Typed First-Class Traits

John Q. Open[1] and Joan R. Access[2]

1 **Dummy University Computing Laboratory, Address/City, Country**
`open@dummyuniversity.org`
2 **Department of Informatics, Dummy College, Address/City, Country**
`access@dummycollege.org`

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.

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Many dynamically typed languages (including JavaScript [1], Ruby [4], Python [2] or Racket [3]) support *first-class classes* [21], 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 parametrized 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 mimick 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 [11] 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 [40]; and Lee et al.'s model of typed first-class classes [24]. Both works provide typed models of first-class classes, and they enable encodings of mixins [12] similar to those employed in dynamically typed languages.

However, as far as we known no previous work supports statically typed *first-class traits*. Traits [38] 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 method implementation in case of conflicts. For example, Scala uses the order of mixin composition to determine which method implementations to pick in the 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. [38] 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 [11] 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 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's type system is based on *disjoint intersection types* [32] and *disjoint polymorphism* [6].

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 [14], we show how to design a source language that can be elaborated into Alpuim et al.'s F_i [6], a polymorphic calculus with records supporting disjoint polymorphism. SEDEL's elaboration into F_i is proved to be both type-safe and coherent. Coherency 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 [31] 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 [33, 36]. The previous state-of-the-art implementations for such operation require employing type-unsafe reflective techniques (including `java.lang.Class`) 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 [6]. The elaboration is inspired by the work by Cook and Palsberg [14] 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 such improved design patterns we re-code the interpreters in Cook's undergraduate Programming Languages book [13] in a modular way in SEDEL.

[1] The implementation, case study code and proofs are submitted as supplementary material.

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 [40]. 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 [18]. 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 illustrative strings.

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

to 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 the programmer 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 classes 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();
```

[2] With C++ templates, it is possible to implement a so-called mixin pattern [39], which enables extending a parametrized 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).

```
            this.mode = "command";
        }
        toggleMode() {
            return "toggle succeeded"
        }
        onKey(key) {
            return "Process " + key + " on modal editor";
        }
    };
};
```

`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 `onKey` method.

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 a `onKey` method of its own. Thus 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 linearity, i.e., the order of mixin applications. Mixins appearing later in the order overrides *all* the identical 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.

The Problem of Mixin Composition From the above discussion, we can see mixin composition is linear: 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.

Traits Because of the linearity and the limited means for resolving conflicts, researchers have proposed a simple compositional model called traits [38]. Traits are lightweight entities and serve as the primitive units of code reuse. Among others, the biggest 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 typed 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 support dynamic inheritance, i.e., what are the types of functions that take classes as arguments and inherit them in the body.
- How to express constraints on method presence and absence (the use of **super** clearly demands that).
- How to ensure that composition of mixins is going to be valid, i.e., how to reflect linearity in a type system.
- 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 result 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 [28]) where we can give a type annotation to the `self` reference. In the type of the self-reference `&` is the construct for type intersections, and `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 a 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. On 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 create first-class traits via trait expressions `trait [self : ...] inherits ... => {...}`. Trait expressions have trait types of the form `Trait[A, B]`. We will explain trait types in Section 3. Despite the structural similarities, there are several significant features that are unique to SEDEL. We discuss these in the following.

Disjoint Polymorphism and Conflict Detection SEDEL uses a type system based on *disjoint intersection types* [32] and *disjoint polymorphism* [6]. 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.

```
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). In summary, `Trait[Editor & Version, Editor & A]` (the assigned type of `base`) specifies both method *presence* and *absence*. 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. On 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 [44] 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 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 (linearity) 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 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

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

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 [38]. Traits provide a mechanism of code reuse in object-oriented programming, which can be used to model disciplined forms 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 the case of SEDEL, detected by the type system, and explicitly resolved 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.

3.2 The 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 [38, 20, 29] 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 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 thing in `spell` 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*, as we will discuss next.

3.3 Trait Types and Trait Requirements

Object Types and Trait Types. SEDEL adopts a relatively standard foundational model of object-oriented constructs [24] 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]`. In general, a trait type **Trait**`[A, B]` specifies both the *requirements* `A` and the *functionality* `B` of a trait. When a trait has no requirements, the absence of a requirement is denoted by using the top type ($\top$). A simplified sugar **Trait**`[B]` is used to denote a trait without requirements, but providing functionality `B`.

Trait Requirements and Functionality. When modeling a trait there are two important aspects: what are the *requirements* of a trait; and what is the *functionality* that a trait provides? The requirements of a trait denote the types/methods that the trait needs to support 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 provides `on_key` and `check`.

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 in 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 an 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. The `IDEEditor` type is defined as an intersection type (`Editor & Version & Spelling & ModalEdit`). Intersection types [15, 35] 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 `A & B` contains exactly those values which can be used as values of type `A` and of type `B`, and as such, `A & B` immediately introduces a subtyping relation between itself and its two constituent types `A` and `B`. 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 to `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 are 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 deal with conflicts. The trait-based approach requires conflicts to be resolved at the level of the composition by the programmer, otherwise the program is rejected by the type system. SEDEL provides a means to help programmers resolve conflicts.

We illustrate this problem 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 a `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 – the programmer must resolve the conflicts *explicitly*.

As mentioned in Section 1, SEDEL's type system is based on intersection types. More concretely, SEDEL uses a type system based on *disjoint intersection types* [32]. Disjointness, in its simplest form, means that the set of values of both types are disjoint. The above definition is ill-typed precisely because there is a conflicting method `on_key`, thus violating the disjointness conditions.

Resolving Conflicts. To resolve the conflict, the programmer needs 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 inherit 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.

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

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

Figure 1 SEDEL core syntax and syntactic abbreviations

4 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 [6], 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 [14]. 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.

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, function types $A \to B$, intersection types $A \,\&\, B$, singleton record types $\{l : A\}$. Inheriting from F_i, types also include two constructs used to support disjoint polymorphism: 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: lambda abstractions $\lambda x.\, E$, function applications $E_1\, E_2$, singleton records $\{l = E\}$, record projections $E.l$ and recursive let bindings $\mathbf{letrec}\, x : A = E_1\, \mathbf{in}\, E_2$. Inheriting from F_i, we have two constructs for polymorphism: 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[4] $\mathbf{trait}\,[\mathsf{self} : B]\, \mathbf{inherits}\, \overline{E_i}^{\,i}\, \{\, \overline{l_j = E'_j}^{\,j} \,\} : A$, which specifies a (possibly empty) list of trait expressions $\overline{E_i}$ in the inherit 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

[4] The abstract syntax of trait expressions is slightly different from the concrete syntax.

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

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

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

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

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

anywhere where an expression is expected. Trait instantiation expressions $\textbf{new}\,[A](\overline{E_i}^{\,i})$ 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 [6]. The subtyping rule for trait types (rule Sub-trait) resembles the one for function types (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 [6] 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. The rationale behind these rules will become clearer when we get to the translation of SEDEL types to F_i types. The axiom rules of the form $A *_{ax} B$ take care of two types with different language constructs. These rules capture the notion that any two types are disjoint unless one of them is an intersection types, a type variable or $\top$.

Typing The typing rules of SEDEL are shown in Fig. 4. For succinctness, the full set of rules can be found in the appendix. The reader is advised to ignore the highlighted parts for now. As with conventional bidirectional type systems, the typing rules of SEDEL employ 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 rule Inf-merge, which plays an essential role behind conflict detection of traits, namely, a merge of two expressions is valid if and only if their types are disjoint. One representative of checking rules is rule Chk-sub, typically known as the subsumption rule,

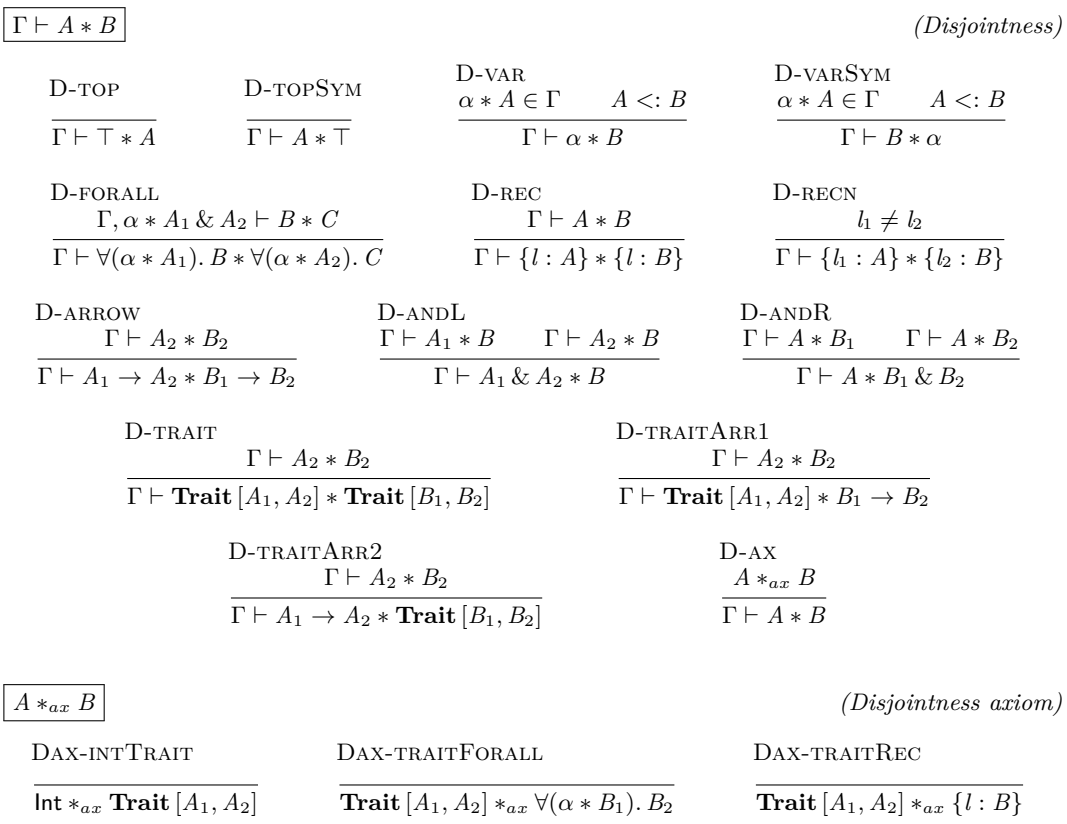

Figure 3 Disjointness rules of SEDEL (excerpt)

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 are well-behaved. The well-formedness judgment $\Gamma \vdash C_1 \,\&\, .. \,\&\, C_n \,\&\, C$ ensures that we do not have conflicting methods (both from 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 a valid trait. There are also several side-conditions, which serve the same purposes as in rule INF-TRAIT. Rule INF-FORWARD dictates how to use the forwarding operator. More concretely, the first operand E_1 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 result record is guaranteed to have no `bar` field. In the same

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

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

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

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

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

$\boxed{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}$ *(Check)*

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

Figure 4 Typing of SEDEL (excerpt)

vein, the use of **override** can be explained using the exclusion operator. The **super** keyword is internally a variable pointing to the inherit clause. We omit these in the meta-theoretic study in order to focus our attention on the essence of first-class traits. 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 these.

Elaboration The operational semantics of SEDEL is given by means of a type-directed translation into an extension of F_i with (lazy) recursive let bindings. 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. 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) 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):

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

Figure 5 F_i syntax

$\boxed{\Delta \vdash e \Rightarrow \tau}$ *(Infer)*

$$\text{TI-TOP}\ \frac{}{\Delta \vdash \top \Rightarrow \top} \qquad \text{TI-INT}\ \frac{}{\Delta \vdash i \Rightarrow \mathsf{Int}} \qquad \text{TI-VAR}\ \frac{x : \tau \in \Delta}{\Delta \vdash x \Rightarrow \tau} \qquad \text{TI-ANNO}\ \frac{\Delta \vdash e \Leftarrow \tau}{\Delta \vdash e : \tau \Rightarrow \tau}$$

$$\text{TI-APP}\ \frac{\Delta \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Delta \vdash e_2 \Leftarrow \tau_1}{\Delta \vdash e_1\ e_2 \Rightarrow \tau_2} \qquad \text{TI-TAPP}\ \frac{\Delta \vdash e \Rightarrow \forall(\alpha * \sigma).\, \tau' \qquad \Delta \vdash \tau \qquad \Delta \vdash \tau * \sigma}{\Delta \vdash e\,\tau \Rightarrow [\tau/\alpha]\tau'}$$

$$\text{TI-MERGE}\ \frac{\Delta \vdash e_1 \Rightarrow \tau \qquad \Delta \vdash e_2 \Rightarrow \sigma \qquad \Delta \vdash \tau * \sigma}{\Delta \vdash e_1 \,,,\, e_2 \Rightarrow \tau \,\&\, \sigma} \qquad \text{TI-REC}\ \frac{\Delta \vdash e \Rightarrow \tau}{\Delta \vdash \{l = e\} \Rightarrow \{l : \tau\}} \qquad \text{TI-PROJ}\ \frac{\Delta \vdash e \Rightarrow \{l : \tau\}}{\Delta \vdash e.l \Rightarrow \tau}$$

$$\text{TI-BLAM}\ \frac{\Delta \vdash \tau \qquad \Delta, \alpha * \tau \vdash e \Rightarrow \sigma}{\Delta \vdash \Lambda(\alpha * \tau).\, e \Rightarrow \forall(\alpha * \tau).\, \sigma} \qquad \text{TI-LETE}\ \frac{\Delta, x : \tau \vdash e_1 \Leftarrow \tau \qquad \Delta, x : \tau \vdash e_2 \Rightarrow \sigma}{\Delta \vdash \mathbf{letrec}\, x : \tau = e_1\, \mathbf{in}\, e_2 \Rightarrow \sigma}$$

$\boxed{\Delta \vdash e \Leftarrow \tau}$ *(Check)*

$$\text{TC-ABS}\ \frac{\Delta \vdash \tau \qquad \Delta, x : \tau \vdash e \Leftarrow \sigma}{\Delta \vdash \lambda x.\, e \Leftarrow \tau \to \sigma} \qquad \text{TC-SUB}\ \frac{\Delta \vdash e \Rightarrow \tau \qquad \tau <: \sigma \qquad \Delta \vdash \sigma}{\Delta \vdash e \Leftarrow \sigma}$$

Figure 6 Typing of F_i

```
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. Note that passing the self parameter to each e_i is the key to the correct implementation of dynamic dispatching. 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;
```

4.3 Type Soundness and Coherence

Since the semantics of SEDEL is defined by elaboration into F_i [6] 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.

To aid understanding of the type safety statement, we show the typing rules of F_i in Fig. 6. The interested reader can refer to [6] for further explanation. We also 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 technical lemmas that are useful in the type safety proof.

▶ **Lemma 1.** *If* $\Gamma \vdash A$ *then* $|\Gamma| \vdash |A|$.

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

▶ **Lemma 2.** *If* $A <: B$ *then* $|A| <: |B|$.

Proof. By structural induction on the subtyping judgment. ◀

▶ **Lemma 3.** *If* $\Gamma \vdash A * B$ *then* $|\Gamma| \vdash |A| * |B|$.

Proof. By structural induction on the disjointness judgment. ◀

▶ Remark. Due to the elaboration semantics, rules D-TRAITARR1 and D-TRAITARR2 are needed to make this lemma hold.

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

▶ **Theorem 4** (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 5** (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 [31] and Extensible VISITORs [30, 41]. Encodings of extensible designs for Object Algebras and Extensible VISITORs have been presented in mainstream languages [30, 41, 31, 33, 36]. However, prior approaches are not entirely satisfactory due to the limitations in existing mainstream OO languages. In the first half of the section, 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,

the second half presents the case study modularizing several orthogonal features of a small JavaScript-like language from a textbook on Programming Languages [13]. The case study illustrates how various features can be modularly developed and composed to assemble a complete language with various operations baked in.

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 [43] in languages like Java. The objective of the expression problem 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 [34]:

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

Note that Object Algebras in languages like Java or Scala do not define the type `Exp` because this would preclude extensibility in those languages. However, as we shall see, this is not a problem in SEDEL. We build a value of `2 + 3` 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 [30] 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. Encodings of extensible visitors in mainstream OO languages usually fail to correctly express these relations, or require sophisticated type system extensions [30].

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 may be useful to create an object that contains both printing and evaluation. Oliveira and Cook [31] 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 [33, 36]. Unfortunately, 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, however, 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 [31], or type-unsafe meta-programming techniques of other approaches [33, 36] 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) a 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 [45, 25]. Figure 7 presents the syntax of the expressions, values and types provided by the features; each line is annotated with the corresponding feature. Starting from a simple arithmetic language, we gradually

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*
		$\mid$	$e_1 \,\&\&\, e_2 \mid e_1 \mid\mid e_2$	*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 7 Mini-JS expressions, values, and types

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 8 Overview of the languages assembled

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, if 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.

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. 8. 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 9 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

Feature	eval	print	check	Lang name	SEDEL	Haskell	% Reduced
natF(7)	23	7	39	`simplenat`	3	29	90%
boolF(4)	9	4	17	`simplebool`	3	12	75%
compF(4)	12	4	20	`natbool`	5	66	92%
logicF(4)	12	4	20	`varbool`	4	20	80%
varF(4)	7	4	7	`varnat`	4	37	89%
funcF(3)	10	3	9	`simplelogic`	4	24	83%
				`varlogic`	6	32	81%
				`arith`	8	86	91%
				`arithlogic`	8	106	92%
				`vararith`	8	99	92%
				`vararithlogic`	8	119	93%
				`mini-JS`	23	140	84%
Total			237		321	770	58%

Figure 9 SLOC statistics: SEDEL implementation vs vanilla AST implementation.

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 the feature and language parts (237 SLOC of all features plus 84 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 58% 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 almost twice SLOC ($40 = 7 + 23 + 7 + 3$) than that of Haskell (which has 29), we still get a huge gain when implementing other languages.

6 Related Work

Typed First-Class Classes/Mixins/Traits. First-class classes have been used in Racket [21], along with mixin support, and have shown great practical value. For example, DrRacket IDE [19] 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. [40] in Typed Racket. They designed a gradual type system that supports first-class

classes. Of particular interest is their use of row polymorphism [44] to type mixins. For example, `modal_mixin` from Section 2 implemented in Typed Racket has type:

```
(All (r / on-key toggle-mode)
     (Class ([on-key : (String → Void)] | r)) →
     (Class ([toggle-mode : (→ Void)] [on-key : (String → Void)] | r)))
```

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. While this is reasonable in the setting of mixins, in some cases, a function taking one class as an argument can return another class that has less methods. 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. In this sense, we argue disjoint polymorphism is more general than row polymorphism in terms of expressive power. As a consequence, Typed Racket ends up with two subtyping mechanisms: one for first-class classes (via row polymorphism) and the other for objects (via normal width subtyping). In contrast, SEDEL uses only one mechanism – disjoint polymorphism – to deal with both. More recently, Lee et al. [24] 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 [21]. 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. In the mixin model this is not a problem because conflicts are implicitly resolved using the order of composition. As far as we know, SEDEL is the first design for typed first-class traits.

Mixin-Based Inheritance. Mixins have become very popular in many OO languages [22, 10, 7]. Bracha and Cook's seminal paper [12] extends Modula-3 with mixins. 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 be resolved explicitly. We want to emphasize that this conflict detection is essential in expressing composition operators for Object Algebras, without running into ambiguities. Bracha's Jigsaw [11] formalized mixin composition, along with a rich trait algebra including merge, restrict, select, project, overriding and rename operators. Allen et al. [5] described how to add first-class generic types – including mixins – to OO languages with nominal typing such as Java. As such, classes and mixins, though enjoy static typing, are still second-class constructs, and thus their system cannot express dynamic inheritance. Bessai et al. [8] showed how to type classes and mixins with intersection types and Bracha-Cook's merge operator [12].

Trait-Based Inheritance. The seminal paper [38] by Schärli el al. introduced traits, where they also documented an implementation of traits in a dynamically typed version of Smalltalk. Since then many formalizations of traits have been proposed [20, 37, 16, 9]. Explicit conflict resolution is the hallmark of trait-based inheritance, compared with mixin-based inheritance. SEDEL differs from prior models of (second-class) traits in that they support *classes* in addition to traits, and consider the interaction between them, whereas in SEDEL the mechanism for

code reuse is purely traits. The deviation from traditional class-based models 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 [42] 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 slots 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 [26] 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 [27, 23] is an object-based language designed for education, where objects are created by *object constructors*. In that regard, Grace is similar to SEDEL in that both are not class-based. 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. For example, Grace imposes the constraint that the object being inherited must be *fresh*. The effect of the freshness constraint is that the expression in the inherit clause must generate a new object. As a consequence, the expression after `inherit` could not be a variable, which seems to preclude dynamic inheritance. Since SEDEL is pure, a relatively simple delegation-based 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*. Still the freshness constraint applies. 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.

Intersection types, Polymorphism and the 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 [17] 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. [32], 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 [6], where they proposed the notion of disjoint polymorphism. 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 [14].

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 fields and recursive types.

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 Eric E. Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2003.
6 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.
7 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam–designing a java extension with mixins. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003.
8 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.
9 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. A prototypical java-like language with records and traits. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, 2010.
10 Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999.
11 Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, 1992.
12 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1990.
13 William R. Cook. *Anatomy of Programming Languages*. 2013. URL: http://www.cs.utexas.edu/~wcook/anatomy/.
14 William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, 1989.
15 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.
16 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006.
17 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.
18 Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015. URL: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.
19 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.
20 Kathleen Fisher and John Reppy. A typed calculus of traits. In *Workshop on Foundations of Object-oriented Programming*, 2004.
21 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Programming Languages and Systems (APLAS)*, 2006.
22 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, 1998.
23 Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. Object inheritance without classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2016.
24 Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

25 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.

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

27 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.

28 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.

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

30 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.

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

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

33 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.

34 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.

35 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.

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

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

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

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

40 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.

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

42 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.

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

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

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