Semantic Predicate Types and Approximation for Class-based Object-Oriented Programming

Steffen van Bakel, Reuben N. S. Rowe

Imperial College London

Research Aims

- A system for *static analysis* of (class-based) object-oriented programs (e.g. Java, C++, C#):
  - more expressive than current type systems for these languages;
  - capture *runtime* properties of programs;
  - based on intersection types.
- *Abstract interpretation* through type-based semantics.

Intersection Types

- A powerful type system for the λ-calculus (and its extensions), as well as other formalisms (e.g. term rewriting systems):
  - allow terms (i.e. function parameters) to have *more* than one type at a time;
  - characterisation of terms with (head) normal forms by assignable types;
  - semantics through interpretation of terms via assignable types;
  - approximation result: each type assignable to a term corresponds to an approximant (a snapshot of computation).

pFJ: Predicate Featherweight Java

We have made some slight modifications to Featherweight Java:

classes	cd	$::=$	$\texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{C}'\ \{\ \overrightarrow{\mathsf{fd}}\ \overrightarrow{\mathsf{md}}\ \}\quad (\mathrm{C} \neq \texttt{Object})$
methods	md	$::=$	$\mathrm{D}\ m(\overrightarrow{\mathrm{C}\ x})\ \{\ \mathsf{e}\ \}$
field declarations	fd	$::=$	$\mathrm{C}\ f$
expressions	e	$::=$	$x \mid \texttt{null} \mid \mathsf{e}.f \mid \mathsf{e}.f = \mathsf{e}' \mid \mathsf{e}.m(\vec{\mathsf{e}}) \mid \texttt{new}\ \mathrm{C}(\vec{\mathsf{e}})$
execution contexts	$\mathcal{E}$	$::=$	$\overrightarrow{\mathsf{cd}}$
programs	P	$::=$	$(\mathcal{E}, \mathsf{e})$

- Removed cast expressions to recover soundness.
- Introduced precursory imperative features:
  - a `null` keyword/value;
  - field assignment (update).

pFJ: Predicate Featherweight Java

Reduction is the contextual closure (e.g. $e \to e' \Rightarrow e.f \to e'.f$) of the following rules:

$$
\begin{array}{rcl}
(\texttt{new}\ C(\vec{e}_n)).f_i & \to & e_i \\
(\texttt{new}\ C(\vec{e}_n)).f_i = e'_i & \to & \texttt{new}\ C(e_1, \ldots, e'_i, \ldots, e_n) \\
(\texttt{new}\ C(\vec{e})).m(\overrightarrow{e'}) & \to & e_b[\overrightarrow{e'/x}, \texttt{new}\ C(\vec{e})/\texttt{this}]
\end{array}
$$

Reduction is:

- more free than call-by-value (e.g. may happen inside objects);
- weak (as in Term Rewriting Systems) - *all* arguments to a method must be supplied.

Type system for pFJ is (almost) identical to that of FJ: $\Gamma \vdash e{:}C$

The Predicate System

We introduce an extra layer of types – *predicates*:

predicates :	ϕ	$::=$	$\top$	$\mid$	ν
normal predicates :	ν	$::=$	$\mathfrak{N}$	$\mid$	σ
object predicates :	σ	$::=$	$\langle \overrightarrow{\ell{:}\tau} \rangle$		
member predicates :	τ	$::=$	ν	$\mid$	$\psi :: \vec{\phi} \to \nu$

Predicate assignment expressed by the judgement: $\Pi \vdash \mathrm{e{:}C} : \phi$

- Predicates provide an analysis of the *functional* behaviour of expressions:
  - play the same role that (intersection) types do in the λ-calculus.
- They are *more* than just record types – they are *implicit intersections*.
- Class types do not allow for multiple analyses (of methods).
- Class types are *recursive*, making type-based termination analysis impossible.

The Predicate System

So, intuitively, what do predicates express?

- $\Pi \vdash \texttt{e}{:}\mathrm{C} : \langle \texttt{f} : \nu, \texttt{m} : \psi :: \vec{\phi} \rightarrow \nu' \rangle$ implies `e` results in an object with:
  - a field `f` behaving as ν,
  - a method `m` which returns a value behaving as ν' (when invoked with appropriately behaved arguments).
- $\Pi \vdash \texttt{e}{:}\mathrm{C} : \mathfrak{N}$ implies that `e` results in the `null` value.
- $\Pi \vdash \texttt{e}{:}\mathrm{C} : \top$ implies that `e` either:
  - results in an error, or
  - disappears during reduction (i.e. does not contribute to the final result).

Properties of the System

Our predicate system has the standard properties of intersection type systems:

Soundness (subject reduction):

$$\Pi \vdash \mathsf{e}{:}\mathrm{C} : \phi \;\&\; \mathsf{e} \rightarrow \mathsf{e}' \Rightarrow \Pi \vdash \mathsf{e}'{:}\mathrm{C} : \phi$$

Completeness (subject expansion):

$$\Pi \vdash \mathsf{e}{:}\mathrm{C} \;\&\; \mathsf{e} \rightarrow \mathsf{e}' \;\&\; \Pi \vdash \mathsf{e}'{:}\mathrm{C} : \phi \Rightarrow \Pi \vdash \mathsf{e}{:}\mathrm{C} : \phi$$

Full intersection type assignment systems are *undecidable*!

- Need to define a *decidable* restriction for practical use.

An Approximation Result for *p*FJ

linking types with semantics

What are Approximants?

Approximants are *snapshots* of a computation

Basic idea: *cover* places in an expression where computation may take place with Ω:

$$\begin{array}{lcl} \mathtt{e} & \equiv & \mathtt{new\ C(\ \ (new\ D).m(new\ Object())\ \ ,new\ E(\ \ x.f\ \ ,new\ D()))} \\ \mathtt{A} & \equiv & \mathtt{new\ C(}\qquad\qquad\qquad \Omega \qquad\qquad\qquad \mathtt{,new\ E(}\ \ \Omega\ \ \mathtt{,new\ D()))} \end{array}$$

- A (a normal form) *directly approximates* e: $\mathtt{A} \sqsubseteq \mathtt{e}$.
- A is an *approximant* of e when it directly approximates some e′ to which e runs: $\mathtt{A} \precsim \mathtt{e} \Leftrightarrow \exists\, \mathtt{e}' \,[\, \mathtt{e} \rightarrow^* \mathtt{e}' \;\&\; \mathtt{A} \sqsubseteq \mathtt{e}' \,]$.
- The set of all approximants of e is denoted by $\mathcal{A}(\mathtt{e}) = \{\, \mathtt{A} \mid \mathtt{A} \precsim \mathtt{e} \,\}$.
- Approximants can be used to define a semantics: $[\![\mathtt{e}]\!] = \mathcal{A}(\mathtt{e})$.

The Approximation Result

The approximation theorem is:

If we can assign a predicate ϕ to an expression e, then e has an approximant A with the same predicate ϕ

$$\Pi \vdash \mathsf{e}{:}\mathrm{C} : \phi \Rightarrow \exists\, \mathsf{A} \in \mathcal{A}(\mathsf{e})\ [\ \Pi \vdash \mathsf{A}{:}\mathrm{C} : \phi\ ]$$

We get characterisation from the following:

- if $\Pi \vdash \mathsf{A}{:}\mathrm{C} : \phi$ with $\phi \neq \top$ then A is in head-normal form (i.e. not Ω).
- The relation $\sqsubseteq$ preserves the structure of expressions

Proof: Key Aspects

1. We used the *computability* technique of Tait.
   - Used by others to show the result for the λ-calculus.
   - Computability Predicate defined inductively over structure of predicates:

$$(\mathit{Comp}(\Pi, \mathrm{e}{:}\mathrm{C}, \langle m : \psi :: \vec{\phi}_n \rightarrow \nu \rangle) \quad \Leftrightarrow \quad (\mathit{Comp}(\Pi, \mathrm{e}{:}\mathrm{C}, \psi) \;\&\; \forall\, i \; [\; \mathit{Comp}(\Pi, \mathrm{e}_i{:}\mathrm{C}_i, \phi_i) \;] \Rightarrow \mathit{Comp}(\Pi, \mathrm{e}.m(\vec{\mathrm{e}}_n){:}\mathrm{D}, \nu)))$$

2. To show the computability of certain expressions, we need predicates to make a statement about what is *visible* in a class! E.g. we need that $\Pi \vdash \mathrm{e}{:}\mathrm{C} : \langle f{:}\nu \rangle$ implies f is visible in C.
   - Introduce notion of the *language* of a class, $\mathcal{L}(\mathrm{C})$, to restrict the predicates that can be assigned.
   - Causes subject expansion to collapse ... problem?

Approximation: Example

```
class C extends Object {
    C m1() { this.m2() }
    C m2() { this }
}
```

$\texttt{(new C()).m1()} \rightarrow \texttt{(new C()).m2()} \rightarrow \texttt{new C()}$

Thus, we have that

$$\texttt{new C()} \in \mathcal{A}\big(\texttt{(new C()).m1()}\big)$$

And we can assign the following predicates:

$$\emptyset \vdash \texttt{(new C()).m1()}:\texttt{C}:\langle\,\rangle$$
$$\emptyset \vdash \texttt{new C()}:\texttt{C}:\langle\,\rangle$$

Approximation: Example

```
class C extends Object {
    C m1() { this.m2() }
    C m2() { this }
}
```

$$\texttt{(new C()).m1()} \rightarrow \texttt{(new C()).m2()} \rightarrow \texttt{new C()}$$

Thus, we have that

$$\texttt{new C()} \in \mathcal{A}(\texttt{(new C()).m1()})$$

And we can assign the following predicates:

$$\emptyset \vdash \texttt{(new C()).m1()} : \texttt{C} : \langle\,\rangle \quad \text{with } \texttt{new C()} : \langle \texttt{m1} : \langle \texttt{m2} : \langle\,\rangle :: \epsilon \rightarrow \langle\,\rangle \rangle :: \epsilon \rightarrow \langle\,\rangle, \texttt{m2} : \langle\,\rangle :: \epsilon \rightarrow \langle\,\rangle \rangle$$

$$\emptyset \vdash \texttt{new C()} : \texttt{C} : \langle\,\rangle$$

Approximation: Example

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

$$\texttt{(new C()).m()} \rightarrow \texttt{(new C()).m()} \rightarrow \ldots$$

What are the approximants of `(new C()).m()`?

$$\mathcal{A}(\texttt{(new C()).m()}) = \{\,\Omega\,\}$$

So, what predicates can we assign?

$$\emptyset \;\vdash\; \Omega\texttt{:C}:\top$$

$$\emptyset \;\vdash\; \texttt{(new C()).m()}\texttt{:C}:\top$$

Future Work

- Extend definition of predicate languages to regain completeness with approximation,
- Predicate inference algorithm: an *interesting* decidable restriction (cf. Rank-2 system for the λ-calculus and TRS),
- Incorporating *state* into the calculus (heaps & pointers),
- Other analyses (e.g. dead code, strictness, type and effect systems). Other class-based OO features?

State of the Art

- For the most part, previous work *not* type based (control-flow/data-flow analysis).
- Centred around optimisation issues:
  - class analysis (to eliminate virtual function calls);
  - class invariants (remove array bounds checks).
- Pointer analysis (catch null pointer dereferences).
- Termination analysis of Java Bytecode (but not of Java programs themselves).