Chai: Traits for Java-like Languages

Charles Smith and Sophia Drossopoulou

Department of Computing, Imperial College London

Abstract. Traits support the factoring out of common behaviour, and its integration into classes in a manner that coexists smoothly with inheritance-based structuring mechanisms.
We designed the language *Chai*, which incorporates statically typed traits into a simple Java-inspired base language, and we discuss three versions of the language: $Chai_1$, where traits are only a mechanism for the creation of classes; $Chai_2$ where traits are a mechanism for the creation of classes, *and* can also introduce types, and $Chai_3$ where traits play a role at runtime, and can can be applied to objects, and change the objects' behaviour. We give formal models for these languages, outline the proof of soundness, and our prototype implementation.

1 Introduction

Traits were designed to facilitate code reuse and to assist in structuring large programs. They are conceptually similar to classes, except that they contain no state, only behaviour, and can be combined using a set of simple composition and modification operators. Elements of behaviour that need to be reused in several different parts of a program can be encapsulated in a trait which may be referenced where necessary, avoiding the need to duplicate code.

Traits first appeared in the object-based language Self[22] where they took the form of parent objects to which an object can delegate some of its behaviour. Subsequent work on Traits was based on the class-based language Smalltalk, for which an extension supporting Traits was created[18, 19]. Use of traits can significantly reduce the overall size of libraries[3].

Mixins[5, 4, 11], Multiple Inheritance[21, 14], Family Polymorphism [9], Delegation Layers, and Aspect Oriented Programming share with Traits the aim of code reuse. Traits, like Mixins, and unlike classes in Multiple Inheritance, have no superclasses, and thus are not tied to a particular location in an inheritance hierarchy. Traits and Mixins usually represent composition of incomplete implementations, and thus support decomposition at a finer grain than classes. When a trait is used by a class have the semantics of the class is the same as if the trait methods were part of the class itself — this is called the *flattening* property.

Smalltalk is the first *class based* language on which traits have been applied. While Smalltalk is dynamically typed, our remit was to apply traits to a statically typed, class based language. In this paper we discuss the design and implementation of *Chai*, an extension of a small Java-like language with traits.

We identified three different rôles for traits in *Chai*, and so, we designed three languages:

Chai$_1$ A language like Java, where traits are purely a mechanism to create classes, and where the application of a trait can only be type-checked after the resulting class has been "flattened".

Chai$_2$ Here we extend the remit of traits so that they may be used as types. This permits checking of traits *before* their application.

Chai$_3$ Finally, we allow traits to be substituted for one another dynamically, supporting runtime changes in object behaviour.

In section 2 of this paper we introduce *Chai* through examples. In sections 3, 4 and 5 we present formal models for *Chai*$_1$, *Chai*$_2$ and *Chai*$_3$. In section 6 we discuss a prototype implementation of these languages, and section 7 contains conclusions and future work.

The prototype implementation and the MSc thesis are available at `http://chai-t.sourceforge.net/`. An appendix with complete definitions and handwritten proofs is available at `http://www.doc.ic.ac.uk/~scd/ChaiApp`.

2 An Example

Figure 1 gives an example of *Chai*$_1$.[1] We define four simple traits: `TScreenShape`, `TPrintedShape`, `TEmptyCircle` and `TFilledCircle`, which describe corresponding components in a simple graphics program: we can form on screen, or print, empty circles or filled circles in any combination in which we are interested, simply by creating a subclass of `Circle` that uses the traits providing the behaviour we want. In the example, we show only two of the four combinations, *i.e.,* classes `ScreenEmptyCircle` and `PrintedFilledCircle`.

A trait T may declare *requirements*, *i.e.,* a list of method signatures, for methods that must be provided by classes or traits using T. Here, trait `TEmptyCircle` requires a method `drawPoint` with return type `void`, and two `int` parameters, and method `getradius` with no parameters, and `int` return type. Class `SreenEmptyCircle` uses `TScreenEmptyCircle`; there, the first method is provided by trait `TScreenShape` and the second by class `Circle`.

Forming the four combinations using single inheritance would mean considerable code duplication, although it could be implemented using multiple inheritance or mixins. See [20] for examples of situations where Traits give more elegant solutions than mixins, and also for examples where mixins give more elegant solutions than traits. Because the trait composition operators are so flexible, *Chai*$_1$ allows much finer code reuse than Java.

Figure 2 gives an example of the additional features of *Chai*$_2$, where we allow traits to be used as types. Any object of a class using the trait `TScreenShape` can be referenced through a variable of type `TScreenShape`. Allowing traits to define types supports more polymorphism, and it allows us to type check traits independently of the classes that use them.

[1] For the sake of simplicity, *Chai* only allows methods with a single parameter called `x`, and does not permit sequences of expressions. These restrictions have minimal implications for the presentation of the features we are interested in, and are not adhered to by the examples in this section, nor by the prototype implementation.

```
class Circle {
   int radius;
   int getRadius() { ... }
}

trait TEmptyCircle {
   requires {
      void drawPoint(int x,int y);
      int getRadius();
   }
   void draw() { ... }
}

trait TFilledCircle {
   requires {
      void drawPoint(int x,int y);
      int getRadius();
   }
   void draw() { ... }
}
```

```
trait TScreenShape {
   void drawPoint(int x,int y) {
      ...
   }
}

trait TPrintedShape {
   void drawPoint(int x,int y) {
      ...
   }
}

class ScreenEmptyCircle
  extends Circle
  uses TEmptyCircle,TScreenShape { }

class PrintedFilledCircle
  extends Circle
  uses TFilledCircle,TPrintedShape
  { }
```

Fig. 1. *Chai*$_1$ Example

In the final example, in figure 3, we show how dynamic substitution of traits can change the behaviour of an object at runtime. The object `circle` starts out as an empty circle on screen, but by substitution of `TFilledCircle` for `TEmptyCircle` we can change it to a filled circle, and subsequently by substituting `TPrintedShape` for `TScreenShape` we can change it to become a printed filled circle.

```
class ScreenShapeStack {
   void push(TScreenShape shape) { ... }
   TScreenShape pop() { ... }
   ...
}

ScreenShapeStack stack = new ScreenShapeStack();
stack.push(new ScreenEmptyCircle());
stack.push(new ScreenFilledCircle());
TScreenShape shape = stack.pop();
```

Fig. 2. *Chai*$_2$ Example

```
class CircleShape extends Circle uses TEmptyCircle,TScreenShape {}

CircleShape circle = new CircleShape(); circle.draw();
       // draws an empty circle on screen
circle<TEmptyCircle -> TFilledCircle>; circle.draw();
       // draws a filled circle on screen
circle<TScreenShape -> TPrintedShape>; circle.draw();
       //  prints a filled circle
```

Fig. 3. *Chai*$_3$ Example

3 The language *Chai*$_1$

3.1 Syntax

For *Chai*$_1$ we adapted Traits from Smalltalk[18] to the Java setting. As in [18], traits can be used to add behaviour to classes or to other traits. Traits may contain method definitions, but no fields — as we said earlier, traits are *pure behaviour* [18]. A trait cannot itself take part in execution (i.e. it cannot be instantiated) — it (or a trait using it) can be *used* by some class, which can then be instantiated.

Traits are not required to be complete - that is, they may require functionality beyond their own to be provided by classes or traits using them. The requirements are declared explicitly in the form of a set of *required methods*.

program	::==	(*trait* $\mid$ *class*)*
trait	::==	**trait** *tr* [**uses** *tr*+] { (*trait-glue* $\mid$ *meth*)* }
field	::==	*type f*;
meth	::==	*meth-sig* { *exp* }
type	::==	*cl*
exp	::==	*exp.f* := *exp* $\mid$ *exp.m*(*exp*) $\mid$ **super**.*m*(*exp*) $\mid$
		new *cl* $\mid$ *var* $\mid$ **this** $\mid$ **null** $\mid$ **x**
trait-glue	::==	**requires** { (*meth-sig* ; $\mid$ *super-sig* ;)* } $\mid$
		exclude { (*t.m* ;)* } $\mid$
		alias { (*t.m* **as** *m* ;)* }
meth-sig	::==	*type m*(*type* x)
super-sig	::==	*type* **super**.*m*(*type* x)
class	::==	**class** *cl* **extends** *cl* [**uses** *tr*+] { (*field* $\mid$ *meth*)* }
cl,tr,m,f	::==	*identifiers*

Fig. 4. *Chai*$_1$ Syntax

A *Chai*$_1$ program consists of trait and class declarations. A trait declaration consists of a name for the trait, an optional list of used traits (whose behaviour it incorporates), and a trait body. The trait body contains method definitions, and "trait glue", *i.e.,* declaration of required methods, exclude declarations (which exclude methods that would otherwise be incorporated from used traits) and alias declarations (which give new labels to methods incorporated from a used trait).

In contrast to the language in [18] which is untyped and where requirements are inferred automatically, in $Chai_1$ provided *and required methods* must be declared using a full type signature.[2] In common with [18], we distinguish required methods into those that must be provided by a class using the trait, and those that must be provided *by the superclass* of the using class. This was necessary because Java allows explicit access to superclass methods (through the `super.m(...)` construct), and so if a trait `tr` is used by a class `cl`, then superclass method calls inside methods of `tr` will resolve to methods from the superclass of `cl`.

A class declaration consists of a name for the class, its superclass, an optional list of used traits (whose behaviour it incorporates), and a class body. The class body contains method definitions and fields.

For simplicity, our model does not support method overloading or field hiding, however we believe it can be easily extended to do so — the implementation of $Chai_1$ supports it.

Note, that in $Chai_1$ classes form types, but traits do not.

3.2 Basic lookup functions

We consider that a program `P` implicitly defines the following eight (partial) lookup functions:

- $\mathtt{P}^{\mathtt{sup}}(\mathtt{cl})$ returns the direct superclass of `cl` in `P`.
- $\mathtt{P}^{\mathtt{fld}}(\mathtt{cl},\mathtt{f})$ returns the type of field `f` as defined in class `cl`.
- $\mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m})$ and $\mathtt{P}^{\mathtt{mth}}(\mathtt{tr},\mathtt{m})$ return the (possibly empty) set of methods with identifier `m` defined in class `cl` or trait `tr`.
- $\mathtt{P}^{\mathtt{use}}(\mathtt{cl})$, or $\mathtt{P}^{\mathtt{use}}(\mathtt{tr})$ return the set of traits directly used in class `cl`, or trait `tr`.
- $\mathtt{P}^{\mathtt{excl}}(\mathtt{tr})$ returns the set of trait and method identifier pairs excluded from trait `tr`.
- $\mathtt{P}^{\mathtt{alias}}(\mathtt{tr},\mathtt{m})$ returns the set of trait and method identifier pairs which are aliases of method `m` in trait `tr`.
- $\mathtt{P}^{\mathtt{req}}(\mathtt{tr})$ returns the set of method signatures mentioned as required in the declaration of `tr`.
- $\mathtt{P}^{\mathtt{req\_sup}}(\mathtt{tr})$ returns the set of method signatures mentioned as required for the superclass (through $\mathtt{t}\ \mathtt{super.m}(\mathtt{t'x})$) in the declaration of `tr`.

Note that the above functions correspond to *direct* lookups in the program text, and do not take class inheritance nor traits use into account. In the next section we will define the functions $\mathcal{F}$, $\mathcal{F}s$, $\mathcal{M}$, $\mathcal{MS}ig$, and $\mathcal{M}^{\mathtt{orig}}$, which lookup fields, and methods, and which *do* take class inheritance and traits use into account.

[2] In a related work [17], a typed language with automatic inference of required methods is described.

3.3 Method or field acquisition through traits use and inheritance

The function $\mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f})$ looks up the field `f` in `cl` or its superclasses, and returns `t` where `t` is the type of `f` in `cl`. The function $\mathcal{F}_\mathrm{s}(\mathtt{P}, \mathtt{cl})$ returns the set of fields defined in `cl`, or inherited from `cl`'s superclasses. The two functions operate only on classes (traits have no fields).

$$\begin{array}{lcl}
\mathcal{F}(\mathtt{P}, \mathtt{Object}, \mathtt{f}) & = & \bot \\
\mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) & = & \begin{cases} \mathtt{P}^{\mathtt{fld}}(\mathtt{cl}, \mathtt{f}) & \text{if } \mathtt{P}^{\mathtt{fld}}(\mathtt{cl}, \mathtt{f}) \neq \bot \\ \mathtt{P}^{\mathtt{fld}}(\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}), \mathtt{f}) & \text{otherwise} \end{cases} \\
\mathcal{F}_\mathrm{s}(\mathtt{P}, \mathtt{cl}) & = & \{\ \mathtt{f}\ \mid \mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) \neq \bot\ \}
\end{array}$$

A class `cl` that uses a trait `tr` acquires the methods from `tr` in such a way that externally there is no way to tell that the methods were not declared by `cl` itself. This forms the basis of the *flattening property* of traits - a trait formed by using existing traits can be viewed as either a composite entity comprising the used traits and the definitions in the new trait, or as a flattened entity containing all the definitions of its constituents.

We now define $\mathcal{M}(\mathtt{P}, \mathtt{cl}, \mathtt{m})$ and $\mathcal{M}(\mathtt{P}, \mathtt{tr}, \mathtt{m})$, which recursively search the used traits and superclasses. Intuitively, these functions embody give precedence to "local" declarations: methods defined in a trait body have highest precedence, and methods that have been aliased have higher precedence than methods acquired from used traits. Methods defined in a class body have highest precedence, and methods acquired from used traits have higher precedence than those acquired from superclasses.

$$\mathcal{M} : \mathtt{program} \times (\mathtt{classId} \cup \mathtt{traitId}) \times \mathtt{methodId} \to \wp(\mathtt{methodBody})$$

If `tr` is a trait :

$$\mathcal{M}(\mathtt{P}, \mathtt{tr}, \mathtt{m}) = \begin{cases} \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) \neq \emptyset \\ \mathit{MsAlias} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) = \emptyset \neq \mathit{MsAlias} \\ \mathit{MsUsed} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) = \emptyset = \mathit{MsAlias}. \end{cases}$$

where

$$\mathit{MsAlias} = \bigcup_{\mathtt{tr}'.\mathtt{m}' \in \mathtt{P}^{\mathtt{alias}}(\mathtt{tr},\mathtt{m})} \mathcal{M}(\mathtt{P}, \mathtt{tr}', \mathtt{m}')$$

$$\mathit{MsUsed} = \bigcup_{\mathtt{tr}'' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr})\ ,\ \mathtt{tr}''.\mathtt{m} \notin \mathtt{P}^{\mathtt{excl}}(\mathtt{tr}.\mathtt{m})} \mathcal{M}(\mathtt{P}, \mathtt{tr}'', \mathtt{m})$$

If `cl` is a class :

$$\mathcal{M}(\mathtt{P}, \mathtt{Object}, \mathtt{m}) = \emptyset$$

$$\mathcal{M}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \begin{cases} \mathtt{P}^{\mathtt{mth}}(\mathtt{cl}, \mathtt{m}) & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl}, \mathtt{m}) \neq \emptyset \\ \mathit{MsUsed} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl}, \mathtt{m}) = \emptyset \neq \mathit{MsUsed} \\ \mathcal{M}(\mathtt{P}, \mathtt{P}^{\mathtt{sup}}(\mathtt{cl}), \mathtt{m}) & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl}, \mathtt{m}) = \emptyset = \mathit{MsUsed}, \end{cases}$$

where

$$\mathit{MsUsed} = \bigcup_{\mathtt{tr} \in \mathtt{P}^{\mathtt{use}}(\mathtt{cl})} \mathcal{M}(\mathtt{P}, \mathtt{tr}, \mathtt{m})$$

Because there are several ways a trait might acquire a method (from any of the traits it uses), the method lookup functions return *sets* of methods. If the precedence rules do not resolve the method lookup to a single method, *i.e.,* if in a class $\mathtt{cl}$, $|\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})| > 1$ for some $\mathtt{m}$, then a conflict occurs, [3].

A class is *complete* if it has no conflicts, and if any call to `super` in any inherited method body resolves without conflict:[4]

$$\frac{\forall \mathtt{m}: \quad \mathtt{e} \text{ contains } \mathtt{super.m}(...) \implies \quad |\mathcal{M}(\mathtt{P},\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{m}')| = 1}{\mathtt{super} \text{ resolves without conflict in } \mathtt{e} \text{ and } \mathtt{cl}}$$

$$\frac{\begin{array}{l}\forall \mathtt{m}:\ |\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})| \leq 1 \\ \forall \mathtt{m},\mathtt{cl}' \quad \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}',\ ...\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m}) \implies \\ \qquad \mathtt{super} \text{ resolves without conflict in } \mathtt{e} \text{ and } \mathtt{cl}\text{'}\end{array}}{\mathtt{P} \vdash_1 \mathtt{cl} \diamond_{\text{cmpl}}}$$

For class $\mathtt{cl}$, and trait $\mathtt{tr}$, we define the functions $\mathcal{MS}\text{ig}_1(\mathtt{P},\mathtt{cl},\mathtt{m})$, $\mathcal{MS}\text{ig}_1(\mathtt{P},\mathtt{tr},\mathtt{m})$, and $\mathcal{MS}\text{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ which return the set of signatures for method $\mathtt{m}$ as found in $\mathtt{cl}$, $\mathtt{tr}$ or the superclass of $\mathtt{cl}$.

$$\begin{array}{ll}\mathcal{MS}\text{ig}_1(\mathtt{P},\mathtt{cl},\mathtt{m}) & = \{\ \mathtt{t\ m(t'\ x)}\ \mid\ \mathtt{t\ m(t'\ x)}\{...\} \in \mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})\ \} \\ \mathcal{MS}\text{ig}_1(\mathtt{P},\mathtt{tr},\mathtt{m}) & = \{\ \mathtt{t\ m(t'\ x)}\ \mid\ \mathtt{t\ m(t'\ x)}\{...\} \in \mathcal{M}(\mathtt{P},\mathtt{tr},\mathtt{m})\ \} \\ \mathcal{MS}\text{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) & = \mathcal{MS}\text{ig}_1(\mathtt{P},\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{m})\end{array}$$

Note, that the look up functions $\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})$ and $\mathcal{MS}\text{ig}_1(\mathtt{P},\mathtt{cl},\mathtt{m})$ abstract from the use of traits. Therefore, as we will see later, we were able to write the operational semantics and type system of *Chai*$_1$ and *Chai*$_2$ without explicit mention of traits.

We define the function $\mathcal{M}^{\mathtt{orig}}$ which determines the "origin" of a method, *i.e.,* the most specific superclass of a class $\mathtt{cl}$ which contains a body for $\mathtt{m}$. We will use $\mathcal{M}^{\mathtt{orig}}$ to model the behaviour of $\mathtt{super.m}(\_)$.[5]

$$\mathcal{M}^{\mathtt{orig}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \begin{cases} \mathtt{cl} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \neq \emptyset \quad \text{or} \quad \exists \mathtt{tr} \text{ with} \\ & \mathtt{tr} \in \mathtt{P}^{\mathtt{use}}(\mathtt{cl}) \text{ and } \mathcal{M}(\mathtt{P},\mathtt{tr},\mathtt{m}) \neq \emptyset, \\ \bot & \text{if } \mathtt{cl} = \mathtt{Object}, \\ \mathcal{M}^{\mathtt{orig}}(\mathtt{P},\mathtt{cl}',\mathtt{m}) & \text{otherwise, for } \mathtt{cl}' = \mathtt{P}^{\mathtt{sup}}(\mathtt{cl}). \end{cases}$$

3.4 Operational Semantics

We give a large step semantics for *Chai*$_1$, where programs map expressions, stacks and heaps, onto results and new heaps. A stack, $\sigma \in$ **stack**, is a triple consisting of the address of the current receiver, the value of the actual parameter

[3] Conflicts can be avoided by overriding the conflicting method in the class where the conflict occurs, or by excluding one of the conflicting methods - in our system without overloading this works only if all conflicting methods have the same signature.

[4] A simpler, but more restrictive, requirement would be to require no conflicts in any of `cl`'s superclasses.

[5] This formalization of `super` has been suggested to us by Andrew Black and Chuan-Kai Lin, and slightly adapted by Rok Strnisa.

$$\begin{array}{ll}
\rightsquigarrow \;:\; & \texttt{program} \;\rightarrow\; \textbf{exp} \times \textbf{stack} \times \textbf{heap} \;\rightarrow\; (\textbf{val} \cup \textbf{dev}) \times \textbf{heap} \\
\textbf{stack} & = \textbf{addr} \times \textbf{val} \times \texttt{classId} \\
\textbf{heap} & = \textbf{addr} \rightarrow \textbf{object} \\
\textbf{val} & = \{ \texttt{null} \} \cup \textbf{addr} \\
\textbf{object} & = \{ \llbracket \texttt{cl} \parallel \texttt{f}_1 : \texttt{v}_1, \ldots, \texttt{f}_r : \texttt{v}_r \rrbracket \mid \texttt{cl} \in \texttt{classId},\, \texttt{f}_1, \ldots, \texttt{f}_r \in \texttt{fldId},\, \texttt{v}_1, \ldots, \texttt{v}_r \in \textbf{val} \} \\
\textbf{addr} & = \{ \iota_n \mid n \text{ is a natural number} \} \\
\textbf{dev} & = \{ \texttt{nllPntrExc}, \texttt{stuckExc} \}
\end{array}$$

Fig. 5. *Chai*$_1$ Runtime

and the class containing the method body currently being executed. The notation $\sigma(\texttt{this})$, $\sigma(\texttt{x})$, and $\sigma(\texttt{this\_class})$ selects the first, second and third component of σ. A heap, $\chi \in$**heap**, maps addresses to objects. Objects contain the class of the object (`cl`), and values ($\texttt{v}_\texttt{i}$) for the object's fields ($\texttt{f}_\texttt{i}$).

The operational semantics of *Chai*$_1$ does not mention traits explicitly, and operates entirely in terms of classes; thus it is very similar to that of a small Java-like language (*e.g.,* ClassicJava[11] or $\mathcal{F}$*ickle*[7]), and is rather standard. It is given in figure 6. The receiver and the parameter are looked up in the heap (**var**). If `null` is dereferenced, a `nullPnterExc` exception is thrown (**null-exception**). Field access is evaluated by looking up the particular field in the object (**field**). Field assignment overrides the corresponding field with the value of the right hand side (**field-assign**). Object creation creates a new object of the appropriate class, and initializes all its fields with `null` (**new**). Method call evaluates the method body found in the *dynamic* class of the receiver; evaluation takes place in a stack consisting of the receiver and actual parameter of the call, and the identifier of the class containing the method body (**method-call**). For the call to `super` the evaluation is similar, but the method is looked up in the *static* superclass of the class given by $\sigma(\texttt{this\_class})$, *i.e.,* the superclass of the class containing the method currently being executed (**super-call**). We require the method lookup functions to return a singleton set, *i.e.,* there should be no conflicting method definitions.

For brevity, we omitted the rules throwing `stuckErr` when conflicting methods are called, or non-existent fields or methods are accessed or called, as well as the rules propagating exceptions `stuckExc` or `nullPntrExc`.

3.5 Type System

In figure 7 we define the judgements $\texttt{P} \vdash \texttt{cl} \leq \texttt{cl}'$ indicating subtypes, and $\texttt{P} \vdash \texttt{cl} \diamond_{\text{class}}$ and $\texttt{P} \vdash \texttt{tr} \diamond_{\text{trait}}$ indicating that `cl` is a class or `tr` is a trait. We also define the judgement $\texttt{P} \vdash \texttt{t} \diamond_{\text{type}}$ indicating that `t` is a type.

For type checking we use a typing environment Γ which maps the receiver, `this`, and the method parameter, `x`, to a class name. The typing judgement $\texttt{P}, \Gamma \vdash_1 \texttt{e} : \texttt{t}$ means that in the context of program `P` and environment Γ, in the type system of *Chai*$_1$, the expression `e` has type `t`. Although in *Chai*$_1$ only classes

null

$$\frac{}{\mathtt{null},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{null}, \chi}$$

null-exception

$$\frac{\mathtt{e},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{null}, \chi'}{\begin{array}{l} \mathtt{e.f} := \mathtt{e}',\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{nllPntrExc}, \chi' \\ \mathtt{e.f},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{nllPnterExc}, \chi' \\ \mathtt{e.m(e')},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{nllPntrExc}, \chi' \end{array}}$$

field

$$\frac{\mathtt{e},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \iota, \chi'}{\mathtt{e.f},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \chi'(\iota)(\mathtt{f}), \chi'}$$

var

$$\frac{}{\begin{array}{l} \mathtt{x},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \sigma(\mathtt{x}), \chi \\ \mathtt{this},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \sigma(\mathtt{this}), \chi \end{array}}$$

field-assign

$$\frac{\begin{array}{l} \mathtt{e},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \iota, \chi'' \\ \mathtt{e}',\ \sigma,\ \chi'' \leadsto_{\mathtt{P}} \mathtt{v}, \chi''' \\ \chi' = \chi'''[\iota \mapsto \chi'''(\iota)[\mathtt{f} \mapsto \mathtt{v}]] \end{array}}{\mathtt{e.f} := \mathtt{e}',\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{v}, \chi'}$$

new

$$\frac{\begin{array}{l} \mathcal{F}_{\mathrm{s}}(\mathtt{P}, \mathtt{cl}) = \mathtt{f}_1, \ldots \mathtt{f}_{\mathtt{r}} \\ \forall \mathtt{k} \in 1, \ldots \mathtt{r} : \mathtt{v}_{\mathtt{k}} = \mathtt{null} \\ \iota \text{ is new in } \chi \end{array}}{\mathtt{new\ cl},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \iota, \chi[\iota \mapsto [\![\, \mathtt{cl} \parallel \mathtt{f}_1 : \mathtt{v}_1, \ldots \mathtt{f}_{\mathtt{r}} : \mathtt{v}_{\mathtt{r}} \,]\!]]}$$

method-call

$$\frac{\begin{array}{l} \mathtt{e}_{\mathtt{r}},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \iota, \chi_0 \\ \mathtt{e}_{\mathtt{a}},\ \sigma,\ \chi_0 \leadsto_{\mathtt{P}} \mathtt{v}_1, \chi_1 \\ \chi_1(\iota) = [\![\, \mathtt{cl} \parallel \ \ldots \,]\!] \\ \mathcal{M}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \{\ \mathtt{t\ m(t'\ x)}\ \{\ \mathtt{e}\ \}\ \} \\ \mathcal{M}^{\mathrm{orig}}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\iota, \mathtt{v}_1, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \leadsto_{\mathtt{P}} \mathtt{v}, \chi' \end{array}}{\mathtt{e}_{\mathtt{r}}\mathtt{.m(e_a)},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{v}, \chi'}$$

super-call

$$\frac{\begin{array}{l} \mathtt{e}_{\mathtt{a}},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{v}_1, \chi_1 \\ \sigma(\mathtt{this\_class}) = \mathtt{cl} \\ \mathtt{P}^{\mathtt{sup}}(\mathtt{cl}) = \mathtt{cl}'' \\ \mathcal{M}(\mathtt{P}, \mathtt{cl}'', \mathtt{m}) = \{\ \mathtt{t\ m(t'\ x)}\ \{\ \mathtt{e}\ \}\ \} \\ \mathcal{M}^{\mathrm{orig}}(\mathtt{P}, \mathtt{cl}'', \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\sigma(\mathtt{this}), \mathtt{v}_1, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \leadsto_{\mathtt{P}} \mathtt{v}, \chi' \end{array}}{\mathtt{super.m(e_a)},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{v}, \chi'}$$

Fig. 6. *Chai*$_1$ Operational Semantics

can be types, the type rules in figure 8 mention types `t` rather than classes `cl`; this generality allows us to reuse these type rules for *Chai*$_2$.

$$\frac{\mathtt{P} = \ldots\ \mathtt{class\ cl\ extends\ cl}'\ \ldots}{\begin{array}{l} \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl} \\ \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}' \\ \mathtt{P} \vdash \mathtt{cl} \diamond_{\mathrm{class}} \\ \mathtt{P} \vdash_1 \mathtt{cl} \diamond_{\mathrm{type}} \end{array}} \qquad \frac{\begin{array}{l} \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}' \\ \mathtt{P} \vdash_1 \mathtt{cl}' \leq \mathtt{cl}'' \end{array}}{\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}''} \qquad \frac{\mathtt{P} = \ldots\ \mathtt{trait\ tr}\ \ldots}{\mathtt{P} \vdash \mathtt{tr} \diamond_{\mathrm{trait}}}$$

Fig. 7. Subclasses and Subtypes in *Chai*$_1$

subsumption

$$\frac{\begin{array}{l}\mathtt{P}, \Gamma \vdash_1 \mathtt{e} : \mathtt{t} \\ \mathtt{P} \vdash_1 \mathtt{t} \leq \mathtt{t}'\end{array}}{\mathtt{P}, \Gamma \vdash_1 \mathtt{e} : \mathtt{t}'}$$

var-this

$$\frac{}{\begin{array}{l}\mathtt{P}, \Gamma \vdash_1 \mathtt{x} : \Gamma(\mathtt{x}) \\ \mathtt{P}, \Gamma \vdash_1 \mathtt{this} : \Gamma(\mathtt{this})\end{array}}$$

new

$$\frac{\mathtt{P} \vdash_1 \mathtt{cl} \diamond_{\text{cmpl}}}{\mathtt{P}, \Gamma \vdash_1 \mathtt{new\,cl} : \mathtt{cl}}$$

null

$$\frac{\mathtt{P} \vdash_1 \mathtt{t} \diamond_{\text{type}}}{\mathtt{P}, \Gamma \vdash_1 \mathtt{null} : \mathtt{t}}$$

field

$$\frac{\begin{array}{l}\mathtt{P}, \Gamma \vdash_1 \mathtt{e} : \mathtt{cl} \\ \mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) = \mathtt{t}\end{array}}{\mathtt{P}, \Gamma \vdash_1 \mathtt{e.f} : \mathtt{t}}$$

field-assign

$$\frac{\begin{array}{l}\mathtt{P}, \Gamma \vdash_1 \mathtt{e} : \mathtt{cl} \\ \mathtt{P}, \Gamma \vdash_1 \mathtt{e}' : \mathtt{t} \\ \mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) = \mathtt{t}\end{array}}{\mathtt{P}, \Gamma \vdash_1 \mathtt{e.f} := \mathtt{e}' : \mathtt{t}}$$

method-call

$$\frac{\begin{array}{l}\mathtt{P}, \Gamma \vdash_1 \mathtt{e_r} : \mathtt{t_r} \\ \mathtt{P}, \Gamma \vdash_1 \mathtt{e_a} : \mathtt{t_a} \\ \mathcal{MS}\text{ig}_1(\mathtt{P}, \mathtt{t_r}, \mathtt{m}) = \{\ \mathtt{t\ m(t}_a\ \mathtt{x)}\ \}\end{array}}{\mathtt{P}, \Gamma \vdash_1 \mathtt{e_r.m(e_a)} : \mathtt{t}}$$

super-call

$$\frac{\begin{array}{l}\Gamma(\mathtt{this}) = \mathtt{t_r} \\ \mathtt{P}, \Gamma \vdash_1 \mathtt{e_a} : \mathtt{t_a} \\ \mathcal{MS}\text{ig}_1^{\mathtt{sup}}(\mathtt{P}, \mathtt{t_r}, \mathtt{m}) = \{\ \mathtt{t\ m(t}_a\ \mathtt{x)}\ \}\end{array}}{\mathtt{P}, \Gamma \vdash_1 \mathtt{super.m(e_a)} : \mathtt{t}}$$

Fig. 8. *Chai*$_1$ Type Rules

The type rules, given in figure 8, *do not explicitly mention traits*, because traits have already been taken into account through $\mathcal{MS}\text{ig}_1(\_,\_,\_)$ and $\mathcal{MS}\text{ig}_1^{\mathtt{sup}}(\_,\_,\_)$. They are standard in all other respects: An expression of a certain type also has any of its supertypes (**subsumption**). The type of the formal parameter and receiver are looked up in the type environment (**var-this**). The creation of a new object has the type of that class, provided that the class is complete (**new**), while `null` has any type (**null**). The type of a method call is the return type of the function found by looking in the class of the first expression through $\mathcal{MS}\text{ig}_1(\mathtt{P}, \mathtt{cl}, \mathtt{m})$, provided that the second expression has the type of the formal parameter type (**method-call**). Similarly for (**super-call**), where the method is looked-up in the superclass through $\mathcal{MS}\text{ig}_1^{\mathtt{sup}}(\mathtt{P}, \mathtt{cl}, \mathtt{m})$.

Note, that required methods do not play any rôle in *Chai*$_1$ type-checking (they do play a rôle in *Chai*$_2$ and *Chai*$_3$ type checking).

In figure 9 we define the notion of a well-formed *Chai*$_1$ class, *i.e.,* $\mathtt{P} \vdash_1 \mathtt{cl}$. A class `cl` is well-formed if:

1. Any field defined in that class has a valid type, and is not defined in its superclass $\mathtt{cl}'$;
2. Any method defined in the class, or acquired though usage of a trait or inheritance from a superclass, has return and parameter type which are valid types, and, in a typing environment which maps `x` to the argument type $\mathtt{t}_1$ and `this` to `cl` (such an environment is written as $\mathtt{t}_1\ \mathtt{x}, \mathtt{cl}\ \mathtt{this}$), the method body has the declared return type $\mathtt{t}_0$. Additionally, if this method is present in the superclass, or any used traits, then it must be defined there with the same return type and parameter type.

The requirement 2. from above is very strong: It checks *all inherited and acquired* methods in a class - rather than just the methods defined in the class itself. Thus, a method defined in a trait will be type checked in all classes using that trait; this is unavoidable, because in Chai_1 method bodies cannot be checked in the traits.

A program is well-formed, $\vdash_1 \mathtt{P}$, if all its classes are well-formed. Note that the traits are not checked. Note also, that we do not require the inheritance hierarchy to be acyclic; although this is convenient, it is not necessary for soundness; in a program with cyclic inheritance, meaning can be given to lookup functions ($\mathcal{M}$, $\mathcal{M}^{\mathtt{orig}}$) through least fixed points.

$$\frac{\begin{array}{l}\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}) = \mathtt{cl}' \\ \forall \mathtt{f}:\ \ \mathtt{P}^{\mathtt{fld}}(\mathtt{cl},\mathtt{f}) = \mathtt{t} \quad \Longrightarrow \quad \mathcal{F}(\mathtt{P},\mathtt{cl}',\mathtt{f}) = \bot,\ \mathtt{P} \vdash_1 \mathtt{t} \diamond_{\text{type}} \\ \forall \mathtt{m}:\ \ \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m}) \quad \Longrightarrow \\ \qquad \mathtt{P} \vdash_1 \mathtt{t}_0 \diamond_{\text{type}} \\ \qquad \mathtt{P} \vdash_1 \mathtt{t}_1 \diamond_{\text{type}} \\ \qquad \mathtt{P}, \mathtt{t}_1\ \mathtt{x},\ \mathtt{cl}\ \mathtt{this} \vdash_1 \mathtt{e}\ :\ \mathtt{t}_0 \\ \qquad \mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \emptyset\ \vee\ \mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \{\,\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\ \{\ldots\}\,\}\end{array}}{\mathtt{P} \vdash_1 \mathtt{cl}}$$

$$\frac{\text{for all classes } \mathtt{cl} \text{ defined in } \mathtt{P}:\quad \mathtt{P} \vdash_1 \mathtt{cl}}{\vdash_1 \mathtt{P}}$$

Fig. 9. Well formed classes and programs in Chai_1

3.6 Type Soundness

The judgement $\mathtt{P}, \sigma \vdash \mathtt{v} \lhd \mathtt{t}$ in figure 10 means that the value `v` agrees with the type `t`. In particular, if `v` is an address, it requires that the object at `v` belongs to a class `cl` which is a subtype of `t`, and for all fields defined in `cl`, the object contains values which agree with the types of the fields as declared in cl.[6] The judgement $\mathtt{P}, \Gamma \vdash_1 \sigma, \chi$ means that the objects in the heap χ agree with their classes, and belong to complete classes (*i.e.,* no conflicts), that the receiver object and argument value agree with their type as given in Γ, and that the class containing the method currently being executed ($\sigma(\mathtt{this\_class})$) is the same as the type of the receiver in the type environment ($\Gamma(\mathtt{this})$).

$$\frac{\mathtt{P} \vdash_1 \mathtt{t} \diamond_{\text{type}}}{\mathtt{P}, \chi \vdash_1 \mathtt{null} \lhd \mathtt{t}} \qquad \frac{\begin{array}{l}\chi(\iota) = [\![\mathtt{cl} \parallel \ \dots]\!] \\ \mathtt{P} \vdash_1 \mathtt{cl} \le \mathtt{t} \\ \mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) = \mathtt{t}' \Longrightarrow \mathtt{P}, \chi \vdash_1 \chi(\iota)(\mathtt{f}) \lhd \mathtt{t}'\end{array}}{\mathtt{P}, \chi \vdash_1 \iota \lhd \mathtt{t}}$$

$$\frac{\begin{array}{l}\forall \iota: \quad \chi(\iota) = [\![\mathtt{cl} \parallel \ \dots]\!] \quad \Longrightarrow \quad \mathtt{P}, \chi \vdash_1 \iota \lhd \mathtt{cl}, \text{and } \mathtt{P} \vdash_1 \mathtt{cl} \diamond_{\text{cmpl}} \\ \mathtt{P}, \sigma \vdash_1 \sigma(\mathtt{this}) \lhd \Gamma(\mathtt{this}) \\ \mathtt{P}, \sigma \vdash_1 \sigma(\mathtt{x}) \lhd \Gamma(\mathtt{x}) \\ \sigma(\mathtt{this\_class}) = \Gamma(\mathtt{this})\end{array}}{\mathtt{P}, \Gamma \vdash_1 \sigma, \chi}$$

Fig. 10. Agreement in *Chai*$_1$

The following lemma is crucial in the proof of soundness, and guarantees that 1-2) the existence and types of fields and methods is preserved to subclasses, 3) that there are no more than one method signature per method in a superclass of a complete class (although there can be several method bodies, and 4) that if a method has a certain signature in a superclass cl', then method lookup in the subclass cl will return a method body which type checks with this signature in the class $\mathtt{cl}''$ which contains this method body (or inherits it from a trait).

Lemma 1 *If* $\vdash_1 \mathtt{P}$ *and* $\mathtt{P} \vdash_1 \mathtt{cl} \le \mathtt{cl}'$ *then*:

1. $\mathcal{F}(\mathtt{P}, \mathtt{cl}', \mathtt{f}) = \mathtt{t} \Longrightarrow \mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) = \mathtt{t}$.
2. $\mathcal{MS}\mathrm{ig}_1(\mathtt{P}, \mathtt{cl}', \mathtt{m}) \subseteq \mathcal{MS}\mathrm{ig}_1(\mathtt{P}, \mathtt{cl}, \mathtt{m})$.
3. $\mathtt{P} \vdash_1 \mathtt{cl} \diamond_{\text{cmpl}} \implies |\, \mathcal{MS}\mathrm{ig}_1(\mathtt{P}, \mathtt{cl}', \mathtt{m}) \,| \le 1$.
4. $\mathtt{t\ m(t'\,x)} \in \mathcal{MS}\mathrm{ig}_1(\mathtt{P}, \mathtt{cl}', \mathtt{m}) \implies \exists \mathtt{cl}'', \mathtt{e}:$
   - $\mathcal{M}^{\text{orig}}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \mathtt{cl}''$, $\quad \mathtt{t\ m(t'\,x)\{e\}} \in \mathcal{M}(\mathtt{P}, \mathtt{cl}, \mathtt{m})$,
   - $\mathtt{P} \vdash_1 \mathtt{cl} \le \mathtt{cl}'' \quad \mathtt{P}, \mathtt{t}'\ \mathtt{x},\ \mathtt{cl}''\ \mathtt{this} \vdash_1 \mathtt{e} : \mathtt{t}$.

We can now prove soundness of the type system:

Theorem 2 (Type Soundness of *Chai*$_1$**)** *For program* P*, typing environment* Γ*, expression* e*, so that* super *resolves without conflict in* e *and* $\Gamma(\mathtt{this\_class})$*, stack* σ*, heap* χ*, and type* t*:*
If

$$\vdash_1 \mathtt{P} \text{ and } \mathtt{P}, \Gamma \vdash_1 \mathtt{e} : \mathtt{t} \text{ and } \mathtt{P}, \Gamma \vdash_1 \sigma, \chi \text{ and } \mathtt{e}, \sigma, \chi \leadsto_{\mathtt{P}} \mathtt{r}, \chi'$$

[6] Although the definition of $\mathtt{P}, \chi \vdash_1 \iota \lhd \mathtt{t}$ is recursive, there exists an equivalent non-recursive definition for it.

then:

$$\mathtt{P}, \Gamma \vdash_1 \sigma, \chi' \qquad \textit{and} \qquad \mathtt{P}, \chi' \vdash_1 \mathtt{r} \triangleleft \mathtt{t} \;\; \textit{or} \;\; \mathtt{r} = \mathtt{nllPntrExc}.$$

In other words, execution of well typed expressions preserves well formedness of the heap and stack, does not get stuck (since `r` is either a value or a null pointer exception), and if it returns a value, then this value is of the same type as the original expression.

4 The language *Chai*$_2$

In *Chai*$_2$ we extended the remit of traits, so that they may be used as types. This has three important repercussions:

First, we can treat in a uniform way objects whose class uses a given trait, *e.g.,* we can write a stack for screenshapes, as in figure 2. Thus, traits support polymorphism, play the role of interfaces, and introduce multiple supertypes.

Second, we can typecheck traits in isolation, and therefore, we will be able to type check a method defined in a trait only once, rather than having to check it again in all the classes using that trait.

Third, we can take required methods into account, and can type check calls to required methods which do not have a method body in the receiver's class or trait. This is safe, because we allow object creation only for *complete* classes, and *Chai*$_2$ complete classes are those that provide method bodies for all required methods.

4.1 *Chai*$_2$ Syntax and Operational Semantics

The only difference between the syntax of *Chai*$_2$ and that of *Chai*$_1$ is that *Chai*$_2$ allows traits to be types, i.e:

$\textit{type} ::= \mathtt{cl} \mid \mathtt{tr}$

The operational semantics of *Chai*$_2$ is identical to that of *Chai*$_1$.

4.2 Required Methods

We fist define *indirect use* of traits, where $\mathcal{U}se^*(\mathtt{P},\mathtt{tr})$ collects the transitive closure of the traits used in `tr`, and $\mathcal{U}se^*(\mathtt{P},\mathtt{cl})$ collects all traits indirectly used by traits used in `cl`, or in `cl`'s superclasses.

$$\mathcal{U}se^*(\mathtt{P},\mathtt{tr}) \;=\; \bigcup_{\mathtt{tr}' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr})} \mathcal{U}se^*(\mathtt{P},\mathtt{tr}') \;\;\cup\;\; \{\, \mathtt{tr} \,\}$$
$$\mathcal{U}se^*(\mathtt{P},\mathtt{cl}) \;=\; \bigcup_{\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}',\; \mathtt{tr} \in \mathtt{P}^{\mathtt{use}}(\mathtt{cl}')} \mathcal{U}se^*(\mathtt{P},\mathtt{tr})$$

A trait `tr` may define a list of *required* methods. A second trait `tr'` which uses `tr` inherits `tr`'s required methods and may add new requirements of its own, through explicit requirements or through exclusion. A class using `tr` inherits the requirements of `tr`.

$$
\begin{array}{lcl}
\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr},\mathtt{m}) & = & \bigcup_{\mathtt{tr}'\in\mathcal{U}se^*(\mathtt{P},\mathtt{tr})} \mathtt{P}^{\mathtt{req}}(\mathtt{tr}') \cup \\
 & & \bigcup_{\mathtt{tr}'\in\mathcal{U}se^*(\mathtt{P},\mathtt{tr})}\{\ \mathtt{t\ m(t'\,x)}\ \mid\ \exists\mathtt{tr}'':\ (\mathtt{tr}'',\mathtt{m})\in\mathtt{P}^{\mathtt{excl}}(\mathtt{tr}'), \\
 & & \quad \mathtt{t\ m(t'\,x)}\in\mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{tr}'',\mathtt{m})\cup\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr}'',\mathtt{m})\ )\} \\
\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{cl},\mathtt{m}) & = & \bigcup_{\mathtt{tr}\in\mathcal{U}se^*(\mathtt{P},\mathtt{cl})} \mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\
\mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m}) & = & \bigcup_{\mathtt{tr}'\in\mathcal{U}se^*(\mathtt{P},\mathtt{tr})} \mathtt{P}^{\mathtt{req\_sup}}(\mathtt{tr}') \\
\mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) & = & \bigcup_{\mathtt{tr}\in\mathcal{U}se^*(\mathtt{P},\mathtt{cl})} \mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m})
\end{array}
$$

Note, that it is possible for a signature to be required in a trait `tr`, and for the trait to have a method body for this signature. Similarly for classes.

A class which is complete in the sense of *Chai*$_1$, and where all required methods have a body is complete for *Chai*$_2$:

$$
\frac{\begin{array}{l}
\forall\mathtt{m}:\ \mathtt{t\ m(t'\,x)}\in\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{cl},\mathtt{m})\implies\ \exists\mathtt{e}:\ \mathtt{t\ m(t'\,x)\{e\}}\in\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m}) \\
\forall\mathtt{m}:\ |\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})|\le 1 \\
\forall\mathtt{m},\mathtt{cl}'\quad \mathtt{P}\vdash_1\mathtt{cl}\le\mathtt{cl}',\ ...\{\mathtt{e}\}\in\mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m})\implies \\
\qquad \mathtt{super}\text{ resolves without conflict in }\mathtt{e}\text{ and}\mathtt{cl}'
\end{array}}{\mathtt{P}\vdash_2\mathtt{cl}\diamond_{\mathrm{cmpl}}}
$$

Thus, a complete subclass of `t` will provide a method body for any method required by `t`. The function $\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})$ returns the signatures of the method that will be provided for `m` by a complete subclass of `t`, while the function $\mathcal{MS}\mathrm{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{t},\mathtt{m})$ returns the signatures of all methods that will be provided in the superclass of a complete subclass of `t`:

$$
\begin{array}{lcl}
\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr},\mathtt{m}) & = & \mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{tr},\mathtt{m})\cup\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\
\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{cl},\mathtt{m}) & = & \mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{cl},\mathtt{m})\cup\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{cl},\mathtt{m}) \\
\mathcal{MS}\mathrm{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m}) & = & \mathcal{MS}\mathrm{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m})\cup\mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\
\mathcal{MS}\mathrm{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) & = & \mathcal{MS}\mathrm{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m})\cup\mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m})
\end{array}
$$

Notice, that $\mathtt{t}\in\mathtt{P}^{\mathtt{use}}(\mathtt{t}')$ implies that $\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})\subseteq\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m})$ for all `m`, and class or trait `t`, and $\mathtt{t}'$.

4.3 Type System

As we define in figure 11, a class or trait is a subtype of any trait that it uses - possibly indirectly. Thus, a class `cl` or trait `tr` that uses a trait $\mathtt{tr}'$ is a subtype of $\mathtt{tr}'$, even if `tr` *requires more* methods than $\mathtt{tr}'$. This may seem surprising, but it is safe for the following reason: even though traits are types, the runtime entities (i.e. the objects) will belong to *complete* classes, which, by definition, provide a method body for any required method. The ensuing subtype relationship is transitive.

Note that $\mathtt{P}\vdash_2\mathtt{t}'\le\mathtt{t}$ implies that $\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})\subseteq\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m})$ - we could have defined subtypes in a structural, rather than a nominal way using the above property.

$$\frac{\mathtt{P}\ \vdash\ \mathtt{tr}\diamond_{\text{class}}}{\mathtt{P}\ \vdash_2 \mathtt{cl}\diamond_{\text{type}}} \qquad \frac{\mathtt{P}\ \vdash\ \mathtt{tr}\diamond_{\text{trait}}}{\mathtt{P}\ \vdash_2 \mathtt{tr}\diamond_{\text{type}}}$$

$$\frac{\mathtt{P}\ \vdash_1 \mathtt{cl} \leq \mathtt{cl}'}{\mathtt{P}\ \vdash_2 \mathtt{cl} \leq \mathtt{cl}'} \qquad \frac{\mathtt{tr} \in \mathcal{U}se^*(\mathtt{P},\mathtt{cl})}{\mathtt{P}\ \vdash_2 \mathtt{cl} \leq \mathtt{tr}} \qquad \frac{\mathtt{tr} \in \mathcal{U}se^*(\mathtt{P},\mathtt{tr}')}{\mathtt{P}\ \vdash_2 \mathtt{tr}' \leq \mathtt{tr}}$$

Fig. 11. Types and Subtypes in *Chai*$_2$

In *Chai*$_2$ traits can be types, therefore in *Chai*$_2$ typing environments may map `this`, and `x` to a trait or a class. The typing rules are the same as those for *Chai*$_1$, with three exceptions. First, the **subsumption** rule uses the new subtype relation $\mathtt{P}\ \vdash_2 \mathtt{t}' \leq \mathtt{t}$. Second, the rules **method-call** and **super-call** take the required method into account, *i.e.,* use $\mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})$ and $\mathcal{MS}\text{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{t},\mathtt{m})$. Third, the rule **new** requires the class to be complete according to $\mathtt{P}\ \vdash_2 \mathtt{cl}\diamond_{\text{cmpl}}$.

A trait `tr` is well formed, *i.e.,* $\mathtt{P}\ \vdash_2\ \mathtt{tr}$ in figure 12, if the methods *directly defined* in that trait are well-typed, and have the same signature as any method with the same identifier acquired from a used trait.

A class `cl` is well formed, *i.e.,* $\mathtt{P}\ \vdash_2\ \mathtt{cl}$, if the fields in that class have well-formed types; and if the methods *directly defined* in that class are well-typed, and have the same signature as any method acquired from a used trait, or inherited from a superclass. A program is well formed, if all its classes and traits are well formed.

Notice that, to establish $\mathtt{P}\ \vdash_2\ \mathtt{t}$ we only check the methods *directly defined* in class or trait `t`; (we use $\mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m})$ - as opposed to $\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})$ in *Chai*$_1$). Also, $\mathtt{P}\ \vdash_1\ \mathtt{t}$ does not *imply* $\mathtt{P}\ \vdash_2\ \mathtt{t}$, and nor does $\mathtt{P}\ \vdash_2\ \mathtt{t}$ imply $\mathtt{P}\ \vdash_1\ \mathtt{t}$.

4.4 Type Soundness

In *Chai*$_2$ we retain the definition of agreement between objects and classes from figure 10, but use the subtype relation $\mathtt{P}\ \vdash_2 \mathtt{t} \leq \mathtt{t}'$, and the definition of complete classes $\mathtt{P}\ \vdash_2 \mathtt{cl}\diamond_{\text{cmpl}}$ from this section.

Thus, we were able to give "uniform" definitions of *Chai*$_1$ and *Chai*$_2$, and distill their similarities and differences.

The following lemma is the counterpart to lemma 1; the difference is that here we talk of types (and thus also of traits) rather than just of classes, we use the *Chai*$_2$ subtype relationship with also incorporates traits usage, and in the *Chai*$_2$ signature lookup function we also take the requirements into account.

Lemma 3 *If* $\vdash_1 \mathtt{P}$ *and classes* `cl`, `cl'` *and types* `t` *and* `t'`, *with* $\mathtt{P}\ \vdash_2 \mathtt{cl} \leq \mathtt{cl}'$, $\mathtt{P}\ \vdash_2 \mathtt{t} \leq \mathtt{t}'$, *and* $\mathtt{P}\ \vdash_2 \mathtt{cl} \leq \mathtt{t}'$, *then*:

1. $\mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m}) \subseteq \mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})$.
2. $\mathtt{P}\ \vdash_2 \mathtt{cl}\diamond_{\text{cmpl}},\ \Longrightarrow\ |\ \mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m})\ | \leq 1$.
3. $\mathtt{P},\mathtt{t_a}\ \mathtt{x},\ \mathtt{t}'\ \mathtt{this}\ \vdash_2 \mathtt{e} : \mathtt{t}'' \quad \Longrightarrow \quad \mathtt{P},\mathtt{t_a}\ \mathtt{x},\ \mathtt{t}\ \mathtt{this}\ \vdash_2 \mathtt{e} : \mathtt{t}''$.
4. $\mathtt{P}\ \vdash_2 \mathtt{cl}\diamond_{\text{cmpl}},\ \ \mathtt{t}''\ \mathtt{m}(\mathtt{t}'''\ \mathtt{x}) \in \mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m}) \quad \Longrightarrow \quad \exists \mathtt{cl}'',\mathtt{e} :$
   - $\mathcal{M}^{\mathtt{orig}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \mathtt{cl}'', \quad \mathtt{t}''\ \mathtt{m}(\mathtt{t}'''\ \mathtt{x})\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl}'',\mathtt{m}),$

$$\frac{\begin{array}{l}\forall \mathtt{m} : \mathtt{t_0}\ \mathtt{m}(\mathtt{t_1}\ \mathtt{x})\{\mathtt{e}\} \in \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) \Longrightarrow \\ \qquad \mathtt{P} \vdash_2 \mathtt{t_0} \diamond_{\text{type}} \\ \qquad \mathtt{P} \vdash_2 \mathtt{t_1} \diamond_{\text{type}} \\ \qquad \mathtt{P},\ \mathtt{t_1}\ \mathtt{x},\ \mathtt{tr}\ \mathtt{this} \vdash_2 \mathtt{e}\ :\ \mathtt{t_0} \\ \qquad \forall \mathtt{tr}' : \mathtt{tr}' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr}) \Longrightarrow \\ \qquad\qquad \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr}',\mathtt{m}) = \emptyset\ \vee\ \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr}',\mathtt{m})\{\mathtt{t_0}\ \mathtt{m}(\mathtt{t_1}\ \mathtt{x})\}\end{array}}{\mathtt{P} \vdash_2 \mathtt{tr}}$$

$$\frac{\begin{array}{l}\mathtt{cl}' = \mathtt{P}^{\mathtt{sup}}(\mathtt{cl}) \\ \forall \mathtt{f}:\ \ \mathtt{P}^{\mathtt{fld}}(\mathtt{cl},\mathtt{f}) = \mathtt{t} \Longrightarrow \mathtt{P} \vdash_2 \mathtt{t} \diamond_{\text{type}}\ ,\ \mathcal{F}(\mathtt{P},\mathtt{cl}',\mathtt{f}) = \bot \\ \forall \mathtt{m}:\ \ \mathtt{t_0}\ \mathtt{m}(\mathtt{t_1}\ \mathtt{x})\{\mathtt{e}\} \in \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \Longrightarrow \\ \qquad \mathtt{P} \vdash_2 \mathtt{t_0} \diamond_{\text{type}} \\ \qquad \mathtt{P} \vdash_2 \mathtt{t_1} \diamond_{\text{type}} \\ \qquad \mathtt{P},\ \mathtt{t_1}\ \mathtt{x},\ \mathtt{cl}\ \mathtt{this} \vdash_2 \mathtt{e}\ :\ \mathtt{t_0} \\ \qquad \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \emptyset\ \vee\ \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \{\mathtt{t_0}\ \mathtt{m}(\mathtt{t_1}\ \mathtt{x})\} \\ \qquad \forall \mathtt{tr} \in \mathtt{P}^{\mathtt{use}}(\mathtt{cl}) : \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr},\mathtt{m}) = \emptyset\ \vee\ \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr},\mathtt{m}) = \{\mathtt{t_0}\ \mathtt{m}(\mathtt{t_1}\ \mathtt{x})\}\end{array}}{\mathtt{P} \vdash_2 \mathtt{cl}}$$

$$\frac{\begin{array}{l}\text{for all classes } \mathtt{cl} \text{ defined in } \mathtt{P}:\ \ \mathtt{P} \vdash_2 \mathtt{cl} \\ \text{for all traits } \mathtt{tr} \text{ defined in } \mathtt{P}:\ \ \mathtt{P} \vdash_2 \mathtt{tr}\end{array}}{\vdash_2 \mathtt{P}}$$

Fig. 12. Well-formed traits, classes and programs in *Chai*₂

– $\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}'' \quad \mathtt{P},\mathtt{t}'''\ \mathtt{x},\ \mathtt{cl}''\ \mathtt{this} \vdash_2 \mathtt{e}''\ :\ \mathtt{t}''$.

With the above lemma we can prove soundness for the type system of *Chai*₂:

Theorem 4 (Type Soundness of *Chai*₂**)** *For any program* $\mathtt{P}$*, environment* Γ*, expression* $\mathtt{e}$ *with* `super` *resolves without conflict in* $\mathtt{e}$ *and* $\Gamma(\mathtt{this\_class})$*, stack* σ*, type* $\mathtt{t}$*, where* $\vdash_2 \mathtt{P}$*, and* $\mathtt{P},\Gamma \vdash_2 \mathtt{e}\ :\ \mathtt{t}$ *and* $\mathtt{P},\Gamma \vdash_2 \sigma,\ \chi$ *and* $\mathtt{e},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{r},\chi'$*:*
$\mathtt{P},\Gamma \vdash_2 \sigma,\ \chi' \qquad$ *and* $\qquad \mathtt{P},\chi \vdash_2 \mathtt{r} \lhd \mathtt{t}$ *or* $\mathtt{r}{=}\mathtt{nllPntrExc}$.

5 The language *Chai*₃

*Chai*₃ introduces *dynamic trait substitution.* Since traits specify pure behaviour, it should be possible to substitute one trait for another at runtime in order to change the behaviour of an object. Outwardly, the interface of the object would remain the same, providing the same fields and methods, but internally the implementation of various methods could be altered.

Although the idea of objects changing behaviour at runtime (dynamic object re-classification) has been presented in several different forms[7, 22], the only time this concept has been explored in the existing literature on traits[7] is relation to the object-based language SELF[1, 22], where dynamic changes in behaviour

[7] The authors of [18] mention using traits to dynamically change object behaviour as an element of future work.

can be obtained by changing which object acts as the parent of the current object. We present a mechanism supporting dynamic traits inspired by the ideas from SELF, but in a class-based language.

5.1 Example

Consider a graphical windowing system: A window in this system may be an `OpenedWindow` or an `IconifiedWindow`. In each state the window will behave differently, and a window may change between these two states at any time.

To implement this in traditional Object Oriented programming, we would need to use wrappers, or some form of the state pattern.

Using dynamic substitution of traits, we can offer a more elegant, and direct solution: we define a class `Window`, and two traits `TOpened` and `TIconified`, where `TOpened` and `TIconified` provide and require the same sets of method signatures, but provide different implementations of the methods and so different behaviour. We define the class `Window` as `class Window uses TOpened ...` (the window begins in the opened state). Then, for a Window object `w` (`Window w = new Window();`) we can change to the iconified state using the statement `w<TOpened ↦TIconified>`. This will result in the substitution of the trait `TIconified` for the trait `TOpened` inside the object `w`.

Since the class `Window` was declared as using the trait `TOpened`, the label `TOpened` becomes a "placeholder" for that trait used by `Window`, and a trait "compatible" with `TOpened` can be substituted for `TOpened` at any time. We use the label `TOpened` in all further substitutions for that trait "placeholder" of `w`. For example, to switch back to the original behaviour of `w`, we write `w<TOpened ↦TOpened>` (and *not*, as might be imagined, `w<TIconified ↦TOpened>`).

5.2 *Chai*$_3$ Syntax and Operational Semantics

We extended the syntax of expressions to allow trait substitution.

$exp ::= \; exp\!< \mathtt{tr} \mapsto \mathtt{tr} > \; | \; \ldots$

Resolving Method Calls Consider the program given in figure 13. If we create an object of class `C`, e.g `C x = new C`, then obviously executing `x.m1`() will return the value 3, and executing `x.m2`() will also return the value 3.

If we execute $\mathtt{x} < \mathtt{TrtB} \mapsto \mathtt{TrtB2} >$ followed by `x.m1`(), then the version of `m1` provided by `TrtA` will be used, since the method `m1` was originally provided to class `C` by trait `TrtA`, and no trait has replaced `TrtA` in `c`.

If we execute $\mathtt{x} < \mathtt{TrtB} \mapsto \mathtt{TrtB2} >$ followed by `x.m2`(), then the situation is more complex. Obviously, the method `m2` defined in `TrtB2` will be executed (since `TrtB` originally provided `m2`, and `TrtB` has been replaced by `TrtB2`). However, there are three possibilities for the binding of `m1` from within the body of `m2`:

1. The version of `m1` from `TrtA` will be used; because invoking a method from within a trait should have the same semantics as invoking it from within the class using the trait. Thus, we resolve methods based on the flattened version of the class using the traits.

```
trait TrtA { int m1() { 3 } }

trait TrtB {
   requires { int m1(); }
   int m2() { this.m1() }
}
```

```
trait TrtB2 {
   int m2() { this.m1() }
   int m1() { 5 }
}

class C uses TrtA,TrtB { }
```

Fig. 13. Resolving Method Calls in *Chai*$_3$

2. The version of `m1` from `TrtB2` will be used; because the methods in `TrtB2` are interrelated, it is likely that the implementor of `TrtB2` intended the call to `m1` to resolve to the method in `TrtB2`. Thus, we resolve methods based on the trait in which the call was found.
3. The situation is illegal; *i.e.,* trait `TrtB2` cannot be substituted for trait `TrtB` because it creates this "ambiguity" regarding the definition of method `m1`.

In this paper, we chose option 1 from above, because of its close relationship to the flattening property which is a crucial element of Traits philosophy.

Object Representation Substitution of traits at runtime is on a per-object basis (rather than a per-class basis). This means that while the list of traits used by any class remains constant, for every object of that class, each used trait may be associated with some (possibly different) trait. Therefore, we extend the representation objects from figure 5 with a list of trait substitutions that have been made to the object.

$$\textbf{object} = \{ \ [\![\ \mathtt{cl} \parallel \mathtt{f}_1 : \mathtt{v}_1, ...\mathtt{f}_\mathtt{r} : \mathtt{v}_\mathtt{r} \parallel \mathtt{tr}_1 : \mathtt{tr}'_1, ...\mathtt{tr}_\mathtt{n} : \mathtt{tr}'_\mathtt{n} \ ]\!] \quad | \quad \mathtt{cl}, \mathtt{f}_1, ...\mathtt{f}_\mathtt{r}, \mathtt{tr}_1, ...\mathtt{tr}_\mathtt{n}, \mathtt{tr}'_1, ...\mathtt{tr}'_\mathtt{n} \text{ identifiers; } \mathtt{v}_1, ...\mathtt{v}_\mathtt{r} \in \textbf{val} \ \}$$

To access and update these trait substitutions for an object $o = [\![\ \mathtt{cl} \parallel ... \parallel \mathtt{tr}_1 : \mathtt{tr}'_1, ...\mathtt{tr}_\mathtt{n} : \mathtt{tr}'_\mathtt{n} \ ]\!]$, we define *trait lookup* $o(\mathtt{tr})$ which finds the current substitution for a given trait name, and *object mutation* $o[\mathtt{tr} \mapsto \mathtt{tr}']$ which replaces the trait named `tr` by $\mathtt{tr}'$.

$$o(\mathtt{tr}) = \begin{cases} \mathtt{tr}'_\mathtt{k} & \text{if } \mathtt{tr}{=}\mathtt{tr}_\mathtt{k} \text{ for some } \mathtt{k} \in 1, ...\mathtt{n} \\ \bot & \text{otherwise.} \end{cases}$$

$$o[\mathtt{tr} \mapsto \mathtt{tr}'] = \begin{cases} [\![\ \mathtt{cl} \parallel ... \parallel \mathtt{tr}_1 {:} \mathtt{tr}'_1 ... \ \mathtt{tr}_\mathtt{k} {:} \mathtt{tr}' \ ...\mathtt{tr}_\mathtt{n} {:} \mathtt{tr}'_\mathtt{n} \ ]\!] & \text{for } \mathtt{tr}{=}\mathtt{tr}_k, \mathtt{k} \in 1,...,\mathtt{n} \\ \bot & \text{otherwise.} \end{cases}$$

Runtime Method Lookup and Operational Semantics Trait substitutions must be taken into account for method call. The function $\mathcal{M}_3$ finds the appropriate method body, taking both the class of the object, and the object itself into account – the latter is needed, in order to find the traits that have replaced the original ones. $\mathcal{M}_3$ first determines which class or trait name is "responsible" for the corresponding method through $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$, which first searches the current class, then the used traits, and then continues with the superclass. If $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ is a class $\mathtt{cl}'$ then the method body is found directly in $\mathtt{cl}'$. If $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ is a trait $\mathtt{tr}$ then the method body is found in trait $\mathtt{tr}'$, which replaces $\mathtt{tr}$ in the current object (*i.e.,* $o(\mathtt{tr}) = \mathtt{tr}'$).

$$\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{tr},\mathtt{m}) = \begin{cases} \{\ \mathtt{tr}\ \} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{tr},\mathtt{m}) \neq \emptyset \\ \bigcup_{\mathtt{tr}' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr})} \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{tr}',\mathtt{m}) & \text{otherwise.} \end{cases}$$

$$\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \begin{cases} \{\ \mathtt{cl}\ \} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \neq \emptyset \\ \mathit{Trts} & \text{where } \mathit{Trts} = \bigcup_{\mathtt{tr} \in \mathtt{P}^{\mathtt{use}}(\mathtt{cl})} \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\ & \text{if } \mathit{Trts} \neq \emptyset = \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \\ \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{m}) & \text{otherwise.} \end{cases}$$

$$\mathcal{M}_3(\mathtt{P},\mathtt{cl},\mathtt{o},\mathtt{m}) = \begin{cases} \mathtt{P}^{\mathtt{mth}}(\mathtt{cl}',\mathtt{m}) & \text{if } \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \{\mathtt{cl}'\} \\ \mathtt{P}^{\mathtt{mth}}(\mathtt{tr}',\mathtt{m}) & \text{if } \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \{\mathtt{tr}\},\ \text{and } o(\mathtt{tr}) = \mathtt{tr}' \\ \bot & \text{otherwise.} \end{cases}$$

A class $\mathtt{cl}$ is complete in *Chai*$_3$ if it provides a method body for any required method, if there are no conflicts for any superclass (this simplifies the treatment of $\mathtt{super}$), and if $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ is empty or a singleton.

$$\frac{\begin{array}{l} \forall \mathtt{m}:\ \mathtt{t\ m(t'\ x)} \in \mathcal{MR}\text{eq}(\mathtt{P},\mathtt{cl},\mathtt{m}) \implies \exists \mathtt{e}:\ \mathtt{t\ m(t'\ x)\{e\}} \in \mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m}) \\ \forall \mathtt{m},\mathtt{cl}' \quad \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}',\ |\mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m})| \leq 1 \\ \forall \mathtt{m}:\ \ |\,\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})\,| \ \leq 1 \end{array}}{\mathtt{P} \vdash_3 \mathtt{cl} \diamond_{\text{cmpl}}}$$

The operational semantics of *Chai*$_3$ differs from that of *Chai*$_1$ and *Chai*$_2$ in the handling of mutation, object creation, and method call, therefore, we extend the semantics from figure 6. A mutate expression substitutes one trait by another (**mutate**). Object creation initializes the fields *and* the list of trait substitutions for new objects through the identity substitution, i.e. associates all traits with themselves (**new**).

mutate

$$\frac{\begin{array}{l} \mathtt{e},\sigma,\chi \leadsto_{\mathtt{P}} \iota,\chi'' \\ \chi' = \chi''[\iota \mapsto \chi''(\iota)[\mathtt{tr} \mapsto \mathtt{tr}']] \end{array}}{\mathtt{e} < \mathtt{tr}\ \mapsto\ \mathtt{tr}' >,\sigma,\chi \leadsto_{\mathtt{P}} \iota,\chi'}$$

new

$$\frac{\begin{array}{l} \mathcal{F}_{\text{s}}(\mathtt{P},\mathtt{cl}) = \{\ \mathtt{f}_1,\ldots,\mathtt{f}_\mathtt{r}\ \} \\ \{\ \mathtt{tr}_1,\ldots\mathtt{tr}_\mathtt{n}\ \} = \mathcal{U}se^*(\mathtt{P},\mathtt{cl}) \\ \iota \text{ is new in } \chi \\ o = [\![\ \mathtt{cl}\ \|\ \mathtt{f}_1\!:\!\mathtt{null}...\mathtt{f}_\mathtt{n}\!:\!\mathtt{null}\ \| \\ \qquad \mathtt{tr}_1\!:\!\mathtt{tr}_1...\mathtt{tr}_\mathtt{n}\!:\!\mathtt{tr}_\mathtt{n}\ ]\!] \end{array}}{\mathtt{new\ cl},\sigma,\chi \leadsto_{\mathtt{P}} \iota,\chi[\iota \mapsto \mathtt{o}]}$$

In method call we use the new method lookup function $\mathcal{M}_3(\mathtt{P},\mathtt{c},\mathtt{o},\mathtt{m})$ (**method-call**). Thus, if a trait is used in class $\mathtt{cl}$ through two different paths (*e.g.,* used by cl, and also by $\mathtt{cl}'$, where $\mathtt{cl}'$ is cl's superclass), then mutation of the trait will affect the behaviour of its methods regardless of the path used to access the object (*e.g.,* as a value of type cl, or $\mathtt{cl}'$) - this is consistent with the flattening property. On the other hand, if a trait tr which uses trait $\mathtt{tr}'$ is replaced by $\mathtt{tr}''$, then only the methods directly provided by tr will be looked up in trait $\mathtt{tr}''$; the ones that were inherited by $\mathtt{tr}'$ will remain unaffected. This is, in some sense, inconsistent with the flattening property, and in further work we would like to investigate alternatives.

method-call

$$\frac{\begin{array}{l} \mathtt{e_r},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \iota, \chi_0 \\ \mathtt{e_a},\ \sigma,\ \chi_0 \leadsto_{\mathtt{P}} \mathtt{v_1}, \chi_1 \\ \chi_1(\iota) = [\![\, \mathtt{cl} \,\|\, \ldots \,]\!] \\ \mathcal{M}_3(\mathtt{P}, \mathtt{cl}, \chi_1(\iota), \mathtt{m}) = \\ \qquad\qquad \{\ \mathtt{t\ m(t'\ x)\ \{\ e\ \}}\ \} \\ \mathcal{M}^{\mathtt{orig}}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\iota, \mathtt{v_1}, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \leadsto_{\mathtt{P}} \mathtt{v}, \chi' \end{array}}{\mathtt{e_r.m(e_a)},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{v}, \chi'}$$

super-call

$$\frac{\begin{array}{l} \mathtt{e_a},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{v_1}, \chi_1 \\ \sigma(\mathtt{this\_class}) = \mathtt{cl} \\ \mathtt{P^{sup}}(\mathtt{cl}) = \mathtt{cl}'' \\ \mathcal{M}_3(\mathtt{P}, \mathtt{cl}'', \chi_1(\iota), \mathtt{m}) = \\ \qquad\qquad \{\ \mathtt{t\ m(t'\ x)\ \{\ e\ \}}\ \} \\ \mathcal{M}^{\mathtt{orig}}(\mathtt{P}, \mathtt{cl}'', \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\sigma(\mathtt{this}), \mathtt{v_1}, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \leadsto_{\mathtt{P}} \mathtt{v}, \chi' \end{array}}{\mathtt{super.m(e_a)},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{v}, \chi'}$$

5.3 Type System

The judgment $\mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}$ says that trait $\mathtt{tr}'$ may replace another trait tr. It requires that $\mathtt{tr}'$ provides all the methods that tr does (with the same signatures, but possibly different bodies), and that any methods provided or required by $\mathtt{tr}'$ are also provided or required in tr.

$$\frac{\begin{array}{l} \mathtt{P} \vdash \mathtt{tr} \diamond_{\text{trait}} \qquad\qquad \mathtt{P} \vdash \mathtt{tr}' \diamond_{\text{trait}} \\ \forall \mathtt{m}:\ \ \mathtt{t_0\ m(t_1\ x)\{\ldots\}} \in \mathtt{P^{mth}}(\mathtt{tr}, \mathtt{m}) \Longrightarrow \mathtt{t_0\ m(t_1\ x)\{\ldots\}} \in \mathtt{P^{mth}}(\mathtt{tr}', \mathtt{m}) \\ \forall \mathtt{m}:\ \ \mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) \subseteq \mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m}) \end{array}}{\mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}}$$

We require $\mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) \subseteq \mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m})$[8] because $\mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}$ and $\mathtt{P},\ \mathtt{t}'\ \mathtt{x}, \mathtt{tr}'\ \mathtt{this} \vdash_3 \mathtt{e} : \mathtt{t}$ should imply $\mathtt{P}, \mathtt{t}'\ \mathtt{x}, \mathtt{tr}\ \mathtt{this} \vdash_3 \mathtt{e} : \mathtt{t}$ – namely, if an object contains a trait placeholder tr, which is replaced by $\mathtt{tr}'$, then it may execute method body e which was defined in $\mathtt{tr}'$. To satisfy $\mathtt{P},\ \mathtt{t}'\ \mathtt{x}, \mathtt{tr}\ \mathtt{this} \vdash_3 \mathtt{e} : \mathtt{t}$ for the case where e=this, we need $\mathtt{P} \vdash_3 \mathtt{tr} \leq \mathtt{tr}'$, which requires $\mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) \subseteq \mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m})$.

In our example, $\_ \vdash \mathtt{TrtB2} \lesssim \mathtt{TrtB}$, and $\_ \not\vdash \mathtt{TrtB} \lesssim \mathtt{TrtB2}$ – because TrtB2 has a method body for m1, and TrtB has not.

Because trait substitutability implies subtypes, in *Chai*$_3$ we extend the subtype relationship from figure 7 as follows:

[8] Andrew Black suggested to us that we could weaken our original requirement of $\mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) = \mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m})$.

$$\frac{\texttt{P} \vdash_2 \texttt{t}' \leq \texttt{t}}{\texttt{P} \vdash_3 \texttt{t}' \leq \texttt{t}} \qquad \frac{\texttt{P} \vdash \texttt{tr}' \lesssim \texttt{tr}}{\texttt{P} \vdash_3 \texttt{tr} \leq \texttt{tr}'} \qquad \frac{\texttt{P} \vdash_3 \texttt{t}' \leq \texttt{t}'' \quad \text{and} \quad \texttt{P} \vdash_3 \texttt{t}'' \leq \texttt{t}}{\texttt{P} \vdash_3 \texttt{t}' \leq \texttt{t}}$$

The type system of *Chai*$_3$ is identical to that of *Chai*$_2$, except for the new definition of subtypes ($\texttt{P} \vdash_3 \texttt{t}' \leq \texttt{t}$) and complete classes ($\texttt{P} \vdash_3 \texttt{cl} \diamond_{\text{cmpl}}$), and the addition of the rule for mutation expressions. It requires that the type of `e` should be any class or trait `t`, that `t` should be using a trait `tr`, and that $\texttt{tr}'$ may replace `trin` `t`. Then, the substitution of `tr` through $\texttt{tr}'$ in `e` has type `t`:

mutate

$$\frac{\begin{array}{l}\texttt{P}, \Gamma \vdash_3 \texttt{e} : \texttt{t} \\ \texttt{tr} \in \mathcal{U}se^*(\texttt{P}, \texttt{t}) \\ \texttt{P} \vdash \texttt{tr}' \lesssim \texttt{tr}\end{array}}{\texttt{P}, \Gamma \vdash_3 \texttt{e} < \texttt{tr} \mapsto \texttt{tr}' > : \texttt{t}}$$

5.4 Type Soundness

Agreement for *Chai*$_3$ is defined in the following. In addition to the properties for agreement in *Chai*$_2$, for *Chai*$_3$ we use the new subtype relation ($\texttt{P} \vdash_3 \texttt{cl} \leq \texttt{t}$), and require that all traits used by class `cl` should appear in the representation of the objects, and that all traits have been replaced by substitutable traits:

$$\frac{\begin{array}{l}\chi(\iota) = [\![\ \texttt{cl} \parallel \ \ldots \ \parallel \texttt{tr}_1 : \texttt{tr}'_1, \ldots, \texttt{tr}_\texttt{n} : \texttt{tr}'_\texttt{n}\ ]\!] \\ \{\ \texttt{tr}_1 \ldots \texttt{tr}_\texttt{n}\ \} = \mathcal{U}se^*(\texttt{P}, \texttt{cl}) \\ \forall \texttt{i} \in 1, \ldots, \texttt{n}: \ \ \texttt{P} \vdash \texttt{tr}'_\texttt{i} \lesssim \texttt{tr}_\texttt{i} \\ \texttt{P} \vdash_3 \texttt{cl} \leq \texttt{t} \\ \mathcal{F}(\texttt{P}, \texttt{cl}, \texttt{f}) = \texttt{t}' \implies \texttt{P}, \chi \vdash_3 \chi(\iota)(\texttt{f}) \triangleleft \texttt{t}'\end{array}}{\texttt{P}, \chi \vdash_3 \iota \triangleleft \texttt{t}}$$

The counterparts to the properties from lemmas 1 and 3 hold for *Chai*$_3$.

Lemma 5 *For program* `P` *with* $\vdash_3 \texttt{P}$*, classes* `cl`*,* $\texttt{cl}'$*, types* `t`*,* $\texttt{t}'$*,* $\texttt{t}''$*, with* $\texttt{P} \vdash_3 \texttt{cl} \leq \texttt{cl}'$*, and* $\texttt{P} \vdash_3 \texttt{t} \leq \texttt{t}'$:

1. $\mathcal{F}(\texttt{P}, \texttt{cl}', \texttt{f}) = \texttt{t} \implies \mathcal{F}(\texttt{P}, \texttt{cl}, \texttt{f}) = \texttt{t}$.
2. $\mathcal{MS}\text{ig}_2(\texttt{P}, \texttt{t}', \texttt{m}) \subseteq \mathcal{MS}\text{ig}_2(\texttt{P}, \texttt{t}, \texttt{m})$.
3. $\texttt{P}, \texttt{t}_\texttt{a}\ \texttt{x},\ \texttt{t}'\ \texttt{this} \vdash_3 \texttt{e} : \texttt{t}'' \implies \texttt{P}, \texttt{t}_\texttt{a}\ \texttt{x},\ \texttt{t}\ \texttt{this} \vdash_3 \texttt{e} : \texttt{t}''$.
4. $\texttt{P}, \sigma \vdash_3 \iota \triangleleft \texttt{cl}$, *and* $\texttt{P} \vdash_3 \texttt{cl} \diamond_{\text{cmpl}}$, *and* $\mathcal{M}^{\texttt{orig}}(\texttt{P}, \texttt{cl}', \texttt{m}) = \{\ \texttt{cl}''\ \}$, *and* $\texttt{t}_0\ \texttt{m}(\texttt{t}_1\ \texttt{x})\{\texttt{e}\} \in \mathcal{M}_3(\texttt{P}, \texttt{cl}, \chi(\iota), \texttt{m})$, $\implies$
   – $\texttt{P} \vdash_3 \texttt{cl}' \leq \texttt{cl}''$
   – $\texttt{P}, \texttt{t}_1\ \texttt{x},\ \texttt{cl}''\ \texttt{this} \vdash_3 \texttt{e}'' : \texttt{t}_0$.
5. $\texttt{t}_0\ \texttt{m}(\texttt{t}_1\ \texttt{x}) \in \mathcal{MS}\text{ig}_2(\texttt{P}, \texttt{t}, \texttt{m})$, *and* $\texttt{P}, \sigma \vdash_3 \iota \triangleleft \texttt{cl}$, *and* $\texttt{P} \vdash_3 \texttt{cl}' \leq \texttt{t}$, *and* $\texttt{P} \vdash_3 \texttt{cl} \diamond_{\text{cmpl}} \implies \mathcal{M}_3(\texttt{P}, \texttt{cl}', \chi(\iota), \texttt{m}) = \{\ \texttt{t}_0\ \texttt{m}(\texttt{t}_1\ \texttt{x})\{\ldots\}\ \}$.

We can now prove soundness for the type system of *Chai*$_3$:

Theorem 6 (Type Soundness of *Chai*$_3$**)** *For any program* $\mathtt{P}$*, environment* Γ*, expression* $\mathtt{e}$*, stack* σ*, heap* χ*, type* $\mathtt{t}$*, where* $\vdash_3 \mathtt{P}$*, and* $\mathtt{P}, \Gamma \vdash_3 \mathtt{e} : \mathtt{t}$ *and* $\mathtt{P}, \Gamma \vdash_3 \sigma, \chi$ *and* $\mathtt{e}, \sigma, \chi \leadsto_{\mathtt{P}} \mathtt{r}, \chi'$:

$\mathtt{P}, \chi' \vdash \mathtt{r} \lhd \mathtt{t}$ *or* $\mathtt{r} = \mathtt{nullPointerExc}$ *and* $\mathtt{P}, \Gamma \vdash_3 \sigma, \chi'$.

6 Implementation

This section describes the translation of a program in *Chai* (the source language) to one in Java (the target language). This is implemented by a mapping from traits and classes in *Chai* to entities in Java[9] There are several possible mappings we could have chosen for this purpose; we could map a class (and all the behaviour it includes from traits) in *Chai* to a single class in Java. Instead, we choose a slightly more complex mapping, which represents traits in Java by classes which are instantiated to give *proxy objects* to which behaviour can be delegated by a class which uses those traits. This allows us to implement the dynamic trait substitution of *Chai*$_3$.

Every trait `tr` is represented by an object of type `tr_impl`, and contains a field called `user_proxy` of type `tr_user`. The `user_proxy` field always stores a reference to an object of the trait or class that uses this trait. Also, for any class or trait, there are fields `tr′_proxy` for all traits `tr′` used by the class or trait. Each of these is a reference to an object of the relevant type `tr′_interface`

Take, for example, a class `D` which uses a trait `T3`, and `T3` uses traits `T1` and `T2`. Because $\mathtt{T3} \in \mathtt{P}^{\mathtt{use}}(\mathtt{D})$, the `D` object contains a reference to a `T3_impl` object. Similarly because $\mathtt{T1} \in \mathtt{P}^{\mathtt{use}}(\mathtt{T3})$ and $\mathtt{T2} \in \mathtt{P}^{\mathtt{use}}(\mathtt{T3})$, the `T3` object contains references to `T1_impl` and `T2_impl` objects.

In order for this arrangement to be type correct, all classes `tr_impl` must implement `tr_interface`, and also `tr′_user` for all `tr′` such that $\mathtt{tr}' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr})$. Additionally, classes that use traits must implement the appropriate `tr_user` interfaces (in the example, $\mathtt{T3} \in \mathtt{P}^{\mathtt{use}}(\mathtt{D})$ and so `D` must implement `T3_user`).

The reason that the type of the fields `tr_proxy` is `tr_interface` (and not `tr_impl`) is to allow different values stored in the field to refer to different trait implementation objects (provided that they implement `tr_interface`), and support trait substitutions (under the restrictions described by *Chai*$_3$).

In more detail, every trait `tr` in *Chai* is mapped to three entities in Java:

1. A *trait interface* containing all the provided methods of `tr`.
2. A *trait-user interface* containing all the required method signatures (i.e. those expected to be provided by the user of the trait `tr`), as well as all the provided method signatures of `tr` (see below).
3. A *trait implementation class* which contains the definitions for the provided methods of the trait, proxy fields for the user of the trait and all used traits, as well as delegation method stubs for acquired methods, which forward method calls to the used trait proxy objects.

[9] Similarly, Java-mixins were implemented through a mapping from Jam into Java[2].

A class in *Chai* is mapped to a class in Java, with the addition of proxy fields for used traits, implements declarations for the trait-user interfaces of traits used by the class, and method stubs for acquired methods and superclass methods required by a used trait.

To preserve the intended semantics of the flattening property (see section 3.3), it is necessary that the use of the expression `this` within a trait proxy is translated to refer the object belonging to the class which uses the trait (note that there may be several levels of intervening trait proxies between the trait proxy and this object). The reason that this is necessary, is that declarations of methods "most local" to the eventual user of a trait have precedence, therefore to preserve the flattening property, we must start the search for a method implementation from this user object itself and work upward into traits represented by proxy objects.

Prototype The prototype implementation of the compiler is written in Java. At present it supports all of the features of $Chai_1$, and would easily accommodate extensions to support $Chai_2$ and $Chai_3$. The compiler, including full source code, is available from `http://chai-t.sourceforge.net/`.

7 Conclusions, related and further Work

We have developed three extensions to a minimal Java-like language incorporating traits, have proven soundness of the type systems, and have outlined our prototype implementation.

The main issues we had to address during the design of *Chai* were:

- The precise semantics of using a trait as part of a class in Java;
- How to perform type-checking on traits, and in particular how to avoid having to type-check the same method body in each class that uses a trait;
- The reflection of calls to `super` in the requirements part of traits
- In how far classes have to be complete, *i.e.,* provide method bodies for all the methods required by the traits they are using;
- Subtype relationships between classes and traits, as required in $Chai_2$; interestingly, a trait may require *more* methods than a supertype trait;
- Dynamic substitution of traits, and the semantics of method lookup in $Chai_3$;
- The trait substitutability relationship in $Chai_3$; interestingly, substitutability in $Chai_3$ does not imply subtype in $Chai_2$.

Recently, and especially after the application of traits to Smalltalk [18, 19], the interest in traits has boomed. In [10] a imperative calculus for traits in the language Moby is developed. The acquisition of methods trough the use of traits is modeled through "class evaluation" which returns flattened classes. As in our work, alias and exclusion of methods in [10] is accompanied by method signatures; unlike our work, traits in [10] may require the presence of fields.

In FTJ [13] traits are added to Featherweight Java[12]; the system is functional, and traits are treated as a class creation mechanism, similar to *Chai*$_1$. The full calculus of FTJ and a proof of soundness of the type system is presented.

Traits are part of the language Scala [15], where they play similar rôle to that of *Chai*$_1$ and *Chai*$_2$. Scala incorporates many advanced features *e.g.,* generics, and dependent types; it is unknown whether its type system is decidable [16].

The Software Composition group at the University of Berne [8] contains a large center for the research around the design, semantics, and application of traits. Tools for Traits for Squeak are being developed, and Microsoft research is sponsoring the design and implementation of traits for C#.

In further work, we would like to refine our model to support overloading. We also want to revisit and reconsider the design decisions in *Chai*$_2$ and *Chai*$_3$; so far they were taken just with the aim to obtain type soundness, but we should explore their implications for the style of programming. We also want to explore the design space for traits, its relation with generic features [6], possibly also incorporate polymorphic features into traits. We also would like to consider generalization of the languages, *e.g.,* allow classes to have trait glue, or allow trait glue to require fields.

Acknowledgments We are deeply grateful to Andrew Black, Tim Sheard, and their research group, for insightful suggestions, pertinent questions, and detailed comments on an earlier version of this work. Their feedback has greatly improved the suggested language features in *Chai*$_1$, *Chai*$_2$ and *Chai*$_3$, the explanations, and presentation – we have not yet had the time to incorporate all of it. Chuan-Kai Lin and Andrew Black suggested the correct fomalization of `super`.

The anonymous ECOOP referees, Alex Buckley, Christopher Anderson, Nick Cameron, and Rok Strnisa gave us valuable feedback. Nobuko Yoshida suggested the distillation of the essential properties used in the proof of soundness for *Chai*$_1$, *Chai*$_2$ and *Chai*$_3$, as in lemmas 1, 3 and 5.

References

1. Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, and David Ungar. *The Self 4.0 Programmer's Reference Manual.* Sun Microsystems, Inc., 1995.
2. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - A Smooth Extension of Java with Mixins. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 154–178. Springer-Verlag, 2000.
3. Andrew Black, Nathanael Schärli, and Stéphane Ducasse. Applying Traits to the Smalltalk Collection Hierarchy. pages 47–64. ACM Conference on Object Oriented Systems, Languages and Applications (OOPSLA), October 2003.
4. Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance.* PhD thesis, University of Utah, 1992.
5. Gilad Bracha and William Cook. Mixin-Based Inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

6. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.
7. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP'01*, LNCS 2072, pages 130–149. Springer, 2001.
8. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, and Roel Wuyts. Traits - Composable Units of Behaviour. University of Berne, Software Composition Group, `http://www.iam.unibe.ch/ scg/Research/Traits/index.html`.
9. Erik Ernst. Family Polymorphism. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 303–326. Springer-Verlag, 2001.
10. Kathleen Fisher and John Reppy. Statically Typed Traits. Technical Report TR-2003-13, Department of Computer Science, University of Chicago, December 2003. presented at FOOL, January 2004.
11. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
12. Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.
13. L. Liquori and A.Spiwack. Featherweight-Trait Java, A Trait-based Extension for FJ. 2004, http://www-sop.inria.fr/mirho/Luigi.Liquori/PAPERS/ftj.ps.gz.
14. Bertrand Meyer. *Eiffel: the Language*. Prentice-Hall, 1988.
15. Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenge. The Scala Language Specification Version 1.0. Technical report, Programming Methods Laboratory, EPFL, Switzerland, 2004.
16. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. ECOOP'03*, Springer LNCS, 2003.
17. Philip J. Quitslund and Andrew P. Black. Java with Traits — Improving Opportunities for Reuse. In *The MASPEGHI Workshop at ECOOP 2004*.
18. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable Units of Behavior. European Conference on Object-Oriented Programming (ECOOP), Springer LNCS 2743, July 2003.
19. Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The Formal Model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002.
20. Charles Smith. Typed Traits, September. MSc thesis - Department of Computing, Imperial College London, September 2004, `http://chai-t.sourceforge.net/`.
21. B. Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.
22. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.