A foundation for trait-based metaprogramming

Aaron Turon

Department of Computer Science
University of Chicago

Joint work with John Reppy
A problem for single inheritance:
What are traits?

A *trait* is a partial class implementation: a flat collection of *provided* methods.

- Methods invoked by a trait but not provided are *required* methods.
- Traits cannot introduce state – they can only provide methods.

Introduced in [Schärli et al.; ECOOP’03].
What are traits?

A *trait* is a partial class implementation: a flat collection of *provided* methods.

- Methods invoked by a trait but not provided are *required* methods.
- Traits cannot introduce state – they can only provide methods.

Introduced in [Schärli et al.; ECOOP’03].

---

**Key idea**

Trait *composition* occurs outside the inheritance hierarchy. Composition is symmetric.
A trait-based solution:
Besides composition, traits support *aliasing* and *excluding* methods to resolve conflicts.
Trait operations

Besides composition, traits support *aliasing* and *excluding* methods to resolve conflicts.

**Exclude and compose**

\[ \text{TCPoint} = \text{TPoint} + (\text{TColored exclude toString}) \]
Trait operations

Besides composition, traits support *aliasing* and *excluding* methods to resolve conflicts.

Exclude and compose

\[ \text{TCPoint} = \text{TPoint} + (\text{TColored exclude toString}) \]

Alias, exclude and compose

\[ \text{TCPoint} = \{ \text{provides} \ \text{toString()} : \text{string} \{ \text{self.strP()} + ":" + \text{self.strC}(); \} \} + ((\text{TPoint alias toString as strP}) \text{exclude toString}) + ((\text{TColored alias toString as strC}) \text{exclude toString}) \]
Deep operations

The **alias** and **exclude** operations are *shallow*: they do not affect the bodies of other methods in the trait.

Deep aliasing $\Rightarrow$ renaming    Deep exclusion $\Rightarrow$ hiding
Deep operations

The **alias** and **exclude** operations are *shallow*: they do not affect the bodies of other methods in the trait.

Deep aliasing $\implies$ renaming  
Deep exclusion $\implies$ hiding

Hide and compose

TCPoint = TPoint + (TColored **hide** toString)
Deep operations

The **alias** and **exclude** operations are *shallow*: they do not affect the bodies of other methods in the trait.

Deep aliasing $\rightarrow$ renaming  
Deep exclusion $\rightarrow$ hiding

Hide and compose

TCPoint = TPoint + (TColored **hide** toString)

Rename and compose

TCPoint = {
    provides toString() : string {
        self.strP() + ": " + self.strC();
    }
} + (TPoint **rename** toString **to** strP)  
+ (TColored **rename** toString **to** strC)
Our research: develop a statically-typed trait calculus giving a semantics for method hiding and renaming.

- Built on the Fisher-Reppy polymorphic trait calculus [Fisher & Reppy 2003].
- Uses Riecke-Stone dictionaries [Riecke & Stone 2002] to provide a realistic model of the deep operations.
- Provides more accurate trait types (requirements are tracked \textit{per-method}, rather than per-trait).
Syntactic forms for runtime trait values

\[
\phi ::= \{ r \mapsto i \mid r \in R \} \quad \text{dictionary}
\]

\[
Mv ::= \{ i \mapsto \mu v_i \mid i \in I \} \quad \text{method suite value}
\]

\[
\mu v ::= [E; \phi; \lambda x.e] \quad \text{method value}
\]

\[
tv ::= \langle Mv; \phi_P; \phi_R \rangle \quad \text{trait value}
\]

where \( r \) ranges over trait method requirements (both self and super) and \( i \) ranges over slots.

Note: these are simplified versions of the forms in the paper.
TBar provides bar.

\[ \phi_P = \{ \text{bar} \mapsto 1 \} \]
\[ \phi_R = \{ \} \]
\[ \phi_1 = \{ \text{bar} \mapsto 1 \} \]
TFoo provides foo, requires bar.

\[ \phi_P = \{ \text{foo} \mapsto 1 \} \]
\[ \phi_R = \{ \text{bar} \mapsto 2 \} \]
\[ \phi_1 = \{ \text{foo} \mapsto 1, \text{bar} \mapsto 2 \} \]
$E ⊢ T_1 \rightarrow \langle \begin{array}{l} I_1 Mv_1; \ M_1 \phi P_1; \ R_1 \phi R_1 \end{array} \rangle$

$E ⊢ T_2 \rightarrow \langle \begin{array}{l} I_2 Mv_2; \ M_2 \phi P_2; \ R_2 \phi R_2 \end{array} \rangle$

$M_1 \cap M_2 \quad Mv_2 = \{ i \mapsto [E_i; \ \phi_i; \ e_i] \ i \in I_2 \}$

$\phi_P = \{ \phi_{P_2}(m) \mapsto \phi_{R_1}(m) \ m \in M_2 \cap R_1 \}$

$\phi_R = \{ \phi_{R_2}(m) \mapsto \phi_{P_1}(m) \ m \in R_2 \cap M_1 \}$

$I'_1 = I_1 \cup \text{rng}(\phi_{R_1}) \quad I'_2 = I_2 \cup \text{rng}(\phi_{R_2})$

$\varphi_F = \text{FS}(I'_2 \setminus \text{dom}(\varphi_R \cup \varphi_P), \ I'_1)$

$\varphi = \varphi_P \cup \varphi_R \cup \varphi_F \quad \phi_P = \phi_{P_1} \cup (\varphi \circ \phi_{P_2})$

$\phi_R = (\phi_{R_1} \cup (\varphi \circ \phi_{R_2})) \setminus (M_1 \cup M_2)$

$Mv = Mv_1 \cup \{ \varphi(i) \mapsto [E_i; \ \varphi \circ \phi_i; \ e_i] \ i \in I_2 \}$

$E ⊢ T_1 + T_2 \rightarrow \langle Mv; \ \phi_P; \ \phi_R \rangle$
\[ E \vdash T_1 \rightarrow \langle I_1 \mathcal{M}v_1; \mathcal{M}_1 \phi P_1; \mathcal{R}_1 \phi R_1 \rangle \]
\[ E \vdash T_2 \rightarrow \langle I_2 \mathcal{M}v_2; \mathcal{M}_2 \phi P_2; \mathcal{R}_2 \phi R_2 \rangle \]

\[ \mathcal{M}_1 \cap \mathcal{M}_2 \quad M\mathcal{v}_2 = \{ i \mapsto [E_i; \phi_i; e_i] \mid i \in I_2 \} \]

\[ \varphi_P = \{ \phi P_2(m) \mapsto \phi R_1(m) \mid m \in \mathcal{M}_2 \cap \mathcal{R}_1 \} \]

\[ \varphi_R = \{ \phi R_2(m) \mapsto \phi P_1(m) \mid m \in \mathcal{R}_2 \cap \mathcal{M}_1 \} \]

\[ I' _1 = I_1 \cup \text{rng}(\phi R_1) \quad I' _2 = I_2 \cup \text{rng}(\phi R_2) \]

\[ \varphi_F = \text{FS}(I'_2 \setminus \text{dom}(\varphi_R \cup \varphi_P), I'_1) \]

\[ \varphi = \varphi_P \cup \varphi_R \cup \varphi_F \quad \phi P = \phi P_1 \cup (\varphi \circ \phi P_2) \]

\[ \phi R = (\phi R_1 \cup (\varphi \circ \phi R_2)) \setminus (\mathcal{M}_1 \cup \mathcal{M}_2) \]

\[ Mv = Mv_1 \cup \{ \varphi(i) \mapsto [E_i; \varphi \circ \phi_i; e_i] \mid i \in I_2 \} \]

\[ E \vdash T_1 + T_2 \rightarrow \langle Mv; \phi P; \phi R \rangle \]
\[ E \vdash T_1 \rightarrow \langle I_1 Mv_1; M_1 \phi_P_1; R_1 \phi_R_1 \rangle \]
\[ E \vdash T_2 \rightarrow \langle I_2 Mv_2; M_2 \phi_P_2; R_2 \phi_R_2 \rangle \]
\[ M_1 \cap M_2 \quad Mv_2 = \{ i \mapsto [E_i; \phi_i; e_i] \mid i \in I_2 \} \]
\[ \varphi_P = \{ \phi_P(m) \mapsto \phi_R(m) \mid m \in M_2 \cap R_1 \} \]
\[ \varphi_R = \{ \phi_R(m) \mapsto \phi_P(m) \mid m \in R_2 \cap M_1 \} \]
\[ I_1' = I_1 \cup \text{rng}(\varphi_R) \quad I_2' = I_2 \cup \text{rng}(\varphi_R) \]
\[ \varphi_F = \text{FS}(I_2' \setminus \text{dom}(\varphi_R \cup \varphi_P), I_1') \]
\[ \varphi = \varphi_P \cup \varphi_R \cup \varphi_F \quad \phi_P = \phi_P_1 \cup (\varphi \circ \phi_P_2) \]
\[ \phi_R = (\phi_R_1 \cup (\varphi \circ \phi_R_2)) \setminus (M_1 \cup M_2) \]
\[ Mv = Mv_1 \cup \{ \varphi(i) \mapsto [E_i; \varphi \circ \phi_i; e_i] \mid i \in I_2 \} \]
\[ E \vdash T_1 + T_2 \rightarrow \langle Mv; \phi_P; \phi_R \rangle \]
\begin{align*}
E \vdash T_1 & \quad \iff \quad \langle \ I_1 Mv_1; \ M_1 \phi_P; \ R_1 \phi_R \ |angle \\
E \vdash T_2 & \quad \iff \quad \langle \ I_2 Mv_2; \ M_2 \phi_P; \ R_2 \phi_R \ |angle \\
M_1 \cap M_2 & = \ Mv_2 = \ \{ \ i \mapsto \ [E_i; \ \phi_i; \ e_i] \ i \in I_2 \} \\
\varphi_P & = \ \{ \ \phi_{P_2}(m) \mapsto \ \phi_{R_1}(m) \ m \in M_2 \cap R_1 \} \\
\varphi_R & = \ \{ \ \phi_{R_2}(m) \mapsto \ \phi_{P_1}(m) \ m \in R_2 \cap M_1 \} \\
I_1' &= I_1 \cup \text{rng}(\phi_R) \\
I_2' &= I_2 \cup \text{rng}(\phi_R) \\
\varphi_F &= \text{FS}(I_2' \setminus \text{dom}(\varphi_R \cup \varphi_P), \ I_1') \\
\varphi &= \ \varphi_P \cup \varphi_R \cup \varphi_F \\
\phi_P &= \ \phi_{P_1} \cup (\varphi \circ \phi_{P_2}) \\
\phi_R &= \ (\phi_{R_1} \cup (\varphi \circ \phi_{R_2})) \ \setminus (M_1 \cup M_2) \\
Mv &= Mv_1 \cup \{ \ \varphi(i) \mapsto \ [E_i; \ \varphi \circ \phi_i; \ e_i] \ i \in I_2 \} \\
E \vdash T_1 + T_2 & \quad \iff \quad \langle \ Mv; \ \phi_P; \ \phi_R \ |angle 
\end{align*}
Aaron Turon

A foundation for trait-based metaprogramming

\[
    E \vdash T_1 \rightarrow \langle | I_1 Mv_1; M_1 \phi_P_1; R_1 \phi_R_1 | \rangle
\]

\[
    E \vdash T_2 \rightarrow \langle | I_2 Mv_2; M_2 \phi_P_2; R_2 \phi_R_2 | \rangle
\]

\[
    M_1 \cap M_2 \quad Mv_2 = \{ i \mapsto [E_i; \phi_i; e_i] \}_{i \in I_2}
\]

\[
    \phi_P = \{ \phi_P_2(m) \mapsto \phi_R_1(m) \mid m \in M_2 \cap R_1 \}
\]

\[
    \phi_R = \{ \phi_R_2(m) \mapsto \phi_P_1(m) \mid m \in R_2 \cap M_1 \}
\]

\[
    I'_1 = I_1 \cup \text{rng}(\phi_R_1) \quad I'_2 = I_2 \cup \text{rng}(\phi_R_2)
\]

\[
    \varphi_F = \text{FS}(I'_2 \setminus \text{dom}(\varphi_R \cup \varphi_P), I'_1)
\]

\[
    \varphi = \varphi_P \cup \varphi_R \cup \varphi_F \quad \phi_P = \phi_P_1 \cup (\varphi \circ \phi_P_2)
\]

\[
    \phi_R = (\phi_R_1 \cup (\varphi \circ \phi_R_2)) \setminus (M_1 \cup M_2)
\]

\[
    Mv = Mv_1 \cup \{ \varphi(i) \mapsto [E_i; \phi \circ \phi_i; e_i] \}_{i \in I_2}
\]

\[
    E \vdash T_1 + T_2 \rightarrow \langle | Mv; \phi_P; \phi_R | \rangle
\]
Traits
Deep operations
Trait-based metaprogramming
Hiding and renaming
Semantics

A foundation for trait-based metaprogramming

\[ \Phi \]

TFooBar = TFoo + TBar

\[ \varphi = \{1 \mapsto 2\} \]

\[ \phi_P = \{\text{foo} \mapsto 1, \text{bar} \mapsto 2\} \]

\[ \phi_R = \{\} \]

\[ \phi_1 = \{\text{foo} \mapsto 1, \text{bar} \mapsto 2\} \]

\[ \phi_2 = \{\text{bar} \mapsto 2\} \]
\[
E \vdash T \longrightarrow \langle \mathcal{I} \mathcal{M} \mathcal{v}; \mathcal{M} \phi_P; \mathcal{R} \phi_R \rangle \\
\text{if } r \in \mathcal{M} \quad r' \notin \mathcal{M} \quad r' \notin \mathcal{R} \\
\phi'_P = (\phi_P \setminus r)[r' \mapsto \phi_P(r)] \\
E \vdash T \text{ rename } r \text{ to } r' \longrightarrow \langle \mathcal{M} \mathcal{v}; \phi'_P; \phi_R \rangle
\]
TFooBaz = TFooBar rename bar to baz

\( \phi_P = \{\text{foo }\mapsto 1, \text{ baz }\mapsto 2\} \)

\( \phi_R = \{\}\)

\( \phi_1 = \{\text{foo }\mapsto 1, \text{ bar }\mapsto 2\} \)

\( \phi_2 = \{\text{bar }\mapsto 2\} \)
\[
E \vdash T \longrightarrow \langle I Mv; M \phi_P; R \phi_R \rangle \quad m \in M
\]

\[
E \vdash T \ hide \ m \longrightarrow \langle Mv; \phi_P \ \backslash \ m; \phi_R \rangle
\]
A foundation for trait-based metaprogramming

\[ \phi_P = \{ \text{foo} \mapsto 1 \} \]
\[ \phi_R = \{} \]
\[ \phi_1 = \{ \text{foo} \mapsto 1, \text{bar} \mapsto 2 \} \]
\[ \phi_2 = \{ \text{bar} \mapsto 2 \} \]

\[ T\text{Foo'} = T\text{FooBaz} \text{ hide baz} \]
Recall the trait-based solution:

```
class CIntRd
  field desc
  method Rd

class CIntWr
  field desc
  method Wr

class CDevice
  field desc

class CIntR
  field desc
  method Rd
  method Wr

class CSIntRd
  field lock
  field desc
  method Rd

class CSIntRW
  field lock
  field desc
  method Rd
  method Wr

class CSIntWr
  field lock
  field desc
  method Wr

trait TIntRd
  required
  field desc
  provided
  method Rd

trait TIntWr
  required
  field desc
  provided
  method Wr

trait TSRd
  required
  field lock
  provided
  override Rd

trait TSWr
  required
  field lock
  provided
  override Wr
```

trait-based metaprogramming
The TSync example

```scala
trait TSync<ty1, ty2> = {
    provides Op(x : ty1) : ty2 {
        self.lock.Acquire();
        super.Op(x) before
        self.lock.Release();
    }

    requires field lock : LockObj
}
```
This time with renaming:
A foundation for trait-based metaprogramming

Aaron Turon
Trait-based metaprogramming

Our calculus provides a *foundation* for “trait-based metaprogramming” by formalizing a new notion of substitution:

\[ \text{TSync rename } \text{Op to Read} \]
Trait-based metaprogramming

Our calculus provides a *foundation* for “trait-based metaprogramming” by formalizing a new notion of substitution:

\[
\left[ \text{TSync Read} \right] = \text{TSync rename Op to Read}
\]
Our calculus provides a *foundation* for “trait-based metaprogramming” by formalizing a new notion of substitution:

\[
\text{TSync} = \lambda \text{Op}. \text{trait} \{ \text{provides} \ \text{Op} \ldots \} \\
\left[ \text{TSync Read} \right] = \text{TSync rename Op to Read}
\]
Method name abstraction is only a first step!

**Research challenge**: design a concrete metalanguage using this notion of substitution.

- Java’s *synchronized* keyword – instantiate TSync?
- Pointcuts and other AOP techniques?
- Type-directed application of abstracted traits?
Summary

*Hiding* and *renaming* are the deep analogs to excluding and aliasing.

- Useful for conflict resolution.
- Useful in isolation (for privacy and fixing “wrong” names).
- Renaming yields a new notion of substitution.

We model these features with a statically typed trait calculus and prove type soundness (see the tech report for proof details).

Our calculus provides a rigorous notion of substitution that can be used to build a trait metalanguage.
Some related work

- *Chai: Traits for Java-like languages*. Smith, C. and S. Drossopoulou. ECOOP’05.
- Aspect-oriented programming.
Thank you.