

Safe language support for feature composition through feature-based dispatch

Adriaan Moors, Jan Smans, Eddy Truyen, Frank Piessens and Wouter Joosen
Computer Science Department

KU Leuven

{adriaan, jans, eddy, frank, wouter}@cs.kuleuven.be

Extended Abstract

Language support for independent feature development and feature composition is a relatively young research topic, and the precise requirements of a feature composition operator are not yet fully understood.

Suppose we are developing a large software system to support a stock management business. The core functionality of such a system is simply adding items to the stock, removing items and retrieving an inventory of the actual stock - or a subset thereof. The essence of the business is based on charging fees for the core services, starting with charging for stocking items. Such a business model requires features for customer (user) management, for tracking the cost of manipulating the stock and for authenticating and billing clients that use our services. Furthermore, to avoid planning problems, such as one of our clients running out of stock, we support features that implement different planning and ordering strategies. Additional features could be: monitoring the availability of certain items in the stock, sending a notification when the availability reaches a certain level and so on.

In the context of the example, the need for different kinds of feature composition arises: some features should be activated only when certain clients use our application, some features must always be composed (eg. the authentication and billing features), other features may be composed dynamically (to change the ordering strategy, for example), some conflict (you may only select one ordering strategy) and again others are optional (the notification feature can only fully function when the monitoring feature is available, but the composition may still be legal when the notification feature is composed without the monitoring feature).

The above examples indicate some of the requirements of a composition operator. In many cases, and especially in the development of large software systems, the use cases for feature-composition can be summarised as follows: there are a number of core abstractions (objects such as a stock item, a client, the warehouse, ...), which may be enriched dynam-

ically by different features (billing, authentication, monitoring, ...) to cater for the needs of individual clients, without affecting the identity of those objects.

To elaborate, the identity of an object should not change when composed with a feature instance: a monitored stock item is still the same item. On the other hand, different clients accessing the same object often require the object to be composed with different sets of features, so there should be a mechanism to distinguish different views on the same object.

Furthermore, the exact set of features that are composed with a certain object is not known statically, but a lower and upper bound may be computed. A lower bound may be based on statically known dependencies whereas the upper bound is simply determined by the deployment of the application: the available set of features is typically fixed at that time.

We call this kind of composition “client-specific run-time customisation”. *Customisation*, because features may only be selected from a certain fixed set, *client-specific*, because it should be easy for clients to customise the same object with different features, depending on the preferences of the client (ie. the caller of a method) and *run-time*, because the exact set of features is determined at run time. This kind of composition is at the core of the Lasagne approach to feature oriented programming, and has been implemented in middleware, and in a Java-based programming language [7, 3, 2]. Technically, the composition is realised through feature-based method dispatch: the set of activated features travels as metadata with every client invocation and influences the method dispatch process.

The fact that lower and upper bounds of features are known statically makes it possible to design a static type system, and this is the main contribution of this paper. We present a formalisation of a small Java-like language that supports feature-based dispatch, we design a static type system for that language, and prove its soundness. To the best of our knowledge, this constitutes the first statically checked language construct to support dynamic, client-specific feature composition.

To illustrate our results, two appendices are attached to this extended abstract. In the first appendix, we illustrate our language informally by means of a simple example application. In the second appendix, we formally define the syntax of the language, and describe the essence of the operational semantics and the static type system.

1. REFERENCES

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MVCDC 2005 San Diego, California USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

- [1] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
- [2] Bo Nørregaard Jørgense and Eddy Truyen. The Lasagne/J homepage, 2005. <http://www.lasagnej.org/>.
- [3] Bo Nørregaard Jørgensen. Language support for incremental integration of independently developed components in java. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1316–1322, New York, NY, USA, 2004. ACM Press.
- [4] Gunter Kniesel. Type-safe delegation for dynamic component adaptation. In *ECOOP Workshops*, pages 136–137, 1998.
- [5] Adriaan Moors, Jan Smans, Eddy Truyen, and Frank Piessens. Support for feature-based dispatch: extending a java-like language. Technical report, Computer Science Department, KU Leuven, May 2005. Draft version available from <http://www.cs.kuleuven.be/~adriaan/report/>.
- [6] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.
- [7] Eddy Truyen, Bart Vanhaute, Bo Nørregaard Jørgensen, Wouter Joosen, and Pierre Verbaeten. Dynamic and selective combination of extensions in component-based applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 233–242, Washington, DC, USA, 2001. IEEE Computer Society.

APPENDIX

A. EXAMPLES

In this appendix we will illustrate our language using a simple example. We consider a program to be a composition of features: a core feature and a set of refining features. The program of fig. 1, a model of a library, contains three such features: a *core* feature called `DigitalLibrary`, which provides the basic abstractions needed to model a library, and two *wrapping* features, which enrich the core with a rating and a billing system. Feature names – also called *feature identifiers* – must be unique.

The core feature, `DigitalLibrary`, consists of two core interfaces that model the basic abstractions: `BookCopy` and `BookManager`. These interfaces are implemented by `BookCopyImpl` and `BookManagerImpl`. Wrapping features, such as `Rating` and `Billing`, consist of interfaces that expect an interface from the feature they refine. An interface that expects another interface is similar to a Java-interface that extends another interface. In addition, this *expects* relation guides class-composition: in our example, `RatedBookCopyImpl`, which implements the `RatedBookCopy` interface, may be composed with `BookCopyImpl` since the latter implements `BookCopy`, which is expected by `RatedBookCopy`. We call `RatedBookCopyImpl` the “wrapper” and `BookCopyImpl` the “wrappee”.

A wrapper may extend the wrappee in two ways: (1) It

```

collaboration DigitalLibrary {
  interface BookCopy {
    String getTitle();
  }
  interface BookManager {
    List searchBooks(Query q);
  }

  class BookCopyImpl
    implements BookCopy {
    String name;
    String getName() {
      this.name
    }
  }
  class BookManagerImpl
    implements BookManager {
    List searchBooks(Query q) {
      // query and return list
    }
  }
}

```

Figure 2: The `DigitalLibrary` feature (the core)

may provide additional methods. As shown in fig. 2, `RatedBookCopy` offers an additional method called `getRating`. (2) It may override a number of the wrappee’s methods. The overriding method can call the overridden method using `inner`, much like `super`-calls in Java. For instance, `RatedBookManagerImpl` overrides the method `search` in order to return a rating-sorted list. See fig. 3.

We strive to enforce encapsulation by separating a type and its implementations, but we will not discuss this in more detail¹. Without loss of generality, we assume a one-to-one relation between an interface and the class that implements it. A combination of an interface and the class that implements it, may tentatively be thought of as a Java class, except that object-based inheritance is used [4].

Static composition selects the set of deployed features and maps each core interface to a class composition that implements the functionality of the deployed features. Classes – to be thought of as mixins – are composed using the mixin-like operator `+`, according to the structure imposed by the *expects*-relation. For example, a rated, billed book is an instance of the composition `BookCopyImpl + RatedBookCopyImpl + BilledBookCopyImpl`. The expected interface of a class must be implemented by a class that occurs to the left of that class (in the composition expression). For example, `RatedBookCopyImpl` is composed after (to the right of) `BookCopyImpl` as `RatedBookCopy` expects `BookCopy`. Methods override synonymous methods in classes that are specified to the left in the composition.

An object is an instance of a *complete composition of classes*, but this is shielded from the programmer by the mapping defined by the deployment. Thus, a programmer may write `new BookCopy` and, based on the deployment in fig. 4, an instance of `BookCopyImpl + RatedBookCopyImpl`

¹Fundamentally, interfaces define types and a class is purely a way to implement one or more interfaces. One interface may have different implementations, which are interchangeable without affecting the typing of a program.

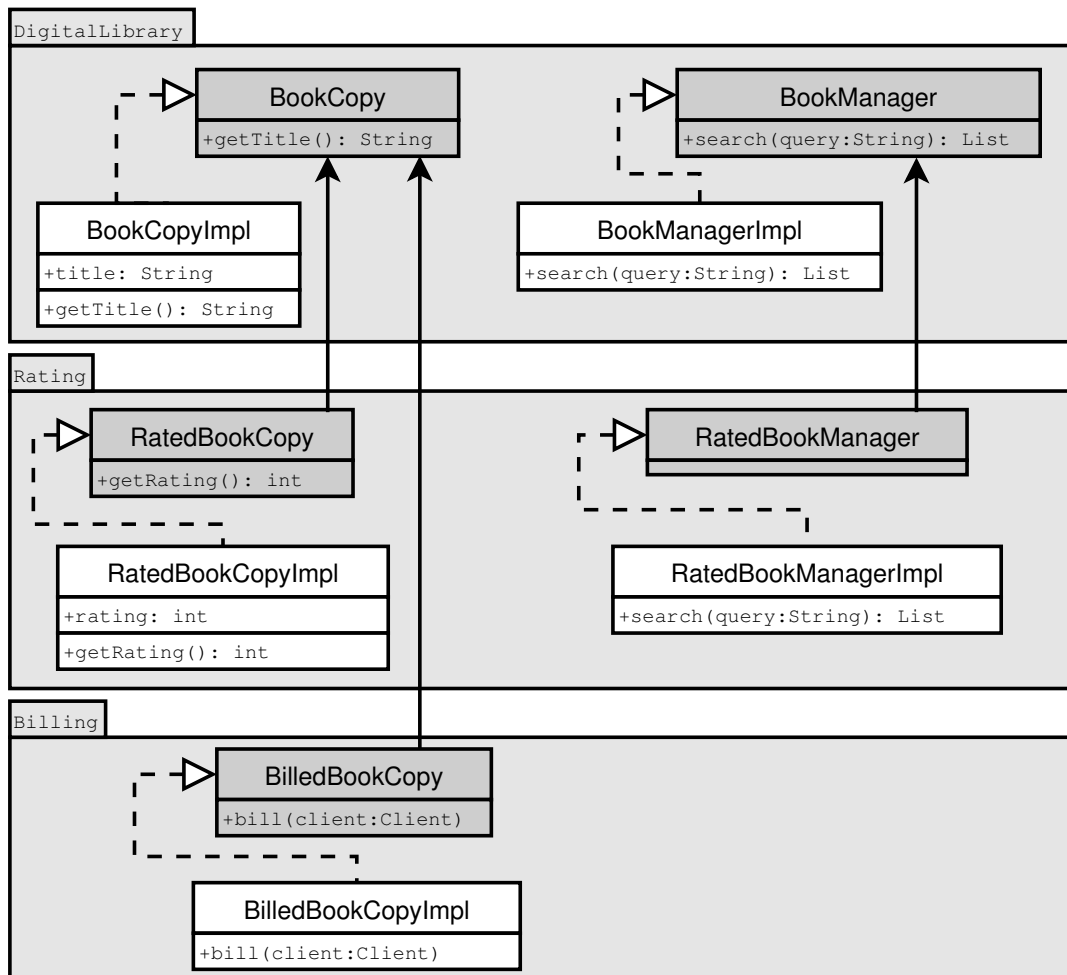


Figure 1: A model for a digital library with two features: **Rating** and **Billing**. The dashed arrows denote the *implements*-relation; the full arrows the *expects*-relation.

+ `BilledBookCopyImpl` will be created. This allows classes to remain private to a feature and the deployment, which increases the potential for reuse.

A.1 Consistency

When a client creates a new book manager (see the main-method in fig. 4) and then searches for a certain book, only the `DigitalLibrary` feature will be active by default. Now, if the client activates `Rating` on a reference to the book manager, a call to `search` on that reference will sort the results according to the rating (this behaviour is implemented in `RatedBookManagerImpl`). During the execution of this search method, the book manager will have to call the `getRating` method on the book copies it searches. At the level of the type system this means that a method in the `Rating`-feature in one class should be allowed to call a method from the `Rating`-feature on another object. Because of this, the programmer does not have to insert explicit casts to be able to use a feature that is statically known to be activated. To ensure that the necessary features will always be activated at run time, the set of active features is propagated automatically.

This propagation is achieved as follows: whenever a method is called on a reference f , the set of features that are acti-

vated on f or on the current `this`-reference are also activated on the new `this`-reference that is seen by the body of the called method. In other words, the set of features that are activated on the new `this`-reference corresponds to the union of the features of f and those of the old `this`-reference.

A.2 Preventing Inconsistencies

It is clear that removing features from the set of active features of a reference may lead to inconsistencies. What is not immediately obvious however, is that even *activating* features may cause inconsistencies: suppose a feature f is activated by a feature g in the middle of the control flow of a method, causing the next method that will be called to be handled by a method in the newly activated feature f . If this method assumes that the feature it is defined in has been active since the beginning of the current control flow, it may fail since this is not the case. To avoid this, feature activation must be restricted to occur only on clear boundaries between different systems. For now, we only consider two systems: the system (think of the application as a service running on a server) and the main method (the client that uses the service). Hence, feature activation is restricted to the main method, as this is the only place we can statically distinguish as “on the border”.

```

collaboration Rating{
  interface RatedBookCopy
    expects BookCopy{
      int getRating();
    }
  interface RatedBookManager
    expects BookManager{}

  class RatedBookCopyImpl
    implements RatedBookCopy{
      int rating;
      int getRating(){
        this.rating
      }
    }
  class RatedBookManagerImpl
    implements RatedBookManager{
      List searchBooks(Query q){
        let booklist = inner(q)
        in booklist.sort(...)
      }
    }
}

```

Figure 3: The Rating feature

```

deploy {
  BookCopy = BookCopyImpl +
    RatedBookCopyImpl +
    BilledBookCopyImpl;
  BookManager = BookManagerImpl +
    RatedBookManagerImpl;
}

// example main method (typically developed
// at the same time the deployment is
// determined)
main {
  BookManager mgr = new BookManager;

  // details omitted

  List results = mgr.searchBooks(
    someQuery);

  List sortedResults =
    mgr@Rating.searchBooks(someQuery);
}

```

Figure 4: Deployment (The Billing feature is not shown, it is included in the deployment for completeness.)

B. FORMALISATION OF FEATURE-BASED DISPATCH IN JAVA

We are working on a full formalisation of a subset of Lasagne/J, based on ClassicJava [1]. The current draft is available as a technical report [5]. In this appendix, we highlight the aspects that are relevant to feature-based dispatch.

We deviate from the ClassicJava approach by not using type elaboration in the static semantics. Instead, we rely on an implied preprocessor tool. In ClassicJava, the dynamic semantics uses term rewriting to reduce an initial expression, which represents the program, to a value, the result of the program. The only information that is used in this reduction is the expression to be evaluated and the store, which models the heap. This allows for a straightforward formulation of the dynamic semantics, but it requires expressions to carry information that is redundant in the context of the static semantics, but essential to the dynamic semantics.

Type elaboration is thus normally employed to enrich the expressions written by the programmer, so that they contain the information needed by the dynamic semantics. Although understanding the extra information is quite easy, the actual elaboration complicates the static semantics. We decided to assume the programmer (or a simple preprocessor tool) writes the already elaborated expressions. For example, an inner-call that occurs in the method `foo`, should be written as `this.inner@foo(bar)` instead of `inner(bar)`, because otherwise, the operational semantics cannot derive the enclosing method (`foo`) from the expression it is rewriting, nor does it have a separate modelling of the `this`-pointer, since we do not model stack frames. The enclosing method and the current object are clearly needed to evaluate the inner-call, however.

B.1 Grammar

Figures 5 and 6 describe the context-free grammar and the meta-variables used throughout this paper. In essence, a program consists of a number of collaborations, the deployment and finally a main-expression. Collaborations consist of interface and class definitions. An interface is defined by the methods it declares and it may also specify an expected interface. A class specifies a number of fields in addition to implementations for methods declared in the interfaces it implements. Finally, the deployment maps each core interface to its corresponding class composition.

Note that this grammar was devised as a vehicle to convey our formalisation, the full language's grammar might, for example, provide syntactic sugar to hide the difference between classes and interfaces.

B.2 Essence of the Static Semantics

B.2.1 Type structure

A feature-dependent type is a natural way of typing a feature-annotated reference: it consists of a set of features and a simple type. For example, a reference to a book that has `Rating` enabled, would be typed as $\{DigitalLibrary, Rating\}.BookCopy$. In general such a type has the form $F.T$, where F is a well-formed set of features and T is a well-formed type. Feature-dependent types are a lightweight kind of dependent type that do not depend on a value. Therefore, they do not induce the same type system complexities as full dependent types.

| | | |
|-------------------|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>program</i> | = | collab_def* deployment main { e } |
| <i>collab_def</i> | = | collaboration f { if_def* class_def* } |
| <i>if_def</i> | = | interface I [expects I] { md_decl* } |
| <i>md_decl</i> | = | <i>F.T</i> md(arg*) |
| <i>arg</i> | = | <i>F.T</i> var |
| <i>class_def</i> | = | class C implements T { fd_def* md_def* } |
| <i>fd_def</i> | = | <i>F.T</i> fd |
| <i>md_def</i> | = | md_decl { e } |
| <i>e</i> | = | new <i>F.I</i> <u>this</u> .inner.md@C(<i>e</i> *) e.md(<i>e</i> *) e@f var null let vn = e in e this :C .fd this :C .fd = e |
| <i>F</i> | = | { f* } |
| <i>var</i> | = | vn this |
| <i>deployment</i> | = | deploy { type_def* } |
| <i>type_def</i> | = | I = C+...+C |

Figure 5: Context-free grammar (Underlined parts are generated by a preprocessor.)

| | |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <i>vn, fd, md</i> | variable-, field- and methodnames |
| <i>f</i> | a feature identifier |
| <i>I</i> | an interface name, which represents a simple type |
| <i>T</i> | a simple type or a new type constructed from simple types (not discussed in this paper) |
| <i>F.T</i> | includes all types which <i>T</i> stands for, plus feature-dependent types |
| <i>T_{meth}</i> | a method-type |
| <i>C</i> | a classname |
| <i>C⁺</i> | a classname (<i>C</i>) or a class composition (such as <i>C₁+C₂</i>) |

Figure 6: The meta-variables and their meaning

Subtyping is standard for interfaces; for feature-dependent types, it corresponds to the superset relation on the set of associated features. More precisely, $F.T$ is a subtype of $F'.T'$ if $F \supseteq F'$ and $T \leq T'$.

B.2.2 Dependencies

When an interface I expects an interface defined in a feature f , I depends on f . An interface also depends on the feature it is defined in. An implementation of a method defined in an interface I may only be executed if all I 's dependencies are in the composition policy. This allows us to statically determine a lower-bound for the composition policy, which is used to type **this** as a feature-dependent type in a method body.

For example, in `RatedBookCopyImpl` from fig. 3, **this** is typed as $\{DigitalLibrary, Rating\}.BookCopy$.

B.2.3 Class composition

A class composition is represented as a concatenation of classnames. For example, $A+B+C$ denotes the composition of three classes, where a method in C overrides methods in A and B that have the same names, and so on. This is quite similar to the mixin-composition of class templates in the ν Obj-calculus [6], except that our calculus preserves the structure of the composition².

Method dispatch “intersects” the class composition with the composition policy to derive the maximal class composition that solely consists of classes that are activated by the composition policy, that is, whose dependencies are a subset of the composition policy. We write this as $C^+ \cap cp$, for example: $BookCopyImpl + RatedBookCopyImpl + BilledBookCopyImpl \cap \{DigitalLibrary, Rating\} = BookCopyImpl + RatedBookCopyImpl$.

B.2.4 Well-formedness

A well-formed program must satisfy a number of predicates which are similar to the ones used in ClassicJava [1]. An important check is the well-formedness of the deployment construct, which must map types to complete implementations of that type. For simplicity, we assume that classes, interfaces and collaborations have unique names.

A set of features is well-formed if it does not have any unresolved dependencies. A type is well-formed if its dependencies are well-formed and if, for every method declaration in that type holds that methods with equal names have equal signatures. Furthermore, we statically prevent “accidental overriding” [4].

B.2.5 Expression Typing

Figure 7 lists the typing rules for expressions. We use different kinds of judgements, of which two are relevant to the subset presented here: $P, \Gamma, F \vdash_e$ and $P, \Gamma, F \vdash_{sub}$. The former types an expression in the context of the program P , the typing of variables Γ and the statically inferred lower-bound of active features F and the latter does the same up to subsumption.

CALL A method call is typed by the return type of the specified method, if the actual arguments’ types conform to the formal arguments’ and if the method is declared in the type $(F^t \cup F).T^t$, where the target expression

²Another difference with ν Obj is that a class-composition is not a value, but this is only a simplification.

| |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\frac{\begin{array}{l} P, \Gamma, F \vdash_e e_t : F^t.T^t \\ \forall j \in [1, n] : P, \Gamma, F \vdash_{sub} e_j : F_j.T_j \\ \langle md, (F_1.T_1 \dots F_n.T_n \longrightarrow F^r.T^r) \rangle \in (F^t \cup F).T^t \end{array}}{P, \Gamma, F \vdash_e e_t.md(e_1 \dots e_n) : F^r.T^r} \text{CALL}$ |
| $\frac{\begin{array}{l} P, \Gamma, F \vdash_{sub} e_j : F_j.T_j \quad (j = 1 \dots n) \\ P, \Gamma, F \vdash_e \mathbf{this} : T^c \quad T^c \text{ expects } T' \\ \langle md, (F_1.T_1 \dots F_n.T_n \longrightarrow F^r.T^r) \rangle \in T' \end{array}}{P, \Gamma, F \vdash_e \mathbf{this}.\mathbf{inner}.md@C(e_1 \dots e_n) : F^r.T^r} \text{INNER}$ |
| $\frac{\begin{array}{l} P, \Gamma, F \vdash_e e : F'.T' \\ F'' = dependenciesOf(f) \quad P \vdash_t (F'' \cup F').T' \end{array}}{P, \Gamma, F \vdash_e e@f : (F'' \cup F').T'} \text{FEATACT}$ |

Figure 7: Expression-typing

| |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle md, (T_{meth}), (var_1, \dots, var_m), e \rangle \in_{\square}^{rt} C_1 + \dots + C_n \Leftrightarrow$ $\left\{ \begin{array}{l} \langle md, (T_{meth}), (var_1, \dots, var_m), e \rangle \in C_i \wedge \\ \forall j \in [i+1, n] : \langle md, (T'_{meth}), (var'_1, \dots, var'_m), e' \rangle \notin C_j \end{array} \right.$ $\langle md, (T_{meth}), (var_1, \dots, var_m), e \rangle \in_{C_s}^{rt} C_1 \dots + C_i + C_s \dots$ $\Leftrightarrow \langle md, (T_{meth}), (var_1, \dots, var_m), e \rangle \in_{\square}^{rt} C_1 + \dots + C_i$ |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Figure 8: Method lookup: the \in^{rt} -relation.

has type $F^t.T^t$ and the features in F are statically known to be active. F is thus a lower bound for the run-time composition policy when this call-expression is evaluated. Similarly, $F^t \cup F$ is a lower bound for the composition policy that will apply when the called method's body is evaluated.

The premise $\langle md, (F_1.T_1 \dots F_n.T_n \longrightarrow F^r.T^r) \rangle \in (F^t \cup F).T^t$ states that a method named md , with formal argument types $F_1.T_1 \dots F_n.T_n$ and return type $F^r.T^r$ must be declared in the type $(F^t \cup F).T^t$. This means that the method must occur lexically in an interface that is a supertype of that type. In the example from fig. 2 and 3, $\langle getRating, (int) \rangle \in \{Rating, DigitalLibrary\}.BookCopy$, since the latter is a subtype of $RatedBookCopy$, which directly declares `int getRating()`. As an informal indication of the soundness of our approach, the statically known composition policy is a lower bound of the run-time composition policy because:

- method dispatch will never select this method unless the actual composition policy of the caller is a superset of the F we use here – the statically known composition policy consists of the dependencies of the class that defines this method –
- the statically known F^t can only result from feature-activation, which will never activate fewer features at run time and
- adding features (at run time) will never cause unsoundness, since we restrict synonymous methods

to have exactly the same signature within a certain type hierarchy.

INNER types an inner-call, which is similar to a regular method call, except that the method must be declared in the expected interface. This is modeled by the premises $P, \Gamma, F \vdash_e \mathbf{this} : T^c$ and T^c expects T' , which ensure that T' is a type expected by the current class³, and $\langle md, (F_1.T_1 \dots F_n.T_n \longrightarrow F^r.T^r) \rangle \in T'$, which enforces that the method is indeed defined in T' .

Note that the preprocessor tool has generated extra information that is incorporated in the expression: the target of the call (always **this**), the method to be called (md) and the class in which this expression occurs (C). This information is used by the evaluation rule OS-INNER, that evaluates inner-calls.

FEATACT types feature-activation. If a feature f is activated on an expression with type $F'.T'$, the type of the whole is $(dependenciesOf(f) \cup F').T'$. To preserve modularity, feature activation may only be used in the main-expression, so that it is possible to check whether the activated feature is actually deployed.

This is not a severe limitation, since the main use case for feature-based dispatch is client code (represented by the main-expression here) that activates a feature when calling a certain method of the software system. This is related to our notion of consistency, which requires that a control flow ‘within the system’ – ie. from the point where it leaves the main-expression until it returns – does not change the composition policy.

B.3 Essence of the Dynamic Semantics

The operational semantics is specified as a reduction relation in a fairly standard way. There are three noteworthy aspects:

- An object reference consists of 2 parts: an object-identifier *objectid* – “the pointer to the object in the heap” – and a set of features F – the minimal set of features that has to be activated when a method is called on the object reference.
- The set of features of the **this**-reference, explicitly modeled as part of the execution state, is propagated with each method call.
- Method dispatch takes the currently active set of features – the composition policy – into account.

B.3.1 Execution state

The current execution state is represented by three components: the expression being evaluated, the store and the current composition policy. The first two are standard, the latter is specific to our extension. The composition policy is the set of currently active features, which can be thought of as the set of features associated to the **this**-reference. However, we have to model the composition policy separately, since the dynamic semantics has no way of knowing the value of the **this**-reference.

The *store* maps object-identifiers to records specifying the class-composition of the object and the values of its fields. More precisely, the store is a function S that takes

³Since synonymous methods must have exactly the same signatures if they are defined in types that are related by subtyping, it doesn't matter which type is chosen.

| |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $ \begin{aligned} e &= \dots \mid \langle \text{object} \mid F \rangle \mid e : \underline{C} . \text{fd} \mid e : \underline{C} . \text{fd} = e \\ &\quad \mid e . \text{inner.md} @ \underline{C}(e^*) \mid e . \text{md}(e^*) \\ &\quad \mid \text{return} (cp, e) \\ v &= \langle \text{object} \mid F \rangle \mid \text{null} \end{aligned} $ |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Figure 9: Description of the syntactic forms that arise only during evaluation.

an *objectid* to a record $\langle C^+, \mathcal{F} \rangle$ describing the corresponding object. \mathcal{F} is a function that maps a field, denoted by a (C, fd) -pair, to its value. A *value* is either an object-reference or the **null** reference. An *object-reference* consists of an object-identifier *objectid* and a set of features F .

During evaluation, expressions deviate from the surface syntax. For example, they may contain object references. Hence, we need a slightly extended grammar to represent the expression being evaluated. The extensions are shown in fig. 9.

B.3.2 Evaluation Rules

For brevity, we only discuss the rules that are relevant to feature-based dispatch and we omit the rules for evaluations that result in an error. The selected rules, that – together with the omitted ones – define the reduction relation, are shown in fig. 10.

OS-CALL reifies the call stack by reducing a method call to a return expression that contains the correct method body-expression – **this** and the arguments substituted by the corresponding values – and the current composition policy. It sets up the new composition policy so that the body-expression is evaluated with the correct set of active features. The evaluation of the return expression will result in the evaluation of the body-expression and will reset the composition policy to the saved one. This makes explicit that the composition policy is only propagated in the “call-direction” of the control flow and not in the “return-direction”.

The \in^{rt} -relation is used to look up the method body in $C^+ \cap cp''$, the “intersection” of the class-composition and the composition policy cp'' : the maximal composition of the classes in C^+ that are activated by cp'' . The \in^{rt} -relation is a straightforward lookup of the most overridden method, ie. the method defined directly in the right-most class in the class-composition. The lookup (fig. 8) has remained simple because we only model wrapping.

OS-INNER is quite similar to OS-CALL, except that the lookup described above, is strengthened to the right-most class *to the left* of the class in which the inner call occurred.

OS-RETURN reduces a return statement whose return expression is a value, to that value. The composition policy is reset to the stored one.

OS-FEATACT specifies how feature activation changes an object reference’s set of features. Activating a feature f on a reference yields a new reference with a set of active features that is the union of the old reference’s active features and f ’s dependencies.

C. SUMMARY OF THE META-THEORY

Below, we have included the formulation of the most interesting definitions and theorems. We refer to [5] for outlines of the proofs. Put very briefly, they state preservation and progress: evaluation preserves a program’s typing (up to subsumption) and for every program state $\langle e, cp, S \rangle$ of P , e is either an answer (a value or an error) or an expression that can be further evaluated.

DEFINITION 1 (STORE CONSISTENCY). *The store is consistent if the fields of every object it contains, correspond to their definition in the class-composition of the object and if every non-null value of every field references an object in the store. See fig. 11.*

THEOREM 1 (PRESERVATION). *Evaluation preserves a program’s typing (up to subsumption). If an expression has type $F.T$ relative to a context Γ and a consistent store S , performing one evaluation-step will result in an expression that has the same type (up to subsumption) and a consistent store.*

$$\begin{aligned}
P, \Gamma, S_n \vdash_{\text{sub}} e_n : F.T \wedge P \vdash_{\sigma} S_n \wedge \langle e_n, cp, S_n \rangle \\
\hookrightarrow \langle e_{n+1}, cp, S_{n+1} \rangle \\
\implies \begin{cases} P, \Gamma, S_{n+1} \vdash_{\text{sub}} e_{n+1} : F.T \\ P \vdash_{\sigma} S_{n+1} \end{cases} \\
\text{or } e_{n+1} \text{ is an error.}
\end{aligned}$$

DEFINITION 2 (FINAL STATE). *A program state $\langle e, cp, S \rangle$ is final if e is a value (null or $\langle \text{object} \mid cp \rangle$) or e is an error*

THEOREM 2 (PROGRESS). *If $\langle e, cp, S \rangle$ is a non-final program state of P such that*

$$P \vdash_{\sigma} S \wedge \exists \Gamma : P, \Gamma, S \vdash_{\text{sub}} e : F.T$$

then

$$\exists \langle e', cp, S' \rangle : \langle e, cp, S \rangle \hookrightarrow \langle e', cp, S' \rangle$$

$$\begin{array}{c}
\frac{S(\text{object}) = \langle C^+, _ \rangle \quad \langle \text{md}, (_), (\text{var}_1, \dots, \text{var}_n), e \rangle \in_{\square}^{rt} C^+ \cap cp \quad cp = cp^c \cup cp^t}{\begin{array}{l} \vdash \langle E [\langle \text{object} | cp^t \rangle . \text{md}(v_1, \dots, v_n)], cp^c, S \rangle \\ \hookrightarrow \langle E [\text{return } (cp^c, [\langle \text{object} | cp \rangle / \text{this}, v_1 / \text{var}_1, \dots, v_n / \text{var}_n] e)], cp, S \rangle \end{array}} \text{OS-CALL} \\
\\
\frac{S(\text{object}) = \langle C^+, _ \rangle \quad \langle \text{md}, (_), (\text{var}_1, \dots, \text{var}_n), e \rangle \in_C^{rt} C^+ \cap cp \quad cp = cp^c \cup cp^t}{\begin{array}{l} \vdash \langle E [\langle \text{object} | cp^t \rangle . \text{inner.md}@C(v_1, \dots, v_n)], cp^c, S \rangle \\ \hookrightarrow \langle E [\text{return } (cp^c, [\langle \text{object} | cp \rangle / \text{this}, v_1 / \text{var}_1, \dots, v_n / \text{var}_n] e)], cp, S \rangle \end{array}} \text{OS-INNER} \\
\\
\frac{}{\vdash \langle E [\text{return } (cp, v)], _, S \rangle \hookrightarrow \langle E [v], cp, S \rangle} \text{OS-RETURN} \\
\\
\frac{F' = F \cup \text{dependenciesOf}(f)}{\vdash \langle E [\langle \text{object} | F \rangle @f], cp, S \rangle \hookrightarrow \langle E [\langle \text{object} | F' \rangle], cp, S \rangle} \text{OS-FEATACT}
\end{array}$$

Figure 10: Essence of the operational semantics

$$\begin{array}{l}
P \vdash_{\sigma} S \\
\Leftrightarrow \\
(\quad S(\text{object}) = \langle C_1 + \dots + C_n, \mathcal{F} \rangle \\
\Rightarrow \\
\Sigma_1 \quad \text{dom}(\mathcal{F}) = \cup_{i \in \{1..n\}} \{ \langle C_i, fd \rangle \mid \langle fd, F.T \rangle \in C_i \} \\
\Sigma_2 \quad \text{and } \text{ref2id}(\text{rng}(\mathcal{F})) \subseteq \text{dom}(S) \cup \{ \text{null} \} \\
\Sigma_3 \quad \text{and } \forall \langle C_i, fd \rangle \in \text{dom}(\mathcal{F}) : \\
\quad (\mathcal{F}(C_i, fd) = \langle \text{object}' | F' \rangle \wedge \langle fd, F.T \rangle \in C_i) \Rightarrow P, [], S \vdash_{\text{sub}} \langle \text{object}' | F' \rangle : F.T \\
)
\end{array}$$

Figure 11: Store consistency ($\text{ref2id}(\langle \text{objectid} | F \rangle) = \text{objectid}$)