Typed First-Class Traits

Xuan Bi
The University of Hong Kong, Hong Kong, China
xbi@cs.hku.hk

Bruno C. d. S. Oliveira
The University of Hong Kong, Hong Kong, China
bruno@cs.hku.hk

Abstract

Many dynamically-typed languages (including JavaScript, Ruby, Python or Racket) support *first-class classes*, or related concepts such as first-class traits and/or mixins. In those languages classes are first-class values and, like any other values, they can be passed as an argument, or returned from a function. Furthermore first-class classes support *dynamic inheritance*: i.e. they can inherit from other classes at *runtime*, enabling programmers to abstract over the inheritance hierarchy. In contrast, type system limitations prevent most statically-typed languages from having first-class classes and dynamic inheritance.

This paper shows the design of SEDEL: a polymorphic statically-typed language with *first-class traits*, supporting *dynamic inheritance* as well as conventional OO features such as *dynamic dispatching* and *abstract methods*. To address the challenges of type-checking first-class traits, SEDEL employs a type system based on the recent work on *disjoint intersection types* and *disjoint polymorphism*. The novelty of SEDEL over core disjoint intersection calculi are *source level* features for practical OO programming, including first-class traits with dynamic inheritance, dynamic dispatching and abstract methods. Inspired by Cook and Palsberg's work on the denotational semantics for inheritance, we show how to design a source language that can be elaborated into Alpuim et al.'s F_i (a core polymorphic calculus with records supporting disjoint polymorphism). We illustrate the applicability of SEDEL with several example uses for first-class traits, and a case study that modularizes programming language interpreters using a highly modular form of visitors.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases traits, extensible designs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.9

Funding Hong Kong Research Grant Council projects number 17210617 and 17258816

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Many dynamically typed-languages (including JavaScript [1], Ruby [4], Python [2] or Racket [3]) support *first-class classes* [26], or related concepts such as first-class mixins and/or traits. In those languages classes are first-class values and, like any other values, they can be passed as an argument, or returned from a function. Furthermore first-class classes support *dynamic inheritance*: i.e., they can inherit from other classes at *runtime*, enabling programmers to abstract over the inheritance hierarchy. Those features make first-class classes very powerful and expressive, and enable highly modular and reusable pieces of code, such as:

```
const mixin = Base => { return class extends Base { ... } };
```

In this piece of JavaScript code, `mixin` is parameterized by a class `Base`. Note that the concrete implementation of `Base` can be even dynamically determined at runtime, for example after reading a configuration file to decide which class to use as the base class. When applied to an argument, `mixin` will create a new class on-the-fly and return that as a result. Later that class can be instantiated and used to create new objects, as any other classes.

In contrast, most statically-typed languages do not have first-class classes and dynamic inheritance. While all statically-typed OO languages allow first-class *objects* (i.e. objects can be passed as arguments and returned as results), the same is not true for classes. Classes in languages such Scala, Java or C++ are typically a second-class construct, and the inheritance hierarchy is *statically determined*. The closest thing to first-class classes in languages like Java or Scala are classes such as `java.lang.Class` that enable representing classes and interfaces as part of their reflective framework. `java.lang.Class` can be used to mimic some of the uses of first-class classes, but in an essentially dynamically-typed way. Furthermore simulating first-class classes using such mechanisms is highly cumbersome because classes need to be manipulated programmatically. For example instantiating a new class cannot be done using the standard **new** construct, but rather requires going through API methods of `java.lang.Class`, such as `newInstance`, for creating a new instance of a class.

Despite the popularity and expressive power of first-class classes in dynamically-typed languages, there is surprisingly little work on typing of first-class classes (or related concepts such as first-class mixins or traits). First-class classes and dynamic inheritance pose well-known difficulties in terms of typing. For example, in his thesis, Bracha [15] comments several times on the difficulties of typing dynamic inheritance and first-class mixins, and proposes the restriction to static inheritance that is also common in statically-typed languages. He also observes that such restriction poses severe limitations in terms of expressiveness, but that appeared (at the time) to be a necessary compromise when typing was also desired. Only recently some progress has been made in statically typing first-class classes and dynamic inheritance. In particular there are two works in this area: Racket's gradually typed first-class classes [51]; and Lee et al.'s model of typed first-class classes [30]. Both works provide typed models of first-class classes, and they enable encodings of mixins [16] similar to those employed in dynamically-typed languages.

However, as far as we known no previous work supports statically-typed *first-class traits*. Traits [47] are an alternative to mixins, and other models of (multiple) inheritance. The key difference between traits and mixins lies on the treatment of conflicts when composing multiple traits/mixins. Mixins adopt an *implicit* resolution strategy for conflicts, where the compiler automatically picks one implementation in case of conflicts. For example, Scala uses the order of mixin composition to determine which implementation to pick in case of conflicts. Traits, on the other hand, employ an *explicit* resolution strategy, where the compositions with conflicts are rejected, and the conflicts are explicitly resolved by programmers.

Schärli et al. [47] make a good case for the advantages of the trait model. In particular, traits avoid bugs that could arise from accidental conflicts that were not detected by programmers. With the mixin model, such conflicts would be silently resolved, possibly resulting in unexpected runtime behaviour due to a wrong method implementation choice. In a setting with dynamic inheritance and first-class classes this problem is exacerbated by not knowing all components being composed statically, greatly increasing the possibility of accidental conflicts. From a modularity point-of-view, the trait model also ensures that composition is *commutative*, thus the order of composition is irrelevant and does not affect the semantics. Bracha [15] claims that "*The only modular solution is to treat the name*

collisions as errors...", strengthening the case for the use of a trait model of composition. Otherwise, if the semantics is affected by the order of composition, global knowledge about the full inheritance graph is required to determine which implementations are chosen. Schärli et al. discuss several other issues with mixins, which can be improved by traits. We refer to their paper for further details.

This paper presents the design of SEDEL: a polymorphic statically-typed (pure) language with *first-class traits*, supporting *dynamic inheritance* as well as conventional OO features such as *dynamic dispatching* and *abstract methods*. Traits pose additional challenges when compared to models with first-class classes or mixins, because method conflicts should be detected *statically*, even in the presence of features such as dynamic inheritance and composition and *parametric polymorphism*. To address the challenges of typing first-class traits and detecting conflicts statically, SEDEL adopts a polymorphic structural type system based on *disjoint polymorphism* [7]. The choice of structural typing is due to its simplicity, but we think similar ideas should also work in a nominal type system.

The main contribution of this paper is to show how to model source language constructs for first-class traits and dynamic inheritance, supporting standard OO features such as *dynamic dispatching* and *abstract methods*. Previous work on disjoint intersection types is aimed at core record calculi, and omits important features for practical OO languages, including (dynamic) inheritance, dynamic dispatching and abstract methods. Based on Cook and Palsberg's work on the denotational semantics for inheritance [19], we show how to design a source language that can be elaborated into Alpuim et al.'s F_i [7], a polymorphic calculus with records supporting disjoint polymorphism. SEDEL's elaboration into F_i is proved to be both type-safe and coherent. Coherence ensures that the semantics of SEDEL is unambiguous. In particular this property is useful to ensure that programs using traits are free of conflicts/ambiguities (even when the types of the object parts being composed are not fully statically know).

We illustrate the applicability of SEDEL with several example uses for first-class traits. Furthermore we conduct a case study that modularizes programming language interpreters using a highly modular form of Object Algebras [39] and VISITORs. In particular we show how SEDEL can easily compose multiple object algebras into a single object algebra. Such composition operation has previously been shown to be highly challenging in languages like Java or Scala [41, 44]. The previous state-of-the-art implementations for such operation require employing type-unsafe reflective techniques to simulate the features of first-class classes. Moreover conflicts are not statically detected. In contrast the approach in this paper is fully type-safe, convenient to use and conflicts are statically detected.

In summary the contributions of this paper are:

- **Typed first-class traits**: We present SEDEL: a statically-typed language design that supports first-class traits, dynamic inheritance, as well as standard high-level OO constructs such as dynamic dispatching and abstract methods.
- **Elaboration of first-class traits into disjoint intersection types/polymorphism:** We show how the semantics of SEDEL can be defined by elaboration into Alpuim et al.'s F_i [7]. The elaboration is inspired by the work of Cook and Palsberg [19] to model inheritance.
- **Implementation and modularization case study:** SEDEL is implemented and available.[1] To evaluate SEDEL we conduct a case study. The case study shows that support for composition of Object Algebras and VISITORs is greatly improved in SEDEL. Using

[1] The implementation, case study code and proofs are available at `https://goo.gl/uFrWkr`.

such improved design patterns we re-code the interpreters in Cook's undergraduate Programming Languages book [18] in a modular way in SEDEL.

2 Overview

This section aims at introducing first-class classes and traits, their possible uses and applications, as well as the typing challenges that arise from their use. We start by describing a hypothetical JavaScript library for text editing widgets, inspired and adapted from Racket's GUI toolkit [51]. The example is illustrative of typical uses of dynamic inheritance/composition, and also the typing challenges in the presence of first-class classes/traits. Without diving into technical details, we then give the corresponding typed version in SEDEL, and informally presents its salient features.

2.1 First-Class Classes in JavaScript

A class construct was officially added to JavaScript in the ECMAScript 2015 Language Specification [23]. One purpose of adding classes to JavaScript was to support a construct that is more familiar to programmers who come from mainstream class-based languages, such as Java or C++. However classes in JavaScript are *first-class* and support functionality not easily mimicked in statically-typed class-based languages.

Conventional Classes. Before diving into the more advanced features of JavaScript classes, we first review the more conventional class declarations supported in JavaScript as well as many other languages. Even for conventional classes there are some interesting points to note about JavaScript that will be important when we move into a typed setting. An example of a JavaScript class declaration is:

```
class Editor {
  onKey(key) { return "Pressing " + key; }
  doCut()    { return this.onKey("C-x") + " for cutting text"; }
  showHelp() { return "Version: " + this.version() + " Basic usage..."; }
};
```

This form of class definition is standard and very similar to declarations in class-based languages (for example Java). The `Editor` class defines three methods: `onKey` for handling key events, `doCut` for cutting text and `showHelp` for displaying help message. For the purpose of demonstration, we elide the actual implementation, and replace it with plain messages.

We wish to bring the readers' attention to two points in the above class. Firstly, note that the `doCut` method is defined in terms of the `onKey` method via the keyword **this**. In other words the call to `onKey` is enabled by the *self* reference and is *dynamically dispatched* (i.e., the particular implementation of `onKey` will only be determined when the class or subclass is instantiated). Secondly, notice that there is no definition of the `version` method in the class body, but such method is used inside the `showHelp` method. In a untyped language, such as JavaScript, using undefined methods is error prone – accidentally instantiating `Editor` and then calling `showHelp` will cause a runtime error! Statically-typed languages usually provide some means to protect us from this situation. For example, in Java, we would need an *abstract* `version` method, which effectively makes `Editor` an abstract class and prevents it from being instantiated. As we will see, SEDEL's treatment of abstract methods is quite different from mainstream languages. In fact, SEDEL has a unified (typing) mechanism for dealing with both dynamic dispatch and abstract methods. We will describe SEDEL's mechanism for dealing with both features and justify our design in Section 3.

First-Class Classes and Class Expressions. Another way to define a class in JavaScript is via a *class expression*. This is where the class model in JavaScript is very different from the traditional class model found in many mainstream OO languages, such as Java, where classes are second-class (static) entities. JavaScript embraces a dynamic class model that treats classes as *first-class* expressions: a function can take classes as arguments, or return them as a result. First-class classes enable programmers to abstract over patterns in the class hierarchy and to experiment with new forms of OOP such as mixins and traits. In particular, mixins become programmer-defined constructs. We illustrate this by presenting a simple mixin that adds spell checking to an editor:

```
const spellMixin = Base => {
  return class extends Base {
    check()    { return super.onKey("C-c") + " for spell checking"; }
    onKey(key) { return "Process " + key + " on spell editor"; }
  }
};
```

In JavaScript, a mixin is simply a function with a superclass as input and a subclass extending that superclass as an output. Concretely, `spellMixin` adds a method `check` for spell checking. It also provides a method `onKey`. The function `spellMixin` shows the typical use of what we call *dynamic inheritance*. Note that `Base`, which is supposed to be a superclass being inherited, is *parameterized*. Therefore `spellMixin` can be applied to any base class at *runtime*. This is impossible to do, in a type-safe way, in conventional statically-typed class-based languages like Java or C++.[2]

It is noteworthy that not all applications of `spellMixin` to base classes are successful. Notice the use of the **super** keyword in the `check` method. If the base class does not implement the `onKey` method, then mixin application fails with a runtime error. In a typed setting, a type system must express this requirement (i.e., the presence of the `onKey` method) on the (statically unknown) base class that is being inherited.

We invite the readers to pause for a while and think about what the type of `spellMixin` would look like. Clearly our type system should be flexible enough to express this kind of dynamic pattern of composition in order to accommodate mixins (or traits), but also not too lenient to allow any composition.

Mixin Composition and Conflicts. The real power of mixins is that `spellMixin`'s functionality is not tied to a particular class hierarchy and is composable with other features. For example, we can define another mixin that adds simple modal editing – as in Vim – to an arbitrary editor:

```
const modalMixin = Base => {
  return class extends Base {
    constructor() {
      super();
      this.mode = "command";
    }
    toggleMode() { return "toggle succeeded"; }
    onKey(key)   { return "Process " + key + " on modal editor"; }
  };
};
```

[2] With C++ templates, it is possible to implement a so-called mixin pattern [49], which enables extending a parameterized class. However C++ templates defer type-checking until instantiation, and such pattern still does not allow selection of the base class at runtime (only at up to class instantiation time).

`modalMixin` adds a `mode` field that controls which keybindings are active, initially set to the command mode, and a method `toggleMode` that is used to switch between modes. It also provides a method `onKey`.

Now we can compose `spellMixin` with `modalMixin` to produce a combination of functionality, mimicking some form of multiple inheritance:

```
class IDEEditor extends modalMixin(spellMixin(Editor)) {
  version() { return 0.2; }
}
```

The class `IDEEditor` extends the base class `Editor` with modal editing and spell checking capabilities. It also defines the missing `version` method.

At first glance, `IDEEditor` looks quite fine, but it has a subtle issue. Recall that two mixins `modalMixin` and `spellMixin` both provide a method `onKey`, and the `Editor` class also defines an `onKey` method of its own. Now we have a name clash. A question arises as to which one gets picked inside the `IDEEditor` class. A typical mixin model resolves this issue by looking at the order of mixin applications. Mixins appearing later in the order overrides *all* the identically named methods of earlier mixins. So in our case, `onKey` in `modalMixin` gets picked. If we change the order of application to `spellMixin(modalMixin(Editor))`, then `onKey` in `spellMixin` is inherited.

Problem of Mixin Composition. From the above discussion, we can see that mixin are composed linearly: all the mixins used by a class must be applied one at a time. However, when we wish to resolve conflicts by selecting features from different mixins, we may not be able to find a suitable order. For example, when we compose the two mixins to make the class `IDEEditor`, we can choose which of them comes first, but in either order, `IDEEditor` cannot access to the `onKey` method in the `Editor` class.

Trait Model. Because of the total ordering and the limited means for resolving conflicts imposed by the mixin model, researchers have proposed a simple compositional model called traits [47, 21]. Traits are lightweight entities and serve as the primitive units of code reuse. Among others, the key difference from mixins is that the order of trait composition is irrelevant, and conflicting methods must be resolved *explicitly*. This gives programmers fine-grained control, when conflicts arise, of selecting desired features from different components. Thus we believe traits are a better model for multiple inheritance in statically-typed OO languages, and in SEDEL we realize this vision by giving traits a first-class status in the language, achieving more expressive power compared with traditional (second-class) traits.

Summary of Typing Challenges. From our previous discussion, we can identify the following typing challenges for a type system to accommodate the programming patterns (first-class classes/mixins) we have just seen in a typed setting:

- How to account for, in a typed way, abstract methods and dynamic dispatch.
- What are the types of first-class classes or mixins.
- How to type dynamic inheritance.
- How to express constraints on method presence and absence (the use of **super** clearly demands that).
- In the presence of first-class traits, how to detect conflicts statically, even when the traits involved are not statically known.

SEDEL elegantly solves the above challenges in a unified way, as we will see next.

2.2 A Glance at Typed First-Class Traits in SEDEL

We now rewrite the above library in SEDEL, but this time with types. The resulting code has the same functionality as the dynamic version, but is statically typed. All code snippets in this and later sections are runnable in our prototype implementation. Before proceeding, we ask the readers to bear in mind that in this section we are not using traits in the most canonical way, i.e., we use traits as if they are classes (but with built-in conflict detection). This is because we are trying to stay as close as possible to the structure of the JavaScript version for ease of comparison. In Section 3 we will remedy this to make better use of traits.

Simple Traits. Below is a simple trait `editor`, which corresponds to the JavaScript class `Editor`. The `editor` trait defines the same set of methods: `on_key`, `do_cut` and `show_help`:

```
trait editor [self : Editor & Version] => {
  on_key(key : String) = "Pressing " ++ key;
  do_cut = self.on_key "C-x" ++ " for cutting text";
  show_help = "Version: " ++ self.version ++ " Basic usage..."
};
```

The first thing to notice is that SEDEL uses a syntax (similar to Scala's self type annotations [36]) where we can give a type annotation to the `self` reference. In the type of `self` we use `&` construct to create intersection types. `Editor` and `Version` are two record types:

```
type Editor  = {on_key : String → String, do_cut : String, show_help : String};
type Version = {version : String};
```

For the sake of conciseness, SEDEL uses **type** aliases to abbreviate types.

Self-Types Encode Abstract Methods. Recall that in the JavaScript class `Editor`, the `version` method is undefined, but is used inside `showHelp`. How can we express this in the typed setting, if not with an abstract method? In SEDEL, self-types play the role of trait requirements. As the first approximation, we can justify the use of `self.version` by noticing that (part of) the type of `self` (i.e., `Version`) contains the declaration of `version`. An interesting aspect of SEDEL's trait model is that there is no need for abstract methods. Instead, abstract methods can be simulated as requirements of a trait. Later, when the trait is composed with other traits, *all* requirements on the self-types must be satisfied and one of the traits in the composition must provide an implementation of the method `version`.

As in the JavaScript version, the `on_key` method is invoked on `self` in the body of `do_cut`. This is allowed as (part of) the type of `self` (i.e., `Editor`) contains the signature of `on_key`. Comparing `editor` to the JavaScript class `Editor`, almost everything stays the same, except that we now have the typed version. As a side note, since SEDEL is currently a pure functional OO language, there is no difference between fields and methods, so we can omit empty arguments and parameter parentheses.

First-Class Traits and Trait Expressions. SEDEL treats traits as first-class expressions, putting them in the same syntactic category as objects, functions, and other primitive forms. To illustrate this, we give the SEDEL version of `spellMixin`:

```
type Spelling = {check : String};
type OnKey    = {on_key : String → String};

spell_mixin [A * Spelling & OnKey] (base : Trait[Editor & Version, Editor & A]) =
  trait [self : Editor & Version] inherits base => {
    override on_key(key : String) = "Process " ++ key ++ " on spell editor";
```

```
    check = super.on_key "C-c"  ++ " for spelling check"
  };
```

This looks daunting at first, but `spell_mixin` has almost the same structure as its JavaScript cousin `spellMixin`, albeit with some type annotations. In SEDEL, we use capital letters (`A`, `B`, ...) to denote type variables, and trait expressions `trait [self : ...] inherits ... => {...}` to create first-class traits. Trait expressions have trait types of the form **Trait** `[T1, T2]` where `T1` and `T2` denote trait requirements and functionality respectively. We will explain trait types in Section 3. Despite the structural similarities, there are several significant features that are unique to SEDEL (e.g., the disjointness operator `*`). We discuss these in the following.

Disjoint Polymorphism and Conflict Detection. SEDEL uses a type system based on *disjoint intersection types* [40] and *disjoint polymorphism* [7]. Disjoint intersections empower SEDEL to detect conflicts statically when trying to compose two traits with identically named features. For example composing two traits `a` and `b` that both provide `foo` gives a type error (the overloaded `&` operator denotes trait composition):

```
trait a => { foo = 1 };
trait b => { foo = 2 };
trait c inherits a & b => {}; -- type error!
```

Disjoint polymorphism, as a more advanced mechanism, allows detecting conflicts even in the presence of polymorphism – for example when a trait is parameterized and its full set of methods is not statically known. As can be seen, `spell_mixin` is actually a polymorphic function. Unlike ordinary parametric polymorphism, in SEDEL, a type variable can also have a disjointness constraint. For instance, `A * Spelling & OnKey` means that `A` can be instantiated to any type as long as it *does not* contain `check` and `on_key`. To mimic mixins, the argument `base`, which is supposed to be some trait, serves as the "base" trait that is being inherited. Notice that the type variable `A` appears in the type of `base`, which essentially states that `base` is a trait that contains at least those methods specified by `Editor`, and possibly more (which we do not know statically). Also note that leaving out the **override** keyword will result in a type error. The type system is forcing us to be very specific as to what is the intention of the `on_key` method because it sees the same method is also declared in `base`, and blindly inheriting `base` will definitely cause a method conflict. As a final note, the use of **super** inside `check` is allowed because the "super" trait `base` implements `on_key`, as can be seen from its type.

Dynamic Inheritance. Disjoint polymorphism enables us to correctly type dynamic inheritance: `spell_mixin` is able to take any trait that conforms with its assigned type, equips it with the `check` method and overrides its old `on_key` method. As a side note, the use of disjoint polymorphism is essential to correctly model the mixin semantics. From the type we know `base` has some features specified by `Editor`, plus something more denoted by `A`. By inheriting `base`, we are guaranteed that the result trait will have everything that is already contained in `base`, plus more features. This is in some sense similar to row polymorphism [55] in that the result trait is prohibited from forgetting methods from the argument trait. As we will discuss in Section 6, disjoint polymorphism is more expressive than row polymorphism.

Typing Mixin Composition. Next we give the typed version of `modalMixin` as follows:

```
type ModalEdit = {mode : String, toggle_mode : String};

modal_mixin [A * ModalEdit & OnKey] (base : Trait[Editor & Version, Editor & A]) =
```

```
trait [self : Editor & Version] inherits base => {
  override on_key(key : String) = "Process " ++ key ++ " on modal editor";
  mode = "command";
  toggle_mode = "toggle succeeded"
};
```

Now the definition of `modal_mixin` should be self-explanatory. Finally we can apply both "mixins" one by one to `editor` to create a concrete editor:

```
type IDEEditor = Editor & Version & Spelling & ModalEdit;

trait ide_editor [self : IDEEditor]
  inherits modal_mixin Spelling (spell_mixin ⊤ editor) => { version = "0.2" };
```

As with the JavaScript version, we need to fill in the missing `version` method. It is easy to verify that the `on_key` method in `modal_mixin` is inherited. Compared with the untyped version, here this behaviour is reasonable because in each mixin we specifically tags the `on_key` method to be an overriding method. Let us take a close look at the mixin applications. Since SEDEL is currently explicitly typed, we need to provide concrete types when using `modal_mixin` and `spell_mixin`. In the inner application (`spell_mixin ⊤ editor`), we use the top type ⊤ to instantiate `A` because the `editor` trait provides exactly those method specified by `Editor` and nothing more (hence ⊤). In the outer application, we use `Spelling` to instantiate `A`. This is where implicit conflict resolution of mixins happens. We know the result of the inner application actually forms a trait that provides both `check` and `on_key`, but the disjointness constraint of `A` requires the absence of `on_key`, thus we cannot instantiate `A` to `Spelling & OnKey` for example when applying `modal_mixin`. Therefore the outer application effectively excludes `on_key` from `spell_mixin`. In summary, the order of mixin applications is reflected by the order of function applications, and conflict resolution code is implicitly embedded. Of course changing the mixin application order to `spell_mixin ModalEdit (modal_mixin ⊤ editor)` gives the expected behaviour.

Admittedly the typed version is unnecessarily complicated as we were mimicking mixins by functions over traits. The final editor `ide_editor` suffers from the same problem as the class `IDEEditor`, since there is no obvious way to access the `on_key` method in the `editor` trait.[3] Section 3 makes better use of traits to simplify the editor code.

3 Typed First-Class Traits

In Section 2 we have seen some examples of first-class traits at work in SEDEL. In this section we give a detailed account of SEDEL's support for typed first-class traits, to complement what has been presented so far. In doing so, we simplify the examples in Section 2 to make better use of traits. Section 4 presents the formal type system of first-class traits.

3.1 Traits in SEDEL

SEDEL supports a simple, yet expressive form of traits [47]. Traits provide a simple mechanism for find-grained code reuse, which can be regarded as a disciplined form of multiple inheritance. A trait is similar to a mixin in that it encapsulates a collection of related methods to be added to a class. The practical difference between traits and mixins is the way conflicting features that typically arise in multiple inheritance are dealt with. Instead of automatically resolved by scoping rules, conflicts are, in SEDEL, detected by the type system, and explicitly resolved

[3] In fact, as we will see in Section 3, we can still access `on_key` in `editor` by the forwarding operator.

by the programmer. Compared with traditional trait models, there are three interesting points about SEDEL's traits: (1) they are *statically typed*; (2) they are *first-class* values; and (3) they support *dynamic inheritance*. The support for such combination of features is one of the key novelties of SEDEL. Another minor difference from traditional traits (e.g., in Scala) is that, due to the use of structural types, a trait name is not a type.

3.2 Two Roles of Traits in SEDEL

Traits as Templates for Creating Objects. An obvious difference between traits in SEDEL and many other models of traits [47, 25, 37] is that they directly serve as templates for objects. In many other trait models, traits are complemented by classes, which take the responsibility for object creation. In particular, most models of traits do not allow constructors for traits. However, a trait in SEDEL has a single constructor of the same name. Take our last trait `ide_editor` in Section 2 for example:

```
a_editor1 = new[IDEEditor] ide_editor;
```

As with conventional OO languages, the keyword **new** is used to create an object. A difference to other OO languages is that the keyword **new** also specifies the intended type of the object. We instantiate the `ide_editor` trait and create an object `a_editor1` of type `IDEEditor`. As we will see in Section 3.4, constructors with parameters can also be expressed.

It is tempting to try to instantiate the `editor` trait such as `new[Editor] editor`. However this results in a type error, because as we discussed, `editor` has no definition of `version`, and blindly instantiating it would cause runtime error. This behaviour is on a par with Java's abstract classes – traits with undefined methods cannot be instantiated on their own.

Traits as Units of Code Reuse. The traditional role of traits is to serve as units of code reuse. SEDEL's traits can have this role as well. Our `spell_mixin` function in Section 2 is more complicated than it should be. This is because we were mimicking classes as traits, and mixins as functions over traits. Instead, traits already provide a mechanism of code reuse. To illustrate this, we simplify `spell_mixin` as follows:

```
trait spell [self : OnKey] => {
  on_key(key : String) = "Process " ++ key ++ " on spell editor";
  check = self.on_key "C-c"  ++ " for spell checking"
};
```

This is much cleaner. The trait `spell` adds a method `check`. It also defines a method `on_key`. A key difference with `spell_mixin` is that `on_key` is invoked on the `self` parameter instead of **super**. Note that this does not necessarily mean `check` will call `on_key` defined in the same trait. As we will see, the actual behaviour entirely depends on how we compose `spell` with other traits. One minor difference is that we do not need to tag `on_key` with the **override** keyword, because `spell` stands as a standalone entity. Another interesting point is that the self-type `OnKey` is not the same as that of the trait body, which also contains the `check` method. In SEDEL, self-types of traits are known as trait *requirements*.

Classes and/or Traits In the literature on traits [21, 47], the aforementioned two roles are considered as competing. One reason of the two roles conflicting in class-based languages is because a class must adopt a fixed position in the class hierarchy and therefore it can be difficult to reuse and resolve conflicts, whereas in SEDEL, a trait is a standalone entity and is not tied to any particular hierarchy. Therefore we can view our traits either as generators of instances, or units of reuse. Another important reason why our model can do just with

traits is because we have a pure language. Mutable state can often only appear in classes in imperative models of traits, which is a good reason for having both classes and traits.

3.3 Trait Types and Trait Requirements

Object Types and Trait Types. SEDEL adopts a relatively standard foundational model of object-oriented constructs [30] where objects are encoded as records with a structural type. This is why the type of the object `a_editor1` is the record type `IDEEditor`. In SEDEL, an object type is different from a trait type. A trait type is specified with the keyword **Trait**. For example, the type of the `spell` trait is **Trait** `[OnKey, OnKey & Spelling]`.

Trait Requirements and Functionality. In general, a trait type **Trait** `[T1, T2]` specifies both the *requirements* `T1` and the *functionality* `T2` of a trait. The requirements of a trait denote the types/methods that the trait needs to support for defining the functionality it provides. Both are reflected in the trait type. For example, `spell` has type **Trait** `[OnKey, OnKey & Spelling]`, which means that `spell` requires some implementation of the `on_key` method, and it provides implementations for the `on_key` and `check` methods. When a trait has no requirements, the absence of a requirement is denoted by using the top type ($\top$). A simplified sugar **Trait** `[T]` is used to denote a trait without requirements, but providing functionality `T`.

Trait Requirements as Abstract Methods. Let us go back to our very first trait `editor`. Note how in `editor` the type of the `self` parameter is `Editor & Version`, where `Version` contains a declaration of the `version` method that is needed for the definition of `show_help`. Note also that the trait itself does not actually contain a `version` definition. In many other OO models a similar program could be achieved by having an *abstract* definition of `version`. In SEDEL there are no abstract definitions (methods or fields), but a similar result can be achieved via trait requirements. Requirements of a trait are met at the object creation point. For example, as we mentioned before, the `editor` trait alone cannot be instantiated since it lacks `version`. However, when it is composed with a trait that provides `version`, the composition can be instantiated, as shown below:

```
trait foo => { version = "0.2" };
bar = new[Editor & Version] foo & editor;
```

SEDEL uses a syntax where the self parameter can be explicitly named (not necessarily named `self`) with a type annotation. When the self parameter is omitted (for example in the `foo` trait above), its type defaults to $\top$. This is different from typical OO languages, where the default type of the self parameter is the same as the class being defined.

Intersection Types Model Subtyping. `IDEEditor` is defined as an intersection type (`Editor & Version & Spelling & ModalEdit`). Intersection types [20, 43] have been woven into many modern languages these days. A notable example is Scala, which makes fundamental use of intersection types to express a class/trait that extends multiple other traits. An intersection type such as `T1 & T2` contains exactly those values which can be used as values of type `T1` and of type `T2`, and as such, `T1 & T2` immediately introduces a subtyping relation between itself and its two constituent types `T1` and `T2`. Unsurprisingly, `IDEEditor` is a subtype of `Editor`.

3.4 Traits with Parameters and First-Class Traits

So far our uses of traits involve no parameters. Instead of inventing another trait syntax with parameters, a trait with parameters is just a function that produces a trait expression,

since functions already have parameters of their own. This is one benefit of having first-class traits in terms of language economy. To illustrate, let us simplify `modal_mixin` in a similar way as in `spell_mixin`:

```
modal (init_mode : String) = trait => {
  on_key(key : String) = "Process " ++ key ++ " on modal editor";
  mode = init_mode;
  toggle_mode = "toggle succeeded"
};
```

The first thing to notice is that `modal` is a function with one argument, and returns a trait expression, which essentially makes `modal` a trait with one parameter. Now it is easy to see that a trait declaration `trait name [self : ...] => {...}` is just syntactic sugar for function definition `name = trait [self : ...] => {...}`. The body of the `modal` trait is straightforward. We initialize the `mode` field to `init_mode`. The `modal` trait also comes with a constructor with one parameter, so we can do `new[ModalEdit] (modal "insert")` for example.

3.5 Detecting and Resolving Conflicts in Trait Composition

A common problem in multiple inheritance is how to detect and/or resolve conflicts. For example, when inheriting from two traits that have the same field, then it is unclear which implementation to choose. There are various approaches to dealing with conflicts. The trait-based approach requires conflicts to be resolved at the level of the composition, otherwise the program is rejected by the type system. SEDEL provides a means to help resolve conflicts.

We start by assembling all the traits defined in this section to create the final editor with the same functionality as `ide_editor` in Section 2. Our first try is as follows:

```
ide_editor (init_mode : String) = trait [self : IDEEditor]
  -- conflict
  inherits editor & spell & modal init_mode => { version = "0.2" };
```

Unfortunately the above trait gets rejected by SEDEL because `editor`, `spell` and `modal` all define an `on_key` method. Recall that in Section 2, when we use a mixin-style composition, the conflict resolution code has been hardwired in the definition. However, in a trait-style composition, this is not the case: conflicts must be resolved *explicitly*. The above definition is ill-typed precisely because there is a conflicting method `on_key`, thus violating the disjointness conditions of disjoint intersection types.

Resolving Conflicts. To resolve the conflict, we need to explicitly state which `on_key` gets to stay. SEDEL provides such a means, the so-called *exclusion* operator (denoted by \), which allows one to exclude a field/method from a given trait. The following matches the behaviour in Section 2 where `on_key` in the `modal` trait is selected:

```
ide_editor (init_mode : String) = trait [self : IDEEditor]
  inherits editor \ {on_key : String → String} &
           spell  \ {on_key : String → String} & modal init_mode =>
    { version = "0.2" };
```

Now the above code type checks. We can also select `on_key` in the `spell` trait as easily:

```
ide_editor2 (init_mode : String) = trait [self : IDEEditor]
  inherits editor \ {on_key : String → String} &
           spell  & (modal init_mode) \ {on_key : String → String} =>
    { version = "0.2" };
```

In Section 2 we mentioned that in the mixin style, it is impossible to select `on_key` in the `editor` trait, but this is not a problem here:

```
ide_editor3 (init_mode : String) = trait [self : IDEEditor]
  inherits editor & spell \ {on_key : String → String} &
           (modal init_mode) \ {on_key : String → String} =>
    { version = "0.2" };
```

The Forwarding Operator. Another operator that SEDEL provides is the so-called *forwarding* operator, which can be useful when we want to access some method that has been explicitly excluded in the **inherits** clause. This is a common scenario in diamond inheritance, where **super** is not enough. Below we show a variant of `ide_editor`:

```
ide_editor4 (init_mode : String) = trait [self : IDEEditor]
  inherits editor \ {on_key : String → String} &
           spell  \ {on_key : String → String} &
           modal init_mode  => {
    version = "0.2";
    override on_key(key : String) =
      super.on_key key ++ " and " ++ (spell ^ self).on_key key
};
```

Notice that `on_key` in `spell` has been excluded. However, we can still access it by using the forwarding operator as in `spell ^ self`, which gives full access to all the methods in `spell`. Also note that using **super** only gives us access to `on_key` in the `modal` trait. To see `ide_editor4` in action, we create a small test:

```
a_editor2 = new[IDEEditor] (ide_editor4 "command");
main = a_editor2.do_cut
-- "Process C-x on modal editor and Process C-x on spell editor for cutting text"
```

3.6 Disjoint Polymorphism and Dynamic Composition

SEDEL supports disjoint polymorphism. The combination of disjoint polymorphism and first-class traits enables the highly modular code where traits with *statically unknown* types can be instantiated and composed in a type-safe way! The following is illustrative of this:

```
merge A [B * A] (x : Trait[A]) (y : Trait[B]) = new[A & B] x & y;
```

The `merge` function takes two traits `x` and `y` of some arbitrary types `A` and `B`, composes them, and instantiates an object with the resulting composed trait. Clearly such composition cannot always work if `A` and `B` can have conflicts. However, `merge` has a constraint `B * A` that ensures that whatever types are used to instantiate `A` and `B` they must be disjoint. Thus, under the assumption that `A` and `B` are disjoint the code type-checks. We want to emphasize that row polymorphism is unable to express this kind of disjointness of two polymorphic types, thus systems using row polymorphism is unable to define the `merge` function, which plays an essential role in Section 5.

4 Formalizing Typed First-Class Traits

This section presents the syntax and semantics of SEDEL. In particular, we show how to elaborate high-level source language constructs (self-references, abstract methods, first-class traits, dynamic inheritance, etc) in SEDEL to F_i [7], a pure record calculus with disjoint polymorphism. The treatment of the self-reference and dynamic dispatching is inspired by Cook and Palsberg's work on the denotational semantics for inheritance [19]. We then prove the elaboration is type safe, i.e., well-typed SEDEL expressions are translated to well-typed F_i terms. Finally we show that SEDEL is coherent. Full proofs can be found in the appendix.

$$
\begin{array}{llll}
\text{Types} & A, B, C & ::= & \top \mid \mathsf{Int} \mid A \to B \mid A \,\&\, B \mid \{l : A\} \mid \alpha \mid \forall\,(\alpha * A).\, B \mid \mathbf{Trait}\,[A, B] \\
\text{Expressions} & E & ::= & \top \mid i \mid x \mid \lambda x.\, E \mid E_1\; E_2 \mid \Lambda\,(\alpha * A).\, E \mid E\; A \mid E_1\;,\; E_2 \mid E : A \\
& & \mid & \{l = E\} \mid E.\, l \mid \mathbf{letrec}\, x : A = E_1\, \mathbf{in}\, E_2 \mid \mathbf{new}\,[A](\,\overline{E_i}^{\,i}\,) \mid E_1\,\hat{}\,E_2 \\
& & \mid & \mathbf{trait}\,[\mathsf{self} : B]\,\mathbf{inherits}\,\overline{E_i}^{\,i}\,\{\,\overline{l_j = E'_j}^{\,j}\,\} : A \\
\text{Contexts} & \Gamma & ::= & \bullet \mid \Gamma, x : A \mid \Gamma, \alpha * A
\end{array}
$$

$$
\begin{array}{lll}
\text{Record types} & \{l_1 : A_1, \ldots, l_n : A_n\} & := \quad \{l_1 : A_1\} \,\&\, \ldots \,\&\, \{l_n : A_n\} \\
\text{Records} & \{l_1 = E_1, \ldots, l_n = E_n\} & := \quad \{l_1 = E_1\}\;,\;\ldots\;,\;\{l_n = E_n\}
\end{array}
$$

Figure 1 SEDEL core syntax and syntactic abbreviations

4.1 Syntax

The core syntax of SEDEL is shown in Fig. 1, with trait related constructs highlighted. For brevity of the meta-theoretic study, we do not consider definitions, which can be added in standard ways.

Types. Metavariables A, B, C range over types. Types include a top type $\top$, type of integers Int, function types $A \to B$, intersection types $A \,\&\, B$, singleton record types $\{l : A\}$, type variables α and disjoint (universal) quantification $\forall\,(\alpha * A).\, B$. The main novelty is the type of first-class traits $\mathbf{Trait}\,[A, B]$, which expresses the requirement A and the functionality B. We will use $[A/\alpha]B$ to denote capture-avoiding substitution of A for α inside B.

Expressions. Metavariable E ranges over expressions. We start with constructs required to encode objects based on records: term variables x, lambda abstractions $\lambda x.\, E$, function applications $E_1\; E_2$, singleton records $\{l = E\}$, record projections $E.\, l$, recursive let bindings $\mathbf{letrec}\, x : A = E_1\, \mathbf{in}\, E_2$, disjoint type abstraction $\Lambda\,(\alpha * A).\, E$ and type application $E\; A$. The calculus also supports a merge construct $E_1\;,\; E_2$ for creating values of intersection types and annotated expressions $E : A$. We also include a canonical top value $\top$ and integer literals i.

First-class traits and trait expressions. The central construct of SEDEL is the trait expression $\mathbf{trait}\,[\mathsf{self} : B]\,\mathbf{inherits}\,\overline{E_i}^{\,i}\,\{\,\overline{l_j = E'_j}^{\,j}\,\} : A$, which specifies a (possibly empty) list of trait expressions $\overline{E_i}$ in the **inherits** clause, an explicit self reference (with type annotation B), and a set of methods $\{\overline{l_j = E'_j}\}$. Intuitively this trait expression has type $\mathbf{Trait}\,[B, A]$. Unlike the conventional trait model, a trait expression denotes a first-class value: it may occur anywhere where an expression is expected. Trait instantiation expressions $\mathbf{new}\,[A](\,\overline{E_i}^{\,i}\,)$ instantiate a composition of trait expressions $\overline{E_i}$ to create an object of type A. Finally $E_1\,\hat{}\,E_2$ is the forwarding expression, where E_1 should be some trait.

Abbreviations. For ease of programming, multiple-field record types are merely syntactic sugar for intersections of single-field record types. Similarly, multi-field record expressions are syntactic sugar for merges of single-field records.

4.2 Semantics

Subtyping and Well-formedness. Figure 2 shows the most relevant subtyping and well-formedness rules for SEDEL. Omitted rules are standard and can be found in previous work [7]. The subtyping rule for trait types (rule Sub-trait) resembles the one for function types

$\boxed{A <: B}$ *(Subtyping)*

$$\frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \textsc{Sub-arr} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{\mathbf{Trait}\,[A_1, A_2] <: \mathbf{Trait}\,[B_1, B_2]}\ \textsc{Sub-trait}$$

$\boxed{\Gamma \vdash A}$ *(Well formedness)*

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \Gamma \vdash A * B}{\Gamma \vdash A \,\&\, B}\ \textsc{WF-and} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash \mathbf{Trait}\,[A, B]}\ \textsc{WF-trait}$$

Figure 2 Subtyping and well-formedness of SEDEL (excerpt)

(rule Sub-arr) in that it is contravariant on the first type A and covariant on the second type B. The well-formedness rule for trait types is straightforward.

Disjointness. Figure 3 shows the disjointness judgment $\Gamma \vdash A * B$, which is used for example in rule WF-and. The disjointness checking is the underlying mechanism of conflict detection. We naturally extend the disjointness rules in F_i to cover trait types. We refer to their paper [7] for further explanation. Here we discuss the rules related with traits. Rule D-trait says that as long as the functionalities that two traits provide are disjoint, the two trait types are disjoint. Rules D-traitArr1 and D-traitArr2 deal with situations where one of the two types is a function type. At first glance, these two look strange because a trait type is *different* from a function type, and they ought to be disjoint as an axiom. The reason is that SEDEL has an elaboration semantics, and as we will see, trait types are translated to function types. In order to ensure the elaboration is type-safe, we have to have special treatment for trait and function types. In principle, if SEDEL has its own semantics, then trait types are always disjoint to function types. The axiom rules of the form $A *_{ax} B$ take care of two types with different language constructs.

Typing Traits. The typing rules of trait related constructs are shown in Fig. 4. The full set of rules can be found in the appendix. The reader is advised to ignore the highlighted parts for now. SEDEL employs two modes: the inference mode ($\Rightarrow$) and the checking mode ($\Leftarrow$). The inference judgment $\Gamma \vdash E \Rightarrow A$ says that we can synthesize a type A for expression E in the context Γ. The checking judgment $\Gamma \vdash E \Leftarrow A$ checks E against A in the context Γ. One representative of inference rules is

$$\frac{\Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Rightarrow B \rightsquigarrow e_2 \qquad \Gamma \vdash A * B}{\Gamma \vdash E_1 \,,, E_2 \Rightarrow A \,\&\, B \rightsquigarrow e_1 \,,, e_2}\ \textsc{Inf-merge}$$

which says that a merge of two expressions is valid only if their types are disjoint. This is the underlying mechanism for conflict detection. One representative of checking rules is

$$\frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \qquad \Gamma \vdash B}{\Gamma \vdash E \Leftarrow B}\ \textsc{Chk-sub}$$

where subtyping is used to coerce expressions of one type to another.

To type-check a trait (rule Inf-trait) we first type-check if its inherited traits $\overline{E_i}$ are valid traits. Note that each trait E_i can possibly refer to self. Methods must all be well-typed in the usual sense. Apart from these, we have several side-conditions to make sure traits

$\boxed{\Gamma \vdash A * B}$ *(Disjointness)*

$$\text{D-TOP}\ \frac{}{\Gamma \vdash \top * A} \qquad \text{D-TOPSYM}\ \frac{}{\Gamma \vdash A * \top}$$

$$\text{D-VAR}\ \frac{\alpha * A \in \Gamma \qquad A <: B}{\Gamma \vdash \alpha * B} \qquad \text{D-VARSYM}\ \frac{\alpha * A \in \Gamma \qquad A <: B}{\Gamma \vdash B * \alpha}$$

$$\text{D-FORALL}\ \frac{\Gamma, \alpha * A_1 \,\&\, A_2 \vdash B * C}{\Gamma \vdash \forall\,(\alpha * A_1).\, B * \forall\,(\alpha * A_2).\, C} \qquad \text{D-REC}\ \frac{\Gamma \vdash A * B}{\Gamma \vdash \{l : A\} * \{l : B\}} \qquad \text{D-RECN}\ \frac{l_1 \neq l_2}{\Gamma \vdash \{l_1 : A\} * \{l_2 : B\}}$$

$$\text{D-ARROW}\ \frac{\Gamma \vdash A_2 * B_2}{\Gamma \vdash A_1 \to A_2 * B_1 \to B_2} \qquad \text{D-ANDL}\ \frac{\Gamma \vdash A_1 * B \qquad \Gamma \vdash A_2 * B}{\Gamma \vdash A_1 \,\&\, A_2 * B} \qquad \text{D-ANDR}\ \frac{\Gamma \vdash A * B_1 \qquad \Gamma \vdash A * B_2}{\Gamma \vdash A * B_1 \,\&\, B_2}$$

$$\text{D-TRAIT}\ \frac{\Gamma \vdash A_2 * B_2}{\Gamma \vdash \mathbf{Trait}\,[A_1, A_2] * \mathbf{Trait}\,[B_1, B_2]} \qquad \text{D-TRAITARR1}\ \frac{\Gamma \vdash A_2 * B_2}{\Gamma \vdash \mathbf{Trait}\,[A_1, A_2] * B_1 \to B_2}$$

$$\text{D-TRAITARR2}\ \frac{\Gamma \vdash A_2 * B_2}{\Gamma \vdash A_1 \to A_2 * \mathbf{Trait}\,[B_1, B_2]} \qquad \text{D-AX}\ \frac{A *_{ax} B}{\Gamma \vdash A * B}$$

$\boxed{A *_{ax} B}$ *(Disjointness axiom)*

$$\text{DAX-INTTRAIT}\ \frac{}{\mathsf{Int} *_{ax} \mathbf{Trait}\,[A_1, A_2]} \qquad \text{DAX-TRAITFORALL}\ \frac{}{\mathbf{Trait}\,[A_1, A_2] *_{ax} \forall\,(\alpha * B_1).\, B_2} \qquad \text{DAX-TRAITREC}\ \frac{}{\mathbf{Trait}\,[A_1, A_2] *_{ax} \{l : B\}}$$

Figure 3 Disjointness rules of SEDEL (excerpt)

are well-behaved. The well-formedness judgment $\Gamma \vdash C_1 \,\&\, .. \,\&\, C_n \,\&\, C$ ensures that we do not have conflicting methods (in inherited traits and the body). The subtyping judgments $\overline{B <: B_i}$ ensure that the self parameter satisfies the requirements imposed by each inherited trait. Finally the subtyping judgment $C_1 \,\&\, .. \,\&\, C_n \,\&\, C <: A$ sanity-checks that the assigned type A is compatible.

Trait instantiation (rule INF-NEW) requires that each instantiated trait is valid. There are also several side-conditions, which serve the same purposes as in rule INF-TRAIT. Rule INF-FORWARD says that the first operand E_1 of the forwarding operator must be a trait. Moreover, the type of the second operand E_2 must satisfy the requirement of E_1.

Treatments of Exclusion, Super and Override. One may have noticed that in Fig. 1 we did not include the exclusion operator in the core SEDEL syntax, neither do **super** and **override** appear. The reason is that in principle all uses of the exclusion operator can be replaced by type annotations. For example to exclude a `bar` field from `{foo = a, bar = b, baz = c}`, all we need is to annotate the record with type `{foo : A, baz : C}` (suppose `a` has type `A`, etc). By rule CHK-SUB, the resulting record is guaranteed to contain no `bar` field. In the same vein, the use of **override** can be explained using the exclusion operator. The **super** keyword is internally a variable pointing to the **inherits** clause (its typing rule is similar to rule INF-TRAIT and can be found in the appendix). We omit all of these features in the meta-theoretic study in order to focus our attention on the essence of first-class traits.

$\boxed{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}$ *(Infer)*

$$\frac{\overline{\Gamma, \mathsf{self} : B \vdash E_i \Rightarrow \mathbf{Trait}\,[B_i, C_i] \rightsquigarrow e_i}^{\,i \in 1..n} \quad \Gamma, \mathsf{self} : B \vdash \{\, \overline{l_j = E'_j}^{\,j \in 1..m} \,\} \Rightarrow C \rightsquigarrow e \quad \overline{B <: B_i}^{\,i \in 1..n} \quad \Gamma \vdash C_1 \,\&\, .. \,\&\, C_n \,\&\, C \quad C_1 \,\&\, .. \,\&\, C_n \,\&\, C <: A}{\Gamma \vdash \mathbf{trait}\,[\mathsf{self} : B]\,\mathbf{inherits}\,\overline{E_i}^{\,i \in 1..n}\,\{\, \overline{l_j = E'_j}^{\,j \in 1..m} \,\} : A \Rightarrow \mathbf{Trait}\,[B, A] \rightsquigarrow \lambda\,(\mathsf{self} : |B|).\,(\overline{(e_i\,\mathsf{self})}^{\,i \in 1..n})\,,,\,e}\ \textsc{Inf-trait}$$

$$\frac{\Gamma \vdash E_1 \Rightarrow \mathbf{Trait}\,[A, B] \rightsquigarrow e_1 \quad \Gamma \vdash E_2 \Leftarrow A}{\Gamma \vdash E_1 \hat{}\, E_2 \Rightarrow B \rightsquigarrow e_1\ e_2}\ \textsc{Inf-forward}$$

$$\frac{\overline{\Gamma \vdash E_i \Rightarrow \mathbf{Trait}\,[A_i, B_i] \rightsquigarrow e_i}^{\,i \in 1..n} \quad \overline{A <: A_i}^{\,i \in 1..n} \quad \Gamma \vdash B_1 \,\&\, .. \,\&\, B_n \quad B_1 \,\&\, .. \,\&\, B_n <: A}{\Gamma \vdash \mathbf{new}\,[A](\overline{E_i}^{\,i \in 1..n}) \Rightarrow A \rightsquigarrow \mathbf{letrec}\,\mathsf{self} : |A| = \overline{(e_i\,\mathsf{self})}^{\,i \in 1..n}\,\mathbf{in}\,\mathsf{self}}\ \textsc{Inf-new}$$

Figure 4 Typing of SEDEL (excerpt)

$$\begin{array}{lrcl}
\text{Types} & \tau, \sigma & ::= & \top \mid \mathsf{Int} \mid \tau \to \sigma \mid \tau \,\&\, \sigma \mid \{l : \tau\} \mid \alpha \mid \forall\,(\alpha * \tau).\,\sigma \\
\text{Expressions} & e & ::= & \top \mid i \mid x \mid \lambda x.\,e \mid\ e_1\ e_2 \mid \Lambda\,(\alpha * \tau).\,e \mid\ e\,\tau \mid\ e_1\,,,\,e_2 \mid\ e : \tau \\
& & \mid & \{l = e\} \mid\ e.\,l \mid \mathbf{letrec}\,x : \tau = e_1\,\mathbf{in}\,e_2
\end{array}$$

Figure 5 Syntax of F_i with let bindings

However in practice, this is rather inconvenient as we need to write down all types we wish to retain rather than the one to exclude. So in our implementation we offer all of them.

Elaboration. The operational semantics of SEDEL is given by means of a type-directed translation into F_i extended with (lazy) recursive let bindings. This extension is standard and type-safe. The syntax of F_i is shown in Fig. 5. Let us go back to Fig. 4, now focusing on the highlighted parts, which denote the elaborated F_i terms. Most of them are straightforward translations and are thus omitted. We explain the most involved rules regarding traits. In rule Inf-trait, a trait is translated into a lambda abstraction with self as the formal parameter. In essence a trait corresponds to what Cook and Palsberg [19] call a *generator*. The translations of the inherited traits (i.e., $\overline{e_i}$) are each applied to self and then merged with the translation of the trait body e. Now it is clear why we require B (the type of self) to be a subtype of each B_i (the requirement of each inherited trait). Note that we abuse the bar notation here with the intention that $\overline{(e_i\,\mathsf{self})}^{\,i \in 1..n}$ means $e_1\,\mathsf{self}\,,,\,..\,,,\,e_n\,\mathsf{self}$. Here is an example of translating the `ide_editor` trait from Section 2 into plain F_i terms equipped with definitions (suppose `modal_mixin` and `spell_mixin` have been translated accordingly):

```
ide_editor = \ (self : IDEEditor) →
  (modal_mixin Spelling (spell_mixin ⊤ editor) self) ,, {version = "0.2"};
```

The gray parts in rule Inf-new show the translation of trait instantiation. First we apply every translation (i.e., e_i) of the instantiated traits to the self parameter, and then merge the applications together. The bar notation is interpreted similarly to the translation in rule Inf-trait. Finally we compute the *lazy* fixed-point of the resulting merge term, i.e., self-reference must be updated to refer to the whole composition. Taking the fixed-point of the traits/generators again follows the denotational inheritance model by Cook and

Palsberg. This is the key to the correct implementation of dynamic dispatching. Finally, rule INF-FORWARD translates forwarding expressions to function applications. We show the translation of the `a_editor1` object in Section 3 to illustrate the translation of instantiation:

```
a_editor1 = letrec self : IDEEditor = ide_editor self in self;
```

One remarkable point is that, while Cook and Palsberg work is done in an untyped setting, here we apply their ideas in a setting with disjoint intersection types and disjoint polymorphism. Our work shows that disjoint intersection types blend in quite nicely with Cook and Palsberg's denotational model of inheritance.

Flattening Property. In the literature of traits [21, 47, 34], a distinguished feature of traits is the so-called *flattening property*. This property says that a (non-overridden) method in a trait has the same semantics as if it were implemented directly in the class that uses the trait. It would be interesting to see if our trait model has this property. One problem in formulating such a property is that flattening is a property that talks about the equivalence between a flattened class (i.e., a class where all trait methods have been inlined) and a class that reuses code from traits. Since SEDEL does not have classes, we cannot state exactly the same property. However, we believe that one way to talk about a similar property for SEDEL is to have something along the lines of the following example:

▶ **Example 1** (Flattening)**.** Suppose we have `m` well-typed (i.e, conflict-free) traits `trait t1 {l11 = E11, ..}, ..., trait tm {lm1 = Em1, ..}`, each with some number of methods, then

```
new (trait inherits t1 & ... & tm {})   =   new (trait {l11 = E11,..,lm1 = Em1,..})
```

If we elaborate these two expressions, the property boils down to whether two merge terms $(E_1 \,,\, E_2) \,,\, E_2$ and $E_1 \,,\, (E_2 \,,\, E_3)$ have the same semantics. As is shown by Bi et al. [13], merges are associative and commutative, so it is not hard to see that the above two expressions are semantically equivalent. We leave it as future work to formally state and prove flattening.

4.3 Type Soundness and Coherence

Since the semantics of SEDEL is defined by elaboration into F_i [7] it is easy to show that key properties of F_i are also guaranteed by SEDEL. In particular, we show that the type-directed elaboration is type-safe in the sense that well-typed SEDEL expressions are elaborated into well-typed F_i terms. We also show that the source language is coherent and each valid source program has a unique (unambiguous) elaboration.

We need a meta-function $|\cdot|$ that translates SEDEL types to F_i types, whose definition is straightforward. Only the translation of trait types deserves attention:

$$|\mathbf{Trait}\,[A, B]| = |A| \to |B|$$

That is, trait types are translated to function types. $|\cdot|$ extends naturally to typing contexts. Now we show several lemmas that are useful in the type-safety proof.

▶ **Lemma 2.** *If* $\Gamma \vdash A$ *then* $|\Gamma| \vdash |A|$.

Proof. By structural induction on the well-formedness judgment. ◀

▶ **Lemma 3.** *If* $A <: B$ *then* $|A| <: |B|$.

Proof. By structural induction on the subtyping judgment. ◀

▶ **Lemma 4.** *If* $\Gamma \vdash A * B$ *then* $|\Gamma| \vdash |A| * |B|$.

Proof. By structural induction on the disjointness judgment. ◀

Finally we are in a position to establish the type safety property:

▶ **Theorem 5** (Type-safe translation). *We have that:*

- *If* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ *then* $|\Gamma| \vdash e \Rightarrow |A|$.
- *If* $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ *then* $|\Gamma| \vdash e \Leftarrow |A|$.

Proof. By structural induction on the typing judgment. ◀

▶ **Theorem 6** (Coherence). *Each well-typed SEDEL expression has a unique elaboration.*

Proof. By examining every elaboration rule, it is easy to see that the elaborated F_i term in the conclusion is uniquely determined by the elaborated F_i terms in the premises. Then by the coherence property of F_i, we conclude that each well-typed SEDEL expression has a unique unambiguous elaboration, thus SEDEL is coherent. ◀

5 Case Study: Modularizing Language Components

To further illustrate the applicability of SEDEL, we present a case study using Object Algebras [39] and Extensible Visitors [38, 52]. Encodings of extensible designs for Object Algebras and Extensible Visitors have been presented in mainstream languages [38, 52, 39, 41, 44]. However, prior approaches are not entirely satisfactory due to the limitations in existing mainstream OO languages. In Section 5.1, we show how SEDEL makes those designs significantly simpler and convenient to use. In particular, SEDEL's encoding of extensible visitors gives true ASTs and supports conflict-free Object Algebra combinators, thanks to first-class traits and disjoint polymorphism. Based on this technique, Section 5.2 gives a bird-view of several orthogonal features of a small JavaScript-like language from a textbook on Programming Languages [18], and illustrates how various features can be modularly developed and composed to assemble a complete language with various operations baked in. Section 5.3 compares our SEDEL's implementation with that of the textbook using Haskell in terms of lines of code.

5.1 Object Algebras and Extensible Visitors in SEDEL

First we give a simple introduction to Object Algebras, a design pattern that can solve the Expression Problem [54] (EP) in languages like Java. The objective of EP is to *modularly* extend a datatype in two dimensions: by adding more cases to the datatype and by adding new operations for the datatype. Our starting point is the following code:

```
type ExpAlg[E] = { lit : Int → E, add : E → E → E };
type IEval = { eval : Int };
trait evalAlg => {
  lit (x : Int) = { eval = x };
  add (x : IEval) (y : IEval) = { eval = x.eval + y.eval }
};
```

`ExpAlg[E]` is the generic interface of a simple arithmetic language with two cases, `lit` for literals and `add` for addition. `ExpAlg[E]` is also called an Object Algebra interface. A concrete Object Algebra will implement such an interface by instantiating `E` with a suitable type. Here we also define one operation `IEval`, modelled by a single-field record type. A concrete Object Algebra that implements the evaluation rules is given by a trait `evalAlg`.

First-Class Object Algebra Values. The actual AST of this simple arithmetic language is given as an internal visitor [42]:

```
type Exp = { accept : forall E . ExpAlg[E] → E };
```

Note that Object Algebras as implemented in languages like Java or Scala do not define the type `Exp` because this would make adding new variants very hard. Although extensible versions of this visitor pattern do exist, they usually require complex types using advanced features of generics [39, 52]. However, as we will see, this is not a problem in SEDEL. We can build a value of `Exp` as follows:

```
e1 : Exp = { accept E f = f.add (f.lit 2) (f.lit 3) };
```

Adding a New Operation. We add another operation `IPrint` to the language:

```
type IPrint = { print : String };
trait printAlg => {
  lit (x : Int) = { print = x.toString };
  add (x : IPrint) (y : IPrint) = {
    print = "(" ++ x.print ++ " + " ++ y.print ++ ")"
  }
};
```

This is done by giving another trait `printAlg` that implements the additional `print` method.

Adding a New Case. A second dimension for extension is to add another case for negation:

```
type ExpExtAlg[E] = ExpAlg[E] & { neg : E → E };
trait negEvalAlg inherits evalAlg => {
  neg (x : IEval) = { eval = 0 - x.eval }
};
trait negPrintAlg inherits printAlg => {
  neg (x : IPrint) = { print= "-" ++ x.print }
};
```

This is achieved by extending `evalAlg` and `printAlg`, implementing missing operations for negation, respectively. We define the actual AST similarly:

```
type ExtExp = { accept: forall E. ExpExtAlg[E] → E };
```

and build a value of `-(2 + 3)` while reusing `e1`:

```
e2 : ExtExp = { accept E f = f.neg (e1.accept E f) };
```

Relations between `Exp` and `ExpExt` At this stage, it is interesting to point out an interesting subtyping relation between `Exp` and `ExtExp`: `ExpExt`, though being an *extension* of `Exp` is actually a *supertype* of `Exp`. As Oliveira [38] observed, these relations are important for legacy and performance reasons since it means that, a value of type `Exp` can be *automatically* and *safely* coerced into a value of type `ExpExt`, allowing some interoperability between new functionality and legacy code. However, to ensure type-soundness, Scala (or other common OO languages) forbids any kind of type-refinement on method parameter types. The consequence of this is that in those languages, it is impossible to express that `ExtExp` is both an extension and a supertype of `Exp`.

Dynamic Object Algebra Composition Support When programming with Object Algebras, oftentimes it is necessary to pack multiple operations in the same object. For example, in the simple language we have been developing it can be useful to create an object that supports

Types	τ	::=	$\mathsf{int} \mid \mathsf{bool}$	
Expressions	e	::=	$i \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \div e_2$	*natF*
		$\mid$	$\mathbb{B} \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	*boolF*
		$\mid$	$e_1 == e_2 \mid e_1 < e_2$	*compF*
		$\mid$	$e_1 \,\&\&\, e_2 \mid e_1 \,\|\|\, e_2$	*logicF*
		$\mid$	$x \mid \textbf{var}\ x = e_1; e_2$	*varF*
		$\mid$	$e_1\ e_2$	*funcF*
Programs	*pgm*	::=	$decl_1 \ldots decl_n\ e$	*funcF*
Functions	*decl*	::=	$\textbf{function}\ f(x : \tau)\{e\}$	*funcF*
Values	v	::=	$i \mid \mathbb{B}$	

Figure 6 Mini-JS expressions, values, and types

both printing and evaluation. Oliveira and Cook [39] addressed this problem by proposing *Object Algebra combinators* that combine multiple algebras into one. However, as they noted, such combinators written in Java are difficult to use in practice, and they require significant amounts of boilerplate. Improved variants of Object Algebra combinators have been encoded in Scala using intersection types and an encoding of the merge construct [41, 44]. However, the Scala encoding of the merge construct is quite complex as it relies on low-level type-unsafe programming features such as dynamic proxies, reflection or other meta-programming techniques. In SEDEL, the combination of first-class traits, dynamic inheritance and disjoint polymorphism allows type-safe, coherent and boilerplate-free composition of Object Algebras.

```
combine A [B * A] (f : Trait[ExpExtAlg[A]]) (g : Trait[ExpExtAlg[B]]) =
  trait inherits f & g => {};
```

That is it. None of the boilerplate in other approaches [39], or type-unsafe meta-programming techniques of other approaches [41, 44] are needed! Two points are worth noting: (1) `combine` relies on *dynamic inheritance*. Notice how `combine` inherits two traits `f` and `g`, for which their implementations are unknown statically; (2) the disjointness constraint (`B * A`) is *crucial* to ensure two Object Algebras (`f` and `g`) are conflict-free when being composed.

To conclude, let us see `combine` in action. We combine `negEvalAlg` and `negPrintAlg`:

```
combinedAlg = combine IEval IPrint negEvalAlg negPrintAlg;
```

The combined algebra `combineAlg` is useful to avoid multiple interpretations of the same AST when running multiple operations. For example, we can create an object `o` that supports both evaluation and printing in one go:

```
o = e2.accept (IEval & IPrint) (new[ExpExtAlg[IEval & IPrint]] combinedAlg);
main = o.print ++ " = " ++ o.eval.toString -- "-(2.0 + 3.0) = -5.0"
```

5.2 Case Study Overview

Now we are ready to see how the same technique scales to modularize different language features. A *feature* is an increment in program functionality [56, 31]. Figure 6 presents the syntax of the expressions, values and types provided by the features; each line is annotated with the corresponding feature name. Starting from a simple arithmetic language, we gradually introduce new features and combine them with some of the existing features to form various languages. Below we briefly explain what constitutes each feature:

- *natF* and *boolF* contain, among others, literals, additions and conditional expressions.
- *compF* and *logicF* introduce comparisons between numbers and logical connectives.
- *varF* introduces local variables and variable declarations.
- *funcF* introduces top-level functions and function calls.

Language	Operations			Data variants					
	eval	print	check	*natF*	*boolF*	*compF*	*logicF*	*varF*	*funcF*
`simplenat`	✓	✓		✓					
`simplebool`	✓	✓			✓				
`natbool`	✓	✓	✓	✓	✓				
`varbool`	✓	✓			✓			✓	
`varnat`	✓	✓		✓				✓	
`simplelogic`	✓	✓			✓		✓		
`varlogic`	✓	✓			✓		✓	✓	
`arith`	✓	✓	✓	✓	✓	✓			
`arithlogic`	✓	✓	✓	✓	✓	✓	✓		
`vararith`	✓	✓	✓	✓	✓	✓		✓	
`vararithlogic`	✓	✓	✓	✓	✓	✓	✓	✓	
`mini-JS`	✓	✓	✓	✓	✓	✓	✓	✓	✓

Figure 7 Overview of the languages assembled

Besides, each feature is packed with 3 operations: evaluator, pretty printer and type checker.

Having the feature set, we can synthesize different languages by selecting one or more operations, and one or more data variants, as shown in Fig. 7. For example `arith` is a simple language of arithmetic expressions, assembled from *natF*, *boolF* and *compF*. On top of that, we also define an evaluator, a pretty printer and a type checker. Note that for some languages (e.g., `simplenat`), since they have only one kind of value, we only define an evaluator and a pretty printer. We thus obtain 12 languages and 30 operations in total. The complete language `mini-JS` contains all the features and supports all the operations. The reader can refer to our supplementary material for the source code of the case study.

5.3 Evaluation

To evaluate SEDEL's implementation of the case study, Figure 8 compares the number of source lines of code (SLOC, lines of code without counting empty lines and comments) for SEDEL's *modular* implementation with the vanilla *non-modular* AST-based implementations in Haskell. The Haskell implementations are just straightforward AST interpreters, which duplicate code across the multiple language components.

Since SEDEL is a new language, we had to write various code that is provided in Haskell by the standard library, so they are not counted for fairness of comparison. In the left part, for each feature, we count the lines of the algebra interface (number beside the feature name), and the algebras for the operations. In the right part, for each language, we count the lines of ASTs, and those to combine previously defined operations. For example, here is the code that is needed to make the `arith` language.

```
type ArithAlg[E] = NatBoolAlg[E] & CompAlg[E];          -- Object Algebra interface
type Arith = { accept : forall E. ArithAlg[E] → E };   -- AST
evalArith (e : Arith) : IEval =                         -- Evaluator
  e.accept IEval (new[ArithAlg[IEval]] evalNatAlg & evalBoolAlg & evalCompAlg);
ppArith (e : Arith) : IPrint =                          -- Pretty printer
  e.accept IPrint (new[ArithAlg[IPrint]] ppNatAlg & ppBoolAlg & ppCompAlg);
tcArith (e : Arith) =                                   -- Type checker
  e.accept ITC (new[ArithAlg[ITC]] tcNatAlg & tcBoolAlg & tcCompAlg);
```

We only need 8 lines in total: 2 lines for the AST, and 6 lines to combine the operations.

Therefore, the total SLOC of SEDEL's implementation is the sum of all the lines in

Feature	eval	print	check	Lang name	SEDEL	Haskell	% Reduced
natF(7)	23	7	39	`simplenat`	3	33	91%
boolF(4)	9	4	17	`simplebool`	3	16	81%
compF(4)	12	4	20	`natbool`	5	74	93%
logicF(4)	12	4	20	`varbool`	4	24	83%
varF(4)	7	4	7	`varnat`	4	41	90%
funcF(3)	10	3	9	`simplelogic`	4	28	86%
				`varlogic`	6	36	83%
				`arith`	8	94	91%
				`arithlogic`	8	114	93%
				`vararith`	8	107	93%
				`vararithlogic`	8	127	94%
				`mini-JS`	33	149	78%
Total			237		331	843	61%

Figure 8 SLOC statistics: SEDEL implementation vs vanilla AST implementation.

the feature and language parts (237 SLOC of all features plus 94 SLOC of ASTs and operations). Although SEDEL is considerably more verbose than a functional language like Haskell, SEDEL's modular implementation for 12 languages and 30 operations in total reduces approximately 60% in terms of SLOC. The reason is that, the more frequently a feature is reused by other languages directly or indirectly, the more reduction we see in the total SLOC. For example, *natF* is used across many languages. Even though `simplenat` itself *alone* has more SLOC ($40 = 7 + 23 + 7 + 3$) than that of Haskell (which has 33), we still get a huge gain when implementing other languages.

Finally, we acknowledge the limitation of our case study in that SLOC is just one metric and we have not measured any other metrics. Nevertheless we believe that the case study is already non-trivial in that we need to solve EP. Note that Scala traits alone are not sufficient on their own to solve EP. While there are solutions to EP in both Haskell and Scala, they introduce significant complexity, as explained in Section 5.1.

6 Related Work

Typed First-Class Classes/Mixins/Traits. First-class classes have been used in Racket [26], along with mixin support, and have shown great practical value. For example, DrRacket IDE [24] makes extensive use of layered combinations of mixins to implement text editing features. The topic of first-class classes with static typing has been explored by Takikawa et al. [51] in Typed Racket. They designed a gradual type system that supports first-class classes. Of particular interest is their use of row polymorphism [55] to type mixins. As with our use of disjoint polymorphism, row polymorphism can express constraints on the presence or absence of members. Unlike disjoint polymorphism, row polymorphism prohibits forgetting class members. For example, in SEDEL we can write:

```
foo [A * {bar : String}] (t : Trait[{bar : String} & A]) : Trait[A] = t;
```

where `foo` drops `bar` from its argument trait `t`, which is impossible to express in Typed Racket. Also as we pointed out in Section 3.6, row polymorphism alone cannot express the `merge` function that is able to compose objects of statically unknown types. In this sense, we argue disjoint polymorphism is more powerful than row polymorphism in terms of expressivity. It would be interesting to investigate the relationship between disjoint polymorphism and row polymorphism. We leave it as future work.

More recently, Lee et al. [30] proposed a model for typed first-class classes based on tagged objects. Like our development, the semantics of their source language is defined by a translation into a target language. One notable difference to SEDEL is that they require the use of a variable rather than an expression in the **extends** clause, whereas we do not have this restriction. In their source language, subclasses define subtypes, which limits its applicability to extensible designs. Also their target calculus is significantly more complex than ours due to the use of dependent function types and dependent sum types. As they admitted, they omit inheritance in their formalization.

Racket also supports a *dynamically-typed model* of first-class traits [26]. However, unlike Racket's first-class classes and mixins, there's no type system supporting the use of first-class traits. A key difficulty is *statically* detecting conflicts. As far as we know, SEDEL is the first design for typed first-class traits.

Mixin-Based Inheritance. Bracha and Cook's seminal paper [16] extends Modula-3 with mixins. Since then, many mixin-based models have been proposed [27, 14, 8]. Mixin-based inheritance requires that mixins are composed linearly, and as such, conflicts are resolved implicitly. In comparison, the trait model in SEDEL requires conflicts to be resolved explicitly. We want to emphasize that conflict detection is essential in expressing composition operators for Object Algebras, without running into ambiguities. Bracha's Jigsaw [15] formalized mixin composition, along with a rich trait algebra including merge, restrict, select, project, overriding and rename operators. Lagorio et al. [29] proposed FJig that reformulates Jigsaw constructs in a Java-like setting. Allen et al. [6] described how to add first-class generic types – including mixins – to OO languages with nominal typing. As such, classes and mixins, though they enjoy static typing, are still second-class constructs, and thus their system cannot express dynamic inheritance. Bessai et al. [9] showed how to type classes and mixins with intersection types and Bracha-Cook's merge operator [16].

Trait-Based Inheritance. Traits were proposed by Schärli et al. [47, 21] as a mechanism for fine-grained code reuse to overcome many limitations of class-based inheritance. The original proposal of traits were implemented in the dynamically-typed class-based language Squeak/Smalltalk. Since then various formalizations of traits in a Java-like (statically-typed) setting have been proposed [25, 46, 50, 34]. In most of the above proposals, trait composition and class-based inheritance live together. SEDEL, in the spirit of *pure trait-based programming languages* [12, 11], embraces traits as the sole mechanism for code reuse. The deviation from traditional class-based inheritance is not only because of its simplicity, but also because we need a very *dynamic* form of inheritance.

Languages with More Advanced Forms of Inheritance. Self [53] is a dynamically-typed, prototype-based language with a simple and uniform object model. Self's inheritance model is typical of what we call *mutable inheritance*, because an object's parent slot may be assigned new values at runtime. Mutable inheritance is rather unstructured, and oftentimes access to any clashing methods will generate a "messageAmbiguous" error at runtime. Although SEDEL's dynamic inheritance is not as powerful as mutable inheritance, its static type system can guarantee that no such errors occur at runtime. Eiffel [33] supports a sophisticated class-based multiple inheritance with deep renaming, exclusion and repeated inheritance. Of particular interest is that in Eiffel, name collisions are considered programming errors, and ambiguities must be resolved explicitly by the programmer (by means of renaming). In this regard, SEDEL is quite like Eiffel. However, the type system in SEDEL is more lenient in

that two identically named methods with different signatures can coexist. Grace [35, 28] is an object-based language designed for education, where objects are created by *object constructors*. Since Grace has mutable fields, it has to consider many concerns when it comes to inheritance, resulting in a rather complex inheritance mechanism with various restrictions. Since SEDEL is pure, a relatively simple encoding of traits with late binding of `self` suffices for our applications. Grace's support for multiple inheritance is based on so-called *instantiable traits*. We believe that there is plenty to be learned from Grace's design of traits if we want to extend our trait model with features such as mutable state. MetaFJig [48] (an extension of FJig) supports *dynamic trait replacement* [50, 10, 21], a feature for changing the behavior of an object at runtime by replacing one trait for another.

Module Systems. In parallel to OOP, the ML module system originally proposed by MacQueen [32] also offers powerful support for flexible program construction, data abstraction and code reuse. Mixin modules in the Jigsaw framework [17] provides a suite of operators for adapting and combining modules. The MixML [45] module system incorporates mixin module composition, while retaining the full expressive powerful of ML modules. Module systems usually put more emphasis on supporting type abstraction. Support for type abstraction adds considerable complexity, which is not needed in SEDEL. SEDEL is focused on OOP and supports, among others, method overriding, self references and dynamic dispatching, which (generally speaking) are all missing features in module systems.

Intersection Types, Polymorphism and Merge Construct. There is a large body of work on intersection types. Here we only talk about work that has direct influences on ours. Dunfield [22] shows significant expressiveness of type systems with intersection types and a merge construct. However his calculus lacks coherence. The limitation was addressed by Oliveira el at. [40], where they introduced the notion of disjointness to ensure coherence. The combination of intersection types, a merge construct and parametric polymorphism, while achieving coherence was first studied in the F_i calculus [7]. F_i serves as the target language of SEDEL. Dynamic inheritance, self-references and abstract methods are all missing from F_i but, as shown in this paper, they can be encoded using an elaboration that employs ideas from Cook and Palsberg's denotational model of inheritance [19].

7 Conclusion

This paper presents SEDEL: the first design for a polymorphic statically-typed language with first-class traits, supporting dynamic inheritance as well as conventional OO features such as dynamic dispatching and abstract methods. The paper also shows how high-level source language constructs can be elaborated into a core record calculus with disjoint polymorphism. Finally the paper illustrates the applicability of SEDEL by showing greatly improved design patterns such as Object Algebras and Extensible Visitors, leveraging first-class traits. As for future work, we are interested to study how first-class traits interacts with features such as mutable state and recursive types. For mutable state, one immediate issue of supporting mutation is how it affects the coherence property of F_i, and we foresee major technical challenges to adjust the previous coherence proof. A more powerful proof method such as logical relations [13, 5] may be needed.

References

1 Javascript. URL: `https://www.javascript.com/`.

2 Python. URL: https://www.python.org/.
3 Racket. URL: https://racket-lang.org/.
4 Ruby. URL: https://www.ruby-lang.org/en/.
5 Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.
6 Eric E. Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2003.
7 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.
8 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam–designing a java extension with mixins. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003.
9 Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. Typing classes and mixins with intersection types. In *Workshop on Intersection Types and Related Systems (ITRS)*, 2014.
10 Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. On flexible dynamic trait replacement for java-like languages. *Science of Computer Programming*, 78(7):907 – 932, 2013.
11 Lorenzo Bettini and Ferruccio Damiani. Xtraitj: Traits for the java platform. *Journal of Systems and Software*, 131:419 – 441, 2017.
12 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. Traitrecordj: A programming language with traits and records. *Science of Computer Programming*, 78(5):521 – 541, 2013.
13 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The essence of nested composition. In *European Conference on Object-Oriented Programming*, 2018.
14 Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999.
15 Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, 1992.
16 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1990.
17 Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, 1992.
18 William R. Cook. *Anatomy of Programming Languages*. 2013. URL: http://www.cs.utexas.edu/~wcook/anatomy/.
19 William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, 1989.
20 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.
21 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, mar 2006.
22 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.
23 Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015. URL: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.
24 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *J. Funct. Program.*, 12(2):159–182, 2002.
25 Kathleen Fisher and John Reppy. A typed calculus of traits. In *Workshop on Foundations of Object-oriented Programming*, 2004.

26 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Programming Languages and Systems (APLAS)*, 2006.

27 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, 1998.

28 Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. Object inheritance without classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2016.

29 Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight jigsaw — replacing inheritance by composition in java-like languages. *Information and Computation*, 214:86 – 111, 2012.

30 Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

31 Roberto E Lopez-Herrejon, Don Batory, and William Cook. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

32 David MacQueen. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming - LFP '84*, 1984.

33 Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices*, 22(2):85–94, 1987.

34 Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journal of Object Technology*, 5(4):129–148, May 2006.

35 James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Timothy Jones. Grace's inheritance. *Journal of Object Technology*, 16(2):2:1–35, 2017.

36 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

37 Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA 2005)*, 2005.

38 Bruno C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *European Conference on Object Oriented Programming (ECOOP)*, 2009.

39 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.

40 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.

41 Bruno C. d. S. Oliveira, Tijs Van Der Storm, Alex Loh, and William R Cook. Feature-oriented programming with object algebras. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013.

42 Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, 2008.

43 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.

44 Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. From object algebras to attribute grammars. In *Object Oriented Programming, Systems Languages and Applications (OOPSLA)*, 2014.

45 Andreas Rossberg and Derek Dreyer. Mixin' up the ML module system. *ACM Transactions on Programming Languages and Systems*, 35(1):1–84, apr 2013.

46 Nathanael Scharli, St Ducasse, Roel Wuyts, Andrew Black, et al. Traits: The formal model. 2003.

47 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

48 Marco Servetto and Elena Zucca. A meta-circular language for active libraries. *Science of Computer Programming*, 95:219 – 253, 2014.

49 Yannis Smaragdakis and Don S. Batory. Mixin-based programming in C++. In *Generative and Component-Based Software Engineering (GCSE)*, 2000.

50 Charles Smith and Sophia Drossopoulou. Chai: Traits for java-like languages. In Andrew P. Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

51 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object-oriented Programming: Systems, Languages and Applications (OOPSLA)*, 2012.

52 Mads Torgersen. The Expression Problem Revisited. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

53 David Ungar and Randall B Smith. Self: the power of simplicity (object-oriented language). In *Compcon Spring'88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, 1988.

54 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

55 Mitchell Wand. Type inference for objects with instance variables and inheritance. *Theoretical aspects of object-oriented programming*, pages 97–120, 1994.

56 Pamela Zave. Faq sheet on feature interaction. *Link: http://www. research. att. com/~ pamela/faq. html*, 1999.