Semantic Types and Approximation for Featherweight Java

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

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK

Abstract

We consider semantics for the class-based object-oriented calculus Featherweight Java based upon *approximation*. We also define an *intersection type assignment systems* for this calculus and show that it is *sound* and *complete*, i.e. types are preserved under conversion. We establish the link with between type assignment and the approximation semantics by showing an approximation result, which leads to a sufficient condition for head-normalisation and termination.

We show the expressivity of our predicate system by defining an encoding of Combinatory Logic into our calculus. We show that this encoding preserves predicate-ability and also 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.

Introduction

Over the years many expressive type systems have been defined and investigated for a variety of calculi. Amongst those, the *intersection type discipline* (ITD) [15, 16, 12, 2], first defined for the Lambda Calculus (LC) [11], stands out as a system that is closed under β-equality and gives rise to a filter model; 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 models and approximation results show that intersection types describe the full semantical behaviour of typeable terms. Intersection type systems have also been employed successfully in analyses for dead code elimination [18], strictness analysis [28], and control-flow analysis [10], proving them a versatile framework for reasoning about programs. Inspired by this expressive power, investigations have taken place of the suitability of intersection type assignment for other computational models: for example, van Bakel and Fernández have studied intersection types in the context of Term Rewriting Systems (TRS) [7, 8] and van Bakel studied them in the context of sequent calculi [4, 5].

The *object-oriented* programming paradigm has also been the subject of extensive theoretical study over the last two decades, as exemplified by languages such as C++ [39], Java [25], C# [20], Ruby [37], ECMAscript (or Javascript) [21] and Python [35]. OO languages come in two broad flavours: the *object* (or prototype) based, and the *class* based. A number of formal models has been developed [13, 14, 31, 22, 23, 1, 27] which attempt to distill the many features of OO into a core set of primitive operations. Of these, the ς-calculus [1] and Featherweight Java (FJ) [27] have been well received as elementary models for object based and class-based OO, respectively. In an attempt to bring intersection types to the context of OO, van Bakel and de'Liguoro presented a system for the ς-calculus [6]; 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.

Semantics is a well-established area of research for both functional and imperative languages; for the functional programming language side, semantics is mainly *denotational*, based on Scott's domain theory [38], whereas for

Email addresses: rnr07@doc.ic.ac.uk (R.N.S. Rowe), svb@doc.ic.ac.uk (S.J. van Bakel)

imperative languages it is mainly *operational* [34]. 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 [27], 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 [30] and Haskell [26]. We illustrate the expressive power of our calculus by showing that it is Turing complete through an embedding of Combinatory Logic (CL) – and thereby also LC. 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 (we will use the terminology *predicates* here, to distinguish our notion from the traditional notion of class types).

For that notion of intersection predicate assignment, we will show that the expected properties of a system based on intersection (i.e. *soundness* and *completeness*) hold. A notion of *approximant* for FJ-programs is defined as a finite, rooted segment of a (head)-normal form, as usual; this is used to show an *approximation result* which states that, for every intersection predicate assignable to a term in our system, an approximant of that term exists which can be assigned the same predicate. Interpreting a term by its set of approximants gives an *approximation semantics* and the approximation result then relates the approximation and the predicate-based semantics; it demonstrates that our predicate system is sound and complete with respect to the approximation semantics, allowing a predicate-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*, as in [8], to show the approximation result we need to consider reduction on predicate derivations.

We then restrict our notion to that of Curry type assignment – for which we can easily show a principal predicate property – and show a predicate preservation result: types assignable to CL-terms in Curry's system correspond to predicates in our system that can be assigned to the interpreted CL-terms. This could easily be extended to the strict intersection type assignment system for LC [2]; combined with the results we show in this paper, this then implies that the collection of predicate-able OO expressions correspond to the terms that are typeable using intersection types, i.e. all λ-terms that are semantically meaningful.

Contents of this paper. In Section 1, we present the calculus $\text{FJ}^{\text{¢}}$, Featherweight Java without casts, for which in Section 2 we define an approximation semantics, interpreting expressions through finite rooted segments of (infinite) normal forms. In Section 3, we define our notion of intersection predicate assignment, and show an subject reduction and expansion result (soundness and completeness). In Section 4 we show how to encode Combinatory Logic into $\text{FJ}^{\text{¢}}$, whilst preserving assignable Curry types. In Section 5 we define a notion of reduction on derivations that follows reduction on $\text{FJ}^{\text{¢}}$-expressions, and show that this notion is strongly normalisable. The two approaches of approximation and intersection types are linked in Section 6, where we show the approximation result. i.e. every type assignment for an expression is valid for one if its approximants, 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; we apply our result to the interpretation of Combinatory Logic. In Section 7, we give some detailed examples and observations, followed by our conclusion.

An extended abstract of this paper will appear as [36]. In [9] 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[1]), which gives a more straightforward presentation of system and proofs. We have also decoupled our intersection predicate system from the existing class type system, which shows that the approximation result does not depend on the class type system in any way.

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

[1] 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`); }` .

objects), which interact with one another by *calling* (also called *invoking*) methods on each other and accessing each other's (or their own) fields. We have removed cast expressions since, as the authors of [27] themselves point out, the presence of *downcasts* is unsound[2]; for this reason we call our calculus FJ$^{¢}$. We also leave the constructor method as implicit.

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

Definition 1 (Notation).
1. A sequence s of n elements $a_1,\ldots,a_n$ is denoted by $\overrightarrow{a_n}$; the subscript can be omitted when the exact number of elements in the sequence is not relevant.
2. We write $a \in \overrightarrow{a_n}$ whenever there exists some $i \in \{1,\ldots,n\}$ such that $a = a_i$.
3. We use $\overline{n}$ (where n is natural number) to represent the sequence $1,\ldots,n$.
4. For a constant term c, $\overrightarrow{\mathsf{c}_n}$ represents the sequence of n occurrences of c.
5. The empty sequence is denoted by ϵ, and concatenation on sequences by $s_1 \cdot s_2$.
6. $\wp$ denotes the *powerset* (set of all subsets) construction.

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). We distinguish the class name `Object` (which denotes the root of the class inheritance hierarchy in all programs) and the self variable `this`[3], used to refer to the receiver object in method bodies.

Definition 2 (FJ$^{¢}$ Syntax). FJ$^{¢}$ programs P consist of a *class table* $\mathcal{CT}$, comprising the *class declarations*, and an *expression* e to be run (corresponding to the body of the `main` method in a real Java program). They are defined by the grammar:

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

From this point, all the concepts defined are program dependent (or more precisely, parametric on the class table); however, since a program is essentially a fixed entity, 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 (e.g. [27]). We only consider programs which conform to some sensible well-formedness criteria: that there are no cycles in the inheritance hierarchy, and that fields and methods in any given branch of the inheritance hierarchy are uniquely named. An exception is made to allow the redeclaration of methods, providing that only the *body* of the method differs from the previous declaration (in the parlance of class-based OO, this is called *method override*).

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

Definition 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.

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

$$\begin{array}{lcl} \mathrm{CN}(\texttt{class } C \texttt{ extends } D\ \texttt{\{}\overrightarrow{fd}\ \overrightarrow{md}\texttt{\}}) & = & C \\ \mathrm{FN}(C\ f) & = & f \end{array}$$

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

$$\mathcal{CT}(C) \quad = \quad cd \quad \textit{if } \mathrm{CN}(cd) = C \textit{ and } cd \in \mathcal{CT}$$

[2] In the sense that typeable expressions can get stuck at runtime.

[3] Not a variable in the traditional sense, since it is not used to express a position in the method's body where a parameter can be passed.

3. 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:

$$\begin{array}{rcll} \mathcal{F}(\texttt{Object}) & = & \epsilon & \\ \mathcal{F}(C) & = & \mathcal{F}(C') \cdot \overrightarrow{f_n} & \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \ \{\overrightarrow{fd_n}\ \overrightarrow{md}\} \\ & & & \text{and } \text{FN}(fd_i) = f_i \text{ for all } i \in \overline{n}. \end{array}$$

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

$$\begin{array}{rcll} \mathcal{M}b(C,m) & = & (\overrightarrow{x_n}, e) & \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \ \{\overrightarrow{fd}\ \overrightarrow{md}\}, \text{ and there exist} \\ & & & C_0, \overrightarrow{C_n} \text{ such that } C_0\ m(C_1\ x_1, \ldots, C_n\ x_n)\ \{\texttt{return } e;\} \in \overrightarrow{md}. \\ \mathcal{M}b(C,m) & = & \mathcal{M}b(C',m) & \text{if } \mathcal{CT}(C) = \texttt{class } C \texttt{ extends } C' \ \{\overrightarrow{fd}\ \overrightarrow{md}\}, \text{ and there are no} \\ & & & C_0, \overrightarrow{C_n}, \overrightarrow{x_n}, e \text{ such that } C_0\ m(C_1\ x_1, \ldots, C_n\ x_n)\ \{\texttt{return } e;\} \in \overrightarrow{md}. \end{array}$$

5. The function VARS returns the set of variables used in an expression.

We impose the additional criterion that well-formed programs satisfy the following property:

$$\text{if } \mathcal{M}b(C,m) = (\overrightarrow{x_n}, e_b) \text{ then } \text{VARS}(e_b) \setminus \{\texttt{this}\} \subseteq \{x_1, \ldots, x_n\}$$

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

Definition 4 (Reduction). 1. A *term substitution* $\mathsf{S} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is defined in the standard way as a total function on expressions that systematically replaces all occurrences of the variables x_i by their corresponding expression e_i. We write e^{S} for $\mathsf{S}(e)$.

2. The reduction relation $\rightarrow$ is the smallest relation on expressions satisfying:

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

We call the left-hand term the *redex* and the right-hand the *contractum*.
We add the usual congruence rules for allowing reduction in subexpressions, and the reflexive and transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

This notion of reduction is *confluent*, which is easily shown by a standard 'colouring' argument (as is done in [11] for LC).

2. Approximation Semantics

In this section, we define an *approximation semantics* for FJ$^{\text{¢}}$. The notion of *approximant* was first introduced by Wadsworth in [41] for LC. 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 $\perp$; 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 (i.e. the observable *output*).

Example 5. *To illustrate this concept, consider* FJ$^{\text{¢}}$ *extended with numerals and arithmetic, and if-then-else construct, and take the class table given in Figure 1. Let the notation* $n_1 : n_2 : \ldots : n_k : []$ *be shorthand for the* FJ$^{\text{¢}}$ *expression:*

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

```
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 1: The class table for the Sieve of Eratosthenes in FJ¢

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

In this case, the output is finite, and the final approximant is the end-result itself. The class table in Figure 1 is also able to calculate a (infinite) list of prime numbers using the well-known 'sieve of Eratosthenes'.

Then (where we abbreviate `removeMultiplesOf` *by* `rMO`*)*		*which has the approximant*
	`new IntList().primes()`	$\bot$
$\rightarrow^*$	`(2:3:4:5:6:7:8:9:10:11:...).sieve()`	$\bot$
$\rightarrow^*$	`2:(3:(4:5:6:7:8:9:10:11:...).rMO(2)).sieve()`	`2:`$\bot$
$\rightarrow^*$	`2:3:(((5:6:7:8:9:10:11:...).rMO(2)).rMO(3)).sieve()`	`2:3:`$\bot$
$\rightarrow^*$	`2:3:5:(((7:8:9:10:11:...).rMO(2)).rMO(3)).rMO(5)).sieve()`	`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.

Approximate expressions and approximate normal forms for FJ¢ are defined below.

Definition 6 (Approximate Expressions). 1. The set of *approximate* FJ$^{\mathfrak{C}}$ expressions is defined, essentially adding $\bot$ as an expression, by the grammar:

$$a \quad ::= \quad x \mid \bot \mid a.f \mid a.m(\overrightarrow{a_n}) \mid \texttt{new}\ C(\overrightarrow{a_n}) \quad (n \geq 0)$$

2. The set of *approximate normal forms*, $\mathbb{A}$, ranged over by A, is a strict subset of the set of approximate expressions and is defined by the following grammar:

$$\begin{array}{rcll} A & ::= & x \mid \bot \mid \texttt{new}\ C(\overrightarrow{A_n}) & (\mathcal{F}(C) = \overrightarrow{f_n}) \\ & & \mid A.f \mid A.m(\vec{A}) & (A \neq \bot, A \neq \texttt{new}\ C(\overrightarrow{A_n})) \end{array}$$

Notice that we consider $\bot.f$ not in approximate normal form: it can be that $\bot$ hides an expression that reduces to an object $\texttt{new}\ C(\overrightarrow{A_n})$, in which case the field invocation can run, so disappears.

As can be expected, when we extend the notion of reduction so that field accesses and method calls on $\bot$ themselves reduce to $\bot$, we find that the approximate normal forms are normal forms with respect to this extended reduction relation.

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

Definition 7 (Approximation Relation). The *approximation relation* $\sqsubseteq$ is defined as the smallest preorder satisfying:

$$\begin{array}{rcl} \bot & \sqsubseteq & A \\ A \sqsubseteq A' \ \&\ \forall i \leq n\ [A_i \sqsubseteq A'_i] & \Rightarrow & \begin{cases} A.f & \sqsubseteq & A'.f \\ \texttt{new}\ C(\overrightarrow{A_n}) & \sqsubseteq & \texttt{new}\ C(\overrightarrow{A'_n}) \\ A.m(\overrightarrow{A_n}) & \sqsubseteq & A'.m(\overrightarrow{A'_n}) \end{cases} \end{array}$$

The relationship between the approximation relation and reduction is characterised by the following result.

Lemma 8. *If $A \sqsubseteq e$ and $e \rightarrow^* e'$, then $A \sqsubseteq e'$.*

PROOF. By induction on the definition of the length of reduction sequences; we only show the base case, which gets shown by induction on the structure of approximate normal forms.

$A = \bot$**:** Immediate, since $\bot \sqsubseteq e'$ by definition.

$A = x$**:** Then $e = e' = x$.

$A = A'.f$**:** Then $e = e''.f$ with $A' \sqsubseteq e''$. Also, since $A' \neq \texttt{new}\ C(\overrightarrow{A_n})$ it follows from Definition 7 that $e'' \neq \texttt{new}\ C(\overrightarrow{e_n})$. Thus e is not a redex and the reduction must take place in e'', that is $e' = e'''.f$ with $e'' \rightarrow e'''$. Then, by induction, $A' \sqsubseteq e'''$ and so $A'.f \sqsubseteq e'''.f$.

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

1. $e_0 \rightarrow e'_0$ and $e' = e'_0.m(\overrightarrow{e_n})$. Then by induction $A' \sqsubseteq e'_0$ and so also $A'.m(\overrightarrow{A_n}) \sqsubseteq e'_0.m(\overrightarrow{e_n})$.
2. $e_j \rightarrow e'_j$ for some $j \in \overline{n}$ and $e' = e_0.m(\overrightarrow{e'_n})$ 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 $A'.m(\overrightarrow{A_n}) \sqsubseteq e_0.m(\overrightarrow{e'_n})$.

$A = \texttt{new}\ C(\overrightarrow{A_n})$**:** Then $e = \texttt{new}\ C(\overrightarrow{e_n})$ with $A_i \sqsubseteq e_i$ for each $i \in \overline{n}$. Also $e_j \rightarrow e'_j$ for some $j \in \overline{n}$ and $e' = \texttt{new}\ C(\overrightarrow{e'_n})$ where $e'_k = e_k$ 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 $k \neq j$ and by induction $A_j \sqsubseteq e'_j$. Thus, by Definition 7, $\texttt{new}\ C(\overrightarrow{A_n}) \sqsubseteq \texttt{new}\ C(\overrightarrow{e'_n})$. □

Notice that this property expresses that the observable behaviour of a program can only increase (in terms of $\sqsubseteq$) through reduction.

We also define a *join* operation on approximate expressions.

Definition 9 (Join Operation). 1. The *join* operation $\sqcup$ on approximate expressions is a partial mapping defined as the reflexive and contextual closure of:

$$\bot \sqcup a = \bot \sqcup a = a$$

2. We extend the join operation to sequences of approximate expressions as follows:

$$\sqcup \epsilon = \bot \quad \sqcup a \cdot \overrightarrow{a_n} = a \sqcup (\sqcup \overrightarrow{a_n})$$

The following lemma shows that $\sqcup$ acts as an upper bound on approximate expressions, and that it is closed over the set of approximate *normal* forms.

Lemma 10. 1. *Let a_1, a_2 and a_3 be approximate expressions, then*

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

Moreover, if a_1 and a_2 are normal, *then so is $a_1 \sqcup a_2$ (when it is defined).*

2. $\sqcup \overrightarrow{A_n} \;=\; A_1 \sqcup \ldots \sqcup A_n$.

PROOF. 1. By induction on the structure of expressions in approximate normal form; 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, we have $a_1' \sqcup a_2' \sqsubseteq a'$, $a_1' \sqsubseteq a_1' \sqcup a_2'$, and $a_2' \sqsubseteq a_1' \sqcup a_2'$. Then, by Definition 7, it immediately follows that $(a_1' \sqcup a_2').f \sqsubseteq a'.f$, $a_1'.f \sqsubseteq (a_1' \sqcup a_2').f$, and $a_2'.f \sqsubseteq (a_1' \sqcup a_2').f$. Then, by Definition 9, $a_1 \sqcup a_2 = (a_1' \sqcup a_2').f$.

Moreover, if a_1 and a_2 are normal, then by definition so are a_1' and a_2', with both a_1' and a_2' being neither $\bot$, nor of the form $\texttt{new}\ C(\overrightarrow{a''_n})$. Then by induction $a_1' \sqcup a_2'$ is also normal, and by Definition 9 the join is neither equal to $\bot$ nor of the form $\texttt{new}\ C(\overrightarrow{a''_n})$. Thus, by Definition 7, $(a_1' \sqcup a_2').f = a_1 \sqcup a_2$ is an approximate normal form.

2. By induction on the length of sequences. □

Definition 11 (Approximants). The function $\mathcal{A}$ 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 is an approximate normal form that approximates some (intermediate) stage of execution of that expression.

As for models of LC, our approximation semantics equates pairs of expressions that are in the reduction relation, as shown by the following theorem.

Theorem 12. $e_1 \to^* e_2 \Rightarrow \mathcal{A}(e_1) = \mathcal{A}(e_2)$.

PROOF. $\supseteq$**:**

$$\begin{array}{lcll} e_1 \to^* e_2 \;\&\; A \in \mathcal{A}(e_2) & \Rightarrow & \text{(Definition 11)} \\ e_1 \to^* e_2 \;\&\; \exists e_3 \, [e_2 \to^* e_3 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \\ \exists e_3 \, [e_1 \to^* e_3 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \text{(Definition 11)} \\ A \in \mathcal{A}(e_1) & & \end{array}$$

$\subseteq$**:**

$$\begin{array}{lcll} e_1 \to^* e_2 \;\&\; A \in \mathcal{A}(e_1) & \Rightarrow & \text{(Definition 11)} \\ e_1 \to^* e_2 \;\&\; \exists e_3 \, [e_1 \to^* e_3 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \text{(Church-Rosser)} \\ \exists e_3, e_4 \, [e_1 \to^* e_2 \;\&\; e_2 \to^* e_4 \;\&\; e_1 \to^* e_3 \;\&\; e_3 \to^* e_4 \;\&\; A \sqsubseteq e_3] & \Rightarrow & \text{(Lemma 8)} \\ \exists e_4 \, [e_2 \to^* e_4 \;\&\; A \sqsubseteq e_4] & \Rightarrow & \text{(Definition 11)} \\ A \in \mathcal{A}(e_2) & & \end{array}$$

□

This result allows us to define a semantics for FJ$^{\mathfrak{C}}$ by interpreting expressions by the set of their approximants:

Definition 13 (Approximation Semantics). An *approximation model* for an FJ$^{\mathfrak{C}}$ program is a structure $\langle \wp(\mathbb{A}), [\![\cdot]\!] \rangle$, where the interpretation function $[\![\cdot]\!]$, mapping expressions to elements of the domain, $\wp(\mathbb{A})$, is defined by $[\![e]\!] = \mathcal{A}(e)$.

3. Type Assignment

Having defined a semantics for $\text{FJ}^{\mathfrak{C}}$, we continue by considering a type system for $\text{FJ}^{\mathfrak{C}}$ which is sound and complete with respect to these 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 subject reduction; moreover, it is the type derivation itself which determines the approximant in question. This relationship is formalised in the next section.

The type assignment system defined below follows in the *intersection type discipline*; it is influenced by the predicate system for the object calculus [6], and is ultimately based upon the strict intersection type system for LC (see [2] 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 (1) the operations that may be performed on it (i.e. accessing a field or invoking a method), and (2) the *outcome* of performing those operations, where dependencies between the inputs and outputs of methods are tracked using (type) variables. In this way, our types express detailed properties about the contexts in which expressions can safely be used. More intuitively, they capture a certain notion of *observational equivalence*: two expressions with the same set of assignable types will be observationally indistinguishable. Our types thus constitute *semantic predicates*, so for this reason (and also to distinguish them from the already existing Java class types) we call them predicates.

Definition 14 (Predicates). The set of *predicates* (ranged over by ϕ, ψ) and its subset of *strict* predicates (ranged over by σ) are defined by the following grammar (where φ ranges over a denumerable set of *predicate variables*, and C ranges over the set of class names):

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

The key feature of predicates 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.

In the language of intersection type systems, our predicates are *strict* in the sense of [2], since they must describe the outcome of performing an operation in terms of a(nother) *single* operation rather than an intersection. We include a predicate 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 predicate constant ω is a *top* (maximal) type, assignable to all expressions.

The *subpredicate* relation facilitates the selection of individual behaviours from an intersection.

Definition 15 (Subpredicate Relation). The subpredicate relation $\leqslant$ is the smallest preorder satisfying the following conditions:

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

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

$$\begin{array}{rcrcl} \sigma \sim \sigma' & \Rightarrow & \langle f{:}\sigma \rangle & \sim & \langle f{:}\sigma' \rangle \\ \forall i \in \overline{n}\,[\phi_i' \sim \phi_i'] \;\&\; \sigma \sim \sigma' & \Rightarrow & \langle m{:}(\phi_1,\ldots,\phi_n) \to \sigma \rangle & \sim & \langle m{:}(\phi_1',\ldots,\phi_n') \to \sigma' \rangle \end{array}$$

We consider predicates modulo $\sim$; in particular, all predicates in an intersection are different and ω does not appear in an intersection. Notice also that $\cap$ is associative, so we will abuse notation slightly and write $\sigma_1 \cap \ldots \cap \sigma_n$ (where $n \geq 2$) to denote a general intersection. In a further abuse of notation, $\phi_1 \cap \ldots \cap \phi_n$ will denote the predicate ϕ_1 when $n = 1$, and ω when $n = 0$.

$$(\text{OBJ}):\ \frac{\Pi \vdash e_1:\phi_1 \quad \ldots \quad \Pi \vdash e_n:\phi_n}{\Pi \vdash \texttt{new}\ C(\overrightarrow{e_n}):C}\ (\mathcal{F}(C)=\overrightarrow{f_n}) \qquad (\text{VAR}):\ \frac{}{\Pi,x{:}\phi \vdash x:\sigma}\ (\phi \lhd \sigma)$$

$$(\text{INVK}):\ \frac{\Pi \vdash e:\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle \quad \Pi \vdash e_1:\phi_1\ \ldots\ \Pi \vdash e_n:\phi_n}{\Pi \vdash e.m(\overrightarrow{e_n}):\sigma} \qquad (\text{FLD}):\ \frac{\Pi \vdash e:\langle f:\sigma\rangle}{\Pi \vdash e.f:\sigma}$$

$$(\text{NEWM}):\ \frac{\texttt{this}{:}\psi,x_1{:}\phi_1,\ldots,x_n{:}\phi_n \vdash e_b:\sigma \quad \Pi \vdash \texttt{new}\ C(\vec{e}):\psi}{\Pi \vdash \texttt{new}\ C(\vec{e}):\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle}\ (\mathcal{M}b(C,m)=(\overrightarrow{x_n},e_b))$$

$$(\text{NEWF}):\ \frac{\Pi \vdash e_1:\phi_1 \quad \ldots \quad \Pi \vdash e_n:\phi_n}{\Pi \vdash \texttt{new}\ C(\overrightarrow{e_n}):\langle f_i:\sigma\rangle}\ (\mathcal{F}(C)=\overrightarrow{f_n}, i \in \overline{n}, \sigma=\phi_i)$$

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

Figure 2: Predicate Assignment for FJ$^{\mathbb{C}}$

Definition 16 (Predicate Environments).
1. A *predicate statement* is of the form $e{:}\phi$, where e is called the *subject* of the statement.
2. An environment Π is a set of predicate statements with (distinct) variables as subjects; $\Pi,x{:}\phi$ stands for the environment $\Pi \cup \{x{:}\phi\}$ (so then either x does not appear in Π or $x{:}\phi \in \Pi$), and $x{:}\phi$ for $\emptyset,x{:}\phi$.
3. We extend $\lhd$ to environments by: $\Pi' \lhd \Pi \iff \forall x{:}\phi \in \Pi\ \exists\ \phi' \lhd \phi\ [x{:}\phi' \in \Pi']$.
4. 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 the subject.

We will now define our notion of predicate assignment, which is a slight variant of the system defined in [9].

Definition 17 (Predicate Assignment). Predicate assignment for FJ$^{\mathbb{C}}$ is defined by the natural deduction system given in Figure 2.

The predicate assignment rules in fact operate on the larger set of approximate expressions, but for clarity 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.

The rules of our predicate assignment system are fairly straightforward generalisations of the rules of the strict intersection type assignment system for LC to OO: e.g. (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 predicate for an object that presents an analysis of a method that is available in that object. It makes sense, however, when viewed as an $(\to I)$ rule. Like that rule, the analysis involves typing the body of the abstraction (i.e. the method body), and the assumptions (i.e. requirements) on the formal parameters are encoded in the derived predicate (to be checked on invocation). However, a method body may also make requirements on the *receiver*, through the use of the variable `this`. In our system we check that these hold *at the same time* as typing the method body (so-called *early self typing*)[4]. 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* predicate 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 permitted by the system.

As is standard for intersection type assignment systems, our system exhibits both subject reduction *and* subject expansion. First we show:

[4]Late self typing would check the type of the receiver at the point of method invocation.

Lemma 18 (Weakening). *Let $\Pi' \leqslant \Pi$; then $\Pi \vdash e:\phi \Rightarrow \Pi' \vdash e:\phi$.*

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 subpredicate relation. □

Lemma 19. 1. ***(Replacement)*** *If $x_1{:}\phi_1,\ldots,x_n{:}\phi_n \vdash e:\phi$ and there exists Π and $\overrightarrow{e_n}$ such that $\Pi \vdash e_i:\phi_i$ for each $i \in \overline{n}$, then $\Pi \vdash e^{\mathsf{S}}:\phi$ where $\mathsf{S} = \{x_1 \mapsto e_1,\ldots,x_n \mapsto e_n\}$.*

2. ***(Extraction)*** *For an expression e and term substitution $\mathsf{S} = \{x_1 \mapsto e_1,\ldots,x_n \mapsto e_n\}$ with $\text{VARS}(e) \subseteq \{x_1,\ldots,x_n\}$, if $\Pi \vdash e^{\mathsf{S}}:\phi$, then there are $\overrightarrow{\phi_n}$ such that $\Pi \vdash e_i:\phi_i$ for each $i \in \overline{n}$ and $x_1{:}\phi_1,\ldots,x_n{:}\phi_n \vdash e:\phi$.*

PROOF. 1. By induction on the structure of derivations.

(ω): Immediate.

(VAR): Then $e = x_i$ for some $i \in \overline{n}$ and $e^{\mathsf{S}} = e_i$. Also, $\phi = \sigma$ with $\phi_i \leqslant \sigma$, thus $\phi_i = \sigma_1 \cap \ldots \cap \sigma_n$ and $\sigma = \sigma_j$ for some $j \in \overline{n}$. Since $\Pi \vdash e_i:\phi_i$ it follows from rule (JOIN) that $\Pi \vdash e_i:\sigma_k$ for each $k \in \overline{n}$. So, in particular, $\Pi \vdash e_i:\sigma_j$.

The other cases follow straightforwardly by induction.

2. Also by induction on the structure of derivations.

(ω): By the (ω) rule, $\Pi \vdash e_i:\omega$ for each $i \in \overline{n}$ and $x_1{:}\omega,\ldots,x_n{:}\omega \vdash e:\omega$.

(VAR): Then ϕ is a strict predicate (hereafter called σ), and $x{:}\psi \in \Pi$ with $\psi \leqslant \sigma$. Also, it must be that $e = x_i$ for some $i \in \overline{n}$ and $e_i = x$. We then take $\phi_i = \sigma$ and $\phi_j = \omega$ for each $j \in \overline{n}$ such that $j \neq i$. By assumption $\Pi \vdash x:\sigma$ (that is $\Pi \vdash e_i:\phi_i$). Also, by the (ω) rule, we can derive $\Pi \vdash e_j:\omega$ for each $j \in \overline{n}$ such that $j \neq i$. Lastly, by (VAR) we have $x_1{:}\omega,\ldots,x_i{:}\sigma,\ldots,x_n{:}\omega \vdash x_i:\sigma$.

(NEWF): Then $e^{\mathsf{S}} = \mathsf{new}\ C(\overrightarrow{e'_{n'}})$ and $\phi = \langle f:\sigma\rangle$ with $\mathcal{F}(C) = \overrightarrow{f_{n'}}$ and $f = f_j$ for some $j \in \overline{n'}$. Also, there is $\overrightarrow{\phi_{n'}}$ such that $\Pi \vdash e'_{k'}:\phi_{k'}$ for each $k' \in \overline{n'}$, and $\sigma \leqslant \phi_j$. There are two cases to consider for e:

(a) $e = x_i$ for some $i \in \overline{n}$. Then $e_i = \mathsf{new}\ C(\overrightarrow{e'_{n'}})$. Take $\phi_i = \langle f:\sigma\rangle$ and $\phi_k = \omega$ for each $k \in \overline{n}$ such that $k \neq i$. By assumption we have $\Pi \vdash \mathsf{new}\ C(\overrightarrow{e'_{n'}}):\langle f:\sigma\rangle$ (that is $\Pi \vdash e_i:\phi_i$). Also, by rule (ω) $\Pi \vdash e_k:\omega$ for each $k \in \overline{n}$ such that $k \neq i$, and lastly by rule (VAR) $\Pi' \vdash x_i:\langle f:\sigma\rangle$ where $\Pi' = x_1{:}\omega,\ldots,x_i{:}\langle f:\sigma\rangle,\ldots,x_n{:}\omega$.

(b) $e = \mathsf{new}\ C(\overrightarrow{e''_{n'}})$ with $e''^{\mathsf{S}}_{k'} = e'_{k'}$ for each $k' \in \overline{n'}$. Notice $\text{VARS}(e''_{k'}) \subseteq \text{VARS}(e) \subseteq \{x_1,\ldots,x_n\}$ for each $k' \in \overline{n'}$. So, by induction, for each $k' \in \overline{n'}$ there is $\overrightarrow{\phi_{k'n}}$ such that $\Pi \vdash e_i:\phi_{k'_i}$ for each $i \in \overline{n}$ and $\Pi_{k'} \vdash e''_{k'}:\phi_{k'}$ where $\Pi_{k'} = x_1{:}\phi_{k'_1},\ldots,x_n{:}\phi_{k'_n}$. Let the environment $\Pi' = \bigcap\overrightarrow{\Pi_{n'}}$, that is $\Pi' = x_1{:}\phi_{1_1} \cap \ldots \cap \phi_{n'_1},\ldots,x_n{:}\phi_{1_n} \cap \ldots \cap \phi_{n'_n}$. Notice that $\Pi' \leqslant \Pi_{k'}$ for each $k' \in \overline{n'}$, so by Lemma 18 $\Pi' \vdash e''_{k'}:\phi_{k'}$ for each $k \in \overline{n'}$. Then by (NEWF) $\Pi' \vdash \mathsf{new}\ C(\overrightarrow{e''_{n'}}):\langle f:\sigma\rangle$. Lastly, by (JOIN) we can derive $\Pi \vdash e_i:\phi_{1_i} \cap \ldots \cap \phi_{n'_i}$ for each $i \in \overline{n}$.

The other cases are similar to that for (NEWF). □

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

Theorem 20 (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 and one inductive case (the others are similar). We show only the reasoning for the case that 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 predicate in the intersection, and then apply the (JOIN) rule.

$\mathcal{F}(C) = \overrightarrow{f_n} \Rightarrow \mathsf{new}\ C(\overrightarrow{e_n}).f_j \to e_j,\ j \in \overline{n}$**: if:** Assume $\Pi \vdash \mathsf{new}\ C(\overrightarrow{e_n}).f_j:\sigma$. The last rule applied in this derivation must be (FLD) so $\Pi \vdash \mathsf{new}\ C(\overrightarrow{e_n}):\langle f_j:\sigma\rangle$. 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 $\phi_j = \sigma$. Thus $\Pi \vdash e_j:\sigma$.

```
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 3: The class table for Object-Oriented Combinatory Logic (OOCL) programs

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 $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e_n}) : \langle f_j : \sigma \rangle$ and by (FLD) also $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e_n}).f_j : \sigma$.

$\mathcal{M}\boldsymbol{b}(C,m) = (\overrightarrow{x_n}, e_\mathrm{b}) \Rightarrow \mathtt{new}\ C(\overrightarrow{e'}).m(\overrightarrow{e_n}) \to {e_\mathrm{b}}^{\mathsf{S}}$ where $\mathsf{S} = \{\mathtt{this} \mapsto \mathtt{new}\ C(\overrightarrow{e'}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$:

if: Assume $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e'}).m(\overrightarrow{e_n}) : \sigma$. The last rule applied in the derivation must be (INVK), so there is $\overrightarrow{\phi_n}$ such that $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle$ and $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. Furthermore, the last rule applied in the derivation of $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle$ must be (NEWM) and so there is some predicate ψ such that $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e'}) : \psi$ and $\Pi' \vdash e_\mathrm{b} : \sigma$ where $\Pi' = \mathtt{this}{:}\psi, x_1{:}\phi_i, \ldots, x_n{:}\phi_n$. Then from Lemma 19(1) it follows that $\Pi \vdash {e_\mathrm{b}}^{\mathsf{S}} : \sigma$.

only if: Assume that $\Pi \vdash {e_\mathrm{b}}^{\mathsf{S}} : \sigma$. Then by Lemma 19(2) it follows that there is ψ, $\overrightarrow{\phi_n}$ such that $\Pi' \vdash e_\mathrm{b} : \sigma$ where $\Pi' = \mathtt{this}{:}\psi, x_1{:}\phi_i, \ldots, x_n{:}\phi_n$ with $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e'}) : \psi$ and $\Pi \vdash e_i : \phi_i$ for each $i \in \overline{n}$. By the (NEWM) rule we can then derive $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle$, and by the (INVK) rule that $\Pi \vdash \mathtt{new}\ C(\overrightarrow{e'}).m(\overrightarrow{e_n}) : \sigma$.

$e \to e' \Rightarrow e.f \to e'.f$**: if:** Assume that $\Pi \vdash e.f : \sigma$. The last rule applied in the derivation must be (FLD) and so we have that $\Pi \vdash e : \langle f : \sigma \rangle$. By induction, $\Pi \vdash e' : \langle f : \sigma \rangle$, and so by (FLD) that $\Pi \vdash e'.f : \sigma$.

only if: Assume that $\Pi \vdash e'.f : \sigma$. The last rule applied in the derivation must be (FLD) and so we have that $\Pi \vdash e' : \langle f : \sigma \rangle$. By induction, $\Pi \vdash e : \langle f : \sigma \rangle$, and so by (FLD) that $\Pi \vdash e.f : \sigma$. □

4. Expressivity

In this section we consider the formal expressivity of our OO calculus and predicate system. We show that FJ¢ is Turing complete by defining an encoding of Combinatory Logic (CL). Through the approximation result of the next

section all normalisable OOCL-expressions can be assigned a non-trivial predicate in our system. Thus, we have a predicate-based characterisation of all (terminating) computable functions in OO (see Theorem 73).

Combinatory Logic is a Turing complete model of computation defined by H.B. Curry [17] independently of LC. It is a higher-order TRS consisting of the function symbols **S**, **K** and the following rewrite rules:

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

Our encoding of CL in FJ$^{¢}$ is based on a Curryfied first-order version of the system above (see [7] for details), where the rules for **S** and **K** are expanded so that each new rewrite rule has a *single* operand, allowing for the partial application of function symbols. 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 21. The encoding of Combinatory Logic (CL) into the FJ$^{¢}$ program OOCL (Object-Oriented Combinatory Logic) is defined using the execution context given in Figure 3 and the function $[\![\cdot]\!]$ which translates terms of CL into FJ$^{¢}$ expressions, and is defined as follows:

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

The reduction behaviour of OOCL mirrors that of CL.

Theorem 22. *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**:

$$\begin{array}{ll} [\![\mathbf{S}\,t_1\,t_2\,t_3]\!] & \triangleq \\ \texttt{((new S().app(}[\![t_1]\!]\texttt{)).app(}[\![t_2]\!]\texttt{)).app(}[\![t_3]\!]\texttt{)} & \to \\ \texttt{((new S}_1\texttt{(}[\![t_1]\!]\texttt{)).app(}[\![t_2]\!]\texttt{)).app(}[\![t_3]\!]\texttt{)} & \to \\ \texttt{(new S}_2\texttt{(this.x,y)).app(}[\![t_3]\!]\texttt{)} \quad [\texttt{this} \mapsto \texttt{new S}_1\texttt{(}[\![t_1]\!]\texttt{), y} \mapsto [\![t_2]\!]] & = \\ \texttt{(new S}_2\texttt{(new S}_1\texttt{(}[\![t_1]\!]\texttt{).x,}[\![t_2]\!]\texttt{)).app(}[\![t_3]\!]\texttt{)} & \to \\ \texttt{(new S}_2\texttt{(}[\![t_1]\!]\texttt{,}[\![t_2]\!]\texttt{).app(}[\![t_3]\!]\texttt{)} & \to \\ \texttt{this.x.app(z).app(this.y.app(z))} \quad [\texttt{this} \mapsto \texttt{new S}_2\texttt{(}[\![t_1]\!]\texttt{,}[\![t_2]\!]\texttt{), z} \mapsto [\![t_3]\!]] & = \\ \texttt{((new S}_2\texttt{(}[\![t_1]\!]\texttt{,}[\![t_2]\!]\texttt{).x.app(}[\![t_3]\!]\texttt{)).app(new S}_2\texttt{(}[\![t_1]\!]\texttt{.}[\![t_2]\!]\texttt{).y.app(}[\![t_3]\!]\texttt{))} & \to^* \\ \texttt{(}[\![t_1]\!]\texttt{.app(}[\![t_3]\!]\texttt{)).app((}[\![t_2]\!]\texttt{).app(}[\![t_3]\!]\texttt{))} & \triangleq \\ [\![t_1\,t_3\,(t_2\,t_3)]\!] & \end{array}$$

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

Given the Turing completeness of CL, this result shows that FJ$^{¢}$ is also Turing complete. Although we are sure this does not come as a surprise, it is a nice formal property for our calculus to have. In addition, our type system can perform the same 'functional' analysis as ITD does for LC and CL. This is illustrated by a *type preservation* result. We describe Curry's type system for CL and then show we can give equivalent types to OOCL programs.

Definition 23 (Curry Type Assignment for CL). 1. The set of *simple types* (or Curry types) is defined by the following grammar:

$$\tau \quad ::= \quad \varphi \mid \tau \to \tau$$

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this}{:}\langle \texttt{x}{:}\langle \texttt{app}{:}(\sigma) \to \langle \texttt{app}{:}(\sigma') \to \sigma''\rangle\rangle\rangle}\,(\textsc{var})}{\Pi \vdash \texttt{this.x}{:}\langle \texttt{app}{:}(\sigma) \to \langle \texttt{app}{:}(\sigma') \to \sigma''\rangle\rangle}\,(\textsc{fld}) \quad \dfrac{}{\Pi \vdash \texttt{z}{:}\sigma}\,(\textsc{var})}{\Pi \vdash \texttt{this.x.app(z)}{:}\langle \texttt{app}{:}(\sigma') \to \sigma''\rangle}\,(\textsc{newM}) \quad \dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this}{:}\langle \texttt{y}{:}\langle \texttt{app}{:}(\sigma) \to \sigma'\rangle\rangle}\,(\textsc{var})}{\Pi \vdash \texttt{this.y}{:}\langle \texttt{app}{:}(\sigma) \to \sigma'\rangle}\,(\textsc{fld}) \quad \dfrac{}{\Pi \vdash \texttt{z}{:}\sigma}\,(\textsc{var})}{\Pi \vdash \texttt{this.y.app(z)}{:}\sigma'}\,(\textsc{invk})}{\Pi \vdash \texttt{this.x.app(z).app(this.y.app(z))}{:}\sigma''}\,(\textsc{invk}) \quad \vdots}{\Pi' \vdash \texttt{new S}_2\texttt{(this.x,y)}{:}\langle \texttt{x}{:}\sigma_1\rangle \cap \langle \texttt{y}{:}\sigma_2\rangle}\,(\textsc{join}) \quad \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}) \quad \vdots}{\Pi' \vdash \texttt{new S}_2\texttt{(this.x,y)}{:}\langle \texttt{app}{:}(\sigma) \to \sigma''\rangle}\,(\textsc{newM}) \quad \dfrac{\dfrac{}{\texttt{this}{:}\texttt{S},\texttt{x}{:}\sigma_1 \vdash \texttt{x}{:}\sigma_1}\,(\textsc{var})}{\texttt{this}{:}\texttt{S},\texttt{x}{:}\sigma_1 \vdash \texttt{new S}_1\texttt{(x)}{:}\langle \texttt{x}{:}\sigma_1\rangle}\,(\textsc{newF}) \quad \dfrac{}{\emptyset \vdash \texttt{new S()}{:}\texttt{S}}\,(\textsc{obj})}{\texttt{this}{:}\texttt{S},\texttt{x}{:}\sigma_1 \vdash \texttt{new S}_1\texttt{(x)}{:}\langle \texttt{app}{:}(\sigma_2) \to \langle \texttt{app}{:}(\sigma) \to \sigma''\rangle\rangle}\,(\textsc{newM}) \quad \vdots}{\emptyset \vdash \texttt{new S()}{:}\langle \texttt{app}{:}(\sigma_1) \to \langle \texttt{app}{:}(\sigma_2) \to \langle \texttt{app}{:}(\sigma) \to \sigma''\rangle\rangle\rangle}\,(\textsc{newM})$$

Figure 4: Derivation schemes for the translations of **S** and **K**

2. A *basis B* is a mapping from variables to Curry types, written as a set of statements of the form $x{:}\tau$ in which each of the variables x is distinct.
3. Simple types are assigned to CL-terms using the following natural deduction system:

$$(\textsc{var}):\ \frac{}{B \vdash_{\text{CL}} x{:}\tau}\,(x{:}\tau \in B) \qquad (\to E):\ \frac{B \vdash_{\text{CL}} t_1{:}\tau \to \tau' \quad B \vdash_{\text{CL}} t_2{:}\tau}{B \vdash_{\text{CL}} t_1 t_2{:}\tau'}$$

$$(\mathbf{K}):\ \frac{}{B \vdash_{\text{CL}} \mathbf{K}{:}\tau \to \tau' \to \tau} \qquad (\mathbf{S}):\ \frac{}{B \vdash_{\text{CL}} \mathbf{S}{:}(\tau \to \tau' \to \tau'') \to (\tau \to \tau') \to \tau \to \tau''}$$

To show type preservation, we need to define what the equivalent of Curry's types are in terms of predicates. To this end, we define the following translation of Curry types.

Definition 24 (Type Translation). The function $[\![\cdot]\!]$, which transforms Curry types into predicates[5], is defined as follows:

$$\begin{array}{rcl} [\![\varphi]\!] & = & \varphi \\ [\![\tau \to \tau']\!] & = & \langle \texttt{app}{:}([\![\tau]\!]) \to [\![\tau']\!]\rangle \end{array}$$

It is extended to bases as follows: $[\![B]\!] = \{x{:}[\![\tau]\!] \mid x{:}\tau \in B\}$.

We can now show the type preservation result.

[5]Note we have *overloaded* the notation $[\![\cdot]\!]$, which we also use for the translation of CL terms to FJ$^{\text{¢}}$ expressions.

Theorem 25 (Preservation of Types). *If* $B \vdash_{\text{CL}} t{:}\tau$ *then* $[\![B]\!] \vdash [\![t]\!] : [\![\tau]\!]$.

PROOF. By induction on the derivation of $B \vdash_{\text{CL}} t{:}\tau$. The cases for (VAR) and $(\to E)$ are trivial. For the rules $(\mathbf{K})$ and $(\mathbf{S})$, Figure 4 gives derivation schemas for assigning the translation of the respective Curry type schemes to the OOCL translations of **K** and **S**. □

Furthermore, since Curry's well-known translation of the simply typed LC into CL preserves typeability, we also construct a type-preserving encoding of LC into FJ$^{\mathfrak{C}}$. 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, since there not all normal forms are typeable using strict types, whereas in our system they are.

5. Strong Normalisation of Derivation Reduction

The approximation result we show in the next section is, as in other systems [3, 8], a direct consequence of the strong normalisability of derivation reduction[6] which we will define in this section.

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

$$\dfrac{\dfrac{\boxed{\mathcal{D}_1} \quad \boxed{\mathcal{D}_n}}{\dfrac{\Pi \vdash e_1 : \phi_1 \quad \cdots \quad \Pi \vdash e_n : \phi_n}{\Pi \vdash \texttt{new}\ C(\overrightarrow{e_n}) : \langle f_i : \sigma \rangle}\ (\text{NEWF})}{\Pi \vdash \texttt{new}\ C(\overrightarrow{e_n}).f_i : \sigma}\ (\text{FLD}) \quad \to_{\mathfrak{D}} \quad \boxed{\mathcal{D}_i}\ \Pi \vdash e_i : \sigma$$

$$\dfrac{\dfrac{\boxed{\mathcal{D}_b} \quad \boxed{\mathcal{D}_{\text{self}}}}{\dfrac{\texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e_b : \sigma \quad \Pi \vdash \texttt{new}\ C(\overrightarrow{e'}) : \psi}{\Pi \vdash \texttt{new}\ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle}\ (\text{NEWM}) \quad \boxed{\mathcal{D}_1} \quad \boxed{\mathcal{D}_n}}{\Pi \vdash \texttt{new}\ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle \quad \Pi \vdash e_1 : \phi_1 \ \cdots\ \Pi \vdash e_n : \phi_n}\ (\text{INVK})}{\Pi \vdash \texttt{new}\ C(\overrightarrow{e'}).m(\overrightarrow{e_n}) : \sigma} \quad \to_{\mathfrak{D}} \quad \boxed{\mathcal{D}_b{}^{\mathcal{S}}}\ \Pi \vdash {e_b}^{\mathsf{S}} : \sigma$$

where $\mathcal{D}_b{}^{\mathcal{S}}$ is the derivation obtained from $\mathcal{D}_b$ by replacing all sub-derivations of the form $\langle \text{VAR} \rangle :: \Pi, x_i{:}\phi_i \vdash x_i : \sigma$ by appropriately typed sub-derivations of $\mathcal{D}_i$, and sub-derivations of the form $\langle \text{VAR} \rangle :: \Pi, \texttt{this}{:}\psi \vdash \texttt{this} : \sigma$ by appropriately typed sub-derivations of $\mathcal{D}_{\text{self}}$. Similarly, ${e_b}^{\mathsf{S}}$ is the expression obtained from e_b by replacing each variable x_i by the expression e_i, and the variable `this` by $\texttt{new}\ C(\overrightarrow{e'})$. 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 $\langle \omega \rangle :: \Pi \vdash e : \omega$ do *not* reduce (although e might) - they are already 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.

Definition 26 (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 $\mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: \Pi \vdash e : \phi$ when the structure of $\mathcal{D}$ 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.

We also 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* within 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.

[6] As in [8], we need to consider derivation reduction; since reduction on expressions is *weak*, the 'normal' approach (as used in [2]) to show the approximation result does not work.

Definition 27 (Position). The *position* p of one (sub) expression – similarly of one (sub) derivation – within 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:

1. Positions within expressions are defined inductively as follows:

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

2. Positions within derivations are defined inductively as follows:

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

   Notice that due to the $(\textsc{join})$ rule, sub-derivations indicated by positions in derivations are not necessarily unique.
3. We define the following terminology:
   - We say that e' ($\mathcal{D}'$) *appears at position* p within e ($\mathcal{D}$) if $pos(e', e) = p$ ($pos(\mathcal{D}', \mathcal{D}) = p$).
   - We say that position p *exists within* e ($\mathcal{D}$) if there exists some e' ($\mathcal{D}'$) that appears at position p within e ($\mathcal{D}$).

Definition 28 (Expression Contexts). 1. An *expression context* $\mathfrak{C}$ is an expression containing a unique 'hole' (denoted by $[\,]$) defined by the following grammar:

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

2. $\mathfrak{C}[e]$ denotes the expression obtained by replacing the hole in $\mathfrak{C}$ with e.
3. We write $\mathfrak{C}_p$ to indicate that the hole in $\mathfrak{C}$ appears at position p.
4. Contexts $\mathfrak{C}_p$ where $p = \overline{0_n}$, for some $n \geq 1$, are called *neutral*.
5. Expressions of the form $\mathfrak{C}[x]$ where $\mathfrak{C}$ is neutral are also called neutral.

The following is easy to show:

Proposition 29. *Approximate expressions of the form $A.f$ and $A.m(\vec{A})$ are neutral.*

We also use the notion of *derivation context* that is like a derivation, but concluding with a statement assigning a strict predicate to a neutral context, essentially adding the inference rule:

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

Definition 30 (Derivation Contexts). 1. A *derivation context* $\mathfrak{D}_{(p,\sigma)}$, where we mark at which position the hole appears, and which strict type it gets assigned, is inductively defined as a generalisation over derivations by:

(a) $\mathfrak{D}_{(0,\sigma)} = \langle [\,] \rangle :: \Pi \vdash [\,] : \sigma$ is a derivation context.

(b) If $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C} : \langle f : \sigma' \rangle$ is a derivation context, then $\mathfrak{D}'_{(0 \cdot p,\sigma)} = \langle \mathfrak{D}, \text{FLD} \rangle :: \Pi \vdash \mathfrak{C}.f : \sigma'$ is also a derivation context.

(c) if $\mathfrak{D}_{(p,\sigma)} :: \Pi \vdash \mathfrak{C} : \langle m : (\overrightarrow{\phi_n}) \to \sigma' \rangle$ is a derivation context and $\overrightarrow{\mathcal{D}_n}$ is a sequence of derivations such that $\mathcal{D}_i :: \Pi \vdash e : \phi_i$ for each $i \in \overline{n}$, then $\mathfrak{D}'_{(0 \cdot p,\sigma)} = \langle \mathfrak{D}, \overrightarrow{\mathcal{D}_n}, \text{INVK} \rangle :: \Pi \vdash \mathfrak{C}.m(\overrightarrow{e_n}) : \sigma'$ is also a derivation context.

2. 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 31 (Weakening). A *weakening*, written $[\Pi' \trianglelefteq \Pi]$ where $\Pi' \trianglelefteq \Pi$, is an operation that replaces environments by sub-environments. For derivations $\mathcal{D} :: \Pi \vdash e : \phi$, $\mathcal{D}[\Pi' \trianglelefteq \Pi]$ is defined as the derivation $\mathcal{D}'$ of exactly the same shape (using the same rules in the same order, deriving the same type for the same expression, but using a different context) as $\mathcal{D}$ such that $\mathcal{D}' :: \Pi' \vdash e : \phi$.

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

Proposition 32. *Let Π_1, Π_2, Π_3 and Π_4 be predicate 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*

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

PROOF. Easy. □

We also show the following property of weakening for derivation contexts and substitutions, which will be used in the proof of Lemma 55 to show that computability is preserved by derivation expansion.

Lemma 33. *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*

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

PROOF. By induction on the structure of derivation contexts. □

We now define two sets of derivations: the 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 34 (Strong and ω-safe Derivations).

1. *Strong derivations* are defined as in Definition 17, but by excluding rule (ω).
2. ω-safe derivations are defined inductively as follows:
   - $\langle \text{VAR} \rangle :: x{:}\phi \vdash x : \sigma$ is ω-safe for any ϕ and σ.
   - $\langle \overrightarrow{\mathcal{D}_n}, \text{JOIN} \rangle$, $\langle \overrightarrow{\mathcal{D}_n}, \text{OBJ} \rangle$ and $\langle \overrightarrow{\mathcal{D}_n}, \text{NEWF} \rangle$ are ω-safe, if each derivation $\mathcal{D}_i$ is ω-safe.
   - $\langle \mathcal{D}, \text{FLD} \rangle$ is ω-safe, if $\mathcal{D}$ is ω-safe.

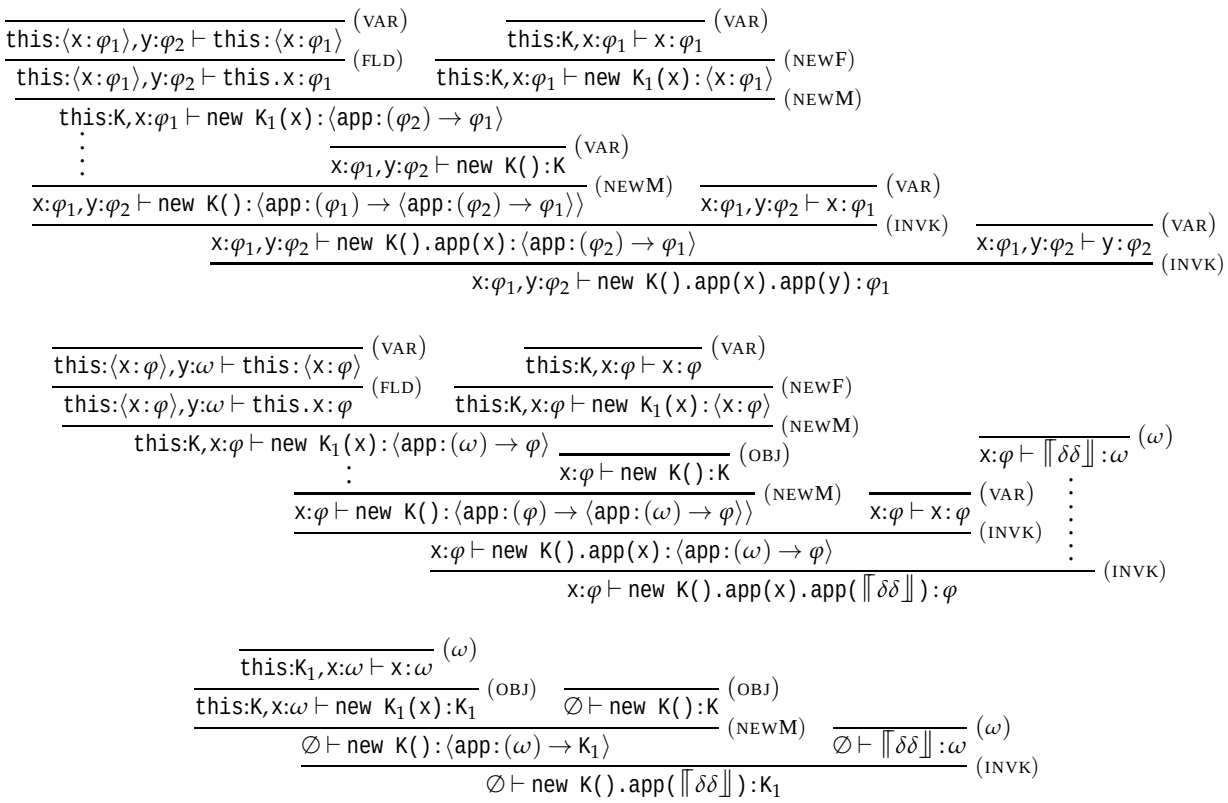

Figure 5: Derivations for Example 35

- $\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \text{INVK} \rangle$ 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 $\langle \omega \rangle :: \Pi \vdash e : \omega$.
- $\langle \mathcal{D}, \mathcal{D}', \text{NEWM} \rangle$ is ω-safe, if both $\mathcal{D}$ and $\mathcal{D}'$ are ω-safe.

We call a predicate ϕ *strong* if it does not contain ω. We call a predicate 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.

Example 35. *Figure 5 shows, respectively,*

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

where δ is the CL *term* $\mathbf{S}\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})$ *– i.e. $\delta\delta$ is an unsolvable term.*

Lemma 36. *If $\mathcal{D} :: \Pi \vdash A : \phi$ with ω-safe $\mathcal{D}$ and Π, then A does not contain $\bot$; moreover, if A is neutral, then ϕ does not contain ω.*

PROOF. By induction on the structure of derivations.

$\langle \omega \rangle$**:** Vacuously true, since $\langle \omega \rangle$ derivations are not ω-safe.

$\langle$VAR$\rangle$**:** Then $A = x$ and so does not contain $\bot$. Since x is neutral, we must also show that ϕ does not contain ω. Notice that ϕ is strict and that there is some $\psi \trianglelefteq \phi$ such that $x{:}\psi \in \Pi$. Since ϕ is strict and $\psi \trianglelefteq \phi$, $\psi \neq \omega$ and since Π is ω-safe it follows that ψ does not contain ω; therefore, by $\trianglelefteq$, neither does ϕ.

$\langle \mathcal{D}', \overrightarrow{\mathcal{D}_n}, \text{INVK} \rangle$**:** Then $A = A'.m(\overrightarrow{A_n})$ and ϕ is strict, hereafter called σ. Also $\mathcal{D}' :: \Pi \vdash A' : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle$ with $\mathcal{D}'$ ω-safe, and $\mathcal{D}_i :: \Pi \vdash A_i : \phi_i$ for each $i \in \overline{n}$. By induction, A' does not contain $\bot$. Also, notice that A must be neutral, and therefore so must A'. Then it also follows by induction that $\langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle$ 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 A_i contains $\bot$ either. Consequently, $A'.m(\overrightarrow{A_n})$ does not contain $\bot$ and σ does not contain ω.

$\langle \mathcal{D}_\mathbf{b}, \mathcal{D}', \text{NEWM} \rangle$**:** Then $\mathcal{D}_\mathrm{b} :: \Pi' \vdash e_\mathrm{b} : \sigma$ with $\mathtt{this}{:}\psi \in \Pi'$ and $\mathcal{D}' :: \Pi \vdash A : \psi$. Since $\mathcal{D}$ is ω-safe so also is $\mathcal{D}'$ and by induction, A does not contain $\bot$.

The other cases follow straightforwardly by induction. □

Continuing with the definition of derivation reduction we point out that, just as 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 subderivations, will form the basis of derivation reduction.

Derivation substitution is formally defined as follows.

Definition 37 (Derivation Substitution). A *derivation substitution* is a partial function from derivations to derivations, defined by:

1. Let $\mathcal{D}_1 :: \Pi' \vdash e_1 : \phi_1, \ldots, \mathcal{D}_n :: \Pi' \vdash e_n : \phi_n$ be derivations, and $x_1, \ldots, x_n$ be distinct variables, then $\mathcal{S} = \{ x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n \}$ 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).
2. If $\mathcal{D} :: \Pi \vdash e : \phi$ is a derivation such that $\Pi \subseteq x_1{:}\phi_1, \ldots, x_n{:}\phi_n$, and $\mathcal{S}$ a derivation substitution, then we say that $\mathcal{S}$ is *applicable* to $\mathcal{D}$, and the result of applying $\mathcal{S}$ to $\mathcal{D}$ (written $\mathcal{D}^\mathcal{S}$) is defined inductively as follows (where S is the term substitution induced by $\mathcal{S}$, i.e. $\mathsf{S} = \{ x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \}$):

   $\mathcal{D} = \langle \text{VAR} \rangle :: \Pi \vdash x : \sigma$**:** Then there are two cases to consider.
   (a) 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
   (b) $x{:}\phi \in \Pi$ 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 $\mathcal{D}_i = \langle \mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}, \text{JOIN} \rangle :: \Pi' \vdash e_i : \phi$ and $\mathcal{D}^\mathcal{S} = \mathcal{D}'_j :: \Pi' \vdash e_i : \sigma_j$.

   $\mathcal{D} = \langle \mathcal{D}_\mathbf{b}, \mathcal{D}', \text{NEWM} \rangle :: \Pi \vdash \mathtt{new}\ C(\vec{e}) : \langle m : (\vec{\phi}) \to \sigma \rangle$**:** Then

$$\mathcal{D}^\mathcal{S} = \langle \mathcal{D}_\mathrm{b}, \mathcal{D}'^\mathcal{S}, \text{NEWM} \rangle :: \Pi \vdash \mathtt{new}\ C(\vec{e})^\mathsf{S} : \langle m : (\vec{\phi}) \to \sigma \rangle$$

   $\mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n, r \rangle :: \Pi \vdash e : \phi, r \notin \{ (\text{VAR}), (\text{NEWM}) \}$**:** 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 $\langle \omega \rangle :: \Pi \vdash e : \omega$ as a special case.
3. We extend the weakening operation to derivation substitutions as follows: for a derivation substitution $\mathcal{S} = \{ x_1 \mapsto \mathcal{D}_1 :: \Pi \vdash e_1 : \phi_1, \ldots, x_n \mapsto \mathcal{D}_n :: \Pi \vdash e_n : \phi_n \}$, $\mathcal{S}[\Pi' \trianglelefteq \Pi]$ is the derivation substitution $\{ x_1 \mapsto \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, x_n \mapsto \mathcal{D}_n[\Pi' \trianglelefteq \Pi] \}$.

Notice that when we substitute

$$\dfrac{\begin{array}{cc}\boxed{\mathcal{D}_1} & \boxed{\mathcal{D}_2} \\ \Pi \vdash e : \phi_1 & \Pi \vdash e : \phi_2\end{array}}{\Pi \vdash e : \phi_1 \cap \phi_2}\ (\text{JOIN})$$

in the derivation for $x{:}\phi_1 \cap \phi_2 \vdash x : \phi_1$, we do not build

$$\dfrac{\dfrac{\begin{array}{cc}\boxed{\mathcal{D}_1} & \boxed{\mathcal{D}_2} \\ \Pi \vdash e : \phi_1 & \Pi \vdash e : \phi_2\end{array}}{\Pi \vdash e : \phi_1 \cap \phi_2}\ (\text{JOIN})}{\Pi \vdash e : \phi_1}\ (?)$$

since we do not have a rule that allows for the last step - after all, the notion of predicate assignment is strict – but define the result of this substitution as:

$$\frac{\boxed{\mathcal{D}_1}}{\Pi \vdash e : \phi_1}$$

so let the collection of derivations used in (JOIN) 'distribute.' It is because the system is strict that we are sure the correct sub-derivation is present.

Example 38. *Consider the derivations below for two expressions e_1 and e_2:*

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

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

$$\frac{\dfrac{}{\Pi' \vdash x : \langle m : (\varphi_1 \cap \varphi_2) \to \sigma \rangle}\ (\text{VAR}) \qquad \dfrac{\dfrac{}{\Pi' \vdash y : \varphi_1}\ (\text{VAR}) \qquad \dfrac{}{\Pi' \vdash y : \varphi_2}\ (\text{VAR})}{\Pi \vdash y : \varphi_1 \cap \varphi_2}\ (\text{JOIN})}{\mathcal{D} :: \Pi' \vdash x.m(y) : \sigma}\ (\text{INVK})$$

Let $\mathcal{S}$ denote the derivation substitution $\{x \mapsto \mathcal{D}_1, y \mapsto \mathcal{D}_2\}$; then the result of substituting $\mathcal{D}_1$ for x and $\mathcal{D}_2$ for y in $\mathcal{D}$ is the following derivation, where instances of the (VAR) rule in $\mathcal{D}$ have been replaced by the appropriate (sub) derivations in $\mathcal{D}_1$ and $\mathcal{D}_2$:

$$\frac{\dfrac{\boxed{\mathcal{D}_1}}{\Pi \vdash e_1 : \langle m : (\varphi_1 \cap \varphi_2) \to \sigma \rangle} \qquad \dfrac{\boxed{\mathcal{D}_2'} \quad \boxed{\mathcal{D}_2''}}{\Pi \vdash e_2 : \varphi_1 \cap \varphi_2}\ (\text{JOIN})}{\mathcal{D}^{\mathcal{S}} :: \Pi \vdash e_1.m(e_2) : \sigma}\ (\text{INVK})$$

Lemma 39 (Soundness of Derivation Substitution). *Let $\mathcal{D} :: \Pi \vdash e : \phi$ and $\mathcal{S}$ be a derivation substitution 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 $\mathcal{S}$.*

PROOF. By easy induction on the structure of derivations. □

Derivation substitution preserves strong and ω-safe derivations.

Lemma 40. *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).*

PROOF. By straightforward induction on the structure of derivations. □

We also show that the operations of weakening and derivation substitution are commutative.

Lemma 41. *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 induction on the structure of derivations. □

Definition 42 (Identity Substitutions). Each environment Π induces a derivation substitution $\mathcal{S}_\Pi$ which is called the *identity substitution* for Π. Let $\Pi = x_1{:}\phi_1, \ldots, x_n{:}\phi_n$; then $\mathcal{S}_\Pi \triangleq \{x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$ where for each $i \in \overline{n}$:

- If $\phi_i = \omega$ then $\mathcal{D}_i = \langle \omega \rangle :: \Pi \vdash x_i : \omega$;
- If ϕ_i is a strict predicate σ then $\mathcal{D}_i = \langle \text{VAR} \rangle :: \Pi \vdash x_i : \sigma$;

$$e \overset{p}{\rightsquigarrow} e' \quad \Rightarrow \quad \mathcal{D} :: \Pi \vdash e:\omega \overset{p}{\rightsquigarrow} \langle\omega\rangle :: \Pi \vdash e':\omega$$

$$\mathcal{D} :: \Pi \vdash e:\langle f:\sigma\rangle \overset{p}{\rightsquigarrow} \mathcal{D}' :: \Pi \vdash e':\langle f:\sigma\rangle \quad \Rightarrow \quad \langle \mathcal{D}, \text{FLD}\rangle \overset{0:p}{\rightsquigarrow} \langle \mathcal{D}', \text{FLD}\rangle$$

$$\mathcal{D} :: \Pi \vdash e:\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle \overset{p}{\rightsquigarrow} \mathcal{D}' :: \Pi \vdash e':\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle \;\&\; \forall i \in \overline{n}\ [\mathcal{D}_i :: \Pi \vdash e_i:\phi_i] \Rightarrow \langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \text{INVK}\rangle \overset{0:p}{\rightsquigarrow} \langle \mathcal{D}', \overrightarrow{\mathcal{D}_n}, \text{INVK}\rangle$$

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

$$\forall i \in \overline{n}\ (n \geq 2)\ [\mathcal{D}_i :: \Pi \vdash e:\sigma_i \overset{p}{\rightsquigarrow} \mathcal{D}'_i :: \Pi \vdash e':\sigma_i] \quad \Rightarrow \quad \langle \overrightarrow{\mathcal{D}_n}, \text{JOIN}\rangle \overset{p}{\rightsquigarrow} \langle \overrightarrow{\mathcal{D}'_n}, \text{JOIN}\rangle$$

$$\mathcal{D} :: \Pi \vdash e:\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle \;\&\; \exists j \in \overline{n}\ [\mathcal{D}_j :: \Pi \vdash e_j:\phi_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j \;\&\; \forall i \neq j \in \overline{n}\ [\mathcal{D}_i :: \Pi \vdash e_i:\phi_i]] \Rightarrow \langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \text{INVK}\rangle \overset{j:p}{\rightsquigarrow} \langle \mathcal{D}, \overrightarrow{\mathcal{D}'_n}, \text{INVK}\rangle :: \Pi \vdash e.m(\overrightarrow{e'_n}):\sigma$$

$$\mathcal{F}(C) = \overrightarrow{f_n} \;\&\; \exists j \in \overline{n}\ [\mathcal{D}_j :: \Pi \vdash e_j:\phi_j \overset{p}{\rightsquigarrow} \mathcal{D}'_j \;\&\; \forall i \neq j \in \overline{n}\ [\mathcal{D}_i :: \Pi \vdash e_i:\phi_i]] \Rightarrow \langle \overrightarrow{\mathcal{D}_n}, \text{OBJ}\rangle \overset{j:p}{\rightsquigarrow} \langle \overrightarrow{\mathcal{D}'_n}, \text{OBJ}\rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'_n}):C$$

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

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

Figure 6: The advance operation on derivations

- If $\phi_i = \sigma_1 \cap \ldots \cap \sigma_{m_i}$ for some $m_i \geq 2$ then $\mathcal{D}_i = \langle \overrightarrow{\mathcal{D}'_{m_i}}, \text{JOIN}\rangle :: \Pi \vdash x_i:\sigma_1 \cap \ldots \cap \sigma_{m_i}$, with $\mathcal{D}'_j = \langle \text{VAR}\rangle :: \Pi \vdash x_i:\sigma_j$ for each $j \in \overline{m_i}$.

Notice that for every environment Π, the identity substitution $\mathcal{S}_\Pi$ is also *based on* Π.

We can of course show that $\mathcal{S}_\Pi$ is indeed the identity for the substitution operation on derivations using Π.

Proposition 43. *Let $\mathcal{D} :: \Pi \vdash e:\phi$ and $\mathcal{S}_\Pi$ be the identity substitution for Π; then $\mathcal{D}^{\mathcal{S}_\Pi} = \mathcal{D}$.*

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 44 (Advancing). 1. 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 $((p,e),e') \in \rightsquigarrow$):

$$\begin{array}{rcl} \mathcal{F}(C) = \overrightarrow{f_n} & \& & e = \mathfrak{C}_p[\texttt{new}\ C(\overrightarrow{e_n}).f_i] \quad \text{with } i \in \overline{n} \quad \Rightarrow \quad e \overset{p}{\rightsquigarrow} \mathfrak{C}_p[e_i] \\ \mathcal{M}b(C,m) = (\overrightarrow{x_n}, e_b) & \& & e = \mathfrak{C}_p[\texttt{new}\ C(\overrightarrow{e'}).m(\overrightarrow{e_n})] \quad \Rightarrow \quad e \overset{p}{\rightsquigarrow} \mathfrak{C}_p[e_b{}^{\mathsf{S}}] \\ & & \text{where } \mathsf{S} = \{\texttt{this} \mapsto \texttt{new}\ C(\overrightarrow{e'}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\} \end{array}$$

2. We extend $\rightsquigarrow$ to derivations via the rules in Figure 6, (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 predicates derived; only subexpressions which are typed with ω are altered.

The following lemma states that this always generates a correct derivation.

Lemma 45 (Soundness of Advancing). *Let $\mathcal{D} :: \Pi \vdash e:\phi$; if a redex appears at position p in e and no derivation redex appears at p in $\mathcal{D}$ (so $e \overset{p}{\rightsquigarrow} e'$ for some e'), then there exists $\mathcal{D}'$ such that $\mathcal{D} \overset{p}{\rightsquigarrow} \mathcal{D}'$, and $\mathcal{D}' :: \Pi \vdash e':\phi$.*

PROOF. By well-founded induction on pairs of position and derivation $(p,\mathcal{D})$. □

$$\langle\langle\overrightarrow{\mathcal{D}_n},\text{NEWF}\rangle,\text{FLD}\rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}).f_i : \sigma \quad \xrightarrow{0}_{\triangleright} \quad \mathcal{D}_i \qquad (\mathcal{F}(C) = \overrightarrow{f_n}, \forall i \in \overline{n})$$

$$\langle\langle\mathcal{D}_b,\mathcal{D}',\text{NEWM}\rangle,\overrightarrow{\mathcal{D}_n},\text{INVK}\rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'}).m(\overrightarrow{e_n}) : \sigma \quad \xrightarrow{0}_{\triangleright} \quad \mathcal{D}_b\mathcal{S}$$
$$(\mathcal{M}b(C,m) = (\overrightarrow{x_n}, e_\text{b}) \;\&\; \mathcal{S} = \{\,\texttt{this} \mapsto \mathcal{D}', x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\,\})$$

$$\mathcal{D} :: \Pi \vdash e : \langle f : \sigma\rangle \xrightarrow{p}_{\triangleright} \mathcal{D}' :: \Pi \vdash e' : \phi \;\Rightarrow\; \langle\mathcal{D},\text{FLD}\rangle :: \Pi \vdash e.f : \sigma \quad \xrightarrow{0\cdot p}_{\triangleright} \quad \langle\mathcal{D}',\text{FLD}\rangle :: \Pi \vdash e'.f : \sigma$$

$$\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}' :: \Pi \vdash e' : \phi \;\Rightarrow\; \langle\mathcal{D},\overrightarrow{\mathcal{D}_n},\text{INVK}\rangle :: \Pi \vdash e.m(\overrightarrow{e_n}) : \sigma \quad \xrightarrow{0\cdot p}_{\triangleright} \quad \langle\mathcal{D}',\overrightarrow{\mathcal{D}_n},\text{INVK}\rangle :: \Pi \vdash e'.m(\overrightarrow{e_n}) : \sigma$$

$$\exists\, j \in \overline{n}\ [\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j :: \Pi \vdash e'_j : \phi] \;\Rightarrow\; \langle\mathcal{D},\mathcal{D}_1,\ldots,\mathcal{D}_n,\text{INVK}\rangle :: \Pi \vdash e.m(\overrightarrow{e_n}) : \sigma \quad \xrightarrow{j\cdot p}_{\triangleright} \quad \langle\mathcal{D},\mathcal{D}'_1,\ldots,\mathcal{D}'_n,\text{INVK}\rangle :: \Pi \vdash e.m(\overrightarrow{e'_n}) : \sigma$$
$$\forall\, i \neq j \in \overline{n}\ [\mathcal{D}'_i = \mathcal{D}_i \;\&\; e'_i = e_i]$$

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

$$\mathcal{D} :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \psi \xrightarrow{p}_{\triangleright} \mathcal{D}' :: \Pi \vdash e : \psi' \;\Rightarrow\; \langle\mathcal{D}_\text{b},\mathcal{D},\text{NEWM}\rangle :: \Pi \vdash \texttt{new}\ C(\vec{e}) : \langle m : (\vec{\phi}) \to \sigma\rangle \quad \xrightarrow{p}_{\triangleright} \quad \langle\mathcal{D}_\text{b},\mathcal{D}',\text{NEWM}\rangle :: \Pi \vdash e : \langle m : (\vec{\phi}) \to \sigma\rangle$$
$$(\mathcal{D}_\text{b} :: \texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n \vdash e_\text{b} : \sigma)$$

$$\exists\, j \in \overline{n}\ [\mathcal{D}_j :: \Pi \vdash e_j : \phi_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j :: \Pi \vdash e'_j : \phi'_j] \;\Rightarrow\; \langle\overrightarrow{\mathcal{D}_n},\text{OBJ}\rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e_n}) : C \quad \xrightarrow{j\cdot p}_{\triangleright} \quad \langle\overrightarrow{\mathcal{D}'_n},\text{OBJ}\rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'_n}) : C$$
$$(\forall\, i \neq j \in \overline{n}\ [\mathcal{D}'_i = \mathcal{D}_i \;\&\; e'_i = e_i])$$

$$\exists\, j \in \overline{n}\ [\mathcal{D}_j \xrightarrow{p}_{\triangleright} \mathcal{D}'_j \;\&\; \forall\, i \neq j \in \overline{n}\ [\mathcal{D}_i \xrightarrow{p}_{\triangleright} \mathcal{D}'_i \vee \mathcal{D}_i \xrightarrow{p}_{\not\triangleright} \mathcal{D}'_i]] \;\Rightarrow\; \langle\mathcal{D}_1,\ldots,\mathcal{D}_n,\text{JOIN}\rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n \quad \xrightarrow{p}_{\triangleright} \quad \langle\mathcal{D}'_1,\ldots,\mathcal{D}'_n,\text{JOIN}\rangle$$

Figure 7: Derivation reduction

The advance operation preserves strong (and ω-safe) typeability.

Lemma 46. *If $\mathcal{D} \xrightarrow{p}_{\not\triangleright} \mathcal{D}'$ is defined, and $\mathcal{D}$ is strong (ω-safe), then $\mathcal{D}'$ is also strong (ω-safe).*

PROOF. 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. within 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 47 (Derivation Reduction).

1. The reduction of a derivation $\mathcal{D}$ at position p to $\mathcal{D}'$ is denoted by $\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}'$, and is defined inductively using the rules in Figure 7.
2. The full reduction relation on derivations $\to_{\mathfrak{D}}$ is defined as the smallest relation on derivations satisfying the condition:
$$\exists\, p\ [\mathcal{D} \xrightarrow{p}_{\triangleright} \mathcal{D}'] \quad\Rightarrow\quad \mathcal{D} \to_{\mathfrak{D}} \mathcal{D}'$$
The reflexive and transitive closure of $\to_{\mathfrak{D}}$ is denoted by $\to^*_{\mathfrak{D}}$.
3. We write $SN(\mathcal{D})$ 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*.

The following properties hold of derivation reduction. They are used in the proofs of Theorem 53 and Lemma 57.

Lemma 48.

1. $SN(\langle\mathcal{D},\text{FLD}\rangle :: \Pi \vdash e.f : \sigma) \;\Leftrightarrow\; SN(\mathcal{D} :: \Pi \vdash e : \langle f : \sigma\rangle)$.
2. $SN(\langle\mathcal{D},\mathcal{D}_1,\ldots,\mathcal{D}_n,\text{INVK}\rangle :: \Pi \vdash e.m(\overrightarrow{e_n}) : \sigma) \;\Rightarrow\; SN(\mathcal{D})\ \&\ \forall\, i \in \overline{n}\ [SN(\mathcal{D}_i)]$.
3. *For neutral contexts* $\mathfrak{C}$, $SN(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x] : \langle m : (\overrightarrow{\phi_n}) \to \sigma\rangle)\ \&\ \forall\, i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)] \Rightarrow$
$$SN(\langle\mathcal{D},\mathcal{D}_1,\ldots,\mathcal{D}_n,\text{INVK}\rangle :: \Pi \vdash \mathfrak{C}[x].m(\overrightarrow{e_n}) : \sigma).$$

4. $SN(\langle \overrightarrow{\mathcal{D}_n}, \text{OBJ} \rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e_n}) : C) \quad \Leftrightarrow \quad \exists\, \overrightarrow{\phi_n}\ [\forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)]]$.
5. $SN(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \text{JOIN} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n) \quad \Leftrightarrow \quad \forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash e : \sigma_i)]$.
6. $SN(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e : \phi) \quad \Leftrightarrow \quad SN(\mathcal{D} :: \Pi \vdash e : \phi)$.
7. *Let C be a class such that* $\mathcal{F}(C) = \overrightarrow{f_n}$*, then for all* $j \in \overline{n}$*:* $SN(\langle \overrightarrow{\mathcal{D}_n}, \text{NEWF} \rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e_n}) : \langle f_j : \sigma \rangle) \quad \Leftrightarrow \quad \exists\, \overrightarrow{\phi_n}\ [\sigma \trianglelefteq \phi_j\ \&\ \forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)]]$.
8. *Let C be a class such that* $\mathcal{F}(C) = \overrightarrow{f_n}$*, then for all* $j \in \overline{n}$*:* $SN(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \sigma)$ & $\forall i \neq j \in \overline{n}\ [\exists\, \phi\ [SN(\mathcal{D}_i :: \Pi \vdash e_i : \phi)]] \quad \Rightarrow \quad SN(\mathfrak{D}_{(p,\sigma')}[\langle \overrightarrow{\mathcal{D}_n}, \text{NEWF} \rangle, \text{FLD}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new}\ C(\overrightarrow{e_n}).f_j] : \sigma)$.
9. *Let C be a class such that* $\mathcal{M}b(C,m) = (\overrightarrow{x_n}, e_b)$ *and* $\mathcal{D}_b :: \texttt{this}{:}\psi, \overrightarrow{x{:}\phi_n} \vdash e_b : \sigma'$*, then for all derivation contexts* $\mathfrak{D}_{(p,\sigma')}$ *and expression contexts* $\mathfrak{C}$*:* $SN(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_b{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_b{}^{\mathsf{S}}] : \sigma)$ & $SN(\mathcal{D}_0 :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'}) : \psi)$ & $\forall i \in \overline{n}\ [SN(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)] \Rightarrow SN(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \text{INVK} \rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new}\ C(\overrightarrow{e'}).m(\overrightarrow{e_n})] : \sigma)$.
   where $\mathcal{D} = \langle \mathcal{D}_b, \mathcal{D}_0, \text{NEWM} \rangle :: \Pi \vdash \texttt{new}\ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma' \rangle$,
   $\mathcal{S} = \{\texttt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$, *and*
   $\mathsf{S} = \{\texttt{this} \mapsto \texttt{new}\ C(\overrightarrow{e'}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$.

PROOF. These all follow straightforwardly from Definition 47. □

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.

Theorem 49 (Soundness of Derivation Reduction). *If* $\mathcal{D} :: \Pi \vdash e : \phi$ *and* $\mathcal{D} \to_{\mathfrak{D}} \mathcal{D}'$*, then* $\mathcal{D}'$ *is a well-defined derivation, that is there exists some e' such that* $\mathcal{D}' :: \Pi \vdash e' : \phi$*; moreover, then* $e \to e'$ *and if* $\mathcal{D}$ *reduces to* $\mathcal{D}'$ *at position p then* $e \overset{p}{\to} e'$.

PROOF. By induction on the definition of derivation reduction.

We can also show that strong and ω-safe derivations are preserved by derivation reduction.

Lemma 50. *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; notice that derivation reduction does not introduce instances of rule (ω) and that, by (Lemma 40), derivation substitution preserves strong and ω-safe derivations. □

The key step in showing the approximation result below is proving that this notion of derivation reduction is *strongly normalising*, i.e. terminating. In other words, all derivations have a *normal form* with respect to $\to_{\mathfrak{D}}$. Our proof uses the well-known technique of *computability* [40]. 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 51 (Computability). 1. The set of *computable* derivations is defined as the smallest set satisfying the following conditions (where $Comp(\mathcal{D})$ denotes that $\mathcal{D}$ is a member of the set of computable derivations):

$$
\begin{aligned}
Comp(\langle \omega \rangle :: \Pi \vdash e : \omega) & \\
Comp(\mathcal{D} :: \Pi \vdash e : \varphi) \quad &\Leftrightarrow \quad SN(\mathcal{D} :: \Pi \vdash e : \varphi) \\
Comp(\mathcal{D} :: \Pi \vdash e : C) \quad &\Leftrightarrow \quad SN(\mathcal{D} :: \Pi \vdash e : C) \\
Comp(\mathcal{D} :: \Pi \vdash e : \langle f : \sigma \rangle) \quad &\Leftrightarrow \quad Comp(\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash e.f : \sigma) \\
Comp(\mathcal{D} :: \Pi \vdash e : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle) \quad &\Leftrightarrow \quad (\forall \overrightarrow{\mathcal{D}_n} [\, \forall i \in \overline{n}\ [Comp(\mathcal{D}_i :: \Pi_i \vdash e_i : \phi_i)] \Rightarrow \\
& \qquad Comp(\langle \mathcal{D}[\textstyle\bigcap \Pi \cdot \overrightarrow{\Pi_n} \trianglelefteq \Pi], \overrightarrow{\mathcal{D}_i[\bigcap \Pi \cdot \overrightarrow{\Pi_n} \trianglelefteq \Pi_i]}, \text{INVK} \rangle :: \textstyle\bigcap \Pi \cdot \overrightarrow{\Pi_n} \vdash e.m(\overrightarrow{e_n}) : \sigma)\ ]) \\
Comp(\langle \mathcal{D}_1, \ldots, \mathcal{D}_n, \text{JOIN} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n) \quad &\Leftrightarrow \quad \forall i \in \overline{n}\ [Comp(\mathcal{D}_i)]
\end{aligned}
$$

2. A derivation substitution $\mathcal{S} = \{x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$ is *computable in an environment* Π, if and only if, for all $x{:}\phi \in \Pi$ there exists some $i \in \overline{n}$ such that $x = x_i$ and $Comp(\mathcal{D}_i)$.

The weakening operation preserves computability:

Lemma 52. $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e:\phi) \quad \Leftrightarrow \quad \mathit{Comp}(\mathcal{D}[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e:\phi)$.

PROOF. By straightforward induction on the structure of predicates; for the base case, we use Lemma 48(6). □

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

Theorem 53. 1. $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e:\phi) \quad \Rightarrow \quad \mathit{SN}(\mathcal{D} :: \Pi \vdash e:\phi)$.
2. *For neutral contexts* $\mathfrak{C}$, $\mathit{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x]:\phi) \quad \Rightarrow \quad \mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x]:\phi)$.

PROOF. By simultaneous induction on the structure of predicates.

ω**:** By Definition 47 in the case of (1), and by Definition 51 in the case of (2).

φ**,** C**:** Immediate, by Definition 51.

$\langle f:\sigma\rangle$**:** 1. $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e:\langle f:\sigma\rangle) \Rightarrow$ (Def. 51)
$\mathit{Comp}(\langle \mathcal{D}, \text{FLD}\rangle :: \Pi \vdash e.f:\sigma) \Rightarrow$ (IH(1))
$\mathit{SN}(\langle \mathcal{D}, \text{FLD}\rangle :: \Pi \vdash e.f:\sigma) \Rightarrow$ (Lem. 48)
$\mathit{SN}(\mathcal{D} :: \Pi \vdash e:\langle f:\sigma\rangle)$

2. Assume $\mathit{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x]:\langle f:\sigma\rangle)$ with $\mathfrak{C}$ a neutral context. Then $\mathit{SN}(\langle \mathcal{D}, \text{FLD}\rangle :: \Pi \vdash \mathfrak{C}[x].f:\sigma)$ by Lemma 48. Now, let $\mathfrak{C}' = \mathfrak{C}.f$; notice that, by Definitions 27 and 28, $\mathfrak{C}'$ is neutral, and $\mathfrak{C}[x].f = \mathfrak{C}'[x]$. Thus $\mathit{SN}(\langle \mathcal{D}, \text{FLD}\rangle :: \Pi \vdash \mathfrak{C}'[x]:\sigma)$, and, by induction, $\mathit{Comp}(\langle \mathcal{D}, \text{FLD}\rangle :: \Pi \vdash \mathfrak{C}'[x]:\sigma)$. Then, from the definition of $\mathfrak{C}'$, it follows that $\mathit{Comp}(\langle \mathcal{D}, \text{FLD}\rangle :: \Pi \vdash \mathfrak{C}[x].f:\sigma)$, and by Definition 51, we have $\mathit{Comp}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x]:\langle f:\sigma\rangle)$.

$\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle$**:** 1. Assume $\mathit{Comp}(\mathcal{D} :: \Pi \vdash e:\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle)$. 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 $\mathcal{D}_i = \langle\omega\rangle :: \Pi_i \vdash x_i:\omega$, with $\Pi_i = \emptyset$;
- If ϕ_i is a strict predicate σ then $\mathcal{D}_i = \langle \text{VAR}\rangle :: \Pi_i \vdash x_i:\sigma$, with $\Pi_i = x_i{:}\sigma$;
- If $\phi_i = \sigma_1 \cap \ldots \cap \sigma_{n'}$ for some $n' \geq 2$ then $\mathcal{D}_i = \langle \mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}, \text{JOIN}\rangle :: \Pi_i \vdash x:\sigma_1 \cap \ldots \cap \sigma_{n'}$, with $\Pi_i = x_i{:}\phi_i$ and $\mathcal{D}'_j = \langle \text{VAR}\rangle :: \Pi_i \vdash x_i:\sigma_j$ for each $j \in \overline{n'}$.

Notice that each $\mathcal{D}_i$ is in normal form, so $\mathit{SN}(\mathcal{D}_i)$ 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 51,

$$\mathit{Comp}(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'_n}, \text{INVK}\rangle :: \Pi' \vdash e.m(\overrightarrow{x_n}):\sigma)$$

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, $\mathit{SN}(\langle \mathcal{D}', \overrightarrow{\mathcal{D}'_n}, \text{INVK}\rangle)$. Lastly, by Lemma 48(2) we have $\mathit{SN}(\mathcal{D}')$, and by Lemma 48(6), $\mathit{SN}(\mathcal{D})$.

2. Assume $\mathit{SN}(\mathcal{D} :: \Pi \vdash \mathfrak{C}[x]:\langle m:(\overrightarrow{\phi_n}) \to \sigma\rangle)$ with $\mathfrak{C}$ a neutral context. Also, assume that there exist derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ such that: $\mathit{Comp}(\mathcal{D}_i :: \Pi_i \vdash e_i:\phi_i)$ for each $i \in \overline{n}$. Then, by the first induction, $\mathit{SN}(\mathcal{D}_i :: \Pi_i \vdash e_i:\phi_i)$ for each $i \in \overline{n}$. Let $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi_n}$; notice that, by Definition 16, $\Pi' \trianglelefteq \Pi$ and $\Pi' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$. Then, by Lemma 48(6), $\mathit{SN}(\mathcal{D}[\Pi' \trianglelefteq \Pi])$ and $\mathit{SN}(\mathcal{D}_i[\Pi' \trianglelefteq \Pi_i])$ for each $i \in \overline{n}$. By Lemma 48(3) we then have

$$\mathit{SN}(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \text{INVK}\rangle :: \Pi' \vdash \mathfrak{C}[x].m(\overrightarrow{e_n}):\sigma)$$

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 $\mathfrak{C}' = \mathfrak{C}.m(\overrightarrow{e_n})$; notice that, since $\mathfrak{C}$ is neutral, by Definitions 27 and 28, $\mathfrak{C}'$ is also a neutral context and $\mathfrak{C}[x].m(\overrightarrow{e_n}) = \mathfrak{C}'[x]$. Thus, by the second induction,

$$\mathit{Comp}(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \text{INVK}\rangle :: \Pi' \vdash \mathfrak{C}[x].m(\overrightarrow{e_n}):\sigma).$$

Since the derivations $\mathcal{D}_1,\ldots,\mathcal{D}_n$ were arbitrary, the following implication holds

$$\forall\overrightarrow{\mathcal{D}_n}\,[\,\forall i\in\overline{n}\;[Comp(\mathcal{D}_i :: \Pi_i \vdash e_i:\phi_i)]\;\;\Rightarrow\;\;Comp(\langle\mathcal{D}',\mathcal{D}'_1,\ldots,\mathcal{D}'_n,\textsc{invk}\rangle :: \Pi' \vdash \mathfrak{C}[x].m(\overrightarrow{e_n}):\sigma)\,]$$

where $\mathcal{D}' = \mathcal{D}[\Pi' \trianglelefteq \Pi]$ and $\mathcal{D}'_i = \mathcal{D}_i[\Pi' \trianglelefteq \Pi_i]$ for each $i\in\overline{n}$ with $\Pi' = \bigcap\Pi\cdot\overrightarrow{\Pi_n}$. So, by Definition 51, we have $Comp(\mathcal{D} :: \Pi \vdash e:\langle m:(\overrightarrow{\phi_n})\to\sigma\rangle)$. □

$\sigma_1\cap\ldots\cap\sigma_n, n\geq 2$**:** By induction.

Another consequence of Theorem 53 is that identity (derivation) substitutions are computable in their own environments.

Lemma 54. *Let* Π *be a predicate environment; then* $\mathcal{S}_\Pi$ *is computable in* Π.

PROOF. Let $\Pi = x_1{:}\phi_1,\ldots,x_n{:}\phi_n$. So, by Definition 42 $\mathcal{S}_\Pi = \{\,x_1\mapsto\mathcal{D}_1 :: \Pi\vdash x_1:\phi_1,\ldots,x_n\mapsto\mathcal{D}_n :: \Pi\vdash x_1:\phi_1\,\}$. Notice that for each $i\in\overline{n}$ the derivation $\mathcal{D}_i$ contains no derivation redexes, i.e. is in normal form and thus $SN(\mathcal{D}_i)$. Notice also that, since $x_i = \mathfrak{C}[x_i]$ where $\mathfrak{C}$ is the empty context $[\,]$ (see Definition 28), $SN(\mathcal{D}_i :: \Pi\vdash\mathfrak{C}[x]:\phi_i)$ for each $i\in\overline{n}$. Then, by Theorem 53(2) it follows that $Comp(\mathcal{D}_i)$. Thus, for each $x{:}\phi\in\Pi$ there is some $i\in\overline{n}$ such that $x = x_i$ and $Comp(\mathcal{D}_i)$ and so, by Definition 51, $\mathcal{S}_\Pi$ is computable in Π. □

Also using Theorem 53, 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 57) below. We first show two auxiliary lemmas, that are needed for the proof of that lemma.

Lemma 55. *Let C be a class such that* $\mathcal{F}(C) = \overrightarrow{f_n}$*, 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* $Comp(\mathfrak{D}_{(p,\sigma')}[\langle\langle\overrightarrow{\mathcal{D}_n},\textsc{newf}\rangle,\textsc{fld}\rangle] :: \Pi\vdash\mathfrak{C}_p[\mathsf{new}\;\;C(\overrightarrow{e_n}).f_j]:\sigma)$.

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 53 it follows that: $SN(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi\vdash\mathfrak{C}_p[e_j]:\varphi)$ and $\exists\,\phi\;[SN(\mathcal{D}_i :: \Pi\vdash e_i:\phi)]$ for each $i\in\overline{n}$ such that $i\neq j$. Then by Lemma 48(8) we have that

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

And, by Definition 51, it follows that $Comp(\mathfrak{D}_{(p,\sigma')}[\langle\langle\overrightarrow{\mathcal{D}_n},\textsc{newf}\rangle,\textsc{fld}\rangle] :: \Pi\vdash\mathfrak{C}_p[\mathsf{new}\;\;C(\overrightarrow{e_n}).f_j]:\varphi)$.

C**:** Similar to the case for predicate variables.

$\langle f{:}\sigma\rangle$**:** 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 51, it follows that $Comp(\langle\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j],\textsc{fld}\rangle :: \Pi\vdash\mathfrak{C}_p[e_j].f:\sigma)$. 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

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

so by the definition of derivation contexts,

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

Then, by Definition 51, we have $Comp(\mathfrak{D}_{(p,\sigma')}[\langle\langle\overrightarrow{\mathcal{D}_n},\textsc{newf}\rangle,\textsc{fld}\rangle] :: \Pi\vdash\mathfrak{C}_p[\mathsf{new}\;\;C(\overrightarrow{e_n}).f_j]:\langle f:\sigma\rangle)$.

$\langle m:(\overrightarrow{\phi_{n'}}) \to \sigma\rangle$**:** Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_j] :: \Pi \vdash \mathfrak{C}_p[e_j] : \langle m:(\overrightarrow{\phi_{n'}}) \to \sigma\rangle)$ 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 51,

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

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

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

then we have $Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_j[\Pi' \trianglelefteq \Pi]])$. Now, by Lemma 52, $\exists\ \phi\ [Comp(\mathcal{D}_i[\Pi' \trianglelefteq \Pi] :: \Pi' \vdash e_i:\phi)]$ for each $i \neq j \in \overline{n}$. Then by induction,

$$Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi], \text{NEWF}\rangle, \text{FLD}\rangle] :: \Pi' \vdash \mathfrak{C}'_{0\cdot p}[\texttt{new}\ \ C(\overrightarrow{e_n}).f_j] : \sigma)$$

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

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\langle\langle \mathcal{D}_1[\Pi' \trianglelefteq \Pi], \ldots, \mathcal{D}_n[\Pi' \trianglelefteq \Pi], \text{NEWF}\rangle, \text{FLD}\rangle], \overrightarrow{\mathcal{D}''_{n'}}, \text{INVK}\rangle \\ :: \Pi' \vdash \mathfrak{C}_p[\texttt{new}\ \ C(\overrightarrow{e_n}).f_j].m(\overrightarrow{e'_{n'}}) : \sigma)$$

And then, by Definition 31,

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\Pi' \trianglelefteq \Pi][\langle\langle \overrightarrow{\mathcal{D}_n}, \text{NEWF}\rangle, \text{FLD}\rangle[\Pi' \trianglelefteq \Pi]], \overrightarrow{\mathcal{D}''_{n'}}, \text{INVK}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new}\ \ C(\overrightarrow{e_n}).f_j].m(\overrightarrow{e'_{n'}}) : \sigma)$$

And by Lemma 33

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}_n}, \text{NEWF}\rangle, \text{FLD}\rangle][\Pi' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}''_{n'}}, \text{INVK}\rangle :: \Pi' \vdash \mathfrak{C}_p[\texttt{new}\ \ C(\overrightarrow{e_n}).f_j].m(\overrightarrow{e'_{n'}}) : \sigma)$$

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

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

where $\mathcal{D} = \mathfrak{D}_{(p,\sigma)}[\langle\langle \overrightarrow{\mathcal{D}_n}, \text{NEWF}\rangle, \text{FLD}\rangle][\Pi' \trianglelefteq \Pi]$. Thus, by Definition 51, it follows that

$$Comp(\mathfrak{D}_{(p,\sigma')}[\langle\langle \overrightarrow{\mathcal{D}_n}, \text{NEWF}\rangle, \text{FLD}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new}\ \ C(\overrightarrow{e_n}).f_j] : \langle m:(\overrightarrow{\phi_{n'}}) \to \sigma\rangle)$$

Lemma 56. *Let $\mathcal{M}b(C,m) = (\overrightarrow{x_n}, e_b)$ and $\mathcal{D}_b :: \Pi' \vdash e_b:\sigma'$ with $\Pi' = \texttt{this}{:}\psi, x_1{:}\phi_1, \ldots, x_n{:}\phi_n$, then for derivation contexts $\mathfrak{D}_{(p,\sigma')}$ and expression contexts $\mathfrak{C}$: if*

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

then $Comp(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \text{INVK}\rangle] :: \Pi \vdash \mathfrak{C}_p[\texttt{new}\ \ C(\overrightarrow{e'}).m(\overrightarrow{e_n})] : \sigma)$, where

$$\begin{array}{rcl} \mathcal{D} & = & \langle \mathcal{D}_b, \mathcal{D}_0, \text{NEWM}\rangle :: \Pi \vdash \texttt{new}\ \ C(\overrightarrow{e'}) : \langle m:(\overrightarrow{\phi_n}) \to \sigma'\rangle, \\ \mathcal{S} & = & \{\texttt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\},\ \textit{and} \\ \mathsf{S} & = & \{\texttt{this} \mapsto \texttt{new}\ \ C(\overrightarrow{e'}), x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\} \end{array}$$

PROOF. By induction on the structure of strict predicates.

φ: Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_{\mathrm{b}}{}^{\mathsf{S}}] : \varphi), Comp(\mathcal{D}_0 :: \Pi \vdash \mathtt{new}\ C(\overrightarrow{e^{\cdot}}) : \psi)$, and $Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for each $i \in \overline{n}$, where $\mathcal{S} = \{\mathtt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$, and S is the term substitution induced by $\mathcal{S}$.

Then by Theorem 53 it follows that $SN(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_{\mathrm{b}}{}^{\mathsf{S}}] : \varphi)$, $SN(\mathcal{D}_0 :: \Pi \vdash \mathtt{new}\ C(\overrightarrow{e^{\cdot}}) : \psi)$, and $SN(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for each $i \in \overline{n}$.

Then, by Lemma 48(9), we have that $SN(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \mathrm{INVK}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ C(\overrightarrow{e^{\cdot}}).m(\overrightarrow{e_n})] : \varphi)$, where

$$\mathcal{D} = \langle \mathcal{D}_{\mathrm{b}}, \mathcal{D}_0, \mathrm{NEWM}\rangle :: \Pi \vdash \mathtt{new}\ C(\overrightarrow{e^{\cdot}}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma\rangle$$

. And, by Definition 51, we know that $Comp(\mathfrak{D}_{(p,\sigma)}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \mathrm{INVK}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ C(\overrightarrow{e^{\cdot}}).m(\overrightarrow{e_n})] : \varphi)$.

C: Similar to the previous case.

$\langle f : \sigma\rangle$: Assume $Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_{\mathrm{b}}{}^{\mathsf{S}}] : \langle f : \sigma\rangle)$, $Comp(\mathcal{D}_0 :: \Pi \vdash \mathtt{new}\ C(\overrightarrow{e^{\cdot}}) : \psi)$, and $Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$ for all $i \in \overline{n}$, where $\mathcal{S} = \{\mathtt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$, and S is the term substitution induced by $\mathcal{S}$. By Definition 51, it follows that $Comp(\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}], \mathrm{FLD}\rangle :: \Pi \vdash \mathfrak{C}_p[e_{\mathrm{b}}{}^{\mathsf{S}}].f : \sigma)$. Take the contexts $\mathfrak{C}'$ and $\mathfrak{D}'$ such that $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.f$ and $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma')}, \mathrm{FLD}\rangle :: \Pi \vdash \mathfrak{C}_p.f : \sigma$. Notice that

$$\langle \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}], \mathrm{FLD}\rangle :: \Pi \vdash \mathfrak{C}_p[e_{\mathrm{b}}{}^{\mathsf{S}}].f : \sigma = \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[e_{\mathrm{b}}{}^{\mathsf{S}}] : \sigma$$

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

$$Comp(\mathfrak{D}'_{(0\cdot p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \mathrm{INVK}\rangle] :: \Pi \vdash \mathfrak{C}'_{0\cdot p}[\mathtt{new}\ C(\overrightarrow{e^{\cdot}}).m(\overrightarrow{e_n})] : \sigma)$$

where $\mathcal{D} = \langle \mathcal{D}_{\mathrm{b}}, \mathcal{D}_0, \mathrm{NEWM}\rangle :: \Pi \vdash \mathtt{new}\ C(\overrightarrow{e^{\cdot}}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma'\rangle$. So by the definition of $\mathfrak{D}'$,

$$Comp(\langle \mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \mathrm{INVK}\rangle], \mathrm{FLD}\rangle :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ C(\overrightarrow{e^{\cdot}}).m(\overrightarrow{e_n})].f : \sigma)$$

Then, by Definition 51, it follows that

$$Comp(\mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \mathrm{INVK}\rangle] :: \Pi \vdash \mathfrak{C}_p[\mathtt{new}\ C(\overrightarrow{e^{\cdot}}).m(\overrightarrow{e_n})] : \langle f : \sigma\rangle)$$

$\langle m' : (\overrightarrow{\phi'_{n'}}) \to \sigma\rangle$: Assume

$$Comp(\mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}_p[e_{\mathrm{b}}{}^{\mathsf{S}}] : \langle m' : (\overrightarrow{\phi'_{n'}}) \to \sigma\rangle),$$
$$Comp(\mathcal{D}_0 :: \Pi \vdash \mathtt{new}\ C(\overrightarrow{e^{\cdot}}) : \psi),$$

and, for all $i \in \overline{n}$,

$$Comp(\mathcal{D}_i :: \Pi \vdash e_i : \phi_i)$$

where $\mathcal{S} = \{\mathtt{this} \mapsto \mathcal{D}_0, x_1 \mapsto \mathcal{D}_1, \ldots, x_n \mapsto \mathcal{D}_n\}$, and S is the term substitution induced by $\mathcal{S}$. Now, take arbitrary derivations $\mathcal{D}'_1, \ldots, \mathcal{D}'_{n'}$ such that $Comp(\mathcal{D}'_k :: \Pi_k \vdash e''_k : \phi'_k)$ for each $k \in \overline{n'}$. By Definition 51, it follows that

$$Comp(\langle \mathcal{D}', \overrightarrow{\mathcal{D}''_{n'}}, \mathrm{INVK}\rangle :: \Pi' \vdash \mathfrak{C}_p[e_{\mathrm{b}}{}^{\mathsf{S}}].m'(\overrightarrow{e''_{n'}}) : \sigma)$$

where $\Pi'' = \bigcap \Pi \cdot \overrightarrow{\Pi_{n'}}$, $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}][\Pi'' \trianglelefteq \Pi]$, and $\mathcal{D}''_k = \mathcal{D}'_k[\Pi'' \trianglelefteq \Pi_k]$ for each $k \in \overline{n'}$. Then, by Lemma 33, $\mathcal{D}' = \mathfrak{D}_{(p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}][\Pi'' \trianglelefteq \Pi] = \mathfrak{D}_{(p,\sigma')}[\Pi'' \trianglelefteq \Pi][\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi]]$. Take the contexts $\mathfrak{C}'$ and $\mathfrak{D}'$ such that $\mathfrak{C}'_{0\cdot p} = \mathfrak{C}_p.m'(\overrightarrow{e''_{n'}})$ and $\mathfrak{D}'_{(0\cdot p,\sigma')} = \langle \mathfrak{D}_{(p,\sigma')}[\Pi'' \trianglelefteq \Pi], \overrightarrow{\mathcal{D}''_{n'}}, \mathrm{INVK}\rangle :: \Pi'' \vdash \mathfrak{C}_p.m'(\overrightarrow{e''_{n'}}) : \sigma$.

Notice that

$$\langle \mathcal{D}', \overrightarrow{\mathcal{D}''_{n'}}, \mathrm{INVK}\rangle = \mathfrak{D}'_{(0\cdot p,\sigma')}[\mathcal{D}_{\mathrm{b}}{}^{\mathcal{S}}[\Pi'' \trianglelefteq \Pi]] :: \Pi'' \vdash \mathfrak{C}'_{0\cdot p}[e_{\mathrm{b}}{}^{\mathsf{S}}] : \sigma$$

So we have

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

And then by Lemma 41

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

Now, by Lemma 52, $Comp(\mathcal{D}_0[\Pi'' \trianglelefteq \Pi] :: \Pi'' \vdash \mathtt{new}\ \ C(\overrightarrow{e'}) : \psi)$ and $Comp(\mathcal{D}_i[\Pi'' \trianglelefteq \Pi] :: \Pi'' \vdash e_i : \phi_i)$ for all $i \in \overline{n}$. Thus, by induction,

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

where $\mathcal{D}'' = \langle \mathcal{D}_{\mathrm{b}}, \mathcal{D}_0[\Pi'' \trianglelefteq \Pi], \textsc{newm}\rangle :: \Pi'' \vdash \mathtt{new}\ \ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma'\rangle$. So by the definition of $\mathfrak{D}'$

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

Then, by Definition 31,

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

where $\mathcal{D} = \langle \mathcal{D}_{\mathrm{b}}, \mathcal{D}_0, \textsc{newm}\rangle :: \Pi \vdash \mathtt{new}\ \ C(\overrightarrow{e'}) : \langle m : (\overrightarrow{\phi_n}) \to \sigma'\rangle$. And by Lemma 33

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

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

$$\forall \overrightarrow{\mathcal{D}'_{n'}}\,[\ \forall i \in \overline{n}\ \ [Comp(\mathcal{D}'_i :: \Pi_i \vdash e''_i : \phi'_i)] \Rightarrow Comp(\langle \mathcal{D}''', \mathcal{D}''_1, \ldots, \mathcal{D}''_{n'}, \textsc{invk}\rangle :: \Pi'' \vdash e.m(\overrightarrow{e_n}) : \sigma)\ ]$$

where $\mathcal{D}''' = \mathfrak{D}_{(p,\sigma')}[\langle \mathcal{D}, \overrightarrow{\mathcal{D}_n}, \textsc{invk}\rangle][\Pi'' \trianglelefteq \Pi]$ and $\mathcal{D}''_k = \mathcal{D}'_k[\Pi'' \trianglelefteq \Pi_k]$ for each $k \in \overline{n'}$.
So, by Definition 51, we have

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

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 [8], where an approximation result is shown for combinator systems, this lemma must be proved using an *encompassment* relation on terms. Since our notion of reduction is weak (as is the case for combinator systems, and TRS in general) one might think that a similar approach would be necessary for FJ$^{\mathfrak{C}}$. This is not the case however, since our type system incorporates a novel feature: 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 the replacement lemma by straightforward induction on derivations.

Lemma 57 (Replacement). *If $\mathcal{D} :: \Pi \vdash e : \phi$ and $\mathcal{S}$ is a derivation substitution computable in Π and applicable to $\mathcal{D}$, 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 55 and 56 respectively. Let

- $\Pi = x_1{:}\phi'_1, \ldots, x_n{}'{:}\phi'_{n'}$ and
- $\mathcal{S} = \{x'_1 \mapsto \mathcal{D}'_1 :: \Pi' \vdash e''_1 : \phi''_1, \ldots, x'_{n''} \mapsto \mathcal{D}'_{n''} :: \Pi' \vdash e''_{n''} : \phi''_{n''}\}$ with $\{x_1, \ldots, x_{n'}\} \subseteq \{x'_1, \ldots, x'_{n''}\}$.

Also, let S be the term substitution induced by $\mathcal{S}$. Note that if $\mathcal{S}$ is applicable to $\mathcal{D}$ then it is also applicable to subderivations of $\mathcal{D}$.

ω**:** Immediately by Definition 51, since $\mathcal{D}^{\mathcal{S}} = \langle \omega\rangle :: \Pi' \vdash e^{\mathsf{S}} : \omega$.

(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 $\mathcal{D}_j' = \langle \overrightarrow{\mathcal{D}_n''}, \text{JOIN} \rangle$ with $\mathcal{D}_k'' :: \Pi' \vdash e_j'':\sigma_k$ for each $k \in \overline{n}$.

  Now, by Definition 37, $\mathcal{D}^{\mathcal{S}} = \mathcal{D}_i'' :: \Pi' \vdash e_j'':\sigma_i$. Since $\mathcal{S}$ is computable in Π it follows that $Comp(\mathcal{D}_j')$ and then, by Definition 51, that $Comp(\mathcal{D}_k'')$ for each $k \in \overline{n}$. Thus, in particular $Comp(\mathcal{D}_i'')$ and so $Comp(\mathcal{D}^{\mathcal{S}})$.

(FLD): Then $\mathcal{D} = \langle \mathcal{D}', \text{FLD} \rangle :: \Pi \vdash e.f:\sigma$ 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 51, $Comp(\langle \mathcal{D}'^{\mathcal{S}}, \text{FLD} \rangle :: \Pi' \vdash e^{\mathsf{S}}.f:\sigma)$. Notice that $\langle \mathcal{D}'^{\mathcal{S}}, \text{FLD} \rangle = \mathcal{D}^{\mathcal{S}}$ and so $Comp(\mathcal{D}^{\mathcal{S}})$.

(INVK): Then $\mathcal{D} = \langle \mathcal{D}_0, \overrightarrow{\mathcal{D}_n}, \text{INVK} \rangle :: \Pi \vdash e_0.m(\overrightarrow{e_n}):\sigma$ with $\mathcal{D}_0 :: \Pi \vdash e_0:\langle m:(\overrightarrow{\phi_n}) \to \sigma \rangle$ and $\mathcal{D}_i :: \Pi \vdash e_i:\phi_i$ for each $i \in \overline{n}$. By induction, we have

$$Comp(\mathcal{D}_0{}^{\mathcal{S}} :: \Pi' \vdash e_0{}^{\mathsf{S}}:\langle m:(\overrightarrow{\phi_n}) \to \sigma \rangle) \;\&\; \forall i \in \overline{n}\ [\,Comp(\mathcal{D}_i{}^{\mathcal{S}} :: \Pi' \vdash e_i{}^{\mathsf{S}}:\phi_i)\,]$$

Then, by Definition 51, it follows that

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

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 it follows that $Comp(\langle \mathcal{D}_0{}^{\mathcal{S}}, \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \text{INVK} \rangle :: \Pi' \vdash e_0{}^{\mathsf{S}}.m(e_1{}^{\mathsf{S}}, \ldots, e_n{}^{\mathsf{S}}):\sigma)$. Notice that $\langle \mathcal{D}_0{}^{\mathcal{S}}, \mathcal{D}_1{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \text{INVK} \rangle = \mathcal{D}^{\mathcal{S}}$ and so $Comp(\mathcal{D}^{\mathcal{S}})$.

(JOIN), (OBJ): By induction.

(NEWF): Then $\mathcal{D} = \langle \overrightarrow{\mathcal{D}_n}, \text{NEWF} \rangle :: \Pi \vdash \texttt{new}\ \ C(\overrightarrow{e_n}):\langle f_j:\sigma \rangle$ with $\mathcal{F}(C) = \overrightarrow{f_n}$ and $j \in \overline{n}$, and there is some $\overrightarrow{\phi_n}$ 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 $\mathfrak{D}_{(0,\sigma)} = \langle [\,] \rangle$ and $\mathfrak{C} = [\,]$. Notice that

$$\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_j{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}[e_j{}^{\mathsf{S}}]:\sigma \quad = \quad \mathcal{D}_j{}^{\mathcal{S}} :: \Pi \vdash e_j{}^{\mathsf{S}}:\phi_j$$

and so $Comp(\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_j{}^{\mathcal{S}}] :: \Pi \vdash \mathfrak{C}[e_j{}^{\mathsf{S}}]:\sigma)$. Then by Lemma 55 it follows that

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

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

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

Then, by Definition 51, we have that $Comp(\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \text{NEWF} \rangle :: \Pi \vdash \texttt{new}\ \ C(e_1{}^{\mathsf{S}}, \ldots, e_n{}^{\mathsf{S}}):\langle f_j:\sigma \rangle)$. Notice that $\langle \mathcal{D}_i{}^{\mathcal{S}}, \ldots, \mathcal{D}_n{}^{\mathcal{S}}, \text{NEWF} \rangle = \mathcal{D}^{\mathcal{S}}$ and so $Comp(\mathcal{D}^{\mathcal{S}})$.

(NEWM): Then $\mathcal{D} = \langle \mathcal{D}_{\mathrm{b}}, \mathcal{D}_0, \text{NEWM} \rangle :: \Pi \vdash \texttt{new}\ \ C(\vec{e}):\langle m:(\overrightarrow{\phi_n}) \to \sigma \rangle$ with $\mathcal{M}b(C,m) = (\overrightarrow{x_n''}, e_{\mathrm{b}})$ such that both $\mathcal{D}_{\mathrm{b}} :: \Pi'' \vdash e_{\mathrm{b}}:\sigma$ and $\mathcal{D}_0 :: \Pi \vdash \texttt{new}\ \ C(\vec{e}):\psi$ where $\Pi'' = \texttt{this}{:}\psi, x_1'':\phi_1, \ldots, x_n'':\phi_n$. By induction, we have $Comp(\mathcal{D}_0{}^{\mathcal{S}} :: \Pi' \vdash \texttt{new}\ \ C(\vec{e})^{\mathsf{S}}:\psi)$. Now, assume there exist derivations $\mathcal{D}_1 :: \Pi_1 \vdash e_1':\phi_1$, …, $\mathcal{D}_1 :: \Pi_n \vdash e_n':\phi_n$ such that $Comp(\mathcal{D}_i)$ for each $i \in \overline{n}$. Let $\Pi''' = \bigcap \Pi' \cdot \overrightarrow{\Pi_n}$; notice that $\Pi''' \trianglelefteq \Pi_i$ for each $i \in \overline{n}$ so from Lemma 32 it follows that $Comp(\mathcal{D}_i[\Pi''' \trianglelefteq \Pi_i] :: \Pi''' \vdash e_i':\phi_i)$ for each $i \in \overline{n}$. Also $\Pi''' \trianglelefteq \Pi'$ and so then too by Lemma 32 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

$$\mathcal{S}' = \{\, \texttt{this} \mapsto \mathcal{D}_0{}^{\mathcal{S}}[\Pi''' \trianglelefteq \Pi'],\ x_1'' \mapsto \mathcal{D}_1[\Pi''' \trianglelefteq \Pi_1],\ \ldots,\ x_n'' \mapsto \mathcal{D}_n[\Pi''' \trianglelefteq \Pi_n] \,\}$$

Notice that $\mathcal{S}'$ is computable in Π'' and applicable to $\mathcal{D}_\mathrm{b}$. So by induction, $Comp(\mathcal{D}_\mathrm{b}{}^{\mathcal{S}'} :: \Pi''' \vdash e_\mathrm{b}{}^{\mathsf{S}'} : \sigma)$ where S' is the term substitution induced by $\mathcal{S}'$. Taking the derivation context $\mathfrak{D}_{(0,\sigma)} = \langle [] \rangle$ and the expression context $\mathfrak{C} = []$, notice that

$$\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_\mathrm{b}{}^{\mathcal{S}'}] :: \Pi''' \vdash \mathfrak{C}[e_\mathrm{b}{}^{\mathsf{S}'}] : \sigma \quad = \quad \mathcal{D}_\mathrm{b}{}^{\mathcal{S}'} :: \Pi''' \vdash e_\mathrm{b}{}^{\mathsf{S}'} : \sigma$$

and so $Comp(\mathfrak{D}_{(0,\sigma)}[\mathcal{D}_\mathrm{b}{}^{\mathcal{S}'}] :: \Pi''' \vdash \mathfrak{C}[e_\mathrm{b}{}^{\mathsf{S}'}] : \sigma)$. From Lemma 56 we then have

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

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

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

Notice that $\mathcal{D}' = \mathcal{D}^{\mathcal{S}}[\Pi''' \trianglelefteqslant \Pi']$. Since the existence of the derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ was assumed, the following implication holds:

$$\forall \overrightarrow{\mathcal{D}_n}\ [Comp(\mathcal{D}_i :: \Pi_i \vdash e'_i : \phi_i)] \quad \Rightarrow \quad Comp(\langle \mathcal{D}', \mathcal{D}'_1, \ldots, \mathcal{D}'_n, \textsc{invk} \rangle :: \Pi''' \vdash \mathsf{new}\ C(\vec{e}).m(\overrightarrow{e'_n}) : \sigma)$$

where $\mathcal{D}'_i = \mathcal{D}_i[\Pi''' \trianglelefteqslant \Pi_i]$ for each $i \in \overline{n}$, with $\Pi''' = \bigcap \Pi' \cdot \overrightarrow{\Pi_n}$. So, by Definition 51, it follows that $Comp(\mathcal{D}^{\mathcal{S}} :: \Pi' \vdash \mathsf{new}\ C(\vec{e})^{\mathsf{S}} : \langle m : (\overrightarrow{\phi_n}) \to \sigma \rangle)$. □

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

Lemma 58. $\mathcal{D} :: \Pi \vdash e : \phi \quad \Rightarrow \quad Comp(\mathcal{D} :: \Pi \vdash e : \phi)$.

PROOF. Suppose $\Pi = x_1{:}\phi_1, \ldots, x_n{:}\phi_n$, then we take the identity substitution

$$\mathcal{S}_\Pi = \{ x_1 \mapsto \mathcal{D}_1 :: \Pi \vdash x_1 : \phi_1, \ldots, x_n \mapsto \mathcal{D}_n :: \Pi \vdash x_n : \phi_n \}$$

Notice that this is computable in Π (Lemma 54). Notice also that, by Definition 37, $\mathcal{S}_\Pi$ is applicable to $\mathcal{D}$. Then from Lemma 57 we have $Comp(\mathcal{D}^{\mathcal{S}_\Pi})$, and since by Proposition 43 $\mathcal{D}^{\mathcal{S}_\Pi} = \mathcal{D}$ it follows that $Comp(\mathcal{D})$. □

Then the key step to the approximation theorem follows directly.

Theorem 59 (Strong Normalisation for Derivation Reduction). *If $\mathcal{D} :: \Pi \vdash e : \phi$ then $SN(\mathcal{D})$.*

PROOF. By Lemma 58 and Theorem 53(1). □

6. Linking Types with Semantics: The Approximation Result

We will now describe 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 for every typeable approximant of an expression, the same type can be assigned to the expression itself, and vice-versa:

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

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 which can be typed. This is a very strong property since, as we will explain, it means that typeability provides a sufficient condition for the (head) normalisation of *expressions*, i.e. a *termination* analysis for FJ$^{\mathfrak{C}}$.

Finally, the following properties of approximants and predicate assignment lead to the approximation result itself.

Lemma 60. *If $\mathcal{D} :: \Pi \vdash a : \phi$ (with $\mathcal{D}$ ω-safe) and $a \sqsubseteq a'$ then there exists a derivation $\mathcal{D}' :: \Pi \vdash a' : \phi$ (where $\mathcal{D}'$ is ω-safe).*

PROOF. By induction on the structure of derivations. □

Lemma 61. *Let $A_1,\ldots,A_n$ be approximate normal forms with $n \geq 2$ and e be an expression such that $A_i \sqsubseteq e$ for each $i \in \overline{n}$. If there are (ω-safe) derivations $\mathcal{D}_1,\ldots,\mathcal{D}_n$ such that $\mathcal{D}_i :: \Pi \vdash A_i : \phi_i$ for each $i \in \overline{n}$, then $\sqcup \overrightarrow{A_n} \sqsubseteq e$ and there are (ω-safe) derivations $\mathcal{D}'_1,\ldots,\mathcal{D}'_n$ such that $\mathcal{D}'_i :: \Pi \vdash \sqcup \overrightarrow{A_n} : \phi_i$ for each $i \in \overline{n}$. Moreover, $\sqcup \overrightarrow{A_n}$ is also an approximate normal form.*

PROOF. By induction on the number of approximants.

$n = 2$**:** Then there are A_1 and A_2 such that $A_1 \sqsubseteq e$ and $A_2 \sqsubseteq e$. By Lemma 10, $A_1 \sqcup A_2 \sqsubseteq e$, with $A_1 \sqcup A_2$ an approximate normal form, 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 $\mathcal{D}_2$), it follows from Lemma 60 that 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 $\mathcal{D}'_2 :: \Pi \vdash A_1 \sqcup A_2 : \phi_2$. The result then follows from the fact that, by Lemma 10, $\sqcup \overrightarrow{A_2} = A_1 \sqcup A_2$.

$n > 2$**:** By assumption, $A_i \sqsubseteq e$ and $\mathcal{D}_i :: \Pi \vdash A_i : \phi_i$ (with $\mathcal{D}_i$ ω-safe) for each $i \in \overline{n}$. Notice that $\overrightarrow{A_n} = A_1 \cdot \overrightarrow{A'_{n'}}$ where $n = n' + 1$ and $A'_i = A_{i+1}$ for each $i \in \overline{n'}$. Thus $A'_i \sqsubseteq e$ for each $i \in \overline{n'}$ and $\mathcal{D}_{i+1} :: \Pi \vdash A'_i : \phi_{i+1}$ for each $i \in \overline{n'}$. Therefore, by induction, $\sqcup \overrightarrow{A'_{n'}} \sqsubseteq e$ with $\sqcup \overrightarrow{A'_{n'}}$ an approximate normal form, and $\mathcal{D}'_i :: \Pi \vdash \sqcup \overrightarrow{A'_{n'}} : \phi_{i+1}$ (with $\mathcal{D}'_i$ ω-safe) for each $i \in \overline{n'}$. Then we have by Lemma 10 that $A_1 \sqcup (\sqcup \overrightarrow{A'_{n'}}) \sqsubseteq e$ with $A_1 \sqcup (\sqcup \overrightarrow{A'_{n'}})$ an approximate normal form, $A_1 \sqsubseteq A_1 \sqcup (\sqcup \overrightarrow{A'_{n'}})$, and $\sqcup \overrightarrow{A'_{n'}} \sqsubseteq A_1 \sqcup (\sqcup \overrightarrow{A'_{n'}})$. So by Lemma 60 there is a derivation $\mathcal{D}'''$ (with $\mathcal{D}'''$ ω-safe) such that $\mathcal{D}''' :: \Pi \vdash A_1 \sqcup (\sqcup \overrightarrow{A'_{n'}}) : \phi_1$ and (ω-safe) derivations $\mathcal{D}''_1,\ldots,\mathcal{D}''_{n'}$ such that $\mathcal{D}''_i :: \Pi \vdash A_1 \sqcup (\sqcup \overrightarrow{A'_{n'}}) : \phi_{i+1}$ for each $i \in \overline{n'}$. The result then follows from the fact that, by Definition 9, $\sqcup \overrightarrow{A_n} = A_1 \sqcup (\sqcup \overrightarrow{A'_{n'}})$. □

Lemma 62. *If $\mathcal{D} :: \Pi \vdash e : \phi$ (with $\mathcal{D}$ ω-safe) and $\mathcal{D}$ is in normal form with respect to $\rightarrow_{\mathfrak{D}}$, then there exists A and (ω-safe) $\mathcal{D}'$ such that $A \sqsubseteq e$ and $\mathcal{D}' :: \Pi \vdash A : \phi$.*

PROOF. By induction on the structure of derivations.

ω**:** Take $A = \bot$. Notice that $\bot \sqsubseteq e$, by Definition 7, and by (ω) we can take $\mathcal{D}' = \langle \omega \rangle :: \Pi \vdash \bot : \omega$.

In the ω-safe version of the result, this case is vacuously true since the derivation $\mathcal{D} = \langle \omega \rangle :: \Pi \vdash e : \omega$ is not ω-safe.

VAR**:** Then $e = x$ and $\mathcal{D} = \langle \text{VAR} \rangle :: \Pi \vdash x : \sigma$ (notice that this is a derivation in normal form). By Definition 6, x is already an approximate normal form and $x \sqsubseteq x$, by Definition 7. So we take $A = x$ and $\mathcal{D}' = \mathcal{D}$. Moreover, notice that, by Definition 34, $\mathcal{D}$ is an ω-safe derivation.

JOIN**:** Then $\mathcal{D} = \langle \overrightarrow{\mathcal{D}_n}, \text{JOIN} \rangle :: \Pi \vdash e : \sigma_1 \cap \ldots \cap \sigma_n$ 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 34, each $\mathcal{D}_i$ is ω-safe too). By induction, there then exist $A_1,\ldots,A_n$ and (ω-safe) derivations $\mathcal{D}'_1,\ldots,\mathcal{D}'_n$ 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 61 it follows that $\sqcup \overrightarrow{A_n} \sqsubseteq e$ with $\sqcup \overrightarrow{A_n}$ normal and that there are (ω-safe) derivations $\mathcal{D}''_1,\ldots,\mathcal{D}''_n$ such that $\mathcal{D}''_i :: \Pi \vdash \sqcup \overrightarrow{A_n} : \sigma_i$ for each $i \in \overline{n}$. Finally, by the (JOIN) rule we can take (ω-safe) $\mathcal{D}' = \langle \overrightarrow{\mathcal{D}''_n}, \text{JOIN} \rangle :: \Pi \vdash \sqcup \overrightarrow{A_n} : \sigma_1 \cap \ldots \cap \sigma_n$.

FLD**:** Then $e = e'.f$ and $\mathcal{D} = \langle \mathcal{D}', \text{FLD} \rangle :: \Pi \vdash e'.f : \sigma$ 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 34, 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 (FLD), $\langle \mathcal{D}'', \text{FLD} \rangle :: \Pi \vdash A.f : \sigma$ and, by Definition 7, $A.f \sqsubseteq e'.f$. Moreover, by Definition 34, when $\mathcal{D}$" is ω-safe so too is $\langle \mathcal{D}'', \text{FLD} \rangle$.

INVK, OBJ, NEWF, NEWM**:** These cases follow straightforwardly by induction similar to (FLD). □

Lemma 60 above simply states the soundness of type assignment with respect to the approximation relation. Lemma 62 is the more interesting, since it is this that 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 $\langle \omega \rangle :: \Pi \vdash e : \omega$ by $\langle \omega \rangle :: \Pi \vdash \bot : \omega$ (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 approximate normal form. 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 63 (Approximation). *$\Pi \vdash e:\phi$ 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 60, $\Pi \vdash e':\phi$, and then by subject expansion (Theorem 20), also $\Pi \vdash e:\phi$.

only if: Let $\mathcal{D} :: \Pi \vdash e:\phi$, then, by Theorem 59, $\mathcal{D}$ is strongly normalising. Take the normal form $\mathcal{D}'$; by the soundness of derivation reduction (Theorem 49), $\mathcal{D}' :: \Pi \vdash e':\phi$ and $e \rightarrow^* e'$. By Lemma 62, there is some approximate normal form A such that $\Pi \vdash A:\phi$ and $A \sqsubseteq e'$. Thus, by Definition 11, $A \in \mathcal{A}(e)$. □

Termination Analysis

As in other intersection type systems [3, 8], the approximation theorem underpins characterisation results for various forms of termination. Our predicate 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 6 correspond in structure to standard expressions in (head) normal form.

Definition 64 (Normal Forms). 1. The set of (well-formed) *head-normal forms* (ranged over by H) is defined by:

$$\begin{array}{rcll} H & ::= & x \mid \mathsf{new}\ C(\overrightarrow{e_n}) & (\mathcal{F}(C) = \overrightarrow{f_n}) \\ & & \mid H.f \mid H.m(\vec{e}) & (H \neq \mathsf{new}\ C(\vec{e})) \end{array}$$

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

$$\begin{array}{rcll} N & ::= & x \mid \mathsf{new}\ C(\overrightarrow{N_n}) & (\mathcal{F}(C) = \overrightarrow{f_n}) \\ & & \mid N.f \mid N.m(\vec{N}) & (N \neq \mathsf{new}\ C(\vec{N})) \end{array}$$

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. Also note that we stipulate that a (head) normal expression of the form $\mathsf{new}\ C(\vec{e})$ *must* have the correct number of field values as defined in the declaration of class C. Expressions of this form with either less or more field values may *technically* constitute (head) normal forms, but we discount them as malformed since they do not 'morally' constitute valid objects according to the class table.

Lemma 65. 1. *If $A \neq \bot$ and $A \sqsubseteq e$, then e is a head-normal form.*
2. *If $A \sqsubseteq e$ and A does not contain $\bot$, then e is a normal form.*

PROOF. By straightforward induction on the structure of A using Definition 7. □

Thus any predicate, or more accurately any predicate derivation other than those of the form $\langle\omega\rangle :: \Pi \vdash e:\omega$ (which correspond to the approximant $\bot$), specifies the structure of a (head) normal form via the normal form of its derivation. From the approximation result, the following characterisation of head-normalisation follows easily.

Lemma 66 (Typeability of (head) normal forms). 1. *If e is a head-normal form then there exists a strict predicate σ and predicate environment Π such that $\Pi \vdash e:\sigma$; moreover, if e is not of the form $\mathsf{new}\ C(\overrightarrow{e_n})$ then for any arbitrary strict predicate σ there is an environment such that $\Pi \vdash e:\sigma$.*
2. *If e is a normal form then there exist strong strict predicate σ, predicate environment Π and derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash e:\sigma$; moreover, if e is not of the form $\mathsf{new}\ C(\overrightarrow{e_n})$ then for any arbitrary strong strict predicate there exist strong $\mathcal{D}$ and Π such that $\mathcal{D} :: \Pi \vdash e:\sigma$.*

PROOF. 1. By induction on the structure of head-normal forms.

x: By the (VAR) rule, $x{:}\sigma \vdash x:\sigma$ for any arbitrary strict predicate.

- $\texttt{new}\ C(\overrightarrow{e_n})$**:** Notice that $\mathcal{F}(C) = \overrightarrow{f_n}$, by definition of the head-normal form. Let us take the empty predicate environment, $\emptyset$. Notice that by rule (ω) we can derive $\emptyset \vdash e_i : \omega$ for each $i \in \overline{n}$. Then, by rule (OBJ) we can derive $\emptyset \vdash \texttt{new}\ C(\overrightarrow{e_n}) : C$.

- $H.f$**:** Notice that, by definition, H is a head-normal expression *not* of the form $\texttt{new}\ C(\overrightarrow{e_n})$, thus by induction for any arbitrary strict predicate σ there is an environment Π such that $\Pi \vdash H : \sigma$. Let us pick some (other) arbitrary strict predicate σ', then there is an environment Π such that $\Pi \vdash H : \langle f : \sigma' \rangle$. Thus, by rule (FLD) we can derive $\Pi \vdash H.f : \sigma'$ for any arbitrary strict predicate σ'.

- $H.m(\overrightarrow{e_n})$**:** This case is very similar to the previous one. Notice that, by definition, H is a head-normal expression *not* of the form $\texttt{new}\ C(\overrightarrow{e_n})$, thus by induction for any arbitrary strict predicate σ there is an environment Π such that $\Pi \vdash H : \sigma$. Let us pick some (other) arbitrary strict predicate σ', then there is an environment Π such that $\Pi \vdash H : \langle m : (\overrightarrow{\omega_n}) \to \sigma' \rangle$. Notice that by rule (ω) we can derive $\Pi \vdash e_i : \omega$ for each $i \in \overline{n}$. Thus, by rule (INVK) we can derive $\Pi \vdash H.m(\overrightarrow{e_n}) : \sigma'$ for any arbitrary strict predicate σ'.

2. By induction on the structure of normal forms.

   - x**:** By the (VAR) rule, $x{:}\sigma \vdash x : \sigma$ for any arbitrary strict predicate, and in particular this holds for any arbitrary *strong* strict predicate. Also, notice that derivations of the form $\langle \text{VAR} \rangle$ are strong by Definition 34.

   - $\texttt{new}\ C(\overrightarrow{N_n})$**:** Notice that $\mathcal{F}(C) = \overrightarrow{f_n}$ by the definition of normal forms. Since each N_i is a normal form for $i \in \overline{n}$, it follows by induction that there are strong strict predicates $\overrightarrow{\sigma_n}$, environments $\overrightarrow{\Pi_n}$ and derivations $\overrightarrow{\mathcal{D}_n}$ such that $\mathcal{D}_i :: \Pi_i \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Let the environment $\Pi' = \bigcap \overrightarrow{\Pi_n}$; notice that, by Definition 16, $\Pi' \trianglelefteqslant \Pi_i$ for each $i \in \overline{n}$, and also that since each Π_i is strong so is Π'. Thus, $[\Pi' \trianglelefteqslant \Pi_i]$ is a weakening for each $i \in \overline{n}$ and $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i] :: \Pi' \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Notice that, by Definition 31, weakening does not change the structure of derivations, therefore for each $i \in \overline{n}$, $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ is a strong derivation. Now, by rule (OBJ) we can derive

   $$\langle \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \text{OBJ} \rangle :: \Pi' \vdash \texttt{new}\ C(\overrightarrow{N_n}) : C$$

   Notice that C is a strong strict predicate, and that since each derivation $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ is strong then, by Definition 34, so is $\langle \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \text{OBJ} \rangle$.

   - $N.f$**:** Notice that, by definition, N is a normal expression *not* of the form $\texttt{new}\ C(\overrightarrow{N_n})$, thus by induction for any arbitrary strong strict predicate σ there is a strong environment Π and derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \sigma$. Let us pick some (other) arbitrary strong strict predicate σ', then there are strong Π and $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \langle f : \sigma' \rangle$. Thus, by rule (FLD) we can derive $\langle \mathcal{D}, \text{FLD} \rangle :: \Pi \vdash N.f : \sigma'$ for any arbitrary strong strict predicate σ'. Furthermore, notice that since $\mathcal{D}$ is strong it follows from Definition 34 that $\langle \mathcal{D}, \text{FLD} \rangle$ is also strong.

   - $N.m(\overrightarrow{N_n})$**:** Since each N_i for $i \in \overline{n}$ is a normal form it follows by induction that there are strong strict predicates $\overrightarrow{\sigma_n}$, environments $\overrightarrow{\Pi_n}$ and derivations $\overrightarrow{\mathcal{D}_n}$ such that $\mathcal{D}_i :: \Pi_i \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Also notice that, by definition, N is a normal expression *not* of the form $\texttt{new}\ C(\overrightarrow{N_n})$, thus by induction for any arbitrary strict predicate σ there is a strong environment Π and derivation $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \sigma$. Let us pick some (other) arbitrary strong strict predicate σ', then $\langle m : (\overrightarrow{\sigma_n}) \to \sigma' \rangle$ is also strong and there are Π and $\mathcal{D}$ such that $\mathcal{D} :: \Pi \vdash N : \langle m : (\overrightarrow{\sigma_n}) \to \sigma' \rangle$. Let the environment $\Pi' = \bigcap \Pi \cdot \overrightarrow{\Pi_n}$ notice that, by Definition 16, $\Pi' \trianglelefteqslant \Pi$ and $\Pi' \trianglelefteqslant \Pi_i$ for each $i \in \overline{n}$, and also that since Π is strong and each Π_i is strong then so is Π'. Thus, $[\Pi' \trianglelefteqslant \Pi]$ is a weakening and $[\Pi' \trianglelefteqslant \Pi_i]$ is a weakening for each $i \in \overline{n}$. Then $\mathcal{D}[\Pi' \trianglelefteqslant \Pi] :: \Pi' \vdash N : \langle m : (\overrightarrow{\sigma_n}) \to \sigma' \rangle$ and $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i] :: \Pi' \vdash N_i : \sigma_i$ for each $i \in \overline{n}$. Notice that, by Definition 31, weakening does not change the structure of derivations, therefore $\mathcal{D}[\Pi' \trianglelefteqslant \Pi]$ is strong and for each $i \in \overline{n}$, $\mathcal{D}_i[\Pi' \trianglelefteqslant \Pi_i]$ is also strong. Now, by rule (INVK)

   $$\langle \mathcal{D}[\Pi' \trianglelefteqslant \Pi], \mathcal{D}_1[\Pi' \trianglelefteqslant \Pi_1], \ldots, \mathcal{D}_n[\Pi' \trianglelefteqslant \Pi_n], \text{INVK} \rangle :: \Pi' \vdash N.m(\overrightarrow{N_n}) : \sigma'$$

   for any arbitrary strong strict predicate σ'. Furthermore, by Definition 34, we have that

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

   is a strong derivation. □

Theorem 67 (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 66(1) there exists a strict predicate σ and a predicate environment Π such that $\Pi \vdash e':\sigma$. Then by subject expansion (Theorem 20) 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 65 *e'* is a head-normal form. □

Recall the Lambda Calculus characterisation of normalisability in ITD:

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

An analogous result does not hold for FJ$^{\mathfrak{C}}$ (see the third example in Example 35 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 68 (Normalisation). $\mathcal{D} :: \Pi \vdash e:\sigma$ *with* $\mathcal{D}$ *and* Π ω*-safe only if 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 50), it follows that $\mathcal{D}'$ is ω-safe and thus by Lemma 36 that A does not contain $\bot$. Then by Lemma 65 we have that *e'* is a normal form. □

On the other hand, the reverse implication does not hold in general since our notion of ω-safe typeability is too fragile: it 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 $\Pi \vdash \texttt{new C}(\overrightarrow{e_n})\texttt{.}f_i:\sigma$ (due to non-termination in the other expressions e_j) even though this expression has the same normal form as e_i. A completeness result *does* hold when we restrict our attention to the image of CL terms in OOCL, as shown later in Theorem 73.

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 69 (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 and therefore each possible reduction of e is matched by a corresponding derivation reduction of $\mathcal{D}$. By Lemma 50 it follows that no reduction of $\mathcal{D}$ introduces subderivations of the form $\langle\omega\rangle$, and so since $\mathcal{D}$ is strongly normalising (Theorem 59) 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. □

We will illustrate our results by applying them in the context of OOCL.

Definition 70 (OOCL normal forms). Let the set of OOCL normal forms be the set of expressions

$$\{\ e \mid \text{there exists a CL term } t \text{ such that } e \text{ is the normal form of } [\![t]\!]\ \}$$

Notice that it can be defined by the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid e\texttt{.app}(e') \quad (e \neq \texttt{new } C(\overrightarrow{e_n})) \mid \\ & & \texttt{new K()} \mid \texttt{new K}_1(e) \mid \texttt{new S()} \mid \texttt{new S}_1(e) \mid \texttt{new S}_2(e_1,e_2) \end{array}$$

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 predicate assignable to the normal form.

Lemma 71. *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 72. *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 predicate* 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 73. *Let t be a* CL*-term: then t is normalisable, if and only if, there are* ω*-safe* $\mathcal{D}$ *and* Π*, and strict predicate* σ *such that* $\mathcal{D} :: \Pi \vdash [\![t]\!] : \sigma$.

PROOF. **if:** Directly by Theorem 68.

only if: Let *t'* be the normal from of *t*; then, by Theorem 22, $[\![t]\!] \to [\![t']\!]$. It is straightforward to show that then $[\![t']\!]$ is normalisable as well; let *e* be the normal form of $[\![t']\!]$. Then by Lemma 66(2) there are strong strict predicate σ, environment Π and derivation $\mathcal{D}$ such that $\Pi \vdash e : \sigma$. Since $\mathcal{D}$ and Π are strong, they are also ω-safe. Then, by Lemma 71 and 72, there exists ω-safe $\mathcal{D}'$ such that $\mathcal{D}' :: \Pi \vdash [\![t]\!] : \sigma$. □

7. Some Worked Examples

We will now give a more concrete idea of how the concepts outlined in the previous section work, 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 predicate 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.

A Fixed-point Construction

The *fixed-point* of a function f is a value x such that $x = f(x)$. A fixed-point *combinator* is a (higher-order) function that returns a fixed-point of its argument (another function). Thus, a fixed-point combinator g has the property that $g\,f = f(g\,f)$ for any function f. Turing's well-known fixed-point combinator in the λ-calculus is the following term:

$$\mathit{Tur} \quad = \quad \Theta\Theta \quad = \quad (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

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

$$\mathit{Tur}\ f \quad = \quad (\lambda xy.y(xxy))\Theta f \quad \to^*_\beta \quad f(\Theta\Theta f) \quad = \quad f(\mathit{Tur}\ f)$$

The term *Tur* itself has the reduction behaviour

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

which implies it has the following set of approximants:

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

Thus, if z is a term variable, the approximants of *Tur* z are $\bot, z\bot, z(z\bot)$, etc. As well as satisfying the characteristic property of fixed-point combinators mentioned above, the term *Tur* satisfies the stronger property that *Tur* $M \rightarrow_\beta^* M(\textit{Tur}\ M)$ for any term M.

It is straightforward to define an FJ¢ program which mirrors this behaviour:

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

Following from the example of Section 4 we have implemented the fixed point combinator *Tur* using a class `T` which conforms to the `Combinator` 'interface', in which term application is modelled via an `app` method. The body of the `app` method in the class `T` encodes the reduction behaviour we saw for *Tur* above.

For any FJ¢ expression e:

$$\texttt{new T().app}(e) \quad \rightarrow \quad e\texttt{.app(new T().app}(e)\texttt{)}$$

So, taking $M = \texttt{new T().app}(e)$, we have

$$M \quad \rightarrow \quad e\texttt{.app}(M)$$

Thus, by Theorem 12, the fixed point M of e (as returned by the fixed point combinator class `T`) is semantically equivalent to $e\texttt{.app}(M)$, and so `new T().app(`$\cdot$`)` does indeed represent a fixed-point constructor.

The (executable) expression $e = \texttt{new T().app(z)}$ has the reduction behaviour

$$\begin{array}{rcl} \texttt{new T().app(z)} & \rightarrow & \texttt{x.app(this.app(x))}\ [\texttt{new T/this,z/x}] \\ & = & \texttt{z.app(new T.app(z))} \\ & \rightarrow & \texttt{z.app(z.app(new T.app(z))} \\ & \vdots & \end{array}$$

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. The derivation $\mathcal{D}_1$ in Figure 8 shows a possible derivation assigning the predicate φ to e. In fact, the normal form of this derivation corresponds to the approximant `z.app(`$\bot$`)`, which we will now demonstrate.

The derivation $\mathcal{D}_1$ comprises a *typed redex*, i.e. a derivation of the form $\langle\langle\cdot,\cdot,\text{NEWM}\rangle,\vec{\cdot},\text{INVK}\rangle$, thus it will reduce. The derivation $\mathcal{D}_2$ shows the result of preforming the reduction step. In this example, the predicate ω is assigned to the receiver `new T()`, since that is the predicate associated with `this` in the environment Π_2 used when typing the method body. It would have been possible to use a more specific predicate for `this` in Π_2 (consequently requiring a more structured subderivation for the receiver), but even had we done so the information contained in this subderivation would have been 'thrown away' by the derivation substitution operation during the reduction step, since the occurrence of the variable `this` in the method body is still covered by ω (i.e. any information about `this` in the environment Π_2 is not used).

The derivation $\mathcal{D}_2$ 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 $\mathcal{D}_3$ as given in the figure.

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.

$$\mathcal{D}_1 :: \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi_2 \vdash \mathtt{x}:\langle \mathtt{app}:(\omega)\to\varphi_2\rangle}\,(\text{VAR}) \quad \dfrac{}{\Pi_2 \vdash \mathtt{this.app(x)}:\omega}\,(\omega)}{\Pi_2 \vdash \mathtt{x.app(this.app(x))}:\varphi}\,(\text{INVK}) \quad \dfrac{}{\Pi_1 \vdash \mathtt{new\ T()}:\omega}\,(\omega)}{\Pi_1 \vdash \mathtt{new\ T()}:\langle \mathtt{app}:(\langle \mathtt{app}:(\omega)\to\varphi\rangle)\to\varphi\rangle}\,(\text{NEWM}) \quad \dfrac{\dfrac{}{\Pi_1 \vdash \mathtt{z}:\langle \mathtt{app}:(\omega)\to\varphi\rangle}\,(\text{VAR})}{\vdots}}{\Pi_1 \vdash \mathtt{new\ T().app(z)}:\varphi}\,(\text{INVK})$$

$$\mathcal{D}_2 :: \dfrac{\dfrac{}{\Pi_1 \vdash \mathtt{z}:\langle \mathtt{app}:(\omega)\to\varphi\rangle}\,(\text{VAR}) \quad \dfrac{}{\Pi_1 \vdash \mathtt{new\ T().app(z)}:\omega}\,(\omega)}{\Pi_1 \vdash \mathtt{z.app(new\ T().app(z))}:\varphi}\,(\text{INVK})$$

$$\mathcal{D}_3 :: \dfrac{\dfrac{}{\Pi_1 \vdash \mathtt{z}:\langle \mathtt{app}:(\omega)\to\varphi\rangle}\,(\text{VAR}) \quad \dfrac{}{\Pi_1 \vdash \bot:\omega}\,(\omega)}{\Pi_1 \vdash \mathtt{z.app}(\bot):\varphi}\,(\text{INVK})$$

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

Figure 8: Predicate Derivations for the Fixed-Point Construction Example

$$\dfrac{\dfrac{\dfrac{}{\mathtt{this}{:}\psi \vdash \mathtt{this}:\langle \mathtt{m}:()\to\varphi\rangle}\,(\text{VAR})}{\mathtt{this}{:}\psi \vdash \mathtt{this.m()}:\varphi}\,(\text{INVK}) \quad \boxed{\mathcal{D}'}\ \emptyset \vdash \mathtt{new\ C()}:\psi}{\mathcal{D} :: \emptyset \vdash \mathtt{new\ C()}:\langle \mathtt{m}:()\to\varphi\rangle}\,(\text{NEWM})}{\emptyset \vdash \mathtt{new\ C().m()}:\varphi}\,(\text{INVK})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{this}{:}\langle \mathtt{m}:()\to\varphi\rangle \vdash \mathtt{this}:\langle \mathtt{m}:()\to\varphi\rangle}\,(\text{VAR})}{\mathtt{this}{:}\langle \mathtt{m}:()\to\varphi\rangle \vdash \mathtt{this.m()}:\varphi}\,(\text{INVK}) \quad \begin{array}{c}\text{DOES NOT EXIST}\\ \vdots\end{array}}{\emptyset \vdash \mathtt{new\ C()}:\langle \mathtt{m}:()\to\varphi\rangle}\,(\text{NEWM}) \quad \vdots}{\emptyset \vdash \mathtt{new\ C()}:\langle \mathtt{m}:()\to\varphi\rangle}\,(\text{INVK})}{\emptyset \vdash \mathtt{new\ C().m()}:\varphi}$$

Figure 9: Predicate Derivations for a Non-Terminating Program

An Unsolvable Program

Let us now examine how the predicate system deals with programs that do not have a head-normal form. The approximation theorem states that any predicate 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 unsolvable expressions (i.e. those without a head-normal form) is that they do *not* have non-trivial approximants: their only approximant is $\bot$. From the approximation result it therefore follows that we cannot build any derivation for these expressions that assigns a predicate other than ω (since that is the only predicate assignable to $\bot$).

To illustrate this, consider the following program which constitutes perhaps the simplest example of unsolvability in OO:

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

This program has a method `m` which simply calls itself recursively.

Figure 9 shows two candidate derivations assigning a non-trivial predicate 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. When we are trying to assign a non-trivial predicate to the invocation of the method `m` on `new C()`, we can proceed without loss of generality by building a derivation assigning a predicate variable φ, since we may then simply substitute any suitable (strict) predicate for φ in the derivation.

The derivation we need to build assigns the predicate φ to a method invocation so we must first build a derivation $\mathcal{D}$ that assigns the method predicate $\langle \mathtt{m}:() \to \varphi \rangle$ to the receiver `new C()`. *This* derivation is constructed by examining the method body – `this.m()` – and finding a derivation assigning to it φ. This analysis reveals that the variable `this` must be assigned a predicate for the method `m` which will be of the form $\langle \mathtt{m}:() \to \varphi \rangle$; `new C()` (the receiver) must also satisfy the predicate ψ used for `this`. Finally, in order for the (VAR) leaf of the derivation to be valid the predicate ψ must satisfy the constraint that $\psi \trianglelefteq \langle \mathtt{m}:() \to \varphi \rangle$.

The second derivation of Figure 9 is an attempt at instantiating the schema that we have just constructed. In order to make the instantiation, we must pick a concrete predicate for ψ satisfying the aforementioned constraint. Perhaps the simplest thing to do would be to pick $\psi = \langle \mathtt{m}:() \to \varphi \rangle$. Next, we must instantiate the derivation $\mathcal{D}'$ assigning this predicate to the receiver `new C()`. Here we run into trouble because, in order to achieve this, we must again type the body of method `m`, i.e. solve the same problem that we started with – we see that our instantiation of the derivation $\mathcal{D}'$ must be of exactly the same shape as our instantiation of the derivation $\mathcal{D}$; of course, this is impossible since $\mathcal{D}'$ is a proper subderivation of $\mathcal{D}$ and so no such derivation exists. Notice however, that the receiver `new C()` itself is *not* unsolvable – indeed, it is a normal form – and so we *can* assign to it a non-trivial predicate: using the (OBJ) rule, $\varnothing \vdash \mathtt{new\ C()}:\mathtt{C}$.

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 FJ$^{\mathfrak{C}}$, 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. Let δ be the following 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. it is unsolvable, 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 the only way we can type this term is using ω[7].

In our type system however we may assign several forms of predicate to $[\![t]\!]$. Most simply we can derive $\varnothing \vdash [\![t]\!] : \mathtt{S}_3$, but even though a 'functional' analysis via the `app` method is impossible, it is still safe to access the fields of the value resulting from $[\![t]\!]$ – both $\varnothing \vdash [\![t]\!] : \langle \mathtt{x}:\mathtt{K}_2 \rangle$ and $\varnothing \vdash [\![t]\!] : \langle \mathtt{y}:\mathtt{K}_2 \rangle$ are also easily derivable statements. In fact, we can derive even more informative types: the expression $[\![\mathbf{K}\ \delta]\!]$ can be assigned predicates 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 equivalent λ-term to *t* is $\lambda y.(\lambda x.xx)(\lambda x.xx)$, which is a *weak* head-normal form without a head-normal form. The 'functional' view is that such terms are observationally indistinguishable from unsolvable terms. When encoded in FJ$^{\mathfrak{C}}$ however, our type system shows that these terms become meaningful (head-normalisable).

The second observation concerns *principal* types. In the LC, each normal form has a *unique* most-specific type: i.e. a 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 FJ$^{\mathfrak{C}}$ does enjoy such a property. Consider the following program:

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

[7] In other intersection type systems (e.g. [12]) $\phi \to \omega$ 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 unsolvable terms.

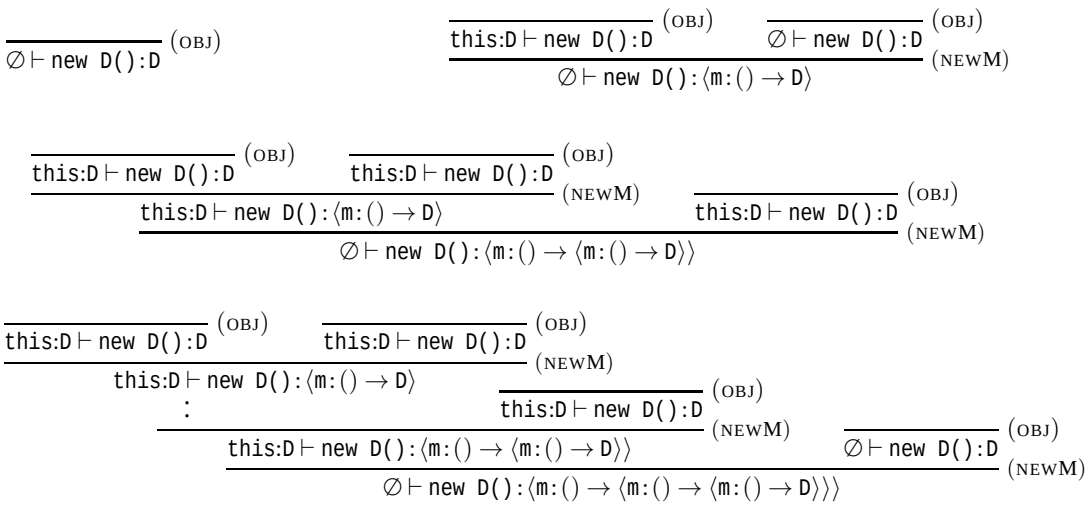

Figure 10: Predicate Derivations for a Program without a Principal Type

The expression `new D()` is a normal form, and so we can assign it a non-trivial predicate, but observe that the set of all predicates 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 10. None of these types may be considered the *most* specific one, since whichever predicate 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 predicates. On the other hand, this seems to preclude the possibility of practical type inference for our system. Notice however that these predicates are not unrelated to one another: they each approximate the 'infinite' predicate $\langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \ldots\rangle\rangle$, 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 not then 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 predicate-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 (i.e. LC and TRS) to the OO paradigm. Through characterisation results we have shown that our predicate system is powerful enough (at least in principle) to form the basis for expressive analyses of OO programs.

Our work has also highlighted where the OO programming style differs from its functional cousin. In particular we have noted that because of the OO 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 [32] shows that a restricted but still highly expressive form of recursive types can still characterise strongly normalising terms, and we hope to fuse this approach with our own to come to an equally precise but more concise and practical predicate-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 $\lambda\mu$ [33], it seems natural to extend our simply typed system to analyse the exception handling features of Java.

References

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

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

[3] 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.

[4] 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.

[5] S. van Bakel. Completeness and Soundness results for $\mathcal{X}$ with Intersection and Union Types. *Fundamenta Informaticae*, 2011. To appear.

[6] S. van Bakel and U. de'Liguoro. Logical equivalence for subtyping object and recursive types. *Theory of Computing Systems*, 42(3):306–348, 2008.

[7] S. van Bakel and M. Fernández. Normalisation Results for Typeable Rewrite Systems. *Information and Computation*, 2(133):73–116, 1997.

[8] S. van Bakel and M. Fernández. Normalisation, Approximation, and Semantics for Combinator Systems. *Theoretical Computer Science*, 290:975–1019, 2003.

[9] 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)*, European Conference on Object-Oriented Programming, 2009. Article No. 3.

[10] A. Banerjee and T.P. Jensen. Modular Control-Flow Analysis with Rank 2 Intersection Types. *Mathematical Structures in Computer Science*, 13(1), 2003.

[11] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[12] 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.

[13] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of data types*, pages 51–67. 1984.

[14] L. Cardelli and J.C. Mitchell. Operations on Records. *Mathematical Structures in Computer Science*, 1(1), 1991.

[15] 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.

[16] 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.

[17] H.B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.

[18] 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.

[19] D.J. Dougherty, P. Lescanne, and L. Liquori. Addressed term rewriting systems: application to a typed object calculus. *Mathematical Structures in Computer Science*, 16(4):667–709, 2006.

[20] ECMA International. The C# Language Specification. ECMA-334, 4th Edition, June 2006. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[21] ECMA International. ECMA Language Specification. ECMA-262, 3rd Edition, December 2006. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

[22] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda Calculus of Objects and Method Specialization. *Nord. J. Comput.*, 1(1), 1994.

[23] 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*. Springer Verlag, 1995.

[24] K. Fisher and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

[25] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification (3rd Edition). ISBN 978-0321246783

[26] 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.

[27] 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), 2001.

[28] 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*. Springer, 2002.

[29] F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi. In JM. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 963–982. Springer, 1999.

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

[31] 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.

[32] H. Nakano. A Modality for Recursion. In *LICS*, 2000.

[33] 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.

[34] G.D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

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

[36] R.N.S. Rowe and S. van Bakel. Approximation Semantics and Expressive Predicate Assignment for Object-Oriented Programming. In *Proceedings of TLCA'11* International Conference on Typed Lambda Calculi and Applications, 2011. To appear.

[37] Ruby online documentation, `http://www.ruby-lang.org/en/`

[38] D. Scott. Domains for Denotational Semantics. In *Proceedings of ICALP'82. 10th International Colloquium on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer Verlag, 1981.

[39] B. Stroustrop. The C++ Programming Language (Third ed.) ISBN 0201889544

[40] W.W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–223, 1967.
[41] C.P. Wadsworth. The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus. *SIAM J. Comput*, 5:488–521, 1976.