Constructing Type-Safe Operators for Software Composition*

Axel Rauschmayer, Andreas Abel, Alexander Knapp, Martin Wirsing

Institut für Informatik
Ludwig-Maximilians-Universität München
{rauschma,abel,knapp,wirsing}@informatik.uni-muenchen.de

Abstract. Grey-box software composition appears in many areas of software engineering: Mixin layers, aspects and traits are just a few examples and even regular inheritance can be viewed as composition. Operators used for these kinds of composition share two deficiencies: (1) They are monolithic and are thus paradoxically supporting reuse without being implemented in a reusable fashion. (2) There is usually no type system that allows us to check if a composition is legal *before* performing it. In this publication, we introduce GRAFT, a small language that aims to "decompose composition" by providing atomic commands for software manipulation that can be combined to express powerful composition operators. Applicability of atomic and composite commands is checked for by a type system. We demonstrate and motivate GRAFT's use by looking at mixin layer composition which is fully expressible in GRAFT.

1 Introduction

Software composition is one of the central mechanisms used in software engineering to enable reuse and structuring in a complex system. Large-scale software composition mainly uses a black box approach, whereas smaller-scale composition usually favors grey boxes as atomic components. With all the hype surrounding large-scale, black-box components such as Enterprise JavaBeans or .NET components, one would think that nobody uses grey-box composition any more. But that is not true: It is the underlying principle of many lower-level reuse infrastructures. Examples are *mixin layers* [Smaragdakis and Batory, 1998] (which we'll take a look at in a moment), *aspects* [Kiczales et al., 2001] and *traits* [Schärli et al., 2003]. Even regular inheritance can be viewed as grey-box composition. Each of these reuse infrastructures comes with its own set of operators for performing composition. This is where we get to a paradox: Even though these operators *enable* software composition, they are not, themselves, implemented in a reusable fashion. This would make a lot of sense, though, because most of these operators are very similar and adhere to certain basic principles. Additionally, there is often a need for many variations in performing composition even

* Supported by Deutsche Forschungsgemeinschaft (DFG) project WI 841/6-1 "InOpSys"

within the same environment. Being able to express these generically would be very helpful.

Our approach is to "decompose composition": We want to find out if composition is really a monolithic operator or rather a composite of more basic operators. If the latter is true and we want to prove that a certain property holds for any (legal) composition operator, we only need to prove the property for the atomic operators and for the operator combination mechanism. In Sect. 2, we present one possible set of atomic manipulation operators, a language we call GRAFT. We also show how GRAFT can be used to express mixin layer composition. In Sect. 3, we introduce a type system that checks for applicability of any combination of operators and whether such combinations are legal. This allows us to generically express and type grey-box software composition. Sect. 4 gives a more formal explanation of GRAFT's type system. The last two sections, 5 and 6, present related work and the conclusions of this document. Consult App. A for a formal specification of the GRAFT language.

2 Finding Operators for Mixin Layer Composition

To find out more about the issues involved in software composition, we start by looking at one approach to object-oriented software composition: *mixin layers*. Fig. 1(a), 1(b) and 1(c) introduce our running example: a system consisting of the program `base` and the mixin layers `undo` and `subonly`. A first clue about composition is that applying a layer to a program can be viewed as applying a function to a constant: Take `base` (Fig. 1(a)) as our base program with the single class `Calculator`. Then layer `undo` (Fig. 1(b)) is a "function" that, if applied to the "constant" `base` produces the new program `undoCalc` = `undo`(`base`) with an incremented version of `Calculator`: It now contains the new field `savedMemory` and the new method `undo` and a statement has been prepended to method `add`. Layer `subonly` (Fig. 1(c)) removes one method and adds another one.

Is there a way to create some kind of programming language to express the changes performed by a mixin layer? We first need to find out what kind of data is to be manipulated by such a programming language. This will hopefully lead us to a better understanding of what operators make sense. How can an object-oriented program in general and a base program in particular be represented in an elegant data structure? Looking at a diagram of `base` (Fig. 2(a)) it is fairly obvious that it is a labeled tree. We often leave the root unlabeled and annotate the arrows with labels because this reflects the data structure for trees that we'll introduce later. Generally, at every level of encapsulation in an object-oriented program, there is an entity that contains a set of named entities: A class contains a set of methods, a package contains a set of classes etc. (Fig. 2(b)). We call sets of entities *collectives* (set Coll) and all entities (including collectives) *units*[1] (set $\mathsf{Unit} \supset \mathsf{Coll}$). Another way to look at a labeled tree is as nested records. Then a collective is a record and each of the collective's members is a field in that

[1] We use this term differently from the *unit type* in type theory!

```
prog base;
class Calculator {
  Int memory;
  Calculator() {
    this.memory = new Int();
  }
  void add(Int i) {
    this.memory.add(i);
  }
}
```

(a) Program `base` provides the initial version of class `Calculator`. `Calculator` contains an internal `memory` and can `add()` values of type `Int` to it.

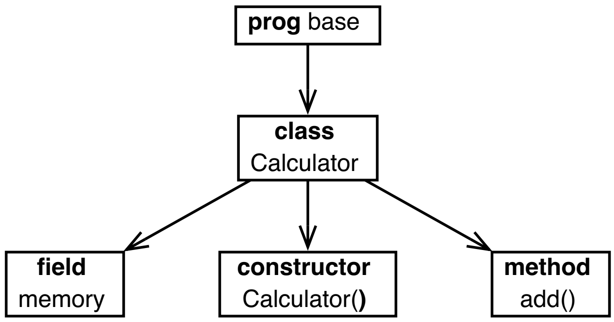
```
layer undo;
class Calculator {
  Int savedMemory;
  @before void add(Int i) {
    this.savedMemory = this.memory;
  }
  void undo() {
    if (this.savedMemory == null) { } // Exception
    this.memory = this.savedMemory;
    this.savedMemory = null;
  }
}
```

(b) Layer `undo` introduces the new member variable `savedMemory` to remember the last value of the `memory` and method `undo()` to restore this value. Additionally, we prepend code to method `add()` to make sure we save the state of `memory` before changing it. Note that we are using the JSR-175-style attribute [Bloch et al., 2002] `@before` to denote that method `add` is to be prepended to an existing method with the same name.

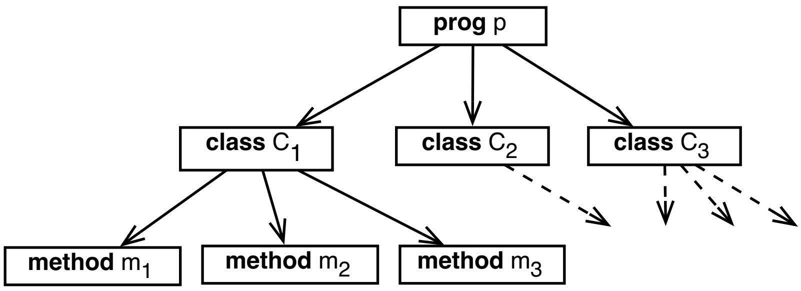
```
layer subonly;
class Calculator {
  @remove add(Int i);
  void sub(Int i) {
    this.memory.sub(i);
  }
}
```

(c) Layer `subonly` removes addition from the calculator and introduces subtraction. We are using the attribute `@remove` to denote that an existing method `add` has to be removed.

Fig. 1. Running example.



(a) Looking at the structure of program `base`, it is obvious that it is a tree.



(b) An object-oriented program p comprises a set of classes which in turn are made up of methods. Representing the system as a labeled tree follows naturally from its structure: The root node has label p, its children the labels C_1, C_2, C_3 etc.

Fig. 2. The structure of object-oriented software.

record. We can now draw on well-established ideas from database theory, more specifically, from tuple calculus to help us with the design of a language that operates on our data structure. Any kind of change that a mixin layer performs can be expressed as a sequence of the following operations; think of using a catalog of comparatively small refactoring steps for arbitrary transformations of a program [Fowler, 1999].

- $\mathsf{insert} : \mathsf{Coll} \times \mathsf{Name} \times \mathsf{Unit} \rightarrow \mathsf{Coll}$ adds a named unit to a collective.
- $\mathsf{delete} : \mathsf{Coll} \times \mathsf{Name} \rightarrow \mathsf{Coll}$ removes a unit (whose name is given by a parameter) from a collective.
- $\mathsf{project} : \mathsf{Coll} \times \mathsf{Name} \rightarrow \mathsf{Unit}$ extracts a unit (whose name is given by a parameter) from a collective.

Let's use these operators to describe how layer `undo` increments program `base` by the before-method `add`: We first have to project method `add` out of the program and prepend a new statement to its body. Then we project class `Calculator`, delete the old version of `add` and insert the new one. The result has to replace the old `Calculator` in `base` in the same fashion. We define $\mathsf{update} : \mathsf{Coll} \times \mathsf{Name} \times \mathsf{Unit} \to \mathsf{Coll}$ to be a combination of delete and insert: $\mathsf{update}(v, n, w) := \mathsf{insert}(\mathsf{delete}(v, n), n, w)$. We also write projection as the infix '.' (dot) operator and thus get the following meta program.

```
1  update(v, Calculator,
2      update(v.Calculator, add,
3          insert(v.Calculator.add, @before,
4              this.savedMemory = this.memory;)))
```

This is what the code does (we explain the lines bottom-up):

- Lines 3 and 4: v`.Calculator.add` means: Project method `add` out of class `Calculator` out of the meta[2] constant[3] v (which holds the complete code of program `base`). Then insert the statement "`this.savedMemory = this.memory;`" before the (body of) the result of this projection.
- Line 2: Project class `Calculator` out of v and replace the existing method `add` with the result from line 3.
- Line 1: Update v with the new `Calculator` from line 2.

This is a nice demonstration of how we only need a few basic mechanisms to implement the changes performed by mixin layers: line 3 and 2 perform an update on a collective and line 1 inserts a unit into a collective. But it is also apparent that this notation is too complicated and needs improvement! We'll explain step by step what is wrong with it and how it can be fixed.

Step 1—eliminate the meta constant v: v keeps the meta code from being generic. Instead of running the code like an imperative program that destructively modifies the base program, we'd like to apply the transformation as a true function. We therefore use lambda notation to abstract the implicit parameter v. Then the meta program gets the prefix "λx" and every occurrence of the constant v is replaced by the variable x.

Step 2—make projection implicit: Lambda abstraction also gives us higher-order functions, allowing us to create a more elegant kind of update operation called $\mathsf{at} : \mathsf{Coll} \times \mathsf{Name} \times (\mathsf{Unit} \to \mathsf{Unit}) \to \mathsf{Coll}$. It applies a function to the member unit u (denoted by its name) of a collective and then replaces u with the result. This does away with projection as a full-blown separate[4] operator and makes our meta code much more readable:

```
λx. at(x, Calculator,
    λy. at(y, add,
        λz.insert(z, @before, this.savedMemory = this.memory;)))
```

[2] v is a constant in a meta language, that's why we call it a meta constant.

[3] v stands for a (completely reduced) *value*, as opposed to functions and partially evaluated terms.

[4] It is of course still implicitly present in the `at` operator.

Here is what happens:

- Line 1: Bind the collective to be modified to variable x. Update member `Calculator` of x by applying...
- Line 2: ... a function that updates member add of its parameter by applying...
- Line 3: ... a function that prepends a statement to the collective given as its argument.

Step 3—make parameter variables implicit: If we change at so that the collective to be modified is the last argument, we can use currying and its signature becomes at : Name $\times$ (Unit $\rightarrow$ Unit) $\rightarrow$ Coll $\rightarrow$ Coll. Then at is a map from a tuple[5] (consisting of a Name and a function from Coll to Coll) to a program transformation (i. e., a function from Coll to Coll). Changing the signature of insert in the same manner eliminates the need for lambda notation, because there are no explicit parameters any more:

```
at(Calculator,
    at(add,
        insert(@before, this.savedMemory = this.memory;)))
```

Step 4—respect order in a collective: Until now, we assumed that a collective was unordered (i. e., a set). But the sequence of statements in a method[6] is also a collective, so that we generally need to deal with *ordered* collectives. This leads to a slightly different set of operators.

$$\begin{aligned}
\mathsf{at} &: \mathsf{Name} \times (\mathsf{Unit} \rightarrow \mathsf{Unit}) \rightarrow \mathsf{Coll} \rightarrow \mathsf{Coll}\\
\mathsf{cons} &: \mathsf{Name} \times \mathsf{Unit} \rightarrow \mathsf{Coll} \rightarrow \mathsf{Coll}\\
\mathsf{snoc} &: \mathsf{Name} \times \mathsf{Unit} \rightarrow \mathsf{Coll} \rightarrow \mathsf{Coll}\\
\mathsf{delete} &: \mathsf{Name} \rightarrow \mathsf{Coll} \rightarrow \mathsf{Coll}\\
\mathsf{override} &: \mathsf{Name} \times \mathsf{Unit} \rightarrow \mathsf{Coll} \rightarrow \mathsf{Coll}
\end{aligned}$$

Explanation: cons and snoc are needed for ordered insertion[7]: cons prepends a unit to a collective, snoc appends it. delete performs the same job as before. Interesting is override as a replacement for update. Whereas update was syntactic sugar for a combined delete/insert operation, we need to think about update differently if nodes in the tree are labeled ordered locations. Then delete removes a unit u *and* its location. We now cannot insert a new unit u' to replace u at the old location (unless we remember where the label occurred). Therefore, we

[5] We could have used currying for the first two arguments, too, but this signature corresponds more closely to how we use at and is useful in Sect. 4 where we give the formal definition of GRAFT's type system.

[6] In formal languages that are not used for programming (e. g., markup languages), order is also often significant. Therefore, this step prepares us for future uses of our framework.

[7] We do not yet need positional insertion, but can easily add it to a future version of our framework should it be necessary.

need an in-place update and override serves that purpose[8]. It replaces an existing member of a collective (whose name equals the first argument) with the second argument. Note that cons, snoc and override are always used with Unit literals.

Step 5—allow sequences of modifications: Right now, we can only apply single modifications to a collective, so we're still missing a mechanism for applying a sequence of operations. As every manipulation is a function, we should obviously draw on function composition. As a slight twist, we don't use the traditional functional composition operator '∘' that evaluates from right to left: `subonly`(`undo`(`base`)) would become `subonly`∘`undo`(`base`). Instead, we prefer diagrammatic composition with the ';' operator which evaluates from left to right. The previous composition is then written as (`base`)`undo`; `subonly`. This gives the meta language more of an imperative feel and makes meta programs easier to read. At last, we have a very compact notation for performing mixin layer composition. We have now arrived at the final version of Graft and demonstrate its use by implementing layer `undo`. Operator cons is used with the underscore '_' as the empty (or anonymous) label. That is, we don't want to label the statement "`this.savedMemory = this.memory`":

```
at(Calculator,
    cons(savedMemory, Int savedMemory);
    at(add,
        cons(_, this.savedMemory = this.memory));
    snoc(undo, void undo() {...}))
```

3 Motivating the Type System

In this document, software composition means either one of two things: We can (1) apply a function to a constant (i. e., apply a mixin layer to a base program) and we can (2) compose two functions (i. e., compose two layers to produce one combined layer). In either case, we want to know if the composition is possible *before* performing it. Guaranteeing safety of an operation is usually handled by a type system. But it is not obvious what the type of a layer should be. We will therefore look at a concrete example to find out what we need for checking the the feasibility of a composition:

1. Applying layer `undo` to a base program v: v cannot be just any program. There are certain constraints imposed on its structure by the operations used in `undo`. For example, there has to be a class `Calculator` and it has to contain a method `add`, otherwise we wouldn't be able to perform the `@before` operation.
2. Composing layer `undo` and `subonly`: Applying layer `undo` after `subonly` will never work, because `subonly` deletes method `add` which makes it impossible for `undo` to prepend the `@before` code.

[8] Another way to look at override is that override(n, v) is "lambda-free" syntactic sugar for at$(n, \lambda x.v)$.

Note that there are two cases of illegal composition that we (for now) do not want to catch:

- Applying `subonly` after `undo` disables undo. This is a semantic problem that we don't treat here.
- The above composed layer, if applied to `base`, still produces a well-typed program (in the sense of the Java language specification [Gosling et al., 2000]). But what if the delete operation had removed a method that is called by another method. Then the modified program won't compile any longer. This is basically a dependence problem and subject of current and future research.

What we do want to find out is what constrains the applicability of atomic and composite operations. It turns out that it depends on the presence or absence of nodes in the tree of the base program:

- $\mathsf{at} : \mathsf{Name} \times (\mathsf{Unit} \to \mathsf{Unit})$: $\mathsf{at}(n, f)$ can be applied to a collective c if in c, there is a unit u whose name is n. Additionally, f has to be applicable to u.
- $\mathsf{cons} : \mathsf{Name} \times \mathsf{Unit}$: $\mathsf{cons}(n, u)$ can be applied to a collective c if there is no unit in c that has the name n.
- The same constraint holds for snoc.
- $\mathsf{delete} : \mathsf{Name}$: We can only perform a delete operation if the name given exists in the collective that is to be modified.
- The same constraint holds for $\mathsf{override}$.

That means that the type system does not need all the information stored in a code tree v to decide if an operation is applicable. It can work with an *abstraction* $\mathsf{abs}(v)$ of v. This abstraction only records the presence of nodes. That is, even though $\mathsf{abs}(v)$ is a labeled tree that is isomorphic to v, it is undecorated and unordered. Given a GRAFT program f, we can use the rules listed above to automatically infer the applicability constraints of f. We encode these constraints in a data structure G that is called *guard*, because it serves the same purpose as guards in functional programming languages. Applicability is then tested by a *guard check operator* '$|$': $\mathsf{abs}(v) \mid G$ holds if and only if f is applicable to v. As guards are matched against abstracted programs, it is natural to also express them as labeled unordered trees. The guard tree is decorated with elements of the set[9] $\{\oplus, \ominus\}$. If a node has to exist, we tag it with a plus $\oplus$. If the node must not exist, we use a minus $\ominus$. Otherwise, we don't care about the node and it does not appear in the guard. Fig. 3 shows the guard tree for one instance of snoc and one instance of delete. Constructing the guard for at is a bit more complicated, because it depends on the guard of the second argument. Fig. 4 shows how we construct at's guard by creating a new root to be the parent of the argument's guard. The child's label is given by the Name argument.

Guards make the feasibility check trivial when applying a layer to a base program, but we have not yet figured out how to perform it for compositions

[9] For uniformity reasons, we decorate nodes in program abstractions with a $\oplus$ from now on.

Fig. 3. Guard trees for snoc(`undo`, ...) and delete(`add`)

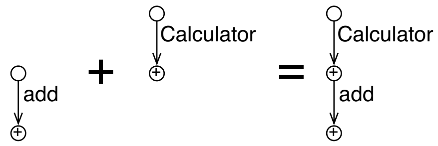


Fig. 4. Guard tree for at(`Calculator`, delete(`add`)): The guard of delete plus a new root is the composite guard of at.

of layers. Two problems have to be solved: (1) How to check if a layer composition is legal and (2) how to calculate the composite guard for the combined layer. The difficulty with the combined guard is that it is not just the union of the two guards, because the changes effected by the first layer might fulfill some of the preconditions of the second guard. This is the case if we write cons(`foo`, ...); delete(`foo`)—we are deleting something that we introduced, therefore the precondition of delete that `foo` has to exist does not have to be propagated to the composite guard. On the other hand, an operation might make subsequent guards impossible to satisfy: In delete(`bar`); override(`bar`, ...), we remove `bar` and cannot override it afterwards. The guard can be seen as an abstract interface of a Graft program f. But it has become obvious that, if we are to compute combined guards, this interface should not only describe the prerequisites for, but also the effects of, f. Thus, we express the interface of f as a pair (G, U), where G is the guard and U is a data structure that encodes how f modifies a program. We write $f \uparrow G \rightsquigarrow U$ to say that f has the interface (G, U). Just like we abstracted code trees, we also abstract the effect of f and are only interested whether it adds or removes nodes. For the same reasons as above, we record U as a labeled unordered tree and call it *update tree*. An update tree has the same format as a guard. It is, however, interpreted differently: If a node is decorated with $\oplus$, it means that an operation adds this node, $\ominus$ indicates a removal. To give an example, we show that the construction of the update tree for at resembles what we have done with guards: We "prepend" a root to the update tree of the argument (Fig. 5). The root is always decorated with a $\oplus$, because any change inside a collective has to guarantee its existence.

Coming back to our problems, we can solve (2), how to compute the combined guard, if we overload operator ';' for (G, U) pairs. Then, if we compose two operations as $f; g$ where $f \uparrow G_1 \rightsquigarrow U_1$ and $g \uparrow G_2 \rightsquigarrow U_2$, we get the combined guard G' from $(G', U') = (G_1, U_1); (G_2, U_2)$. Problem (2), how to check for legal

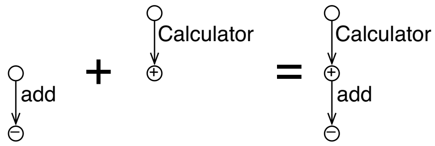


Fig. 5. Update tree for $\mathsf{at}(\mathtt{Calculator}, \mathsf{delete}(\mathtt{add}))$

layer compositions, is solved by letting the composition operator return *undefined* if U_1 and G_2 are incompatible. This makes composition a partial function.

4 Type System

In this section, we give a more rigorous and formal definition of GRAFT's type system. Recall that a transformation f of a typed source program v into a target program w may be unsuccessful for one of the following reasons:

1. One of the operations involved in f fails (e. g., modifying a non-existent method).
2. The transformed program does not compile any more.
3. The target program does not exhibit the expected behavior.

We now develop a type system which helps to detect some of these failures statically (i. e., before actually carrying out the transformation). Errors of category 3 are clearly out of reach for automatic analyses; to ensure preservation of well-typedness (cat. 2) would be desirable, but for now, we only guarantee that all operations can be performed (cat. 1).

4.1 Typing

Our type system results from an *abstract interpretation* [Cousot, 2001] of programs v, w (as *types* A, B) and operations f.

$$\begin{tikzcd} v \arrow[r, "f"] \arrow[d, "\mathsf{abs}"] & w \arrow[d, "\mathsf{abs}"] \\ A \arrow[r, "\mathsf{abs}(f)"] & B \end{tikzcd}$$

It has the following soundness property: If the abstract operation $\mathsf{abs}(f)$ succeeds on the abstract program A, then the concrete operation f also succeeds on the concrete program v. We follow standard programming language theory and introduce these notions:

$v : A$ The program v has type A. From now on, we call the abstraction function from programs to types typeof, therefore $A = \mathsf{typeof}(v)$

$f : A \to B$ Operation f takes programs of type A to programs of type B. Equivalently, we can say that the abstract operation $\mathsf{abs}(f)$ maps an abstract program A to an abstract program B.

In our case, a program v is just a labeled tree with some data d at each node. Its type A abstracts away the stored data and only tells us which nodes exist in the tree. Thus, to get A, we simply replace all data in v by a special symbol $\oplus$ (Fig. 6). Alternatively, A can be seen as a function from paths p into the set $\{\oplus, \ominus\}$ where $A(p) = \oplus$ if the path p exists in v, otherwise $A(p) = \ominus$.

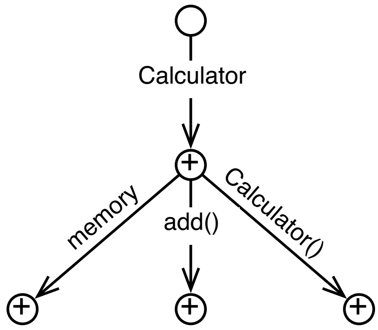


Fig. 6. The abstract interpretation of program `prog` is a tree. Note that the order of children is not significant.

A well-typed operation $f : A \to B$ is applicable to all trees v which have shape A and results in trees of type B. The result type B can be computed from A by performing the abstract version of the operation f on A. For instance,

$$\frac{}{\mathsf{cons}(n, v) : A \to B} \quad \text{if } A(n) = \ominus \text{ and } B = A[n \mapsto \mathsf{typeof}(v)]$$

Here "$A(n) = \ominus$" says that the singleton path n does not exist in A, and "$B = A[n \mapsto \mathsf{typeof}(v)]$" expresses that B results from adding the type of v as a subtree named n to A. For the other operators, we get similar typing rules (see appendix).

A term is either a *value* (i. e. a program) v or a GRAFT operation f applied to a value as $(v)f$. Evaluation of terms is defined using two relations: $t \Downarrow v$ means that term t reduces to value v. $v \xrightarrow{f} w$ means that applying operation f to value v results in value w.

The type system has the desired properties: each well-typed program transformation can be executed.

Theorem 1 (Type preservation and normalization).

1. *If* $t : A$ *then* $A \neq \ominus$, $t \Downarrow v$ *and* $v : A$.

2. If $v : A$ and $f : A \to B$ then $v \text{ —}f\text{⟶ } w$ and $w : B$.

The proof is easy after a complete formalization of the type system (see appendix).

4.2 Type inference

At the current state, we can compute the type A of a tree v and the type B of applying a transformation f to v. This is not yet satisfactory; we want to compute the type of f independently of its argument, because this enables us to type-check each mixin layer f, g individually and then check whether a composition $f; g$ is possible. The problem with the type system so far is that we can assign an infinite number of types $A \to B$ to an operation f. (For example, the domain A of operation $\mathsf{delete}(n)$ can be *any* type with $A(n) = \oplus$.)

In the following, we overcome this problem by specifying an inference system for *most general types* $f \uparrow G \rightsquigarrow U$. A most general type of an operation f consists of a *guard* G and an *update* U. Given the most general type, f can be assigned a type $A \to B$ if the domain A fulfills guard G, written $A \mid G$, and if B results from A updated by U, written $B = A[U]$.

In essence, guards G are partial types, specifying only these parts of the abstract tree of A which it has to contain for an operation to be executed. Formally, guards are partial functions from paths into the set $\{\oplus, \ominus\}$, or, equivalently, total functions in $\mathsf{Path} \to \{\oplus, \ominus, ?\}$. Here, $G(p) = ?$ means that does not impose any conditions on path p. If otherwise $G(p) \in \{\oplus, \ominus\}$, then $A(p)$ has to match $G(p)$. This is expressed in the following definition: The relation $A \mid G$ holds iff for all paths p, $A(p)$ coincides with $G(p)$ unless $G(p) = ?$.

Updates U are of the same shape as guards. The effect of an update $B = A[U]$ can be described pointwise for all paths p: if $U(p) = ?$, i. e., no update is specified, then nothing changes and $B(p) = A(p)$; otherwise, the old content is overwritten as specified by U and $B(p) = U(p)$.

We are now ready to give most general types for the basic operations, e. g.

$$\frac{}{\mathsf{cons}(n, v) \uparrow \{n \mapsto \ominus\} \rightsquigarrow \{n \mapsto A\}} \quad \text{where } A = \mathsf{typeof}(v)$$

Here, the guard G is a partial function which maps n to $\ominus$ and is completely undefined elsewhere. The update U specifies that name n should now be mapped to subtree A. By looking at the original typing rule of cons we can immediately verify that this guard/update pair is the most general type of $\mathsf{cons}(n, v)$.

A little more work has to be done for composite operators $f; g$. Assume $f \uparrow G_1 \rightsquigarrow U_1$ and $g \uparrow G_2 \rightsquigarrow U_2$, what should guard G and update U of the composite operation $f; g$ look like? If f and g are disjoint, i. e., modifying different regions of a program, then the common guard $G = G_1 \cup G_2$ is the union of the two partial functions G_1 and G_2 (and similar for U). For the case that f and g interfere, the common update is simply given by $U = U_1[U_2]$, which is defined in the same way as the update on types $B = A[U]$ (see above). In this case, g may override some updates performed by f.

To compute the common guard G, let us first consider that U_1 does not interfere with G_2, i. e., f does not update parts of the tree G_2 speaks about. The joint guard G is written $G_1 \vee G_2$ and does not exist if G_1 contradicts G_2. This is, for instance, the case if G_1 requires some path p to be present and G_2 requires it to be absent (or the other way round). Otherwise $G_1 \vee G_2 = G_1 \cup G_2$. Some type A passes the two single guards, $A \mid G_1$ and $A \mid G_2$, if and only if the joint guard $G = G_1 \vee G_2$ is defined and $A \mid G$ holds.

In some cases the first update U_1 will fulfill some requirements of the second guard G_2. When we compute the joint guard, these requirements can be dropped. To this end, we define a partial simplification operation $G \setminus U$ with the following behavior: $A[U] \mid G$ if and only if $G' = G \setminus U$ is defined and $A \mid G'$. The new guard G' is defined if U does not contradict G, i. e., if $G \vee U$ is defined. In this case, G' can be computed pointwise: if the old requirement $G(p)$ if fulfilled by the update $U(p)$, it is dropped and $G(p') = ?$, otherwise it is kept and $G(p) = G(p')$. Now the typing rule for composition should be clear:

$$\frac{f \uparrow G_1 \rightsquigarrow U_1 \qquad g \uparrow G_2 \rightsquigarrow U_2}{f;g \uparrow G_1 \vee (G_2 \setminus U_1) \rightsquigarrow U_1[U_2]} \text{ if } G_1 \vee (G_2 \setminus U_1) \text{ defined}$$

For completeness, we present the rules for the type inference $t \uparrow A$ for terms. The typing rule for application of operations f to trees t connects most general types to ordinary types:

$$\frac{}{v \uparrow \mathsf{typeof}(v)} \qquad \frac{v \uparrow A \qquad f \uparrow G \rightsquigarrow U}{(v)f \uparrow A[U]} \text{ if } A \mid G$$

It remains to show that type inference is sound, which can be done by inspecting the typing rules, using the fundamental properties of guard and update joining.

Theorem 2 (Soundness of type inference).

1. *If* $t \uparrow A$ *then* $t : A$.
2. *If* $f \uparrow G \rightsquigarrow U$ *and* $A \mid G$ *then* $f : A \to A[U]$.

For a proof and a complete formalization, consult the appendix.

4.3 Example

Let's demonstrate type inference for the layers `subonly` and `undo` and show how typing of the composite operation `subonly;undo` fails. Layer `subonly` is implemented in Graft as follows:

$$\mathtt{subonly} = \mathsf{at}(\mathtt{Calculator},\ \mathsf{delete}(\mathtt{add});\ \mathsf{snoc}(\mathtt{sub}, \dots))$$

We give the result of type inference for `subonly` step by step:

$$\begin{array}{rll} \mathsf{delete}(\texttt{add}) & \uparrow \{\texttt{add} \mapsto \oplus\} & \rightsquigarrow \{\texttt{add} \mapsto \ominus\} \\ \mathsf{snoc}(\texttt{sub}, \ldots) & \uparrow \{\texttt{sub} \mapsto \ominus\} & \rightsquigarrow \{\texttt{sub} \mapsto \oplus\} \\ \mathsf{delete}(\ldots); \mathsf{snoc}(\ldots) & \uparrow \{\texttt{add} \mapsto \oplus, \texttt{sub} \mapsto \ominus\} & \rightsquigarrow \{\texttt{add} \mapsto \ominus, \texttt{sub} \mapsto \oplus\} \end{array}$$

$$\begin{array}{rl} \mathsf{at}(\texttt{Calculator}, \ldots) & \uparrow \{\texttt{Calculator} \mapsto \{\texttt{add} \mapsto \oplus, \texttt{sub} \mapsto \ominus\}\} =: G_1 \\ & \rightsquigarrow \{\texttt{Calculator} \mapsto \{\texttt{add} \mapsto \ominus, \texttt{sub} \mapsto \oplus\}\} =: U_1 \end{array}$$

For layer `undo` we get the guard

$$G_2 = \{\texttt{Calculator} \mapsto \{\texttt{savedMemory} \mapsto \ominus, \texttt{add} \mapsto \oplus, \texttt{undo} \mapsto \ominus\}\}$$

In the process of calculating the joint guard for $(\texttt{subonly}; \texttt{undo})$, the simplification operation $G_2 \setminus U_1$ fails already, since the operands differ on the existence of the path $(\texttt{Calculator}, \texttt{add})$:

$$\begin{array}{l} U_1(\texttt{Calculator}, \texttt{add}) = \ominus \\ G_2(\texttt{Calculator}, \texttt{add}) = \oplus \end{array}$$

Hence type inference fails for the composition of the two layers, which indicates an inconsistency.

5 Related Work

Related work can be grouped into the following categories:

- Composition paradigms: Each of these paradigms supports one kind of composition. Additionally, and this is similar to GRAFT, they enable validity checks for their operations. The advantage of concentrating on only one way of composition is that these validity checks are very evolved and well adapted for this one purpose. GRAFT's goals for the future are broader: We want a less sophisticated framework (with specialized "plug-ins" for some features) to support *several* ways of composition and more artifacts than just code. Examples: Detection and resolution of aspect interactions [Douence et al., 2002], invasive composition [Aßmann, 2003], traits [Schärli et al., 2003], composition of class hierarchies [Snelting and Tip, 2002].
- Object-oriented language calculi: This category is even more specialized than the last one, but shares the use of formal methods with GRAFT. Examples: Reduction semantics for classes and mixins [Flatt et al., 1999], core calculus of classes and mixins [Bono et al., 1999].
- General meta programming: These are full-blown languages which enable many powerful program generation tasks that GRAFT can't perform. In contrast, we aim for simplicity and our focus on program transformation is slightly different. Examples: Logic meta programming [Brichau et al., 2002], OpenJava [Tatsubori et al., 2000], multi-stage languages [Calcagno et al., 2003].

– Program transformation: This category has many similarities to the last one and is closely related to GRAFT. Again, we sacrifice generality for simplicity which makes many consistency checks easier to perform. Examples: Strategic programming [Lämmel et al., 2003], program transformation through the manipulation of semantic graphs [Gosling, 2003].
– Separate compilation: Separate compilation is mainly concerned with providing self-sufficient typing for program fragments. This is an area that is currently completely ignored by GRAFT. On the other hand, separate compilation is only concerned with composition as linking and other manipulations are—if at all—limited. Examples: Fragment calculus [Drossopoulou et al., 1999], true separate compilation of Java classes [Ancona et al., 2002].

Two publications deserve particular mention, because they are the most similar in nature to GRAFT:

– Design rule checking [Batory and Geraci, 1997]: Design rules are complementary to our approach. They are manual annotations of units in propositional logic and capture semantics. Whereas GRAFT relies on syntactic analysis that is performed automatically.
– Reuse contracts: are concerned with step-wise evolution of programs while checking consistency constraints [Mens, 2001]. They also have a well-defined and powerful type system and have already been applied to a wide variety of artifacts [Mens and D'Hondt, 2000]. While our work is currently less ambitious, using trees and a language for their manipulation makes GRAFT simpler and easier to understand.

6 Conclusions and Future Research

In this paper, we have presented GRAFT, a simple meta language for generically expressing composition. It is complemented by a type system that allows one to check the applicability of a GRAFT program before actually applying it. As a practical example, we demonstrated GRAFT's ability to express mixin layer composition. The main contributions of our work are GRAFT's simplicity, elegance and universality; its rigorous definition (where many composition mechanisms are not that clearly defined); and the type-safety of its programs, which includes the use of inference to save human effort. The intended "end-user" of GRAFT is not the programmer. Instead, tool implementors will write compilers that translate source code plus annotations into GRAFT in order to perform checks and verification. Another possible use case is to let small GRAFT programs execute type-safe refactorings.

We strove to make this first version of GRAFT a minimal, clean core to help us understand the issues involved in creating a composition language. These attributes make the core a strong foundation for the following future experiments and extensions:

– We have started to implement GRAFT in Haskell (which, in this case, is the meta meta language). This version of GRAFT operates on MJ [Bierman et al., 2003], a subset of Java.

– We cannot yet guarantee the well-typedness of a program after it has been manipulated by Graft code. To achieve this, we have experimented with partial types and plan to make them part of a future version of Graft.
– Graft should also support more features of the language that is to be modified (the *object language*): Reflective features, pre/post conditions, call history enforcement and others come to mind. For that, more of the semantics of the object language has to be made explicit.
– We will investigate what other programming language features (such as conditionals, loops, quantifiers, procedures etc.) make sense for Graft.
– In creating Graft, we lay some groundwork for the future support of artifact types other than code (documentation, UML diagrams etc.). AHEAD [Batory et al., 2003] already gives some ideas how mixin layer composition can be extended to non-code artifacts. Frame-based programming [Bassett, 1996] can also be used to produce all sorts of artifacts and motivated some of Graft's basic features. We hope that our type system will remain generic enough to be applicable across artifact kinds. Consistency enforcement could then be cross-cutting, too (e. g., if a class is part of the source code, it also has to appear in the UML class diagram).
– Finally, we have to find out how Graft can express other composition mechanisms than mixin layers, such as aspects.

Acknowledgements. Thanks to Jörg Striegnitz for recommending MJ.

A Graft Specification

A.1 Trees

$\mathsf{Name} \ni n, m$			Names
$\mathsf{Path} \ni p$	$::= n_1, \ldots, n_k$	$(k \geq 0)$	Paths
$\mathsf{D} \ni d$			Data (tree decoration)
$\mathsf{Unit} \ni u$	$::= n{::}v$		Named units
$\mathsf{Coll} \ni c$	$::= u_1, \ldots, u_k$	$(k \geq 0)$	Collectives
$\mathsf{Tree} \ni v, w$	$::= \langle d;\, c\rangle$		Decorated trees (values)

Collectives can be viewed as partial functions from names to trees resp. as total functions to $\mathsf{Tree}^{\ominus} := \mathsf{Tree} \cup \{\ominus\}$.

$$\mathsf{Coll} \subseteq \mathsf{Name} \to \mathsf{Tree}^{\ominus}$$

$$(\overrightarrow{n{::}v})(m) = v_i \quad \text{if } m = n_i$$
$$(\overrightarrow{n{::}v})(m) = \ominus \quad \text{if } m \notin \overrightarrow{n}$$

Similarily, trees can be viewed as as total functions from paths into $\mathsf{D}^{\ominus} := \mathsf{D} \cup \{\ominus\}$. We extend the domain of this coercion to $\mathsf{Tree}^{\ominus}$.

$$\mathsf{Tree}^{\ominus} \subseteq \mathsf{Path} \to \mathsf{D}^{\ominus}$$

$\ominus(p)$	$= \ominus$	empty tree
$\langle d;\, c\rangle()$	$= d$	empty path
$\langle d;\, c\rangle(n, p)$	$= c(n)(p)$	composite path

A.2 Operations and Terms

$$
\begin{array}{llll}
\mathsf{Op} & \ni f, g ::= & \mathsf{cons}(n, v) & \text{prepend unit } n{::}v \text{ to current collective} \\
 & & \mid \mathsf{snoc}(n, v) & \text{append unit} \\
 & & \mid \mathsf{override}(n, v) & \text{replace unit named } n \text{ by new unit } n{::}v \\
 & & \mid \mathsf{delete}(n) & \text{delete unit named } n \text{ from current collective} \\
 & & \mid \mathsf{at}(n, f) & \text{apply operation } f \text{ to unit named } n \\
 & & \mid f; g & \text{operation sequence} \\
\mathsf{Term} \ni t & ::= & v & \text{tree value} \\
 & & \mid (v)f & \text{(left hand side) application of operation}
\end{array}
$$

Transformation of trees $v \text{ —}f\text{⟶ } w$.

$$\frac{c \text{ —}f\text{⟶ } c'}{\langle d;\ c\rangle \text{ —}f\text{⟶ } \langle d;\ c'\rangle}$$

Transformation of collectives $c \text{ —}f\text{⟶ } c'$.

$$\frac{}{c \text{ —}\mathsf{cons}(n, v)\text{⟶ } n{::}v, c}\ \text{if } n \notin c \qquad \frac{}{c \text{ —}\mathsf{snoc}(n, v)\text{⟶ } c, n{::}v}\ \text{if } n \notin c$$

$$\frac{}{c, n{::}v, c' \text{ —}\mathsf{override}(n, v')\text{⟶ } c', n{::}v', c'} \qquad \frac{}{c, n{::}v, c' \text{ —}\mathsf{delete}(n)\text{⟶ } c, c'}$$

$$\frac{v \text{ —}f\text{⟶ } v'}{c, n{::}v, c' \text{ —}\mathsf{at}(n, f)\text{⟶ } c, n{::}v', c'} \qquad \frac{c_1 \text{ —}f\text{⟶ } c_2 \qquad c_2 \text{ —}g\text{⟶ } c_3}{c_1 \text{ —}f; g\text{⟶ } c_3}$$

Evaluation $t \Downarrow v$.

$$\frac{}{v \Downarrow v} \qquad \frac{v \text{ —}f\text{⟶ } w}{(v)f \Downarrow w}$$

A.3 Tree Types and Typing

$$
\begin{array}{llll}
\mathsf{Type} & = & (\mathsf{Name} \to \mathsf{Type}) \cup \{\ominus\} & \\
\mathsf{Type} & \ni & A, B & \text{Types} \\
\mathsf{Type} & \subseteq & \mathsf{Path} \to \{\oplus, \ominus\} & \\
\ominus(p) & = & \ominus & \\
A() & = & \oplus & \text{if } A \neq \ominus \\
A(n, p) & = & A(n)(p) &
\end{array}
$$

On the other hand, for each $B \in \mathsf{Path} \to \{\oplus, \ominus\}$, there exists an equivalent $A \in \mathsf{Type}$: If $B(p) = \ominus$ for all $p \in \mathsf{Path}$, then $A := \ominus$. Otherwise, let recursively $A(n)$ be the type equivalent to the function $B(n, \_)$. By this isomorphism, we can conveniently specify the type of a tree:

$$\mathsf{typeof} \in \mathsf{Tree} \to \mathsf{Type}$$

$$
\begin{array}{ll}
\mathsf{typeof}(v)(p) = \ominus & \text{if } v(p) = \ominus \\
\mathsf{typeof}(v)(p) = \oplus & \text{else}
\end{array}
$$

Type update $A[n \mapsto B] \in \mathsf{Type}$.

$$
\begin{array}{lll}
\ominus[n \mapsto B] & = \ominus & \\
A[n \mapsto B](n) & = B & \text{if } A \neq \ominus \\
A[n \mapsto B](m) & = A(m) & \text{if } A \neq \ominus \text{ and } m \neq n
\end{array}
$$

Typing of terms $t : A$.

$$
\frac{}{v : \mathsf{typeof}(v)} \qquad \frac{v : A \qquad f : A \to B}{(v)f : B}
$$

Typing of operations $f : A \to B$.

$$
\text{If } A(n) = \ominus : \quad \frac{v : A'}{\mathsf{cons}(n, v) : A \to A[n \mapsto A']} \quad \frac{v : A'}{\mathsf{snoc}(n, v) : A \to A[n \mapsto A']}
$$

$$
\text{If } A(n) \neq \ominus : \quad \frac{v : A'}{\mathsf{override}(n, v) : A \to A[n \mapsto A']} \quad \frac{}{\mathsf{delete}(n) : A \to A[n \mapsto \ominus]}
$$

$$
\frac{f : A(n) \to A'}{\mathsf{at}(n, f) : A \to A[n \mapsto A']} \text{ if } A(n) \neq \ominus \qquad \frac{f : A_1 \to A_2 \qquad g : A_2 \to A_3}{f; g : A_1 \to A_3}
$$

Theorem 1 (Type preservation and normalization).

1. *If $t : A$ then $A \neq \ominus$, $t \Downarrow v$ and $v : A$.*
2. *If $v : A$ and $f : A \to B$ then $v \stackrel{f}{\longrightarrow} w$ and $w : B$.*

Proof. Simultaneously by structural induction on the judgments $t : A$ and $f : A \to B$.

A.4 Guards and Updates

$$
\begin{array}{lll}
G \in \mathsf{Guard} & = \mathsf{Path} \to \{\oplus, \ominus, ?\} & \text{conditions on types} \\
U \in \mathsf{Update} & = \mathsf{Path} \to \{\oplus, \ominus, ?\} & \text{modifications of types}
\end{array}
$$

Guard checking $A \mid G$.

$$
\begin{array}{lll}
A \mid G & \iff A(p) \mid G(p) & \text{for all } p \in \mathsf{Path} \text{ where} \\
a \mid g & \iff g = ? \text{ or } a = g &
\end{array}
$$

Applying updates $U_1[U_2] \in \mathsf{Update}$. (Defines also $P[U] \in \mathsf{Type}$ since $\mathsf{Type} \subseteq \mathsf{Update}$.)

$$
\begin{array}{lll}
U_1[U_2](p) = U_1(p)[U_2(p)] & & \text{where} \\
u_1[u_2] & = u_1 & \text{if } u_2 = ? \\
u_1[u_2] & = u_2 & \text{else}
\end{array}
$$

Lemma 1 (Associativity). $U_1[U_2][U_3] = U_1[U_2[U_3]]$.

Guard joining $G_1 \vee G_2 \in \mathsf{Guard}$ (symmetric partial operation).

$$\begin{array}{lll}
(G_1 \vee G_2)(p) & = G_1(p) \vee G_2(p) & \text{where} \\
g_1 \vee g_2 & = g_1 & \text{if } g_1 = g_2 \text{ or } g_2 = \,? \\
g_1 \vee g_2 & = g_2 & \text{if } g_1 = \,? \\
g_1 \vee g_2 & = \text{undefined} & \text{otherwise}
\end{array}$$

$G_1 \vee G_2$ is only defined if $G_1(p) \vee G_2(p)$ is defined for all $p \in \mathsf{Path}$.

Lemma 2 (Joint guard checking). *$A \mid G_1$ and $A \mid G_2$ iff $G_{12} := G_1 \vee G_2$ is defined and $A \mid G_{12}$.*

Guard updates $G \setminus U \in \mathsf{Guard}$ (partial operation).

$$\begin{array}{lll}
(G \setminus U)(p) & = G(p) \setminus U(p) & \text{where} \\
g \setminus u & = g & \text{if } g = \,? \text{ or } u = \,? \\
g \setminus u & = \,? & \text{if } g = u \\
g \setminus u & = \text{undefined} & \text{otherwise}
\end{array}$$

Lemma 3 (Updated guard checking). *$A[U] \mid G$ iff $G' := G \setminus U$ is defined and $A \mid G'$.*

A.5 Type Inference

Type inference $t \uparrow A$.

$$\frac{}{v \uparrow \mathsf{typeof}(v)} \qquad \frac{v \uparrow A \qquad f \uparrow G \rightsquigarrow U}{(v)f \uparrow A[U]} \text{ if } A \mid G$$

Most general types for operations $f \uparrow G \rightsquigarrow U$.

$$\frac{v \uparrow A}{\mathsf{cons}(n, v) \uparrow \{n \mapsto \ominus\} \rightsquigarrow \{n \mapsto A\}} \qquad \frac{v \uparrow A}{\mathsf{snoc}(n, v) \uparrow \{n \mapsto \ominus\} \rightsquigarrow \{n \mapsto A\}}$$

$$\frac{v \uparrow A}{\mathsf{override}(n, v) \uparrow \{n \mapsto \oplus\} \rightsquigarrow \{n \mapsto A\}} \qquad \frac{}{\mathsf{delete}(n) \uparrow \{n \mapsto \oplus\} \rightsquigarrow \{n \mapsto \ominus\}}$$

$$\frac{f \uparrow G \rightsquigarrow U}{\mathsf{at}(n, f) \uparrow \{n \mapsto G\} \rightsquigarrow \{n \mapsto U\}}$$

$$\frac{f_1 \uparrow G_1 \rightsquigarrow U_1 \qquad f_2 \uparrow G_2 \rightsquigarrow U_2}{f_1; f_2 \uparrow G_1 \vee (G_2 \setminus U_1) \rightsquigarrow U_1[U_2]} \text{ if } G_1 \vee (G_2 \setminus U_1) \text{ defined}$$

Theorem 2 (Soundness of type inference).

1. *If $t \uparrow A$ then $t : A$.*
2. *If $f \uparrow G \rightsquigarrow U$ and $A \mid G$ then $f : A \to A[U]$.*

Proof. Simultaneously by structural induction on the judgments $t \uparrow A$ and $f \uparrow G \rightsquigarrow U$.

References

Davide Ancona, Giovanni Lagorio, and Elena Zucca. True separate compilation of Java classes. In *Proc. 4th Conf. Principles and Practice of Declarative Programming (PPDP)*, pages 189–200. ACM Press, 2002.

Uwe Aßmann. *Invasive Software Composition*. Springer, 2003.

Paul G. Bassett. *Framing Software Reuse: Lessons From the Real World*. Prentice Hall, 1996.

Don Batory and Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Trans. Software Engineering*, 23(2):67–82, 1997.

Don Batory, Jack Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. In *Proc. 25st IEEE Int. Conf. Software Engineering (ICSE)*. IEEE, 2003.

Gavin M. Bierman, Matthew J. Parkinson, and Andrew M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge, 2003.

Joshua Bloch et al. A Metadata Facility for the Java Programming Language, 2002. URL `http://www.jcp.org/en/jsr/detail?id=175`. Java Specification Request 175.

Viviana Bono, Amit Patel, and Vitaly Shmatikov. A Core Calculus of Classes and Mixins. In Rachid Guerraoui, editor, *Proc. 13th Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1628 of *Lect. Notes Comp. Sci.*, pages 43–66. Springer, 1999.

Johan Brichau, Kim Mens, and Kris De Volder. Building Composable Aspect-Specific Languages with Logic Metaprogramming. In *Proc. 1st Conf. Generative Programming and Component Engineering (GPCE)*, volume 2487 of *Lect. Notes Comp. Sci.*, pages 110–127, October 2002.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-Stage Languages Using ASTs, Gensym, and Reflection. 2003.

Patrick Cousot. Abstract Interpretation Based Formal Methods and Future Challenges. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lect. Notes Comp. Sci.*, pages 138–156. Springer, 2001.

Rémi Douence, Pascal Fradet, and Mario Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Proc. 1st Conf. Generative Programming and Component Engineering (GPCE)*, volume 2487 of *Lect. Notes Comp. Sci.*, pages 173–188, 2002.

Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *Proc. 14th IEEE Symp. Logic in Computer Science*, pages 147–156. IEEE, 1999.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes Comp. Sci.*, pages 241–269. Springer, 1999.

Martin Fowler. *Refactoring*. Addison Wesley, 1999.

James Gosling. Program Transformations Through the Manipulation of Semantic Graphs, January 2003. URL `http://www.sigs.de/kongresse/oop_2003/key_gosling.zip`. Keynote SIGS-DATACOM Conf. Object-Oriented Programming (OOP 2003).

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2000.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proc. 15^{th} Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2072 of *Lect. Notes Comp. Sci.*, pages 327–353. Springer, 2001.

Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic Programming Meets Adaptive Programming. In *Proc. 2^{nd} Conf. Aspect-Oriented Software Development (AOSD)*. ACM Press, 2003.

Tom Mens. A Formal Foundation for Object-Oriented Software Evolution. In *Proc. Int. Conf. Software Maintenance (ICSM)*, pages 549–552. IEEE, 2001.

Tom Mens and Theo D'Hondt. Automating Support for Software Evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable Units of Behavior. In *Proc. 17^{th} Europ. Conf. Object-Oriented Programming (ECOOP)*. Springer, 2003. To appear.

Yannis Smaragdakis and Don Batory. Implementing Layered Design with Mixin Layers. In Eric Jul, editor, *Proc. 12^{th} Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1445 of *Lect. Notes Comp. Sci.*, pages 550–570, 1998. URL `http://www.cs.utexas.edu/users/schwartz/pub.htm#ecoop-templates`.

Gregor Snelting and Frank Tip. Semantics-Based Composition of Class Hierarchies. In *Proc. 16^{th} Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2374 of *Lect. Notes Comp. Sci.*, pages 562–584. Springer, 2002.

Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A Class-Based Macro System for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lect. Notes Comp. Sci.*, pages 117–133. Springer, 2000.