld} \rangle$$

$$\mathcal{D} :: \Pi \vdash e : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle \overset{p}{\rightsquigarrow} \mathcal{D}' :: \Pi \vdash e' : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle \;\&\; \forall i \in \underline{n}\ [\mathcal{D}_i :: \Pi \vdash e_i : \phi_i] \Rightarrow \langle \mathcal{D}, \vec{\mathcal{D}}_n, \textsc{invk} \rangle \overset{0 \cdot p}{\rightsquigarrow} \langle \mathcal{D}', \vec{\mathcal{D}}_n, \textsc{invk} \rangle$$

$$\mathcal{M}b(C,m) = (\vec{x}_n, e_b) \;\&\; \texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e_b : \sigma \;\&\; \mathcal{D} :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \psi \overset{p}{\rightsquigarrow} \mathcal{D}' \Rightarrow \langle \mathcal{D}_b, \mathcal{D}, \textsc{newM} \rangle \overset{p}{\rightsquigarrow} \langle \mathcal{D}_b, \mathcal{D}', \textsc{newM} \rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle$$

$$\forall i \in \underline{n}\ (n \geq 2)\ [\mathcal{D}_i :: \Pi \vdash e : \sigma_i \overset{p}{\rightsquigarrow} \mathcal{D}'_i :: \Pi \vdash e' : \sigma_i] \Rightarrow \langle \vec{\mathcal{D}}_n, \textsc{join} \rangle \overset{p}{\rightsquigarrow} \langle \vec{\mathcal{D}'}_n, \textsc{join} \rangle$$

$$\mathcal{D} :: \Pi \vdash e : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle \;\&\; \exists j \in \underline{n}\,[\mathcal{D}_j :: \Pi \vdash e_j : \phi_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j \;\&\; \forall i \neq j \in \underline{n}\ [\mathcal{D}_i :: \Pi \vdash e_i : \phi_i]] \Rightarrow \langle \mathcal{D}, \vec{\mathcal{D}}_n, \textsc{invk} \rangle \overset{j \cdot p}{\rightsquigarrow} \langle \mathcal{D}, \vec{\mathcal{D}'}_n, \textsc{invk} \rangle :: \Pi \vdash e.m(\vec{e'}_n) : \sigma$$

$$\mathcal{F}(C) = \vec{f}_n \;\&\; \exists j \in \underline{n}\,[\mathcal{D}_j :: \Pi \vdash e_j : \phi_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j \;\&\; \forall i \neq j \in \underline{n}\ [\mathcal{D}_i :: \Pi \vdash e_i : \phi_i]] \Rightarrow \langle \vec{\mathcal{D}}_n, \textsc{obj} \rangle \overset{j \cdot p}{\rightsquigarrow} \langle \vec{\mathcal{D}'}_n, \textsc{obj} \rangle :: \Pi \vdash \texttt{new}\ C(\vec{e'}_n) : C$$

$$\mathcal{F}(C) = \vec{f}_n \;\&\; \exists j \in \underline{n}\,[\mathcal{D}_j :: \Pi \vdash e_j : \phi_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j \;\&\; \forall i \neq j \in \underline{n}\ [\mathcal{D}_i :: \Pi \vdash e_i : \phi_i] \;\&\; \phi_j \sim \sigma] \Rightarrow \langle \vec{\mathcal{D}}_n, \textsc{newF} \rangle \overset{j \cdot p}{\rightsquigarrow} \langle \vec{\mathcal{D}'}_n, \textsc{newF} \rangle :: \Pi \vdash \texttt{new}\ C(\vec{e'}_n) : \langle f_j{:}\sigma \rangle$$

For the last three cases, $e_j \overset{p}{\rightsquigarrow} e'_j$ and $\forall i \neq j \in \underline{n}\ [\mathcal{D}'_i = \mathcal{D}_i \;\&\; e'_i = e_i]$.

Figure 4: The advance operation on derivations

$$\mathcal{F}(C) = \vec{f}_n \;\&\; e = \mathfrak{C}_p[\texttt{new}\ C(\vec{e}_n).f_i] \quad \text{with } i \in \overline{n} \;\Rightarrow\; e \overset{p}{\rightsquigarrow} \mathfrak{C}_p[e_i]$$

$$\mathcal{M}b(C,m) = (\vec{x}_n, e_b) \;\&\; e = \mathfrak{C}_p[\texttt{new}\ C(\vec{e'}).m(\vec{e}_n)] \;\Rightarrow\; e \overset{p}{\rightsquigarrow} \mathfrak{C}_p[e_b{}^{\mathsf{S}}] \quad \text{where } \mathsf{S} = \langle \texttt{this} \mapsto \texttt{new}\ C(\vec{e'}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \rangle$$

ii) We extend $\rightsquigarrow$ to derivations via the rules in Figure 4 (where we write $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$ to mean $((p,\mathcal{D}),\mathcal{D}') \in \rightsquigarrow$).

Notice that the advance operation does not change the *structure* of derivations. Exactly the same rules are applied and the same types derived; only subexpressions which are typed with ω are altered.

The following lemma states that this always generates a correct derivation and that the advance operation preserves strong (and ω-safe) typeability.

Lemma 4.15 (Soundness of Advancing) *i) Let $\mathcal{D} :: e : \phi$; if a redex appears at position p in e (so $e \overset{p}{\rightsquigarrow} e'$ for some e') and no derivation redex appears at p in $\mathcal{D}$, then there exists $\mathcal{D}'$ such that $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$, and $\mathcal{D}' :: \Pi \vdash e' : \phi$.*

ii) If $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$ is defined, and $\mathcal{D}$ is strong (ω-safe), then $\mathcal{D}'$ is also strong (ω-safe).

Proof: *i*) By well-founded induction on pairs of position and derivation $(p,\mathcal{D})$.

ii) By induction on the definition of the advance operation for derivations. □

The notion of derivation reduction is defined in two stages. First, the more specific notion of reduction at a certain position (*i.e.* in a given subderivation) is introduced. The full notion of derivation reduction is then a straightforward generalisation of this position-specific reduction over all positions.

Definition 4.16 (Derivation Reduction) *i*) The reduction of a derivation $\mathcal{D}$ at position p to $\mathcal{D}'$ is denoted by $\mathcal{D} \overset{p}{\rightarrowtriangle} \mathcal{D}'$, and is defined inductively using the rules in Figure 5.

ii) The reduction relation on derivations $\to_{\mathfrak{D}}$ is defined by:

$$\mathcal{D} \to_{\mathfrak{D}} \mathcal{D}' \;\overset{\Delta}{=}\; \exists p\,[\mathcal{D} \overset{p}{\rightarrowtriangle} \mathcal{D}']$$

The reflexive and transitive closure of $\to_{\mathfrak{D}}$ is denoted by $\to^*_{\mathfrak{D}}$.

$$\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle :: \Pi \vdash \texttt{new}\ C(\vec{e})\,.\,f_i : \sigma \quad \xrightarrow{0}_{\triangleright}\ \mathcal{D}_i \qquad (\mathcal{F}(C) = \vec{f}_n, \forall i \in \underline{n})$$

$$\langle\langle \mathcal{D}_b :: \texttt{this}{:}\psi, \overrightarrow{x{:}\phi}_n \vdash e_b : \sigma, \mathcal{D}', \textsc{newM}\rangle, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash \texttt{new}\ C(\vec{e'})\,.\,m(\vec{e}_n) : \sigma \xrightarrow{0}_{\triangleright}\ \mathcal{D}_b\mathcal{S}$$
$$(\mathcal{M}b(C,m) = (\vec{x}_n, e_b)\ \&\ \mathcal{S} = \langle \texttt{this}{:}\psi \mapsto \mathcal{D}', \overrightarrow{x{:}\phi \mapsto \mathcal{D}}_n\rangle)$$

$$\mathcal{D} :: \Pi \vdash e : \langle f{:}\sigma\rangle \xrightarrow{p}_{\triangleright} \mathcal{D}' :: \Pi \vdash e' : \phi \ \Rightarrow\ \langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash e\,.\,f : \sigma \xrightarrow{0:p}_{\triangleright} \langle \mathcal{D}', \textsc{fld}\rangle :: \Pi \vdash e'\,.\,f : \sigma$$

$$\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}' :: \Pi \vdash e' : \phi \ \Rightarrow\ \langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash e\,.\,m(\vec{e}_n) : \sigma \xrightarrow{0:p}_{\triangleright} \langle \mathcal{D}', \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle :: \Pi \vdash e'\,.\,m(\vec{e}_n) : \sigma$$

$$\exists j \in \underline{n}\,[\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j :: \Pi \vdash e'_j : \phi] \ \Rightarrow\ \langle \mathcal{D}, \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{invk}\rangle :: \Pi \vdash e\,.\,m(\vec{e}_n) : \sigma \xrightarrow{j:p}_{\triangleright} \langle \mathcal{D}, \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{invk}\rangle :: \Pi \vdash e\,.\,m(\overrightarrow{e'}_n) : \sigma$$
$$(\forall i \neq j \in \underline{n}\ [\mathcal{D}'_i = \mathcal{D}_i\ \&\ e'_i = e_i])$$

$$\exists j \in \underline{n}\,[\mathcal{D}_j :: \Pi \vdash e_j : \phi_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j :: \Pi \vdash e'_j : \phi'_j\ \&\ \phi_j \sim \sigma] \ \Rightarrow\ \langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}_n) : \langle f{:}\sigma\rangle \xrightarrow{j:p}_{\triangleright} \langle \overrightarrow{\mathcal{D}'}_n, \textsc{newF}\rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'}_n) : \langle f{:}\sigma\rangle$$
$$(\forall i \neq j \in \underline{n}\ [\mathcal{D}'_i = \mathcal{D}_i\ \&\ e'_i = e_i])$$

$$\mathcal{D} :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \psi \xrightarrow{p}_{\triangleright} \mathcal{D}' :: \Pi \vdash e : \psi' \ \Rightarrow\ \langle \mathcal{D}_b, \mathcal{D}, \textsc{newM}\rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \langle m{:}(\vec{\phi})\to\sigma\rangle \xrightarrow{p}_{\triangleright} \langle \mathcal{D}_b, \mathcal{D}', \textsc{newM}\rangle :: \Pi \vdash e : \langle m{:}(\vec{\phi})\to\sigma\rangle$$
$$(\mathcal{D}_b :: \texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e_b : \sigma)$$

$$\exists j \in \underline{n}\,[\mathcal{D}_j :: \Pi \vdash e_j : \phi_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j :: \Pi \vdash e'_j : \phi'_j] \ \Rightarrow\ \langle \overrightarrow{\mathcal{D}}_n, \textsc{obj}\rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}_n) : C \xrightarrow{j:p}_{\triangleright} \langle \overrightarrow{\mathcal{D}'}_n, \textsc{obj}\rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'}_n) : C$$
$$(\forall i \neq j \in \underline{n}\ [\mathcal{D}'_i = \mathcal{D}_i\ \&\ e'_i = e_i])$$

$$\exists j \in \underline{n}\,[\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j\ \&\ \forall i \neq j \in \underline{n}\ [\mathcal{D}_i \xrightarrow{p}_{\triangleright} \mathcal{D}'_i \vee \mathcal{D}_i \overset{p}{\rightsquigarrow}_{\triangleright} \mathcal{D}'_i]] \ \Rightarrow\ \langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join}\rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n \xrightarrow{p}_{\triangleright} \langle \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{join}\rangle$$

Figure 5: Derivation reduction

iii) We write $\mathcal{SN}(\mathcal{D})$ whenever the derivation $\mathcal{D}$ is strongly normalising with respect to $\to_{\mathfrak{D}}$.

Similarly to reduction for expressions, if $\mathcal{D} \xrightarrow{0}_{\triangleright} \mathcal{D}'$ then we call $\mathcal{D}$ a *derivation redex* and $\mathcal{D}'$ its *derivation contractum*.

Our notion of derivation reduction is not only *sound* (*i.e.* produces valid derivations) but, most importantly, we can show that it corresponds to reduction on expressions. We can also show that strong and ω-safe derivations are preserved by derivation reduction.

Theorem 4.17 (Soundness of Derivation Reduction) *i) If $\mathcal{D} :: e : \phi$ and $\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}'$, then $\mathcal{D}'$ is a well-defined derivation, that is there exists some e' such that $\mathcal{D}' :: \Pi \vdash e' : \phi$; moreover, then $e \overset{p}{\rightsquigarrow} e'$.*

ii) If $\mathcal{D}$ is strong (ω-safe) and $\mathcal{D} \to_{\mathfrak{D}} \mathcal{D}'$, then $\mathcal{D}'$ is strong (ω-safe).

Proof: By induction on the definition of derivation reduction; for the second part, notice that derivation reduction does not introduce instances of rule (ω) and that, by Lemma 4.11, derivation substitution preserves strong and ω-safe derivations. □

We can also show that derivation reduction is strongly normalisable; the (full construction of the) proof can be found in the appendix. The main result shown there is:

Theorem 4.18 (Strong Normalisation for Derivation Reduction) *If $\mathcal{D} :: \Pi \vdash e : \phi$ then $\mathcal{D}$ is strongly normalisable with respect to $\to_{\mathfrak{D}}$.*

5 Linking Types with Semantics: The Approximation Result

We will now study the relationship that the type system from Section 3 has with the semantics that we defined in Section 2. This takes the form of an *approximation theorem*, which states that

every type we can assign to an approximant of an expression can be assigned to the expression itself, and vice-versa:

$$\Pi \vdash e : \phi \;\Leftrightarrow\; \exists\, A \in \mathcal{A}(e)\,[\Pi \vdash A : \phi]$$

This expresses that every type we can derive for an expression describes a finite part of its (potentially infinite) head normal form and execution behaviour by describing that part of the output that is reached after a finite amount of steps. We will show that this result is a direct consequence of the strong normalisability of derivation reduction we achieved in the previous section: the structure of the normal form of a given derivation exactly corresponds to the structure of the approximant of the term that is typed. This is a very strong property since, as it implies that typeability provides a sufficient condition for the (head) normalisation of *expressions*, *i.e.* a *termination* analysis for $\mathbf{FJ}^{\not\in}$.

The following properties of approximants and type assignment lead to the approximation result itself.

Lemma 5.1 If $\mathcal{D} :: \Pi \vdash a : \phi$ (with $\mathcal{D}$ ω-safe) and $a \sqsubseteq a'$ then there exists $\mathcal{D}' :: \Pi \vdash a' : \phi$ (where $\mathcal{D}'$ is ω-safe).

Proof: By induction on the definition of $\sqsubseteq$. The main case is $\bot \sqsubseteq a'$: then $\phi = \omega$, and the result follows. □

Lemma 5.2 Let $\vec{A}_n$ be apns with $n \geq 2$ and e be an expression such that $A_i \sqsubseteq e$ for each $i \in \overline{n}$. Then $\sqcup \vec{A}_n$ is also an apn and $\sqcup \vec{A}_n \sqsubseteq e$, and if there are ($\omega$-safe) derivations $\mathcal{D}_i :: \Pi \vdash A_i : \phi_i$ for each $i \in \overline{n}$, there are (ω-safe) derivations $\mathcal{D}'_i :: \Pi \vdash \sqcup \vec{A}_n : \phi_i$ for each $i \in \overline{n}$.

Proof: By induction on the number of approximants. We just deal with the base case $n = 2$.

$(n = 2)$: Then there are A_1 and A_2 such that $A_1 \sqsubseteq e$ and $A_2 \sqsubseteq e$. By Lemma 2.6, $A_1 \sqcup A_2 \sqsubseteq e$, with $A_1 \sqcup A_2$ an *apn*, and also $A_1 \sqsubseteq A_1 \sqcup A_2$ and $A_1 \sqsubseteq A_2 \sqcup A_2$. Therefore, given that $\mathcal{D}_1 :: \Pi \vdash A_1 : \phi_1$ and $\mathcal{D}_2 :: \Pi \vdash A_2 : \phi_2$ (with ω-safe $\mathcal{D}_1$ and $\mathcal{D}_2$), by Lemma 5.1 there exist derivations $\mathcal{D}'_1$ and $\mathcal{D}'_2$ (both ω-safe) such that $\mathcal{D}'_1 :: \Pi \vdash A_1 \sqcup A_2 : \phi_1$ and $\mathcal{D}'_2 :: \Pi \vdash A_1 \sqcup A_2 : \phi_2$. Then by Lemma 2.6, $\sqcup \vec{A}_2 = A_1 \sqcup A_2$. □

The following lemma states that a derivation in normal form corresponds to a derivation for an *apn*.

Lemma 5.3 If $\mathcal{D} :: e : \phi$ (with $\mathcal{D}$ ω-safe) and $\mathcal{D}$ is in $\rightarrow_{\mathfrak{D}}$-normal form, then there exists A and (ω-safe) $\mathcal{D}'$ such that $A \sqsubseteq e$ and $\mathcal{D}' :: \Pi \vdash A : \phi$, and $\mathcal{D}$ and $\mathcal{D}'$ have the same structure in terms of applied rules and types.

Proof: By induction on the structure of derivations.

$((\omega))$: Take $A = \bot$. Notice that $\bot \sqsubseteq e$, by Definition 2.3, and by (ω) we can take $\mathcal{D}' = \langle \mathrm{Q} \rangle \omega :: \Pi \vdash \bot : \omega$. (In the ω-safe version of the result, this case is vacuously true since the derivation $\mathcal{D} = \langle \mathrm{Q} \rangle \omega :: \Pi \vdash e : \omega$ is not ω-safe.)

$((\textsc{var}))$: Then $e = x$ and $\mathcal{D} = \langle \mathrm{Q} \rangle var :: \Pi \vdash x : \sigma$ (notice that this is a derivation in normal form). By Definition 2.2, x is already an *apn* and $x \sqsubseteq x$, by Definition 2.3. So we take $A = x$ and $\mathcal{D}' = \mathcal{D}$. Moreover, notice that, by Definition 4.7, $\mathcal{D}$ is an ω-safe derivation.

$((\textsc{join}))$: Then $\mathcal{D} = \langle \vec{\mathcal{D}}_n, \textsc{join} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n$ with $n \geq 2$ and $\mathcal{D}_i :: \Pi \vdash e : \sigma_i$ for each $i \in \overline{n}$. Since $\mathcal{D}$ is in normal form it follows that each $\mathcal{D}_i\,(i \in \overline{n})$ is in normal form too (and also, if $\mathcal{D}$ is ω-safe then, by Definition 4.7, each $\mathcal{D}_i$ is ω-safe too). By induction, there exist $\vec{A}_n$ and (ω-safe) derivations $\overrightarrow{\mathcal{D}'}_n$ such that, for each $i \in \overline{n}$, $A_i \sqsubseteq e$ and $\mathcal{D}'_i :: \Pi \vdash A_i : \sigma_i$. Now, by Lemma 5.2 it follows that $\sqcup \vec{A}_n \sqsubseteq e$ with $\sqcup \vec{A}_n$ normal and that there are (ω-safe) derivations $\overrightarrow{\mathcal{D}'}_n$ such that $\mathcal{D}'_i :: \Pi \vdash \sqcup \vec{A}_n : \sigma_i$ for each $i \in \overline{n}$. Finally, by the $(\textsc{join})$ rule we

can take (ω-safe) $\mathcal{D}' = \langle \overrightarrow{\mathcal{D}''}_n, \textsc{join} \rangle :: \Pi \vdash \sqcup \vec{A}_n : \sigma_1 \cap \ldots \cap \sigma_n$.

$((\textsc{fld}))$: Then $e = e'.f$ and $\mathcal{D} = \langle \mathcal{D}', \textsc{fld} \rangle :: \Pi \vdash e'.f : \sigma$ with $\mathcal{D}' :: \Pi \vdash e' : \langle f{:}\sigma \rangle$. Since $\mathcal{D}$ is in normal form, so too is $\mathcal{D}'$. Furthermore, if $\mathcal{D}$ is ω-safe then, by Definition 4.7, so too is $\mathcal{D}'$. By induction, there is some A and (ω-safe) derivation $\mathcal{D}''$ such that $A \sqsubseteq e'$ and $\mathcal{D}'' :: \Pi \vdash A : \langle f{:}\sigma \rangle$. Then by rule $(\textsc{fld})$, $\langle \mathcal{D}'', \textsc{fld} \rangle :: \Pi \vdash A.f : \sigma$ and, by Definition 2.3, $A.f \sqsubseteq e'.f$. Moreover, by Definition 4.7, when $\mathcal{D}''$ is ω-safe, so too is $\langle \mathcal{D}'', \textsc{fld} \rangle$.

$((\textsc{invk}), (\textsc{obj}), (\textsc{newf}), (\textsc{newm}))$: These cases follow straightforwardly by induction similar to $(\textsc{fld})$. □

Lemma 5.1 above simply states the soundness of type assignment with respect to the approximation relation. Lemma 5.3 is the more interesting, since it expresses the relationship between the structure of a derivation and the typed approximant. The derivation $\mathcal{D}'$ is constructed from $\mathcal{D}$ by replacing sub-derivations of the form $\langle \mathrm{Q} \rangle \omega :: \Pi \vdash e : \omega$ by $\langle \mathrm{Q} \rangle \omega :: \Pi \vdash \bot : \omega$ (thus covering any redexes appearing in e). Since $\mathcal{D}$ is in normal form, there are also no *typed* redexes, ensuring that the expression typed in the conclusion of $\mathcal{D}'$ is an *apn*. The 'only if' part of the approximation result itself then follows easily from the fact that $\rightarrow_{\mathfrak{D}}$ corresponds to reduction of expressions, so A is also an *approximant* of e. The 'if' part follows from the first property above and subject expansion.

Theorem 5.4 (Approximation Theorem) *$\Pi \vdash e : \phi$ if and only if there exists $A \in \mathcal{A}(e)$ such that $\Pi \vdash A : \phi$.*

Proof: *(if)*: There is an approximant A of e such that $\Pi \vdash A : \phi$, so $e \rightarrow^* e'$ with $A \sqsubseteq e'$. Then, by Lemma 5.1, $\Pi \vdash e' : \phi$, and then by subject expansion (Theorem 3.8), also $\Pi \vdash e : \phi$.

(only if): Let $\mathcal{D} :: \Pi \vdash e : \phi$, then, by Theorem 4.18, $\mathcal{D}$ is strongly normalising, with normal form $\mathcal{D}'$, say; by the soundness of derivation reduction (Theorem 4.17), $\mathcal{D}' :: \Pi \vdash e' : \phi$ and $e \rightarrow^* e'$. By Lemma 5.3, there is some *apn* A such that $\Pi \vdash A : \phi$ and $A \sqsubseteq e'$. Also, by Definition 2.7, $A \in \mathcal{A}(e)$. □

Termination Analysis As in other intersection type systems [8, 16, 9, 11], the approximation theorem underpins characterisation results for various forms of termination. Our type system is *sound* with respect to the approximation semantics (as shown by the Approximation Theorem), and so typeability gives a guarantee of termination since our normal approximate forms of Definition 2.2 correspond in structure to standard expressions in (head) normal form.

Definition 5.5 ((Head) Normal Forms) *i)* The set of (well-formed) *head-normal forms* (ranged over by H) is defined by:

$$H \ ::= \ x \mid \texttt{new}\ C(\vec{e}_n) \mid H.f \mid H.m(\vec{e}) \quad (H \neq \texttt{new}\ C(\vec{e}))$$

ii) The set of (well-formed) *normal* forms (ranged over by N) is defined by:

$$N \ ::= \ x \mid \texttt{new}\ C(\vec{N}_n) \mid N.f \mid N.m(\vec{N}) \quad (N \neq \texttt{new}\ C(\vec{N}))$$

Notice that the difference between these two notions sits in the second and fourth alternatives, where head-normal forms allow arbitrary expressions to be used.

Lemma 5.6 *i) If $A \neq \bot$ and $A \sqsubseteq e$, then e is a head-normal form.*

ii) If $A \sqsubseteq e$ and A does not contain $\bot$, then e is a normal form.

Proof: By straightforward induction on the structure of *apn*s using Definition 2.3. □

From the approximation result, the following characterisation of head-normalisation follows easily.

Lemma 5.7 (Typeability of (head) normal forms) *i) If e is a head-normal form then there exists a strict type σ and type environment Π 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 σ there is an environment such that $\Pi \vdash e : \sigma$.*

ii) If e is a normal form then there exist strong strict type σ, type environment Π and derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash e : \sigma$; moreover, if e is not of the form $\texttt{new } C(\vec{e}_n)$ then for any arbitrary strong strict type there exist strong $\mathcal{D}$ and Π such that $\mathcal{D} :: \Pi \vdash e : \sigma$.

Proof: *i*) By induction on the structure of head-normal forms; we only show some of the cases:

($\texttt{new } C(\vec{e}_n)$): Notice that $\mathcal{F}(C) = \vec{f}_n$, by definition of the head-normal form. Notice that by rule (ω) we have $\emptyset \vdash e_i : \omega$ for each $i \in \overline{n}$; by rule (OBJ) we have $\emptyset \vdash \texttt{new } C(\vec{e}_n) : C$.

($H.f$): Take σ' a strict type, then, in particular, $\langle f{:}\sigma' \rangle$ is strict. Notice that, by definition, H is a head-normal expression *not* of the form $\texttt{new } C(\vec{e}_n)$, thus by induction there exists Π such that $\Pi \vdash H : \langle f{:}\sigma' \rangle$. Thus, by rule (FLD) we have $\Pi \vdash H.f : \sigma'$ for any arbitrary strict type σ'.

ii) By induction on the structure of normal forms.

(x): By the (VAR) rule, $x{:}\sigma \vdash x : \sigma$ for any arbitrary strict type (in particular, for any arbitrary *strong* strict type). Also, notice that derivations of the form $\langle \mathrm{Q} \rangle \mathrm{var}$ are strong by Definition 4.7.

($\texttt{new } C(\vec{N}_n)$): Notice that $\mathcal{F}(C) = \vec{f}_n$ by the definition of normal forms. Since each N_i is a normal form, by induction there are strong strict types $\vec{\sigma}_n$, $\overline{\Pi}_n$ and $\overline{\mathcal{D}}_n$ such that $\mathcal{D}_i :: \Pi_i \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Let $\Pi' = \bigcap \overline{\Pi}_n$; notice that, by Definition 3.3, $\Pi' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$, and also that since each Π_i is strong so is Π'. Thus, $[\Pi' \trianglelefteq \Pi_i]$ is a weakening for each $i \in \overline{n}$ and thus $\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i] :: \Pi' \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Notice that, by Definition 4.6, weakening does not change the structure of derivations, therefore for each $i \in \overline{n}$, $\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ is a strong derivation. Now, by rule (OBJ) we can derive

$$\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi_n], \textsc{obj} \rangle :: \Pi' \vdash \texttt{new } C(\vec{N}_n) : C$$

Notice that C is a strong strict type, and that since each derivation $\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ is strong then, by Definition 4.7, so is $\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi_n], \textsc{obj} \rangle$.

($N.f$): Notice that, by definition, N is a normal expression *not* of the form $\texttt{new } C(\vec{N}_n)$, thus by induction, with σ' a strong strict type, there are strong Π and $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \langle f{:}\sigma' \rangle$. Thus, by rule (FLD) we have $\langle \mathcal{D}, \textsc{fld} \rangle :: \Pi \vdash N.f : \sigma'$. Notice that since $\mathcal{D}$ is strong, by Definition 4.7 also $\langle \mathcal{D}, \textsc{fld} \rangle$ is strong.

($N.m(\vec{N}_n)$): Since each N_i for $i \in \overline{n}$ is a normal form, by induction there are strong strict types $\vec{\sigma}_n$, $\overline{\Pi}_n$ and $\overline{\mathcal{D}}_n$ such that $\mathcal{D}_i :: \Pi_i \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Take σ' a strong strict type, then $\langle m{:}(\vec{\sigma}_n) \to \sigma' \rangle$ is also strong. Notice that, by definition, N is a normal expression *not* of the form $\texttt{new } C(\vec{N}_n)$, thus by induction there is a strong environment Π and derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \langle m{:}(\vec{\sigma}_n) \to \sigma' \rangle$. Let $\Pi' = \bigcap \Pi \cdot \overline{\Pi}_n$ notice that, by Definition 3.3, $\Pi' \trianglelefteq \Pi$ and $\Pi' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$, and also that since Π is strong and each Π_i is strong then so is Π'. Thus, $[\Pi' \trianglelefteq \Pi]$ is a weakening and $[\Pi' \trianglelefteq \Pi_i]$ is a weakening for each $i \in \overline{n}$. Then $\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash N : \langle m{:}(\vec{\sigma}_n) \to \sigma' \rangle$ and $\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i] :: \Pi' \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Notice that, by Definition 4.6, weakening does not change the structure of derivations, therefore $\mathcal{D}[\Pi' \trianglelefteq \Pi]$ is strong and for each $i \in \overline{n}$, $\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ is also strong. Now, by rule (INVK)

$$\langle \mathcal{D}[\Pi' \trianglelefteq \Pi], \mathcal{D}_1[\Pi' \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi_n], \textsc{invk} \rangle :: \Pi' \vdash N.m(\vec{N}_n) : \sigma'$$

for any arbitrary strong strict type σ'. Furthermore, by Definition 4.7, we have that

$$\langle \mathcal{D}[\Pi' \trianglelefteqslant \Pi], \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \textsc{invk} \rangle$$

is a strong derivation. □

Theorem 5.8 (Head-normalisation) *$\Pi \vdash e : \sigma$ if and only if e has a head-normal form.*

Proof: (*if*): Let e' be a head-normal of e. By Lemma 5.7(1) there exists a strict type σ and a type environment Π such that $\Pi \vdash e' : \sigma$. Then by subject expansion (Theorem 3.8) it follows that $\Pi \vdash e : \sigma$.

(*only if*): By the approximation theorem, there is an approximant A of e such that $\Pi \vdash A : \sigma$. Thus $e \to^* e'$ with $A \sqsubseteq e'$. Since σ is strict, it follows that $A \neq \bot$, so by Lemma 5.6 e' is a head-normal form. □

For LC, normalisability can be characterised in ITD as follows:

$$\Gamma \vdash M : \sigma \text{ with } \Gamma \text{ and } \sigma \text{ strong} \Leftrightarrow M \text{ has a normal form}$$

An analogous result does not hold for $\textbf{FJ}^{\not\subset}$ (see the third example in Example 6.12 for a counterexample); however, we can obtain such a result *modulo* certain kinds of derivations – namely the *ω-safe* derivations (and also, as we will explain, modulo certain kinds of programs – namely OOCL ones).

One half of the implication holds in general:

Theorem 5.9 (Normalisation) *If $\mathcal{D} :: \Pi \vdash e : \sigma$ with $\mathcal{D}$ and Π ω-safe then e has a normal form.*

Proof: By the approximation theorem, there is an approximant A of e and derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi \vdash A : \sigma$ and $\mathcal{D} \to^*_{\mathfrak{D}} \mathcal{D}'$. Thus $e \to^* e'$ with $A \sqsubseteq e'$. Also, since derivation reduction preserves ω-safe derivations (Lemma 4.17), it follows that $\mathcal{D}'$ is ω-safe and thus by Lemma 4.8 that A does not contain $\bot$. Then by Lemma 5.6 we have that e' is a normal form. □

The reverse implication does not hold in general since our notion of ω-safe typeability is too fragile: it is not preserved by (derivation) expansion. Consider that while an ω-safe derivation may exist for $\Pi \vdash e_i : \sigma$, no ω-safe derivation may exist for $\Pi \vdash \texttt{new C}(\vec{e}_n)\texttt{.}f_i : \sigma$ (due to non-termination in the other expressions e_j with $j \neq i$) even though this expression has the same normal form as e_i. Such a completeness result *can* hold for certain particular programs, and we consider such an example in the following section.

We can however show that the set of strongly normalising expressions are exactly those typeable using strong derivations. This follows from the fact that in such derivations, all redexes in the typed expression correspond to redexes in the derivation, and then any reduction step that can be made by the expression (via $\to$) is then matched by a corresponding reduction of the derivation (via $\to_{\mathfrak{D}}$).

Theorem 5.10 (Strong Normalisation for Expressions) *e is strongly normalisable if and only if $\mathcal{D} :: \Pi \vdash e : \sigma$ with $\mathcal{D}$ strong.*

Proof: (*if*): Since $\mathcal{D}$ is strong, all redexes in e are typed with a strict type and therefore each possible reduction of e is matched by a corresponding derivation reduction of $\mathcal{D}$. By Lemma 4.17 it follows that no reduction of $\mathcal{D}$ introduces subderivations of the form $\langle \mathrm{Q} \rangle \omega$, and so since $\mathcal{D}$ is strongly normalising (Theorem 4.18) so too is e.

(*only if*): By induction on the maximum lengths of left-most outer-most reduction sequences for strongly normalising expressions, using the fact that all normal forms are typeable with strong derivations and that strong typeability is preserved under left-most outer-most redex expansion. □

6 Curry type assignment

Although the nominal type system for Java is so far the accepted standard, 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 briefly studying 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 a true restriction of our semantical type system; the restriction consists of removing the type constant ω as well as intersection types from the type language, but not completely: we will still allow for types to be combined as by rule (JOIN) above, but only if they are of the shape $\langle f : \cdot \rangle$ or $\langle m : \cdot \rangle$, and the labels involved are different: the intersection types we allow, thereby, correspond to *records*.

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 ω) we could not show that type assignment is closed under conversion.

Definition 6.1 (CURRY TYPE ASSIGNMENT FOR FJ$^{\not\in}$) *i) Curry (object) types* for FJ are defined by:

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

We will call a type of the shape $\langle\cdots\rangle$ a *record* type, and let ρ range over those; we write ℓ for arbitrary labels, $\langle \ell{:}\sigma\rangle \in \rho$ when $\ell{:}\sigma$ occurs in ρ, and assume that all labels are distinct in records.

ii) A *Curry context* is a mapping from term variables (including `this`) to Curry types.

iii) Curry type assignment for FJ is defined through the rules:

$$(\text{NEWM}):\ \frac{\texttt{this}{:}\tau, x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n \vdash e_b : \sigma \quad \Pi \vdash \texttt{new}\ C(\vec{e}) : \tau}{\Pi \vdash \texttt{new}\ C(\vec{e}) : \langle m{:}(\vec{\sigma}_n)\to\sigma\rangle}\ (\mathcal{M}b(C,m) = (\vec{x}_n, e_b))$$

$$(\text{NEWF}):\ \frac{\Pi \vdash e_1 : \sigma_1 \quad \ldots \quad \Pi \vdash e_n : \sigma_n}{\Pi \vdash \texttt{new}\ C(\vec{e}_n) : \langle f_i{:}\sigma_i\rangle}\ (\mathcal{F}(C) = \vec{f}_n, i\in\overline{n}, n>0)$$

$$(\text{OBJ}):\ \frac{\Pi \vdash f_1 : \sigma_1 \quad \ldots \quad \Pi \vdash f_n : \sigma_n}{\Pi \vdash \texttt{new}\ C(\vec{e}_n) : C}\ (\mathcal{F}(C) = \vec{f}_n) \qquad (\text{VAR}):\ \frac{}{\Pi, x{:}\sigma \vdash x : \sigma}$$

$$(\text{INVK}):\ \frac{\Pi \vdash e : \langle m{:}(\vec{\sigma}_n)\to\sigma\rangle \quad \Pi \vdash e_1 : \sigma_1 \ \ldots\ \Pi \vdash e_n : \sigma_n}{\Pi \vdash e.m(\vec{e}_n) : \sigma} \qquad (\text{FLD}):\ \frac{\Pi \vdash e : \langle f{:}\sigma\rangle}{\Pi \vdash e.f : \sigma}$$

$$(\text{REC}):\ \frac{\Gamma \vdash e : \langle \ell_1{:}\sigma_1\rangle \quad \cdots \quad \Gamma \vdash e : \langle \ell_n{:}\sigma_n\rangle}{\Gamma \vdash e : \langle \ell_1{:}\sigma_1,\ \ldots,\ \ell_n{:}\sigma_n\rangle} \qquad (\text{PROJ}):\ \frac{\Gamma \vdash e : \rho}{\Gamma \vdash e : \langle \ell{:}\sigma\rangle}\ (\ell{:}\sigma\in\rho)$$

We write $\Gamma \vdash_C e : \sigma$ for statements derivable using those rules; the last two rules could be omitted without affecting the obtainable results.

We will normally drop the adjective "Curry".

It is straightforward to check that this system is a true restriction of our intersection type system by translating record types into intersections, as described above, and then noting that the (REC) rule corresponds to (JOIN) and (PROJ) corresponds to a derivable subsumption rule with respect to $\leqslant$; for the other rules, in case that σ is a record type, the premise can be translated into an appropriate intersection constructed from all the strict types contained in the record type σ. The normalisation results as shown above therefore still hold. In particular, since ω is not used, all typeable terms are strongly normalisable.

We make no claim about the possibility to define a notion of principal pair for FJ$^{\not\in}$ expressions for this system, nor how to show completeness and decidability of (Curry) type

assignment. Since we focus in this paper on semantics, and not on implementation, we do not study such properties. Notice that this system, as the one of Definition 3.4, does not associate types to classes, as does the nominal system of Definition 1.6;[8] however, a decidable restriction would need to do this, as well as switch to *early self typing*.

We can, however, relate this notion of type assignment to one from the world of functional programming, by defining an encoding of Combinatory Logic [34] (CL) into $\text{FJ}^{\not\in}$, and showing that assignable types are preserved by this encoding.

Definition 6.2 (COMBINATORY LOGIC) CL consists of the function symbols $\mathbf{S}, \mathbf{K}$ with terms 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:

$$\begin{aligned} \mathbf{K}\ x\ y &\to x \\ \mathbf{S}\ x\ y\ z &\to x\ z\ (y\ z) \end{aligned}$$

CL can be seen as a higher-order TRS.

Through our embedding - and the results we have shown above - we can achieve a type-based characterisation of all (terminating) computable functions in OO (see Theorem 6.11). Since CL is a Turing-complete model of computation, as a side effect we show that $\text{FJ}^{\not\in}$ is Turing-complete.[9] 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.

Our encoding of CL in $\text{FJ}^{\not\in}$ is based on a Curryfied first-order version of the system above (see [15] 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. Application, the basic engine of reduction in TRS, is modelled 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 6.3 The encoding of Combinatory Logic (CL) into the $\text{FJ}^{\not\in}$ program OOCL (Object-Oriented Combinatory Logic) is defined using the class table given in Figure 6 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.

[8] We will leave a system based on this one, that types classes as well and has polymorphic method types, for future research.

[9] As a remark, it is not straightforward to embed the higher-order abstraction of LC into $\text{FJ}^{\not\in}$ without resorting to bracket abstraction, as is used for the encoding of LC into CL. The approach we follow here seems to be the most straightforward.

```
class Combinator extends Object {
      Combinator app(Combinator x) { return this; }
}
class K extends Combinator {
      Combinator app(Combinator x) { return new K1(x); }
}
class K1 extends K {
      Combinator x;
      Combinator app(Combinator y) { return this.x; }
}
class S extends Combinator {
      Combinator app(Combinator x) { return new S1(x); }
}
class S1 extends S {
      Combinator x;
      Combinator app(Combinator y) { return new S2(this.x, y); }
}
class S2 extends S1 {
      Combinator y;
      Combinator app(Combinator z) { return this.x.app(z).app(this.y.app(z)); }
}
```

Figure 6: The class table for Object-Oriented Combinatory Logic (OOCL) programs

Theorem 6.4 (SOUNDNESS OF $[\![\cdot]\!]$) *If t_1, t_2 are terms of* CL *and $t_1 \to^* t_2$, then $[\![t_1]\!] \to^* [\![t_2]\!]$ in* OOCL.

Proof: By induction on the definition of reduction in CL; we only show the case for **S**:

$$
\begin{array}{ll}
[\![\mathbf{S}\,t_1\,t_2\,t_3]\!] & \stackrel{\Delta}{=} \\
\texttt{((new S()).app(}[\![t_1]\!]\texttt{)).app(}[\![t_2]\!]\texttt{)).app(}[\![t_3]\!]\texttt{)} & \to \\
\texttt{((new S1(}[\![t_1]\!]\texttt{)).app(}[\![t_2]\!]\texttt{)).app(}[\![t_3]\!]\texttt{)} & \to \\
\texttt{(new S2(this.x,y)).app(}[\![t_3]\!]\texttt{)} \quad [\texttt{this} \mapsto \texttt{new S1(}[\![t_1]\!]\texttt{), y} \mapsto [\![t_2]\!]] & = \\
\texttt{(new S2(new S1(}[\![t_1]\!]\texttt{).x,}[\![t_2]\!]\texttt{)).app(}[\![t_3]\!]\texttt{)} & \to \\
\texttt{new S2(}[\![t_1]\!]\texttt{,}[\![t_2]\!]\texttt{).app(}[\![t_3]\!]\texttt{)} & \to \\
\texttt{this.x.app(z).app(this.y.app(z))} \quad [\texttt{this} \mapsto \texttt{new S2(}[\![t_1]\!]\texttt{,}[\![t_2]\!]\texttt{), z} \mapsto [\![t_3]\!]] & = \\
\texttt{(new S2(}[\![t_1]\!]\texttt{,}[\![t_2]\!]\texttt{).x.app(}[\![t_3]\!]\texttt{)).app(new S2(}[\![t_1]\!]\texttt{.}[\![t_2]\!]\texttt{).y.app(}[\![t_3]\!]\texttt{))} & \to^* \\
\texttt{(}[\![t_1]\!]\texttt{.app(}[\![t_3]\!]\texttt{)).app((}[\![t_2]\!]\texttt{).app(}[\![t_3]\!]\texttt{))} & \stackrel{\Delta}{=} \\
[\![t_1\,t_3\,(t_2\,t_3)]\!] &
\end{array}
$$

The case for **K** is similar, and the rest is straightforward. □

The reverse of this result also holds, that is if $[\![t_1]\!] \to^* [\![t_2]\!]$ in OOCL, then $t_1 \to^* t_2$ in CL. Notice that this only relates reduction between OOCL expressions which are the *images* of CL terms. Consider that there are OOCL expressions (and typeable ones, at that) which have no counterpart in CL, such as `new S2(new K(), new K()).x`; see also Example 7.1; this implies that we cannot show an *operational completeness* result.

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 6.5 (CURRY TYPE ASSIGNMENT FOR CL [46]) *i*) The set of *simple types* (also known

$$\dfrac{\dfrac{\dfrac{}{\texttt{this}{:}\langle \texttt{x}{:}\sigma\rangle,\texttt{y}{:}\tau \vdash \texttt{this} : \langle \texttt{x}{:}\sigma\rangle}\,(\textsc{var})}{\texttt{this}{:}\langle \texttt{x}{:}\sigma\rangle,\texttt{y}{:}\tau \vdash \texttt{this.x} : \sigma}\,(\textsc{fld}) \quad \dfrac{\dfrac{}{\texttt{x}{:}\sigma \vdash \texttt{x} : \sigma}\,(\textsc{var})}{\texttt{x}{:}\sigma \vdash \texttt{new K}_1\texttt{(x)} : \langle \texttt{x}{:}\sigma\rangle}\,(\textsc{newF})}{\dfrac{\texttt{x}{:}\sigma \vdash \texttt{new K}_1\texttt{(x)} : \langle \texttt{app}{:}(\tau){\to}\sigma\rangle}{\emptyset \vdash \texttt{new K()} : \langle \texttt{app}{:}(\sigma){\to}\langle \texttt{app}{:}(\tau){\to}\sigma\rangle\rangle}\,(\textsc{newM})}\,(\textsc{newM})$$

Let $\sigma_1 = \langle \texttt{app}{:}(\sigma){\to}\langle \texttt{app}{:}(\tau){\to}\mu\rangle\rangle$, and $\sigma_2 = \langle \texttt{app}{:}(\sigma){\to}\tau\rangle$, $\Pi' = \texttt{this}{:}\langle \texttt{x}{:}\sigma_1,\texttt{y}{:}\sigma_2$, and $\Pi = \texttt{this}{:}\langle \texttt{x}{:}\sigma_1,\texttt{y}{:}\sigma_2\rangle,\texttt{z}{:}\sigma$. Then

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this} : \langle \texttt{x}{:}\sigma_1,\texttt{y}{:}\sigma_2\rangle}\,(\textsc{var})}{\Pi \vdash \texttt{this} : \langle \texttt{x}{:}\langle \texttt{app}{:}(\sigma){\to}\langle \texttt{app}{:}(\tau){\to}\mu\rangle\rangle\rangle}\,(\textsc{proj})}{\Pi \vdash \texttt{this.x} : \langle \texttt{app}{:}(\sigma){\to}\langle \texttt{app}{:}(\tau){\to}\mu\rangle\rangle}\,(\textsc{fld}) \quad \vdots}{\Pi \vdash \texttt{this.x.app(z)} : \langle \texttt{app}{:}(\tau){\to}\mu\rangle}\,(\textsc{invk}) \quad \dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this} : \langle \texttt{x}{:}\sigma_1,\texttt{y}{:}\sigma_2\rangle}\,(\textsc{var})}{\Pi \vdash \texttt{this} : \langle \texttt{y}{:}\langle \texttt{app}{:}(\sigma){\to}\tau\rangle\rangle}\,(\textsc{proj})}{\Pi \vdash \texttt{this.y} : \langle \texttt{app}{:}(\sigma){\to}\tau\rangle}\,(\textsc{fld}) \quad \dfrac{}{\Pi \vdash \texttt{z} : \sigma}\,(\textsc{var})}{\Pi \vdash \texttt{this.y.app(z)} : \tau}\,(\textsc{invk})}{\Pi \vdash \texttt{this.x.app(z).app(this.y.app(z))} : \mu}\,(\textsc{invk})$$

$$\vdots$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi' \vdash \texttt{this} : \langle \texttt{x}{:}\sigma_1\rangle}\,(\textsc{var})}{\Pi' \vdash \texttt{this.x} : \sigma_1}\,(\textsc{fld}) \quad \dfrac{}{\Pi' \vdash \texttt{y} : \sigma_2}\,(\textsc{var})}{\Pi' \vdash \texttt{new S}_2\texttt{(this.x,y)} : \langle \texttt{x}{:}\sigma_1\rangle}\,(\textsc{newF}) \quad \dfrac{\dfrac{\dfrac{}{\Pi' \vdash \texttt{this} : \langle \texttt{x}{:}\sigma_1\rangle}\,(\textsc{var})}{\Pi' \vdash \texttt{this.x} : \sigma_1}\,(\textsc{fld}) \quad \dfrac{}{\Pi' \vdash \texttt{y} : \sigma_2}\,(\textsc{var})}{\Pi' \vdash \texttt{new S}_2\texttt{(this.x,y)} : \langle \texttt{y}{:}\sigma_2\rangle}\,(\textsc{newF})}{\Pi' \vdash \texttt{new S}_2\texttt{(this.x,y)} : \langle \texttt{x}{:}\sigma_1,\texttt{y}{:}\sigma_2\rangle}\,(\textsc{rec})}{\Pi' \vdash \texttt{new S}_2\texttt{(this.x,y)} : \langle \texttt{app}{:}(\sigma){\to}\mu\rangle}\,(\textsc{newM}) \quad \dfrac{\dfrac{}{\texttt{x}{:}\sigma_1 \vdash \texttt{x} : \sigma_1}\,(\textsc{var})}{\texttt{x}{:}\sigma_1 \vdash \texttt{new S}_1\texttt{(x)} : \langle \texttt{x}{:}\sigma_1\rangle}\,(\textsc{newF})$$

$$\vdots$$

$$\dfrac{\dfrac{\texttt{x}{:}\sigma_1 \vdash \texttt{new S}_1\texttt{(x)} : \langle \texttt{app}{:}(\sigma_2){\to}\langle \texttt{app}{:}(\sigma){\to}\mu\rangle\rangle}{\emptyset \vdash \texttt{new S()} : \langle \texttt{app}{:}(\sigma_1){\to}\langle \texttt{app}{:}(\sigma_2){\to}\langle \texttt{app}{:}(\sigma){\to}\mu\rangle\rangle\rangle}\,(\textsc{newM})}\,(\textsc{newM})$$

Figure 7: Derivation schemes for the translations of **S** and **K**

as Curry types) is defined by the following grammar:

$$A, B \;::=\; \varphi \mid A \to B$$

ii) A *basis* Γ 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.

iii) Simple types are assigned to CL-terms using the following natural deduction 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 this approach is that we can now link types assigned to combinators to types assignable to object-oriented programs. To show this type preservation, we need to define what the equivalent of Curry's types are in terms of our FJ$^{\not\in}$ types.

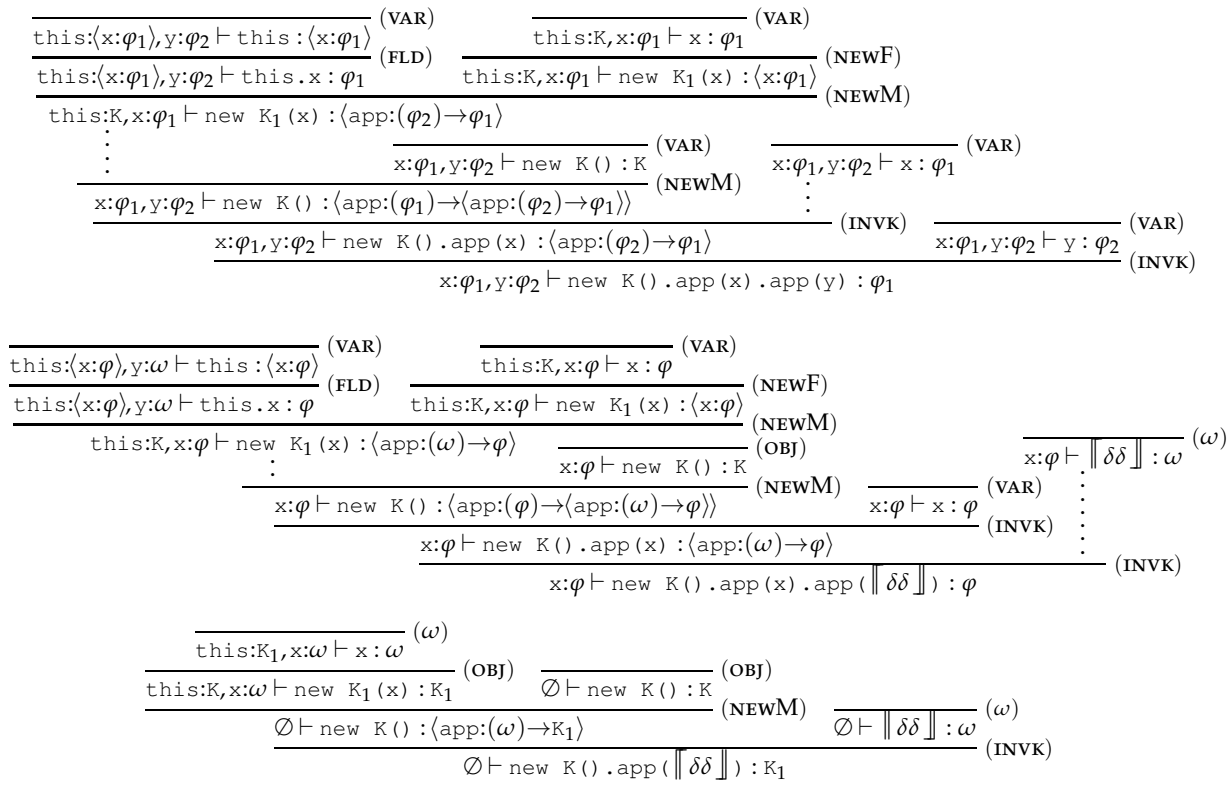
Definition 6.6 (TYPE TRANSLATION) The function $\lceil\!\lceil \cdot \rfloor\!\rfloor$, which transforms Curry types,[10] is defined as follows:

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

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 result.

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



where δ 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.

Figure 8: Derivations for Example 6.12

Theorem 6.7 (Preservation of Types) *If* $\Gamma \vdash_{\mathrm{CL}} t\colon A$ *then* $\lceil\lceil \Gamma \rfloor\rfloor \vdash \llbracket t \rrbracket : \lceil\lceil A \rfloor\rfloor$.

Proof: By induction on the derivation of $\Gamma \vdash_{\mathrm{CL}} t\colon A$. The cases for (var) and $(\to E)$ are trivial. For the rules (**K**) and (**S**), Figure 7 gives derivation schemas for assigning the translation of the respective Curry type schemes to the OOCL translations of **K** and **S**. □

Notice that, in the nominal system, we can at most show $\vdash$ `new K() : K` and $\vdash$ `new S() : S`, and that those types do not express an applicative character.

Furthermore, since Curry's well-known translation of the simply typed LC into CL preserves typeability (see [16]), we can also construct a type-preserving encoding of LC into $\mathrm{FJ}^{\not\in}$; 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. Indeed, our type system actually has more power than intersection type systems for CL as presented in [16], since there not all normal forms are typeable using strict types, whereas in our system they are; this is mainly because we can assign types to encoded terms that do not correspond to encoded types.

First we will illustrate our termination results by applying them in the context of OOCL.

Definition 6.8 (OOCL normal forms) Let the set of OOCL-normal forms be the set of expressions

$$\{\, e \mid \text{there exists a CL-term } t \text{ such that } e \text{ is the normal form of } \llbracket t \rrbracket \,\}$$

Notice that OOCL-normal forms can be defined by the following grammar:

$$n ::= x \mid \texttt{new K()} \mid \texttt{new K}_1\texttt{(}n\texttt{)} \mid \texttt{new S()} \mid \texttt{new S}_1\texttt{(}n\texttt{)} \mid \texttt{new S}_2\texttt{(}n_1\texttt{,}n_2\texttt{)} \mid n\texttt{.app(}n'\texttt{)} \quad (n \neq \texttt{new C(}\vec{e}_n\texttt{)})$$

Each OOCL normal form corresponds to a CL normal form, the translation of which can also by typed with an ω-safe derivation for each type assignable to the normal form.

Lemma 6.9 If e is an OOCL *normal form, then there exists a* CL *normal form t such that $[\![t]\!] \to^* e$ and for all ω-safe $\mathcal{D}$ and Π such that $\mathcal{D} :: \Pi \vdash e : \sigma$, there exists an ω-safe derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi \vdash [\![t]\!] : \sigma$.*

Proof: By induction on the structure of OOCL normal forms. □

We can also show that ω-safe typeability is preserved under expansion for the images of CL-terms in OOCL.

Lemma 6.10 Let t_1 and t_2 be CL*-terms such that $t_1 \to t_2$; if there is an ω-safe derivation $\mathcal{D}$ and environment Π, and a strict type σ such that $\mathcal{D} :: \Pi \vdash [\![t_2]\!] : \sigma$, then there exists another ω-safe derivation $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi \vdash [\![t_1]\!] : \sigma$.*

Proof: By induction on the definition of reduction for CL. □

This property of course also extends to multi-step reduction.

Together with the lemma preceding it (and the fact that all normal forms can by typed with an ω-safe derivation), this leads to both a sound and *complete* characterisation of normalisability for the images of CL-terms in OOCL.

Theorem 6.11 *Let t be a* CL*-term: then t is normalisable, if and only if, there are ω-safe $\mathcal{D}$ and Π, and strict type σ such that $\mathcal{D} :: \Pi \vdash [\![t]\!] : \sigma$.*

Proof: (*if*): Directly by Theorem 5.9.

(*only if*): Let t' be the normal form of t; then, by Theorem 6.4, $[\![t]\!] \to [\![t']\!]$. Since reduction in CL is confluent, $[\![t']\!]$ is normalisable as well; let n be the normal form of $[\![t']\!]$. Then by Lemma 5.7(2) there are strong strict type σ, environment Π and derivation $\mathcal{D}$ such that $\Pi \vdash n : \sigma$. Since $\mathcal{D}$ and Π are strong, they are also ω-safe. Then, by Lemma 6.9 and 6.10, there exists ω-safe $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi \vdash [\![t]\!] : \sigma$. □

To conclude this section, we give some example derivations of OOCL programs that demonstrate these results.

Example 6.12 Figure 8 shows, respectively,

- a strong derivation typing a strongly normalising expression of OOCL;
- an ω-safe derivation of a normalising (but not strongly normalising) expression of OOCL; and
- a non-ω-safe derivation deriving a non-trivial type for a head-normalising (but not normalising) OOCL expression,

7 Some Worked Examples

We will now give a more concrete idea of the concepts outlined above, by giving a couple of examples. The first is based upon the familiar concept of a fixed-point combinator from the world of functional programming: we will show how a simple yet non-trivial type can be derived for our construction, and then demonstrate how this derivation reduces to a normal form whose structure directly corresponds to an approximant of the original term. The second example is actually a non-example demonstrating how a non-terminating program (*i.e.* one having no approximants other than $\perp$) is not typeable. The third will show that, in our system, we catch the 'message not understood' run-time error.

$$\mathcal{D}_1 :: \dfrac{\dfrac{\dfrac{\overline{\Pi_2 \vdash \texttt{x} : \langle \texttt{app}{:}(\omega)\to\varphi\rangle}\ (\textsc{var}) \quad \overline{\Pi_2 \vdash \texttt{this.app(x)} : \omega}\ (\omega)}{\Pi_2 \vdash \texttt{x.app(this.app(x))} : \varphi}\ (\textsc{invk})}{\Pi_1 \vdash \texttt{new T()} : \langle \texttt{app}{:}(\langle \texttt{app}{:}(\omega)\to\varphi\rangle)\to\varphi\rangle}\ (\textsc{newM}') \quad \overline{\Pi_1 \vdash \texttt{z} : \langle \texttt{app}{:}(\omega)\to\varphi\rangle}\ (\textsc{var})}{\Pi_1 \vdash \texttt{new T().app(z)} : \varphi}\ (\textsc{invk})$$

$$\mathcal{D}_2 :: \dfrac{\overline{\Pi_1 \vdash \texttt{z} : \langle \texttt{app}{:}(\omega)\to\varphi\rangle}\ (\textsc{var}) \quad \overline{\Pi_1 \vdash \texttt{new T().app(z)} : \omega}\ (\omega)}{\Pi_1 \vdash \texttt{z.app(new T().app(z))} : \varphi}\ (\textsc{invk})$$

$$\mathcal{D}_3 :: \dfrac{\overline{\Pi_1 \vdash \texttt{z} : \langle \texttt{app}{:}(\omega)\to\varphi\rangle}\ (\textsc{var}) \quad \overline{\Pi_1 \vdash \bot : \omega}\ (\omega)}{\Pi_1 \vdash \texttt{z.app}(\bot) : \varphi}\ (\textsc{invk})$$

$\Pi_1 = \texttt{z}{:}\langle \texttt{app}{:}(\omega)\to\varphi\rangle,\ \ \Pi_2 = \texttt{x}{:}\langle \texttt{app}{:}(\omega)\to\varphi\rangle$

Figure 9: Type Derivations for the Fixed-Point Construction Example

7.1 A Fixed-point Construction

The *fixed point* of a function F is a term M such that $M = F(M)$; a fixed-point *combinator* is a (higher-order) function that returns a fixed-point of its argument (another function). Thus, a fixed-point combinator F has the property that $F f = f(F f)$ for any function f. Turing's well-known fixed-point combinator in LC is the following term:

$$Tur \ = \ \Theta\Theta \ = \ (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

That *Tur* provides a fixed-point constructor is easy to check:

$$Tur\ M \ = \ (\lambda xy.y(xxy))\Theta M \ \to_\beta^* \ M(\Theta\Theta M) \ = \ M(Tur\ M)$$

Tur itself has the reduction behaviour

$$\begin{aligned} Tur = (\lambda xy.y(xxy))\Theta \ &\to_\beta \ \lambda y.y(\Theta\Theta y) \\ &\to_\beta \ \lambda y.y((\lambda z.z(\Theta\Theta z))y) \\ &\to_\beta \ \lambda y.y(y(\Theta\Theta y)) \\ &\to_\beta \ \lambda y.y(y(y(\Theta\Theta y))) \\ &\ \ \vdots \end{aligned}$$

which implies it has the following set of approximants:

$$\{\bot,\ \lambda y.y\bot,\ \lambda y.y(y\bot),\ \lambda y.y(y(y\bot)),\ \ldots\}$$

Thus, if z is a term variable, the approximants of *Tur* z are $\bot, z\bot, z(z\bot)$, etc. Based on this, it is straightforward to define an $\textbf{FJ}^{\not\in}$ program which mirrors this behaviour:

```
class T extends Combinator {
    combinator app(Combinator x) { return x.app(this.app(x)); }
}
```

The body of the `app` method in the class `T` encodes the reduction behaviour we saw for *Tur* above: for any $\textbf{FJ}^{\not\in}$ expression `e` we have:

```
new T().app(e)  →  e.app(new T().app(e))
```

So, taking $t =$ `new T().app(e)`, we have $t \ \to$ `e.app(`t`)`. Thus, by Theorem 2.8, the fixed point t of `e` (as returned by the fixed-point combinator class `T`) is semantically equivalent to `e.app(`t`)`, and so `new T().app(`$\cdot$`)` does indeed represent a fixed-point constructor.

The (executable) expression `new T().app(z)` has the reduction behaviour

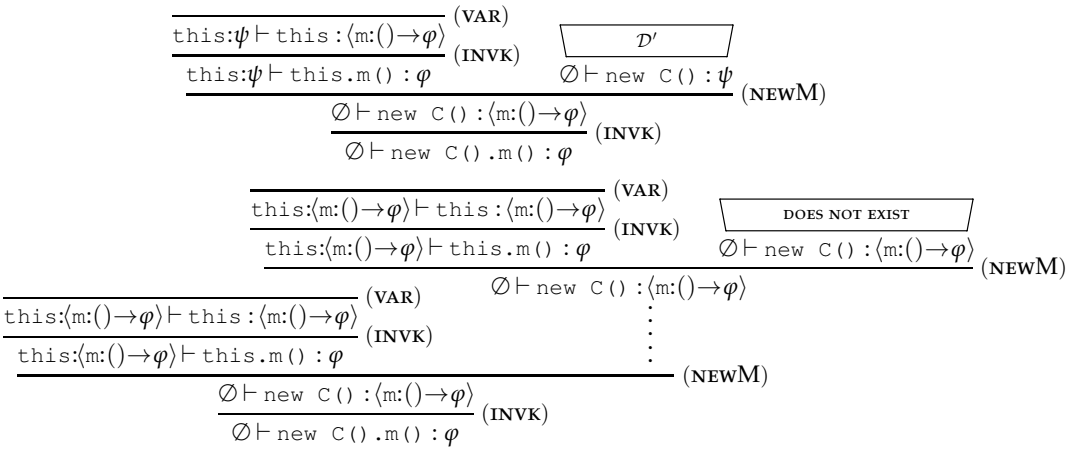


Figure 10: Type Derivations for a Non-Terminating Program

$$\begin{aligned} \texttt{new T().app(z)} &\rightarrow \texttt{z.app(new T.app(z))} \\ &\rightarrow \texttt{z.app(z.app(new T.app(z)))} \\ &\vdots \end{aligned}$$

so has the following (infinite) set of approximants:

$$\{\bot, \texttt{z.app(}\bot\texttt{)}, \texttt{z.app(z.app(}\bot\texttt{))}, \ldots\}$$

Notice that these exactly correspond to the set of the approximants for the λ-term *Tur* z that we considered above.

$\mathcal{D}_1$ in Figure 9 shows a possible derivation assigning the type φ to `new T().app(z)`. In fact, the normal form of this derivation corresponds to the approximant `z.app(`$\bot$`)`. Observe that the derivation $\mathcal{D}_1$ comprises a *typed redex*, in this case a derivation of the form $\langle\langle\cdot,\cdot,\textsc{newm}\rangle,\vec{\cdot},\textsc{invk}\rangle$, thus it will reduce, creating the derivation $\mathcal{D}_2$. This is now in *normal form* since although the expression that it types still contains a redex, that redex is covered by ω and so no further (derivation) reduction can take place there. The structure of this derivation therefore dictates the structure of an approximant of `e`: the approximant is formed by replacing all sub-expressions typed with ω by the element $\bot$. When we do this, we obtain the derivation $\mathcal{D}_3$.

Although this example is relatively simple (we chose the derivation corresponding to the simplest non-trivial approximant), it does demonstrate the central concepts involved in the approximation theorem.

7.2 A program without head-normal form

We now examine how the type system deals with programs that do not have a head-normal form. The approximation theorem states that any type which we can assign to an expression is also assignable to an approximant of that expression. As we mentioned in Section 2, approximants are snapshots of evaluation: they represent the information computed during evaluation. But by their very nature, programs which do not have a head-normal form do not compute any information as they have no observable behaviour. Formally, then, the characteristic property of expressions without a head-normal form is that they do *not* have non-trivial approximants: their only approximant is $\bot$. From the approximation result therefore follows that we cannot build any derivation for these expressions that assigns a type other than ω (since that is the only type assignable to $\bot$).

To illustrate this, consider the following program 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*:

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

so, in particular, `new C().m()` has no normal form, not even a head-normal form.

Figure 10 shows two candidate derivations assigning a non-trivial type to the expression `new C().m()`, the first of which we can more accurately call a derivation *schema* since it specifies the form that any such derivation must take. The second derivation of Figure 10 is an attempt at instantiating the schema that we have just constructed, which clearly fails: the requirements for this derivation to exist is that it is identical to a proper sub-derivation, which is impossible. Notice however, that the receiver `new C()` itself *is* a head normal form – indeed, it is a normal form – and so we *can* assign to it a non-trivial type: using the (OBJ) rule, $\emptyset \vdash$ `new C():C`.

7.3 Cops and Cars

To give the reader a more intuitive understanding of both the differences and advantages of our approach over the conventional nominal approach to object-oriented static analysis (as exemplified in Featherweight Java), we will now consider an example which presents certain challenges to the nominal approach, but is handled by our type system naturally since it is a *semantics*-based one.

We will model a situation involving cars and drivers so we write classes `Car` and `Driver`; we will focus on a single aspect: the action of the driver starting the car. For our purposes, we will assume that a car is started when its driver turns the ignition key and so the classes `Car` and `Driver` might contain the following code:

```
class Car {
    Driver driver;
 ⋮
    Car start() { return this.driver.turnIgnition(this); }
}

class Driver {
 ⋮
    Car turnIgnition(Car c) { return c; }
}
```

Since we are working with a featherweight model of the language, we have had to abstract away some detail and are subject to certain restrictions. For instance, since in Featherweight Java we do not have a `void` return type, we return the `Car` object itself from the `start` and `turnIgnition` methods.

We define a special type of car - a *police* car: it may chase other cars, however in order to do so the police officer driving the car must report to the headquarters. Thus, only police officers may initiate car chases. We write a `PoliceCar` class that *extends* `Car` and make the `Cop` class extend `Driver` so that police officers are capable of driving cars (including police cars). Here we run into a problem, however: the nominal approach imposes that when we override method definitions we must use the *same type signature* (we are not allowed to specialise or change the argument or return types, nor are we allowed to specialise the types of fields that

are inherited[11]). Thus, we must define our new classes as follows:

```
class PoliceCar extends Car {
    PoliceCar chaseCar(Car c) { return this.driver.reportChase(this); }
 ⋮
}

class Cop extends Driver {
 ⋮
    PoliceCar reportChase(PoliceCar c) { return c; }
}
```

Before considering the type safety of our extra classes, let us examine their behaviour from a purely operational point of view. As desired, a police car driven by a police officer is able to chase another car (the method invocation results in an object, *i.e.* a well-formed normal form):

```
new PoliceCar(new Cop()).chaseCar(new Car(new Driver()))
    → new PoliceCar(new Cop()).driver.reportChase(new PoliceCar(new Cop()))
    → new Cop().reportChase(new PoliceCar(new Cop()))
    → new PoliceCar(new Cop())
```

However, if a police car driven by a *Driver* attempts to chase a car we run into trouble:

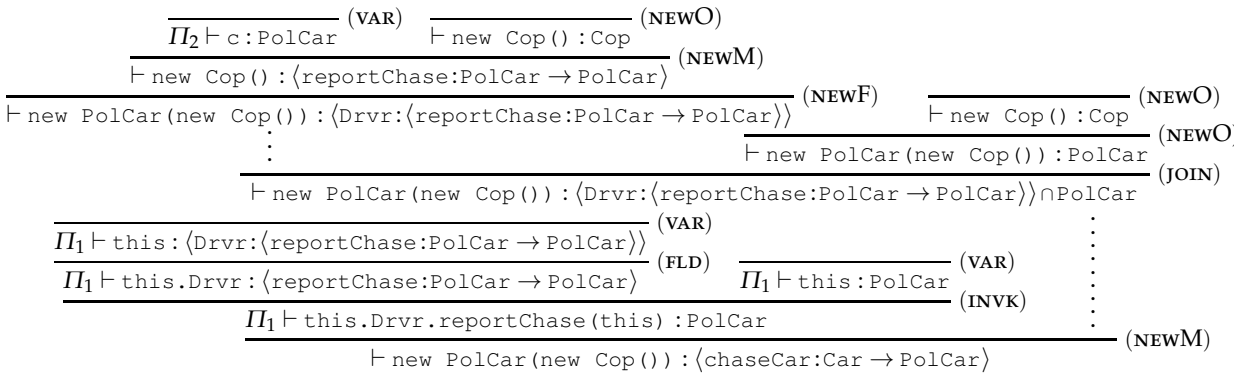
```
new PoliceCar(new Driver()).chaseCar(new Car(new Driver()))
    → new PoliceCar(new Driver()).driver.reportChase(new PoliceCar(new Driver()))
    → new Driver().reportChase(new PoliceCar(new Driver()))
```

Here, we get stuck trying to invoke the `reportChase` method on a `Driver` object since the `Driver` class does not contain such a method. This is the infamous '*message not understood*' error.

The nominal approach to static type analysis is twofold: firstly, to *ensure* that the values assigned to the fields of an object match their declared type; and then secondly, to enforce within the bodies of the methods that the fields are used in a way consistent with their declared type. Thus, while it is type safe to assign a `Cop` object to the `driver` field of a `PoliceCar` (since `Cop` is a subtype of `Driver`), trying to invoke the `reportChase` method on the `driver` field in the body of the `chaseCar` method is *not* type safe since such an action is not consistent with the declared type (`Driver`) of the `driver` field. In such a situation, where a method body uses a field inconsistently, the nominal approach is to brand the entire class unsafe and prevent any instances being created. Thus, in Featherweight Java (as in full Java), the *subexpression* `new PoliceCar(new Driver())` is not well-typed, consequently entailing that the full expression `new PoliceCar(new Driver()).chaseCar(new Car(new Driver()))` is not well typed.

This leaves us in an uncomfortable position, since we have seen that *some* instances of the `PoliceCar` class (namely, those that have `Cop` drivers) are perfectly safe, and thus preventing us from creating any instances at all seems a little heavy-handed. There are two solutions to this problem. The first is to rewrite the `PoliceCar` and `Cop` classes so that they do *not* extend the classes `Car` and `Driver`. That way, we are free to specify the constructor (and any setter methods) to take an argument of `Cop`. However, this would mean having to *reimplement* all the functionality of `Car` and `Driver`. The other solution is to use *casts*: in the body of the `chaseCar`

[11] The full Java language allows fields to be declared in a subclass with the same name as fields that exists in the superclasses, however the semantics of this construction is that a *new* field is created which *hides* the previously declared field; while this serves to mitigate the specific problem we are discussing here, it does introduce its own new problems.



where $\Pi_1 = \{\texttt{this} : \langle \texttt{Drvr}:\langle \texttt{reportChase}:\texttt{PolCar} \to \texttt{PolCar}\rangle\rangle \cap \texttt{PolCar}, \texttt{c} : \texttt{Car}\}$
$\Pi_2 = \{\texttt{this} : \texttt{Cop}, \texttt{c} : \texttt{PolCar}\}$

Figure 11: Typing derivation for the `chaseCar` method of a `PolCar` object with a `Cop` Drvr.

method we cast the `driver`, telling the type system that it is safe to consider the `driver` field to be of type `Cop`:

```
class PolicarCar extends Car {
 ⋮
    PolicarCar chaseCar(Car c) { return ((Cop) this.driver).reportChase(this); }
}
```

Now, the `PolicarCar` class is type safe: we can create instances of it and `PolicarCar` objects with `Cop` drivers can chase cars:

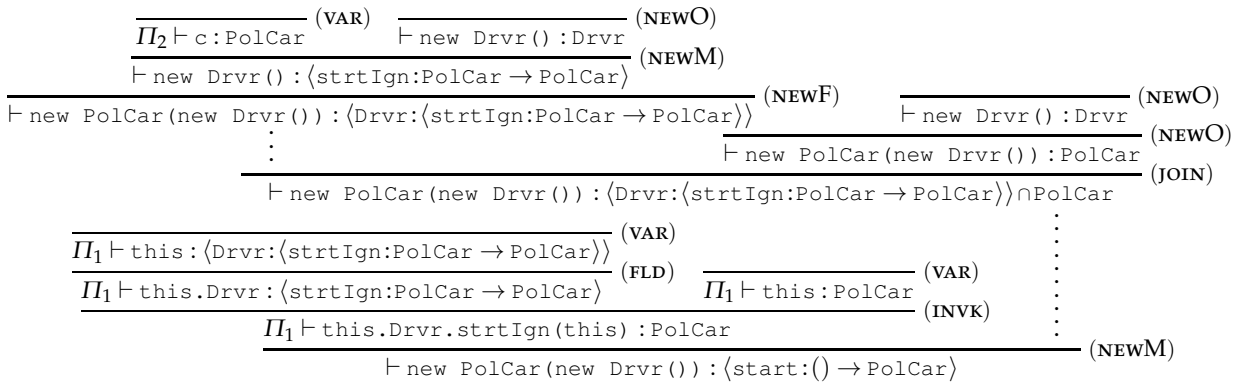
```
new PolicarCar(new Cop()).chaseCar(new Car(new Driver()))
  → ((Cop) new PolicarCar(new Cop()).driver).reportChase(new PolicarCar(new Cop()))
  → ((Cop) new Cop()).reportChase(new PolicarCar(new Cop()))
  → new Cop().reportChase(new PolicarCar(new Cop()))
  → new PolicarCar(new Cop())
```

However, we are not entirely home and dry, since to regain type soundness in the presence of casts we now have to *check at run-time* that the cast is valid:

```
new PolicarCar(new Driver()).chaseCar(new Car(new Driver()))
   → ((Cop) new PolicarCar(new Driver()).driver)
                          .reportChase(new PolicarCar(new Driver()))
   → ((Cop) new Driver()).reportChase(new PolicarCar(new Driver()))
```

As the above reduction sequence shows, the 'message not understood' error from before has merely been *transformed* into a run-time 'cast exception' which occurs when we try to cast the `new Driver()` object to a `Cop` object. Using the nominal approach to static typing, we are forced to choose the 'lesser of many evils', as it were: being unable to write typeable programs that implement what we desire; being unable to share implementations between classes; or having to allow some run-time exceptions (albeit only with the explicit permission of the programmer). We should point out here that some other solutions to this particular problem have been proposed in the literature (see the work on family polymorphism [39, 49]), but these solutions persist in the nominal typing approach and can thus only be achieved by extending the language itself.

The $\text{FJ}^{\not\in}$ intersection type system we have presented in this paper has two main characteristics that distinguish it from the traditional (nominal) type systems for object-orientation: *i)*



$$\textit{where } \Pi_1 = \{\texttt{this} : \langle \texttt{Drvr}:\langle \texttt{strtIgn}:\texttt{PolCar} \to \texttt{PolCar}\rangle\rangle \cap \texttt{PolCar}\}$$
$$\Pi_2 = \{\texttt{this} : \texttt{Drvr}, \texttt{c} : \texttt{PolCar}\}$$

Figure 12: Typing derivation for the `start` method of a `PolCar` object with a `Drvr` driver.

our types are structural and so provide a fully functional analysis of the behaviour of objects; ii) we keep the analysis of methods and fields independent *from one another, allowing for a fine-grained analysis.* This means that not all methods *need* be typeable - we do not reject instances of a class as ill-typed simply because they cannot satisfy *all* of the interface specified by the class (in terms of being able to *safely* - in a semantic sense - invoke all the methods). In other words, if we cannot assign a type to any particular method body from a given class, then this does not prevent us from creating instances of the class if other methods may be safely invoked and typed.

In Figure 11 we can see a typing derivation in our system that assigns a type for the `chaseCar` method to a `PoliceCar` object with `Cop` driver. Now consider replacing the `Cop` object in this derivation with a `Driver` object, as we would have to do if we wanted to try and assign this type to a `PoliceCar` object with a `Driver` driver. In doing so, we would run into problems since we would ultimately have to assign a type for the `reportChase` method to the driver (as has been done in the topmost subderivation in Figure 11) - an obviously impossible task seeing as no such method exists in the `Driver` class. This does not mean however that we should not be able to create such `PoliceCar` objects. After all, `PoliceCar`s are supposed to behave in all other respects as ordinary cars, so perhaps we might want ordinary `Driver`s to be able to use them as such. In Figure 12 we can see a typing derivation assigning a type for the `start` method to a `PoliceCar` object with a `Driver` driver, showing that this is indeed possible. Notice that this is also sound from an operational point of view:

```
new PoliceCar(new Driver()).start()
  → new PoliceCar(new Driver()).driver.turnIgnition(new PoliceCar(new Driver()))
  → new Driver().turnIgnition(new PoliceCar(new Driver()))
  → new PoliceCar(new Driver())
```

The second characteristic is that our type system is a true type *inference* system - that is, no type annotations are required in the program itself in order for the type system to verify its correctness.[12] In the type *checking* approach, the programmer specifies the type that their program must satisfy. As our example shows, this can sometimes lead to inflexibility: in

[12] It is true that our calculus retains class type annotations, however this is a syntactic legacy due to the fact that we would like our calculus to be considered a true sibling of Featherweight Java, and nominal class type no longer constitute true types in our system.

some cases, multiple types may exist for a given program (as in a system without finitely representable principal types) and then the programmer is forced to choose just one of them; in the worst case, a suitable type may not even be expressible in the language. This is the case for our nominally typed cars example: the same `PoliceCar` class may give rise to objects which behave differently depending on the particular values assigned to their fields; this should be expressed through multiple different typings, however in the nominal system there is no way to express them. Our system does not force the programmer to choose a type for the program, thus retaining flexibility. Moreover, since our system is semantically complete, all safe behaviour is typeable and so it provides the *maximum* flexibility possible. Lastly, and more importantly, we have achieved this result without having to extend the programming language in any way.

7.4 Some Observations

In this paper we have shown how the ITD approach can be applied to class-based OO, preserving the main expected properties of intersection type systems. There are however some notable differences between our type system and previous work on LC and TRS upon which our research is based.

Firstly, we point out that when considering the encoding of CL (and via that, LC) in $\mathbf{FJ}^{\not\in}$, our system provides *more* than the traditional analysis of terms as functions: there are untypeable LC and CL-terms which have typeable images in OOCL.

Example 7.1 Let δ be the CL-term $\mathbf{S}\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})$. Notice that $\delta\delta \to^* \delta\delta$, *i.e.* has no head-normal form, and thus can only be given the type ω (this is also true for $[\![\delta\delta]\!]$). Now, consider the term $t = \mathbf{S}\ (\mathbf{K}\ \delta)\ (\mathbf{K}\ \delta)$. Notice that it is a normal form ($[\![t]\!]$ has a normal form also), but that for any term t', $\mathbf{S}\ (\mathbf{K}\ \delta)\ (\mathbf{K}\ \delta)\ t' \to^* \delta\delta$. In a strict system, no functional analysis is possible for t since $\phi \to \omega$ is not a type and so we can only type t with ω.[13]

In our type system however, we may assign several forms of type to $[\![t]\!]$. Most simply, we can derive $\emptyset \vdash [\![t]\!] : \mathtt{S}_2$, but even though a 'functional' analysis via the `app` method is impossible, it is still safe to access the fields of the object resulting from running $[\![t]\!]$ – both $\emptyset \vdash [\![t]\!] : \langle \mathtt{x}{:}\mathtt{K}_1 \rangle$ and $\emptyset \vdash [\![t]\!] : \langle \mathtt{y}{:}\mathtt{K}_1 \rangle$ are also easily derivable statements. In fact, we can derive even more informative types: the expression $[\![\mathbf{K}\ \delta]\!]$ can be assigned types of the form $\sigma_{\mathbf{K}\delta} = \langle \mathtt{app}{:}(\sigma_1) \to \langle \mathtt{app}{:}(\sigma_2 \cap \langle \mathtt{app}{:}(\sigma_2) \to \sigma_3 \rangle) \to \sigma_3 \rangle \rangle$, and so we can also assign $\langle \mathtt{x}{:}\sigma_{\mathbf{K}\delta} \rangle$ and $\langle \mathtt{y}{:}\sigma_{\mathbf{K}\delta} \rangle$ to $[\![t]\!]$. Notice that the λ-term equivalent to t is $\lambda y.(\lambda x.xx)(\lambda x.xx)$, which is a *weak* normal form without a head-normal form. The 'functional' view is that such terms are observationally indistinguishable from terms without head-normal form. When encoded in $\mathbf{FJ}^{\not\in}$ however, our type system shows that these terms become meaningful (head-normalisable).

The second observation concerns *principal* types. In LC, each normal form has a *unique* most-specific type: *i.e.* a principal type from which all the other assignable types may be generated (this property is important for practical type *inference*). It is not clear if our intersection type system for $\mathbf{FJ}^{\not\in}$ does enjoy such a property. Consider the following program:

```
class D extends Object {
    D m() { return new D(); }
}
```

[13] In other intersection type systems (*e.g.* [21]) $\phi \to \omega$ is a permissible type, but is equivalent to ω (that is $\omega \leq (\phi \to \omega) \leq \omega$) and so semantics based on these type systems identify terms of type $\phi \to \omega$ with terms that do not have a head-normal form.

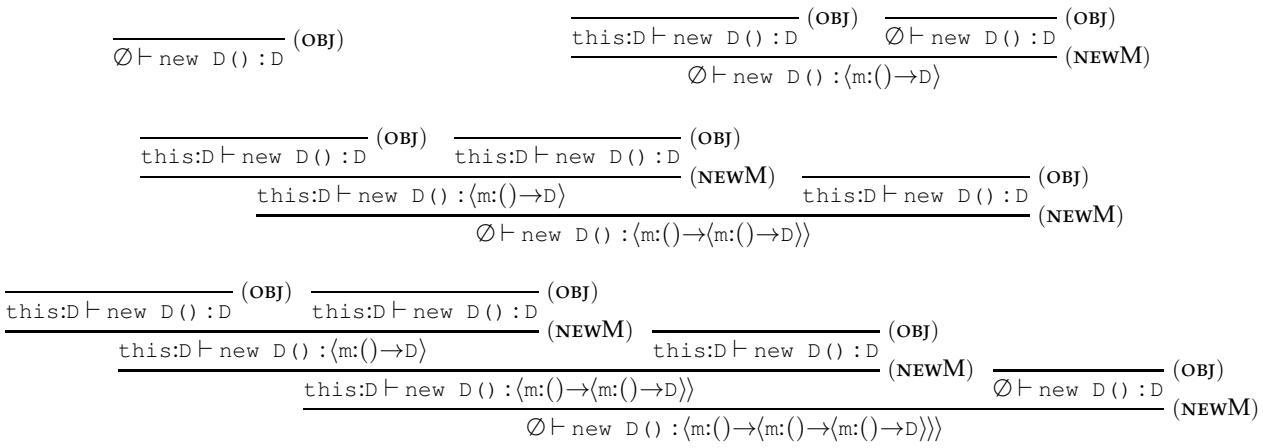


Figure 13: Type Derivations for a Program without a Principal Type

The expression `new D()` is a normal form, and so we can assign it a non-trivial type, but observe that the set of all types which may be assigned to this expression is the *infinite* set $\{\mathtt{D}, \langle \mathtt{m}:() \to \mathtt{D}\rangle, \langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \mathtt{D}\rangle\rangle, \ldots\}$, as illustrated in Figure 13.[14] None of these types may be considered the *most* specific one, since whichever type we pick we can always derive a more informative (larger) one. On the one hand, this is exactly what we want: we may make a series of any finite number of calls to the method `m` and this is expressed by the types. On the other hand, this seems that a practical type inference for our system will not be straightforwardly defined. Notice however that these types are not unrelated to one another: they each approximate the 'infinite' type $\langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \ldots\rangle\rangle$, which can be finitely represented by the recursive type $\mu X.\langle \mathtt{m}:() \to X\rangle$. This type concisely captures the reduction behaviour of `new D()`, showing that when we invoke the method `m` on it we again obtain our original term. In LC such families of types arise in connection with fixed-point operators. This is not a coincidence: the class `D` was *recursively* defined, and in the face of such self-reference it is then not surprising that this is reflected in our type analysis.

Conclusions & Future Work

We have considered an approximation-based denotational semantics for class-based OO-programs and related this to a type-based semantics defined using an intersection type approach. Our work shows that the techniques and strong results of this approach can be transferred straightforwardly from other programming formalisms (LC and TRS) to the OO-paradigm. Through our characterisation results we have shown that our type system is powerful enough (at least in principle) to form the basis for expressive analyses of OO-programs.

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. Since a class type represents the entire interface defined in the class declaration, the programmer acts on the assumption that they may safely call any method within this interface. Consequently, to keep up their end of the 'bargain', the programmer is under an obligation to ensure that the value returned by their program safely provides the *whole* interface of its declared type.

[14] That principal types can be infinitely large is also the case in LC, typically for terms with an infinite number of approximants (like a fixed-point combinator).

In the approach suggested by our type system, by firstly removing the requirement to safely implement a full collection of methods regardless of the input values, the programmer is afforded a certain expressive freedom. Secondly, while they can no longer rely on the fact that all objects of a given class provide a particular interface, this apparent problem is obviated by 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 point out that our type system is not the only type system for oo in the literature with these characteristics: for example, the work of Palsberg for the ς-calculus, who showed decidable type inference [57], and that of Eifrig, Smith and Trifonov [38, 37]. But our system is, we believe, the first such system which is faithful to a semantic model of the language, and this is the main contribution of our work.

The case for the nominal type checking approach, based as it is on providing sound, decidable static analyses is a strong one. Our full semantic system is obviously undecidable but we believe that decidable restrictions of our system exist which could give it the edge over current approaches.

Our work has also highlighted where the oo-programming style differs from its functional cousin. In particular, we have noted that because of oo's facility for *self-reference*, it is no longer clear if all normal forms have a most specific (or principal) type. The types assignable to such normal forms do however seem to be representable using recursive definitions. This observation further motivates and strengthens the case (by no means a new concept in the analysis of oo) for the use of recursive types in this area. Some recent work by Nakano [56] shows that a restricted but still highly expressive form of recursive types can still guarantee head normalisation, and we hope to fuse this approach with our own to come to an equally precise but more concise and practical type-based treatment of oo.

We would also like to 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 $\lambda\mu$ [58], it seems natural to extend our simply typed system to analyse the exception handling features of Java.

References

[1] *The C# Language Specification (ECMA-334), 4th Edition*. ECMA International, June 2006.

[2] *ECMA Language Specification (ECMA-262)*. ECMA International, June 2011.

[3] M. Abadi and L. Cardelli. A Semantics of Object Types. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 332–341. IEEE Computer Society Press, 1994.

[4] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[5] J. Alves-Foss and F.S. 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 Verlag, 1999.

[6] S. van Bakel. Complete restrictions of the Intersection Type Discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[7] S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 151(2):385–435, 1995.

[8] S. van Bakel. Cut-Elimination in the Strict Intersection Type Assignment System is Strongly Normalising. *Notre Dame journal of Formal Logic*, 45(1):35–63, 2004.

[9] S. van Bakel. The Heart of Intersection Type Assignment; Normalisation proofs revisited. *Theoretical Computer Science*, 398:82–94, 2008.

[10] 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.
[11] S. van Bakel. Strict intersection types for the Lambda Calculus. *ACM Computing Surveys*, 43:20:1–20:49, April 2011.
[12] S. van Bakel. Completeness and Soundness results for $\mathcal{X}$ with Intersection and Union Types. *Fundamenta Informaticae*, 121:1–41, 2012.
[13] S. van Bakel and U. de'Liguoro. Logical equivalence for subtyping object and recursive types. *Theory of Computing Systems*, 42(3):306–348, 2008.
[14] S. van Bakel and M. Fernández. Strong Normalisation of Typeable Rewrite Systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Proceedings of HOA'93. First International Workshop on Higher Order Algebra, Logic and Term Rewriting,* Amsterdam, the Netherlands. *Selected Papers*, volume 816 of *Lecture Notes in Computer Science*, pages 20–39. Springer Verlag, 1994.
[15] S. van Bakel and M. Fernández. Normalisation Results for Typeable Rewrite Systems. *Information and Computation*, 2(133):73–116, 1997.
[16] S. van Bakel and M. Fernández. Normalisation, Approximation, and Semantics for Combinator Systems. *Theoretical Computer Science*, 290:975–1019, 2003.
[17] S. van Bakel and P. Lescanne. Computation with Classical Sequents. *Mathematical Structures in Computer Science*, 18:555–609, 2008.
[18] S. van Bakel and R. Rowe. Semantic Predicate Types for Class-based Object Oriented Programming. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs (FTfJP'09)*, 2009. Article No. 3.
[19] A. Banerjee and T.P. Jensen. Modular Control-Flow Analysis with Rank 2 Intersection Types. *Mathematical Structures in Computer Science*, 13(1):87–124, 2003.
[20] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.
[21] 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.
[22] K.B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.
[23] K.B. Bruce, L. Cardelli, and B.C. Pierce. Comparing Object Encodings. *Information and Computation*, 155(1-2):108–133, 1999.
[24] M.T. Burt. *Games, Call-by-Value and Featherweight Java*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, England, 2004.
[25] L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen, and G.D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer Verlag, June 27-29 1984.
[26] L. Cardelli and J.C. Mitchell. Operations on Records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991.
[27] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.
[28] A. Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.
[29] W.R. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In G. Bosworth, editor, *Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89)*, pages 433–443, New Orleans, Louisiana, USA, 1989. ACM.
[30] W.R. Cook and J. Palsberg. A Denotational Semantics of Inheritance and Its Correctness. *Information and Computation*, 114(2):329–350, 1994.
[31] M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame journal of Formal Logic*, 21(4):685–693, 1980.
[32] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.
[33] P.-L. Curien and H. Herbelin. The Duality of Computation. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 233–243. ACM, 2000.
[34] H.B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.
[35] F. Damiani and F. Prost. Detecting and Removing Dead-Code using Rank 2 Intersection. In *Proceedings of International Workshop TYPES'96, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 66–87. Springer Verlag, 1998.
[36] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of*

Theoretical Computer Science, volume B, chapter 6, pages 245–320. North-Holland, 1990.

[37] J. Eifrig, S.F. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. In R. Wirfs-Brock, editor, *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), Austin, Texas, USA*, pages 169–184. ACM, October 15-19 1995.

[38] J. Eifrig, S.F. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1:132–153, 1995.

[39] E. Ernst. Family polymorphism. In J. Lindskov Knudsen, editor, *Object-Oriented Programming, 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326, Budapest, Hungary, 2001. Springer Verlag.

[40] S. Feferman. A language and axioms for explicit mathematics. In J. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*. Springer Verlag, 1975.

[41] K. Fisher, F. Honsell, and J.C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[42] K. Fisher and J.C. Mitchell. A Delegation-based Object Calculus with Subtying. In *Fundamentals of Computation Theory, 10th International Symposium, FCT '95, Dresden, Germany, August 22-25, 1995, Proceedings*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer Verlag, 1995.

[43] D. Flanagan and Y. Matsumoto. *The Ruby programming language - everything you need to know: covers Ruby 1.8 and 1.9*. O'Reilly, 2008.

[44] N. Glew. An efficient class and object encoding. In M.B. Rosson and D. Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA'00), Minneapolis, Minnesota, USA, October 15-19, 2000*, pages 311–324. ACM, 2000.

[45] J. Gosling, W.N. Joy, and G.L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[46] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[47] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell. *ACM SIGPLAN Notices*, 27(5):1–64, 1992.

[48] 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.

[49] A. Igarashi, C. Saito, and M. Viroli. Lightweight family polymorphism. In K. Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 161–177. Springer Verlag, 2005.

[50] T.P. Jensen. Types in Program Analysis. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th Birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 204–222. Springer Verlag, 2002.

[51] S.N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In *POPL'88*, pages 80–87, 1988.

[52] S.N. Kamin and U.S. Reddy. Two semantic models of object-oriented languages. In C.A. Gunter and J.C. Mitchell, editors, *Theoretical aspects of object-oriented programming*, pages 463–495. MIT Press, Cambridge, MA, USA, 1994.

[53] J.W. Klop. Term Rewriting Systems: a tutorial. *EATCS Bulletin*, 32:143–182, 1987.

[54] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[55] J.C. Mitchell. Type Systems for Programming Languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 415–431. North-Holland, 1990.

[56] H. Nakano. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science*, pages 255–266, Santa Barbara, California, USA, 2000. IEEE Computer Society.

[57] Jens Palsberg. Efficient Inference of Object Types. *Information and Computation*, 123(2):198–209, 1995.

[58] M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proceedings of 3rd International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'92)*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Verlag, 1992.

[59] U.S. Reddy. Objects as Closures: Abstract Semantics of Object-Oriented Languages. In *LISP and Functional Programming*, pages 289–297, 1988.

[60] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[61] G. van Rossum and F.L. Drake, editors. *Python Language Reference*. PythonLabs, 2003.

[62] R.N.S. Rowe and S.J. van Bakel. Approximation Semantics and Expressive Predicate Assignment for Object-Oriented Programming. In L. Ong, editor, *Proceedings of* 10th International Conference on Typed Lambda Calculi and Applications *(TLCA'11)*, volume 6690 of *Lecture Notes in Computer Science*, pages 229–244. Springer Verlag, 2011.
[63] B. Stroustrup. *The C++ programming language (3. ed.)*. Addison-Wesley-Longman, 1997.
[64] Th. Studer. Constructive Foundations for Featherweight Java. In R. Kahle, P. Schroeder-Heister, and R.F. Stärk, editors, *Proof Theory in Computer Science, International Seminar, PTCS'01, Dagstuhl Castle, Germany*, volume 2183 of *Lecture Notes in Computer Science*, pages 202–238. Springer Verlag, October 7-12 2001.
[65] W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.
[66] R. Viswanathan. Full Abstraction for First-Order Objects with Recursive Types and Subtyping. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 380–391. IEEE Computer Society, 1998.
[67] C.P. Wadsworth. The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus. *SIAM Journal on Computing*, 5:488–521, 1976.
[68] C.P. Wadsworth. Approximate Reduction and Lambda Calculus Models. *SIAM Journal on Computing*, 7(3):337–356, 1978.

Appendix A Proof of the approximation result

The following properties hold of derivation reduction. They are used in the proofs of Theorem A.4 and Lemma A.10.

Lemma A.1 *i)* $\mathcal{SN}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash \mathtt{e.f} : \sigma) \;\Leftrightarrow\; \mathcal{SN}(\mathcal{D} :: \Pi \vdash \mathtt{e} : \langle \mathtt{f}{:}\sigma\rangle)$.

ii) $\mathcal{SN}(\langle \mathcal{D}, \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{invk}\rangle :: \Pi \vdash \mathtt{e.m}(\vec{\mathtt{e}}_n) : \sigma) \;\Rightarrow\; \mathcal{SN}(\mathcal{D})\ \&\ \forall i \in \overline{n}\ [\mathcal{SN}(\mathcal{D}_i)]$.

iii) For neutral contexts $\mathfrak{C}$, $\mathcal{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle m{:}(\vec{\phi}_n) \to \sigma\rangle)\ \&\ \forall i \in \overline{n}\ [\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi_i)] \Rightarrow \mathcal{SN}(\langle \mathcal{D}, \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{invk}\rangle :: \Pi \vdash \mathfrak{C}[x].m(\vec{\mathtt{e}}_n) : \sigma)$.

iv) $\mathcal{SN}(\langle \overline{\mathcal{D}}_n, \textsc{obj}\rangle :: \Pi \vdash \mathtt{new\ C}(\vec{\mathtt{e}}_n) : \mathtt{C}) \;\Leftrightarrow\; \exists \vec{\phi}_n\, [\forall i \in \overline{n}\ [\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi_i)]]$.

v) $\mathcal{SN}(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join}\rangle :: \Pi \vdash \mathtt{e} : \sigma_1 \cap \ldots \cap \sigma_n) \;\Leftrightarrow\; \forall i \in \overline{n}\ [\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash \mathtt{e} : \sigma_i)]$.

vi) $\mathcal{SN}(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash \mathtt{e} : \phi) \;\Leftrightarrow\; \mathcal{SN}(\mathcal{D} :: \Pi \vdash \mathtt{e} : \phi)$.

vii) Let $\mathtt{C}$ be a class such that $\mathcal{F}(\mathtt{C}) = \vec{\mathtt{f}}_n$, then for all $j \in \overline{n}$: $\mathcal{SN}(\langle \overline{\mathcal{D}}_n, \textsc{newf}\rangle :: \Pi \vdash \mathtt{new\ C}(\vec{\mathtt{e}}_n) : \langle \mathtt{f}_j{:}\sigma\rangle) \;\Leftrightarrow\; \exists \vec{\phi}_n\, [\sigma \trianglelefteq \phi_j\ \&\ \forall i \in \overline{n}\ [\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi_i)]]$.

viii) B Let $\mathtt{C}$ be such that $\mathcal{F}(\mathtt{C}) = \vec{\mathtt{f}}_n$, then for all $j \in \overline{n}$: $\mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[\mathtt{e}_j] : \sigma)\ \&\ \forall i \neq j \in \overline{n}\ [\exists \phi\, [\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi)]] \;\Rightarrow\; \mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\langle \langle \overline{\mathcal{D}}_n, \textsc{newf}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new\ C}(\vec{\mathtt{e}}_n).\mathtt{f}_j] : \sigma)$.

ix) Let $\mathtt{C}$ be such that $\mathcal{M}b(\mathtt{C}, m) = (\vec{x}_n, \mathtt{e_b})$ and $\mathcal{D}_b :: \mathtt{this} : \psi, \overrightarrow{x{:}\phi}_n \vdash \mathtt{e_b} : \sigma'$, then for all derivation contexts $\mathfrak{D}_{(p,\sigma')}$ and expression contexts $\mathfrak{C}$: $\mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[\mathtt{e_b}^{\mathsf{S}}] : \sigma)\ \&\ \mathcal{SN}(\mathcal{D}_0 :: \Pi \vdash \mathtt{new\ C}(\overrightarrow{\mathtt{e}}) : \psi)\ \&$
$\forall i \in \overline{n}\ [\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash \mathtt{e}_i : \phi_i)] \Rightarrow \mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overline{\mathcal{D}}_n, \textsc{invk}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new\ C}(\overrightarrow{\mathtt{e}}).m(\vec{\mathtt{e}}_n)] : \sigma)$.
where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newm}\rangle :: \Pi \vdash \mathtt{new\ C}(\overrightarrow{\mathtt{e}}) : \langle m{:}(\vec{\phi}_n) \to \sigma'\rangle$,
$\mathcal{S} = \langle \mathtt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\rangle$, *and*
$\mathsf{S} = \langle \mathtt{this} \mapsto \mathtt{new\ C}(\overrightarrow{\mathtt{e}}), x_1 \mapsto \mathtt{e}_1, \ldots, x_n \mapsto \mathtt{e}_n\rangle$.

Proof: These all follow straightforwardly from Definition 4.16. □

Our proof uses the well-known technique of *computability* [65]. As is standard, our notion is defined inductively over the structure of types (predicates), and is defined in such a way as to guarantee that computable derivations are strongly normalising.

Definition A.2 (Computability) *i*) The set of *computable* derivations is defined as the smallest set satisfying the following conditions (where $\mathit{Comp}(\mathcal{D})$ denotes that $\mathcal{D}$ is a member of

the set of computable derivations):

$$
\begin{aligned}
&\mathit{Comp}(\langle \mathrm{Q}\rangle\omega :: \Pi \vdash e : \omega)\\
\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \varphi) \;&\Leftrightarrow\; \mathcal{SN}(\mathcal{D} :: \Pi \vdash e : \varphi)\\
\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : C) \;&\Leftrightarrow\; \mathcal{SN}(\mathcal{D} :: \Pi \vdash e : C)\\
\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \langle f{:}\sigma\rangle) \;&\Leftrightarrow\; \mathit{Comp}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash e.f : \sigma)\\
\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \langle m{:}(\vec{\phi}_n) \to \sigma\rangle) \;&\Leftrightarrow\; (\forall \vec{\mathcal{D}}_n\,[\;\forall i \in \overline{n}\;\; [\mathit{Comp}(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)] \;\Rightarrow\\
&\quad \mathit{Comp}(\langle \mathcal{D}[\textstyle\bigcap \Pi \cdot \overrightarrow{\Pi}_n \trianglelefteq \Pi], \overrightarrow{\mathcal{D}_i[\bigcap \Pi \cdot \overrightarrow{\Pi}_n \trianglelefteq \Pi_i]}, \textsc{invk}\rangle :: \textstyle\bigcap \Pi \cdot \overrightarrow{\Pi}_n \vdash e.m(\vec{e}_n) : \sigma)\;])\\
\mathit{Comp}(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \textsc{join}\rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n)&\\
\Leftrightarrow\; \forall i \in \overline{n}\;\; [\mathit{Comp}(\mathcal{D}_i)]&
\end{aligned}
$$

ii) A derivation substitution $\mathcal{S}$ is *computable in* Π, if and only if, $\mathit{Comp}(\mathcal{S}(x{:}\phi))$ for all $x{:}\phi \in \Pi$.

Computability is preserved by weakening:

Lemma A.3 $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \phi) \;\Leftrightarrow\; \mathit{Comp}(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \phi)$.

Proof: By straightforward induction on the structure of predicates; for the base case, we use Lemma A.1((*vi*)). □

The key property of computable derivations is that they are strongly normalising as shown in the first part of the following theorem.

Theorem A.4 *i)* $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \phi) \;\Rightarrow\; \mathcal{SN}(\mathcal{D} :: \Pi \vdash e : \phi)$.

ii) For neutral contexts $\mathfrak{C}$, $\mathcal{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \phi) \;\Rightarrow\; \mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \phi)$.

Proof: By simultaneous induction on the structure of predicates.

(ω): By Definition 4.16 in the case of (1), and by Definition A.2 in the case of (2).

(φ, C): Immediate, by Definition A.2.

$(\langle f{:}\sigma\rangle)$: *a)* $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \langle f{:}\sigma\rangle) \;\Rightarrow$ (Def. A.2) $\mathit{Comp}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash e.f : \sigma) \;\Rightarrow$ (IH(1))
$\mathcal{SN}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash e.f : \sigma) \;\Rightarrow$ (Lem. A.1) $\mathcal{SN}(\mathcal{D} :: \Pi \vdash e : \langle f{:}\sigma\rangle)$

b) Assume $\mathcal{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle f{:}\sigma\rangle)$ with $\mathfrak{C}$ a neutral context. Then $\mathcal{SN}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}[x].f : \sigma)$ by Lemma A.1. Now, let $\mathfrak{C}' = \mathfrak{C}.f$; notice that, by Definitions 4.2 and 4.3, $\mathfrak{C}'$ is neutral, and $\mathfrak{C}[x].f = \mathfrak{C}'[x]$. Thus $\mathcal{SN}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}'[x] : \sigma)$, and, by induction, $\mathit{Comp}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}'[x] : \sigma)$. Then, from the definition of $\mathfrak{C}'$, it follows that $\mathit{Comp}(\langle \mathcal{D}, \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}[x].f : \sigma)$, and by Definition A.2, we have $\mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle f{:}\sigma\rangle)$.

$(\langle m{:}(\vec{\phi}_n) \to \sigma\rangle)$: *a)* Assume $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e : \langle m{:}(\vec{\phi}_n) \to \sigma\rangle)$. For each $i \in \overline{n}$, we take a fresh variable x_i and construct a derivation $\mathcal{D}_i$ as follows:

- If $\phi_i = \omega$ then $\mathcal{D}_i = \langle \mathrm{Q}\rangle\omega :: \Pi_i \vdash x_i : \omega$, with $\Pi_i = \emptyset$;
- If ϕ_i is a strict predicate σ then $\mathcal{D}_i = \langle \mathrm{Q}\rangle var :: \Pi_i \vdash x_i : \sigma$, with $\Pi_i = x_i{:}\sigma$;
- If $\phi_i = \sigma_1 \cap \ldots \cap \sigma_{n'}$ for some $n' \geq 2$ then $\mathcal{D}_i = \langle \mathcal{D'}_1, \ldots, \mathcal{D'}_{n'}, \textsc{join}\rangle :: \Pi_i \vdash x : \sigma_1 \cap \ldots \cap \sigma_{n'}$, with $\Pi_i = x_i{:}\phi_i$ and $\mathcal{D'}_j = \langle \mathrm{Q}\rangle var :: \Pi_i \vdash x_i : \sigma_j$ for each $j \in \overline{n'}$.

Notice that each $\mathcal{D}_i$ is in normal form, so $\mathcal{SN}(\mathcal{D}_i)$ for each $i \in \overline{n}$. Notice also that $\mathcal{D}_i :: \Pi_i \vdash \mathfrak{C}[x_i] : \phi_i$ for each $i \in \overline{n}$ where $\mathfrak{C}$ is the neutral context $[\,]$. So, by the second induction $\mathit{Comp}(\mathcal{D}_i)$ for each $i \in \overline{n}$.
Then, by Definition A.2,

$$Comp(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_n, \textsc{invk} \rangle :: \Pi' \vdash e.m(\vec{x}_n) : \sigma)$$

where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteq \Pi]$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ for each $i \in \overline{n}$ with $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi}_n$. So, by the first induction, $\mathcal{SN}(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_n, \textsc{invk} \rangle)$. Lastly, by Lemma A.1((*ii*)) we have $\mathcal{SN}(\mathcal{D}')$, and by Lemma A.1((*vi*)), $\mathcal{SN}(\mathcal{D})$.

b) Assume $\mathcal{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle m : (\vec{\phi}_n) \to \sigma \rangle)$ with $\mathfrak{C}$ a neutral context. Also, assume that there exist derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ such that: $Comp(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)$ for each $i \in \overline{n}$. Then, by the first induction, $\mathcal{SN}(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)$ for each $i \in \overline{n}$. Let $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi}_n$; notice that, by Definition 3.3, $\Pi' \trianglelefteq \Pi$ and $\Pi' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$. Then, by Lemma A.1((*vi*)), $\mathcal{SN}(\mathcal{D}[\Pi' \trianglelefteq \Pi])$ and $\mathcal{SN}(\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i])$ for each $i \in \overline{n}$. By Lemma A.1((*iii*)) we then have

$$\mathcal{SN}(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{invk} \rangle :: \Pi' \vdash \mathfrak{C}[x].m(\vec{e}_n) : \sigma)$$

where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteq \Pi]$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ for each $i \in \overline{n}$. Take the context $\mathfrak{C}' = \mathfrak{C}.m(\vec{e}_n)$; notice that, since $\mathfrak{C}$ is neutral, by Definitions 4.2 and 4.3, $\mathfrak{C}'$ is also a neutral context and $\mathfrak{C}[x].m(\vec{e}_n) = \mathfrak{C}'[x]$. Thus, by the second induction,

$$Comp(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{invk} \rangle :: \Pi' \vdash \mathfrak{C}[x].m(\vec{e}_n) : \sigma).$$

So, by Definition A.2, we have $Comp(\mathcal{D} :: \Pi \vdash e : \langle m : (\vec{\phi}_n) \to \sigma \rangle)$.

$(\sigma_1 \cap \ldots \cap \sigma_n, n \geq 2)$: By induction. □

A consequence of Theorem A.4 is that identity (derivation) substitutions are computable in their own environments.

Lemma A.5 Let Π be a predicate environment; then Id_Π is computable in Π.

Proof: Let $\Pi = \overrightarrow{x:\phi}$, then $Id_\Pi = \langle \overrightarrow{x \mapsto \mathcal{D} :: \Pi \vdash x : \phi} \rangle$ with each $\mathcal{D}_i$ in normal form and thus $\mathcal{SN}(\mathcal{D}_i)$. Notice also that, since $x_i = \mathfrak{C}[x_i]$ where $\mathfrak{C}$ is the empty context $[\,]$, $\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash \mathfrak{C}[x] : \phi_i)$ for each $i \in \overline{n}$. Then $Comp(\mathcal{D}_i)$ by Theorem A.4(2). Thus, for each $x{:}\phi \in \Pi$, $Comp(\mathcal{S}(x{:}\phi))$ and so, by Definition A.2, Id_Π is computable in Π. □

Also using Theorem A.4, we can show that computability is closed for derivation expansion - that is, if $\mathcal{D}'$ is computable and $\mathcal{D} \to_{\mathfrak{D}} \mathcal{D}'$, then also $\mathcal{D}$ is computable. This property will be important when showing the *replacement* lemma (Lemma A.10) below. We first show two auxiliary expansion lemmas, that are needed for the proof of that lemma.

Lemma A.6 (Field expansion) *Let C be a class such that $\mathcal{F}(C) = \overline{f}_n$, then for all $j \in \overline{n}$: if $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \sigma)$ and $\forall i \neq j \in \overline{n}\ [\exists \phi\ [Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi)\,]]$, then $Comp(\mathfrak{D}_{(p,\sigma')}[\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF} \rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new}\ C(\vec{e}_n).f_j] : \sigma)$.*

Proof: By induction on the structure of strict predicates.

(φ): Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \varphi)$ and $\exists \phi\ [Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi)\ ]$ for each $i \in \overline{n}$ such that $i \neq j$. By Theorem A.4, $\mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \varphi)$ and $\exists \phi\ [\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash e_i : \phi)\ ]$ for each $i \in \overline{n}$ such that $i \neq j$. Then by Lemma A.1((*viii*)) we have

$$\mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF} \rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new}\ C(\vec{e}_n).f_j] : \varphi)$$

And, by Definition A.2, $Comp(\mathfrak{D}_{(p,\sigma')}[\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF} \rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new}\ C(\vec{e}_n).f_j] : \varphi)$.

(C): Similar to the previous case.

$(\langle f{:}\sigma \rangle)$: Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \langle f{:}\sigma \rangle)$ and $\exists \phi\ [Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi)\ ]$ for each $i \in \overline{n}$ such that $i \neq j$. By Definition A.2, $Comp(\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[e_j].f : \sigma)$. Take the contexts $\mathfrak{C}'$ and $\mathfrak{D}'$ such that: $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.f$ and $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma')}, \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p.f : \sigma$. Notice that

$$\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[e_j].f : \sigma \;=\; \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[e_j] : \sigma,$$

so we have $Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[e_j] : \sigma)$. Then by induction we have

$$Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new C}(\vec{e}_n).f_j] : \sigma),$$

so by the definition of derivation contexts,

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n).f_j].f : \sigma).$$

Then, by Definition A.2, we have $Comp(\mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n).f_j] : \langle f{:}\sigma\rangle)$.

$(\langle m{:}(\vec{\phi}_{n'}) \to \sigma\rangle)$: Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \langle m{:}(\vec{\phi}_{n'}) \to \sigma\rangle)$ and that $\exists \phi\, [Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi)]$for each $i \neq j \in \overline{n}$. Now, take arbitrary derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ such that, for each $k \in \overline{n'}$, $Comp(\mathcal{D}'_k :: \Pi_k \vdash e'_k : \phi_k)$. By Definition A.2,

$$Comp(\langle \mathcal{D}', \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle) :: \Pi' \vdash \mathfrak{C}_p[e_j].m(\overrightarrow{e'}_{n'}) : \sigma,$$

where $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi}_{n'}$, $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j][\Pi' \trianglelefteq \Pi]$, and $\mathcal{D}''_k = \mathcal{D}'_k[\Pi' \trianglelefteq \Pi_k]$ for each $k \in \overline{n}$.

By Lemma A.9, $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j][\Pi' \trianglelefteq \Pi] = \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\mathcal{D}_j[\Pi' \trianglelefteq \Pi]]$; take the contexts $\mathfrak{C}'$ and $\mathfrak{D}'$ such that: $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.m(\overrightarrow{e'}_{n'})$ and $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma)}[\Pi' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p.m(\overrightarrow{e'}_{n'}) : \sigma$. Notice that

$$\langle \mathcal{D}', \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle \;=\; \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j[\Pi' \trianglelefteq \Pi]] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[e_j] : \sigma,$$

then we have $Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j[\Pi' \trianglelefteq \Pi]])$. Now, by Lemma A.3, $\exists \phi\, [Comp(\mathcal{D}_i[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e_i : \phi)$] for each $i \neq j \in \overline{n}$. Then by induction,

$$Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi], \textsc{newF}\rangle, \textsc{fld}\rangle] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new C}(\vec{e}_n).f_j] : \sigma)$$

So by the definition of $\mathfrak{D}'$,

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\langle\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi], \textsc{newF}\rangle, \textsc{fld}\rangle], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n).f_j].m(\overrightarrow{e'}_{n'}) : \sigma)$$

And then, by Definition 4.6,

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle[\Pi' \trianglelefteq \Pi]], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n).f_j].m(\overrightarrow{e'}_{n'}) : \sigma)$$

And by Lemma A.9

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle][\Pi' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n).f_j].m(\overrightarrow{e'}_{n'}) : \sigma)$$

Since the derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ were arbitrary, the following implication holds:

$$\forall \overrightarrow{\mathcal{D}'}_{n'} [\forall i \in \overline{n'}\; [Comp(\mathcal{D}'_i :: \Pi_i \vdash e'_i : \phi_i)] \;\Rightarrow\; Comp(\langle \mathcal{D}, \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n).f_j].m(\overrightarrow{e'}_{n'}) : \sigma)]$$

where $\mathcal{D} = \mathfrak{D}_{(p,\sigma)}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle][\Pi' \trianglelefteq \Pi]$. Thus, by Definition A.2,

$$Comp(\mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}}_n, \textsc{newF}\rangle, \textsc{fld}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new C}(\vec{e}_n).f_j] : \langle m{:}(\vec{\phi}_{n'}) \to \sigma\rangle) \qquad \square$$

Lemma A.7 (Method expansion) *Let* $\mathcal{M}b(C,m) = (\vec{x}_n, e_b)$ *and* $\mathcal{D}_b :: \Pi' \vdash e_b : \sigma'$ *with* $\Pi' = \texttt{this} : \psi, \overrightarrow{x{:}\phi}$, *then for contexts* $\mathfrak{D}_{(p,\sigma')}$ *and* $\mathfrak{C}$: *if* $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \sigma)$, $Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ *for all* $i \in \overline{n}$, *and* $Comp(\mathcal{D}_0 :: \Pi \vdash \texttt{new C}(\overrightarrow{e'}) : \psi)$, *then*

$$Comp(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new C}(\overrightarrow{e'}).m(\vec{e}_n)] : \sigma),$$

where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM}\rangle :: \Pi \vdash \texttt{new C}(\overrightarrow{e'}) : \langle m{:}(\vec{\phi}_n) \to \sigma'\rangle$, $\mathcal{S} = \langle \texttt{this} : \psi \mapsto \mathcal{D}_0, \overrightarrow{x{:}\phi \mapsto \mathcal{D}}\rangle$, *and* S *is the term substitution induced by* $\mathcal{S}$.

Proof: By induction on the structure of strict predicates.

(φ): Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \varphi)$, $Comp(\mathcal{D}_0 :: \Pi \vdash \texttt{new } C(\overrightarrow{e}) : \psi)$, and, for each $i \in \overline{n}$, $Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$. Then by Theorem A.4

$\mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \varphi)$, $\mathcal{SN}(\mathcal{D}_0 :: \Pi \vdash \texttt{new } C(\overrightarrow{e}) : \psi)$, and $\mathcal{SN}(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$

for each $i \in \overline{n}$. Then $\mathcal{SN}(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new } C(\overrightarrow{e}).m(\vec{e}_n)] : \varphi)$ by Lemma A.1((*ix*)), where

$$\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM}\rangle :: \Pi \vdash \texttt{new } C(\overrightarrow{e}) : \langle m : (\vec{\phi}_n) \to \sigma\rangle$$

And, by Definition A.2, $Comp(\mathfrak{D}_{(p,\sigma)}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle])$.

(C): Similar to the previous case.

$(\langle f{:}\sigma\rangle)$: Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \langle f{:}\sigma\rangle)$, $Comp(\mathcal{D}_0 :: \Pi \vdash \texttt{new } C(\overrightarrow{e}) : \psi)$, and $Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for all $i \in \overline{n}$. By Definition A.2, it follows that $Comp(\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}].f : \sigma)$. Take the contexts $\mathfrak{C}'$ and $\mathfrak{D}'$ such that $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.f$ and $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma')}, \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p.f : \sigma$. Notice that

$$\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}].f : \sigma \;=\; \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma$$

So we have $Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma)$, and then by induction

$$Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new } C(\overrightarrow{e}).m(\vec{e}_n)] : \sigma)$$

where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM}\rangle :: \Pi \vdash \texttt{new } C(\overrightarrow{e}) : \langle m : (\vec{\phi}_n) \to \sigma'\rangle$. So by the definition of $\mathfrak{D}'$,

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle], \textsc{fld}\rangle :: \Pi \vdash \mathfrak{C}_p[\texttt{new } C(\overrightarrow{e}).m(\vec{e}_n)].f : \sigma)$$

Then, by Definition A.2, $Comp(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle])$.

$(\langle m' : (\overrightarrow{\phi'}_{n'}) \to \sigma\rangle)$: Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \langle m' : (\overrightarrow{\phi'}_{n'}) \to \sigma\rangle)$,, $Comp(\mathcal{D}_0 :: \Pi \vdash \texttt{new } C(\overrightarrow{e}) : \psi)$, and, for all $i \in \overline{n}$, $Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$. Now, take $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ such that $Comp(\mathcal{D}'_k :: \Pi_k \vdash e'_k : \phi'_k)$ for each $k \in \overline{n'}$. By Definition A.2, $Comp(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_{n'}, \textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}].m'(\overrightarrow{e'}_{n'}) : \sigma)$, where $\Pi'' = \bigcap \Pi \cdot \overrightarrow{\Pi}_{n'}$, $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}][\Pi'' \trianglelefteq \Pi]$, and $\mathcal{D}''_k = \mathcal{D}'_k[\Pi'' \trianglelefteq \Pi_k]$ for each $k \in \overline{n'}$. Then, by Lemma A.9, $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}][\Pi'' \trianglelefteq \Pi] = \mathfrak{D}_{(p,\sigma')}[\Pi'' \trianglelefteq \Pi][\mathcal{D}_b{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi]]$. Take the contexts $\mathfrak{C}'$ and $\mathfrak{D}'$ such that $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.m'(\overrightarrow{e'}_{n'})$ and $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma')}[\Pi'' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}'}_{n'}, \textsc{invk}\rangle :: \Pi'' \vdash \mathfrak{C}_p.m'(\overrightarrow{e'}_{n'}) : \sigma$.

Notice that

$$\langle \mathcal{D}', \overrightarrow{\mathcal{D}'}_{n'}, \textsc{invk}\rangle \;=\; \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi]] :: \Pi'' \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma$$

So we have

$$Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi]] :: \Pi'' \vdash \mathfrak{C}'_{0\cdot p}[e_b{}^{\mathsf{S}}] : \sigma)$$

So, by Lemma 4.11 $Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}[\Pi'' \trianglelefteq \Pi]}])$. Now, by Lemma A.3, $Comp(\mathcal{D}_0[\Pi'' \trianglelefteq \Pi] :: \Pi'' \vdash \texttt{new } C(\overrightarrow{e}) : \psi)$ and $Comp(\mathcal{D}_i[\Pi'' \trianglelefteq \Pi] :: \Pi'' \vdash e_i : \phi_i)$ for all $i \in \overline{n}$. Thus, by induction,

$$Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle \mathcal{D}'', \mathcal{D}_1[\Pi'' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi'' \trianglelefteq \Pi], \textsc{invk}\rangle] :: \Pi'' \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new } C(\overrightarrow{e}).m(\vec{e}_n)] : \sigma)$$

where $\mathcal{D}'' = \langle \mathcal{D}_b, \mathcal{D}_0[\Pi'' \trianglelefteq \Pi], \textsc{newM}\rangle :: \Pi'' \vdash \texttt{new } C(\overrightarrow{e}) : \langle m : (\vec{\phi}_n) \to \sigma'\rangle$. So by the definition of $\mathfrak{D}'$

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi'' \trianglelefteq \Pi][\langle \mathcal{D}'', \mathcal{D}_1[\Pi'' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi'' \trianglelefteq \Pi], \textsc{invk}\rangle], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi'' \vdash \mathfrak{C}_p[\texttt{new } C(\overrightarrow{e}).m(\vec{e}_n)].m'(\overrightarrow{e''}_{n'}) : \sigma)$$

Then, by Definition 4.6,

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi'' \trianglelefteq \Pi][\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle[\Pi'' \trianglelefteq \Pi]], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle \\ :: \Pi'' \vdash \mathfrak{C}_p[\texttt{new}\ C(\overrightarrow{e'}).m(\vec{e}_n)].m'(\overrightarrow{e''}_{n'}) : \sigma)$$

where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newM}\rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'}) : \langle m{:}(\vec{\phi}_n) \to \sigma'\rangle$. And by Lemma A.9

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle][\Pi'' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}''}_{n'}, \textsc{invk}\rangle :: \Pi'' \vdash \mathfrak{C}_p[\texttt{new}\ C(\overrightarrow{e'}).m(\vec{e}_n)].m'(\overrightarrow{e''}_{n'}) : \sigma)$$

So, by Definition A.2, we have $Comp(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}}_n, \textsc{invk}\rangle])$. □

The following two basic properties of the weakening operation on derivations will be needed later when showing that it preserves computability.

Proposition A.8 Let Π_1, Π_2, Π_3 and Π_4 be type environments such that $\Pi_2 \trianglelefteq \Pi_1$, and $\Pi_3 \trianglelefteq \Pi_1$; $\Pi_4 \trianglelefteq \Pi_2$, and $\Pi_4 \trianglelefteq \Pi_3$; and $\mathcal{D}$ be a derivation such that $\mathcal{D} :: \Pi_1 \vdash e : \phi$. Then

i) $(\mathcal{D}[\Pi_2 \trianglelefteq \Pi_1])[\Pi_4 \trianglelefteq \Pi_2] = \mathcal{D}[\Pi_4 \trianglelefteq \Pi_1]$.

ii) $(\mathcal{D}[\Pi_2 \trianglelefteq \Pi_1])[\Pi_4 \trianglelefteq \Pi_2] = (\mathcal{D}[\Pi_3 \trianglelefteq \Pi_1])[\Pi_4 \trianglelefteq \Pi_3]$.

We also show the following property of weakening for derivation contexts and substitutions, which will be used in the proof of Lemma A.6 to show that computability is preserved by derivation expansion.

Lemma A.9 Let $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C}_p : \phi$ be a derivation context and $\mathcal{D} :: \Pi \vdash e : \sigma$ be a derivation. Also, let $[\Pi' \trianglelefteq \Pi]$ be a weakening. Then

$$\mathfrak{D}_{(p,\sigma)}[\mathcal{D}][\Pi' \trianglelefteq \Pi] \;=\; \mathfrak{D}_{(p,\sigma)}[\Pi' \trianglelefteq \Pi][\mathcal{D}[\Pi' \trianglelefteq \Pi]]$$

Proof: By induction on the structure of derivation contexts. □

The final piece of the strong normalisation proof is the derivation replacement lemma, which shows that when we perform derivation substitution using computable derivations we obtain a derivation that is overall computable. In [16], where an approximation result is shown for combinator systems, this lemma must be proved using an *encompassment* relation on terms. Since we have sub-derivations for the constituents of each redex that will appear during reduction, we are able to prove the replacement lemma by straightforward induction on the structure of derivations.

Lemma A.10 (Replacement) *If $\mathcal{D} :: \Pi \vdash e : \phi$ and $\mathcal{S}$ is a derivation substitution computable in Π, then $Comp(\mathcal{D}^{\mathcal{S}})$.*

Proof: By induction on the structure of derivations. The (newF) and (newM) cases are particularly tricky, and use Lemmas A.6 and A.7 respectively. Let $\Pi = x_1{:}\phi'_1, \ldots, x_n{:}\phi'_{n'}$ and $\mathcal{S} = \langle \overrightarrow{x' \mapsto \mathcal{D}' :: \Pi' \vdash e'' : \phi''}_{n''} \rangle$ with $\{\vec{x}_{n'}\} \subseteq \{\overrightarrow{x'}_{n''}\}$. Also, let S be the term substitution induced by $\mathcal{S}$.

(ω): Immediately by Definition A.2, since $\mathcal{D}^{\mathcal{S}} = \langle \mathrm{Q}\rangle\omega :: \Pi' \vdash e^{\mathsf{S}} : \omega$.

(var): Then $\mathcal{D} :: \Pi \vdash x : \sigma$. We examine the different possibilities for $\mathcal{D}^{\mathcal{S}}$:

- $x{:}\sigma \in \Pi$, so $x = x'_i$ for some $i \in \overline{n''}$ and $\mathcal{D}'_i :: \Pi' \vdash e'_i : \sigma$. Then $\mathcal{D}^{\mathcal{S}} = \mathcal{D}'_i$. Since $\mathcal{S}$ is computable in Π it follows that $Comp(\mathcal{D}'_i)$, and so $Comp(\mathcal{D}^{\mathcal{S}})$.
- $x{:}\phi \in \Pi$ for some $\phi \trianglelefteq \sigma$, so $\phi = \sigma_1 \cap \ldots \cap \sigma_n$ with $\sigma = \sigma_i$ for some $i \in \overline{n}$. Also, $x = x'_j$ for some $j \in \overline{n''}$ and $\mathcal{D}'_j :: \Pi' \vdash e'_j : \phi$, so $\mathcal{D}'_j = \langle \overrightarrow{\mathcal{D}''}_n, \textsc{join}\rangle$ with $\mathcal{D}''_k :: \Pi' \vdash e'_j : \sigma_k$ for each $k \in \overline{n}$. Now, by Definition 4.9, $\mathcal{D}^{\mathcal{S}} = \mathcal{D}''_i :: \Pi' \vdash e'_j : \sigma_i$. Since $\mathcal{S}$ is computable in Π, $Comp(\mathcal{D}'_j)$ and then, by Definition A.2, $Comp(\mathcal{D}''_k)$ for each $k \in \overline{n}$. Thus, in particular $Comp(\mathcal{D}''_i)$ and so $Comp(\mathcal{D}^{\mathcal{S}})$.

(fld): Then $\mathcal{D} = \langle \mathcal{D}', \textsc{fld}\rangle :: \Pi \vdash e.f : \sigma$ and $\mathcal{D}' :: \Pi \vdash e : \langle f{:}\sigma\rangle$. By induction, $Comp(\mathcal{D}'^{\mathcal{S}} ::$

$\Pi' \vdash e^{\mathsf{S}} : \langle f{:}\sigma \rangle$).

Then, by Definition A.2, $Comp(\langle \mathcal{D}'^{\mathcal{S}}, \textsc{fld} \rangle :: \Pi' \vdash e^{\mathsf{S}}.f : \sigma)$. Notice that $\langle \mathcal{D}'^{\mathcal{S}}, \textsc{fld} \rangle = \mathcal{D}^{\mathcal{S}}$ and so $Comp(\mathcal{D}^{\mathcal{S}})$.

($\textsc{invk}$): Then $\mathcal{D} = \langle \mathcal{D}_0, \overrightarrow{\mathcal{D}}_n, \textsc{invk} \rangle :: \Pi \vdash e_0.m(\vec{e}_n) : \sigma$ with $\mathcal{D}_0 :: \Pi \vdash e_0 : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle$ and $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. By induction, we have $Comp(\mathcal{D}_0{}^{\mathcal{S}} :: \Pi' \vdash e_0{}^{\mathsf{S}} : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle)$ and $\forall i \in \overline{n}\ [Comp(\mathcal{D}_i{}^{\mathcal{S}} :: \Pi' \vdash e_i{}^{\mathsf{S}} : \phi_i)]$. Then, by Definition A.2,

$$Comp(\langle \mathcal{D}_0{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi'], \mathcal{D}_1{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi'], \ldots, \mathcal{D}_n{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi'], \textsc{invk} \rangle :: \Pi'' \vdash e_0{}^{\mathsf{S}}.m(e_1{}^{\mathsf{S}}, \ldots, e_n{}^{\mathsf{S}}) : \sigma)$$

where $\Pi'' = \bigcap \Pi' \cdot \overrightarrow{\Pi}_n$ and $\Pi_i = \Pi'$ for each $i \in \overline{n}$. Notice that $\Pi'' = \Pi'$ and that for all $\mathcal{D} :: \Pi \vdash e : \phi$, $\mathcal{D}[\Pi \trianglelefteq \Pi] = \mathcal{D}$, so $Comp(\langle \mathcal{D}_0{}^{\mathcal{S}}, \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{invk} \rangle)$. Notice that $\langle \mathcal{D}_0{}^{\mathcal{S}}, \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{invk} \rangle = \mathcal{D}^{\mathcal{S}}$.

($\textsc{join}$), ($\textsc{obj}$): By induction.

($\textsc{newf}$): Then $\mathcal{D} = \langle \overrightarrow{\mathcal{D}}_n, \textsc{newf} \rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}_n) : \langle f_j{:}\sigma \rangle$ with $\mathcal{F}(C) = \vec{f}_n$ and $j \in \overline{n}$, and there is some $\vec{\phi}_n$ such that $\mathcal{D}_i :: \Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$ with $\phi_j = \sigma$. By induction, $Comp(\mathcal{D}_i{}^{\mathcal{S}} :: \Pi \vdash e_i : \phi_i)$ for each $i \in \overline{n}$. Now, take $\mathfrak{D}_{(0,\sigma)} = \langle \mathrm{Q} \rangle [\,]$ and $\mathfrak{C} = [\,]$. Notice that $\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_j{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}[e_j{}^{\mathsf{S}}] : \sigma = \mathcal{D}_j{}^{\mathcal{S}} :: \Pi \vdash e_j{}^{\mathsf{S}} : \phi_j$ and so $Comp(\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_j{}^{\mathcal{S}}])$. Then by Lemma A.6,

$$Comp(\mathfrak{D}_{(0,\sigma)}[\langle \langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{newf} \rangle, \textsc{fld} \rangle] :: \Pi \vdash \mathfrak{C}[\texttt{new}\ C(e_1{}^{\mathsf{S}}, \ldots, e_n{}^{\mathsf{S}}).f_j] : \sigma),$$

and from the definitions of $\mathfrak{D}_{(0,\sigma)}$ and $\mathfrak{C}$ that

$$Comp(\langle \langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{newf} \rangle, \textsc{fld} \rangle :: \Pi \vdash \texttt{new}\ C(e_1{}^{\mathsf{S}}, \ldots, e_n{}^{\mathsf{S}}).f_j : \sigma)$$

Then, by Definition A.2, $Comp(\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{newf} \rangle :: \Pi \vdash \texttt{new}\ C(e_1{}^{\mathsf{S}}, \ldots, e_n{}^{\mathsf{S}}) : \langle f_j{:}\sigma \rangle)$. Notice that $\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \textsc{newf} \rangle = \mathcal{D}^{\mathcal{S}}$ and so $Comp(\mathcal{D}^{\mathcal{S}})$.

($\textsc{newm}$): Then $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \textsc{newm} \rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle$ with $\mathcal{M}b(C,m) = (\overrightarrow{x''}_n, e_b)$ such that both $\mathcal{D}_b :: \Pi'' \vdash e_b : \sigma$ and $\mathcal{D}_0 :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \psi$ where $\Pi'' = \texttt{this}{:}\psi, \overrightarrow{x'':\phi}_n$. By induction, we have $Comp(\mathcal{D}_0{}^{\mathcal{S}} :: \Pi' \vdash \texttt{new}\ C(\vec{e})^{\mathsf{S}} : \psi)$. Now, assume that for every $i \in \overline{n}$ there exist a derivation $\mathcal{D}_i :: \Pi_i \vdash e'_i : \phi_i$ such that $Comp(\mathcal{D}_i)$. Let $\Pi''' = \bigcap \Pi' \cdot \overrightarrow{\Pi}_n$; notice that $\Pi''' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$ so by Lemma A.8 $Comp(\mathcal{D}_i[\Pi''' \trianglelefteq \Pi_i] :: \Pi''' \vdash e'_i : \phi_i)$ for each $i \in \overline{n}$. Also $\Pi''' \trianglelefteq \Pi'$ and so then too by Lemma A.8 we have

$$Comp(\mathcal{D}_0{}^{\mathcal{S}}[\Pi''' \trianglelefteq \Pi'] :: \Pi''' \vdash \texttt{new}\ C(\vec{e})^{\mathsf{S}} : \psi).$$

Now consider the derivation substitution

$$\mathcal{S}' = \langle \texttt{this}{:}\psi \mapsto \mathcal{D}_0{}^{\mathcal{S}}[\Pi''' \trianglelefteq \Pi'],\ \overrightarrow{x'':\phi \mapsto \mathcal{D}[\Pi''' \trianglelefteq \Pi]}_n \rangle$$

Notice that $\mathcal{S}'$ is computable in Π'' and applicable to $\mathcal{D}_b$. So by induction, $Comp(\mathcal{D}_b{}^{\mathcal{S}'} :: \Pi''' \vdash e_b{}^{\mathsf{S}'} : \sigma)$ where S' is the term substitution induced by $\mathcal{S}'$. Taking the derivation context $\mathfrak{D}_{(0,\sigma)} = \langle \mathrm{Q} \rangle [\,]$ and the expression context $\mathfrak{C} = [\,]$, notice that $\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_b{}^{\mathcal{S}'}] :: \Pi''' \vdash \mathfrak{C}[e_b{}^{\mathsf{S}'}] : \sigma = \mathcal{D}_b{}^{\mathcal{S}'} :: \Pi''' \vdash e_b{}^{\mathsf{S}'} : \sigma$ and so $Comp(\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_b{}^{\mathcal{S}'}] :: \Pi''' \vdash \mathfrak{C}[e_b{}^{\mathsf{S}'}] : \sigma)$. From Lemma A.7 we then have

$$Comp(\mathfrak{D}_{(0,\sigma)}[\langle \mathcal{D}', \mathcal{D}_1[\Pi''' \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi''' \trianglelefteq \Pi_n], \textsc{invk} \rangle] :: \Pi''' \vdash \mathfrak{C}[\texttt{new}\ C(\vec{e})^{\mathsf{S}}.m(\vec{e'}_n)] : \sigma)$$

where $\mathcal{D}' = \langle \mathcal{D}_b, \mathcal{D}_0{}^{\mathcal{S}}[\Pi''' \trianglelefteq \Pi'], \textsc{newm} \rangle$. So, from the definitions of $\mathfrak{D}_{(0,\sigma)}$ and $\mathfrak{C}$,

$$Comp(\langle \mathcal{D}', \mathcal{D}_1[\Pi''' \trianglelefteq \Pi_1], \ldots, \mathcal{D}_n[\Pi''' \trianglelefteq \Pi_n], \textsc{invk} \rangle :: \Pi''' \vdash \texttt{new}\ C(\vec{e})^{\mathsf{S}}.m(\vec{e'}_n) : \sigma).$$

Notice that $\mathcal{D}' = \mathcal{D}^{\mathcal{S}}[\Pi''' \trianglelefteq \Pi']$. So, by Definition A.2, it follows that $Comp(\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash \texttt{new}\ C(\vec{e})^{\mathsf{S}} : \langle m{:}(\vec{\phi}_n) \to \sigma \rangle)$. □

Using this result, we can show that all valid derivations are computable.

Lemma A.11 $\mathcal{D} :: \Pi \vdash e : \phi \ \Rightarrow\ Comp(\mathcal{D} :: \Pi \vdash e : \phi)$.

Proof: Suppose $\Pi = x_1{:}\phi_1, \ldots, x_n{:}\phi_n$, and take the identity substitution Id_Π which is computable in Π by Lemma A.5. Then from Lemma A.10 we have $Comp(\mathcal{D}^{Id_\Pi})$, and since by Proposition 4.13 $\mathcal{D}^{Id_\Pi} = \mathcal{D}$ it follows that $Comp(\mathcal{D})$. □

Then the strong normalisation result for derivation reduction follows directly.

Theorem A.12 (Strong Normalisation for Derivation Reduction) *If* $\mathcal{D} :: \Pi \vdash e : \phi$ *then* $\mathcal{SN}(\mathcal{D})$.

Proof: By Lemma A.11 and Theorem A.4(1). □