

# Type-Checking AHEAD\*

Axel Rauschmayer, Alexander Knapp, Martin Wirsing  
Institut für Informatik  
Ludwig-Maximilians-Universität München  
{rauschma, knapp, abel, wirsing}@informatik.uni-muenchen.de

November 16, 2003

## Abstract

Ubiquitous computing increases the pressure on the software industry to produce ever more and error-free code. As an answer, *generative programming* increases the level of abstraction in software development by describing problems in high-level *domain-specific* languages and translating these to executable code. AHEAD (*Algebraic Hierarchical Equations for Application Design*) is a framework for generative programming that achieves additional productivity gains by recognizing that in many areas, we need a *family* of programs that are very similar. It avoids duplication of work by generating programs out of a common base of *features* that can be freely composed. Our contribution is GRAFT, a calculus that gives a formal foundation to AHEAD and provides several mechanisms for making sure that feature combinations are legal and that features in themselves are consistent.

## 1 Introduction

As manufacturers turn almost every household device into a computer, they increase the pressure on the software industry to produce ever more robust and bug-free code. *Generative programming* [Czarnecki and Eisenecker, 2000] meets the challenge by raising the level of abstraction and translates higher-level *domain-specific languages* [van Deursen et al., 2000] (DSLs) into lower-level implementation languages. AHEAD [Batory et al., 2003ab] is a framework for generative programming that solves several problems in this area: As a typical software system comprises a mixture of artifacts (written in human language, programming languages, domain-specific languages etc.), AHEAD manages the synthesis of *all* of them. It keeps the artifacts in a tree (which, in the current incarnation of AHEAD is defined by a file system directory). This approach has the advantage of easily scaling from small to large software systems. AHEAD realizes additional productivity gains by observing that there are often *families* of programs to be created that are very similar. Traditionally, a lot of work is being duplicated here (see the *variability* problem in software product line engineering [Jaring and Bosch, 2002]). AHEAD generates a member of a program family by composing it from a set of features.

The subject of this work is *First-Order Graft* (short: GRAFT), a calculus that provides a theoretical foundation for the current incarnation of AHEAD<sup>1</sup>. The clear and formal definition of what an AHEAD feature is allows us to perform a variety of analyses. We can perform *intra-checks* for features: Is a feature internally consistent (e.g., do I add the same node twice?), does it fulfill internal semantic constraints, are existential dependencies between nodes fulfilled? In addition to just checking these conditions, GRAFT can also be used as a diagnostic tool: If a condition can still be fulfilled externally, it shows up in an inferred interface of a feature. This interface permits *inter-checks* between features: Does a given combination of features make sense? As before, we check semantic properties, consistency and dependencies. One interesting aspect of GRAFT is that the language used for composition is separate from the language used for defining artifacts. This is a necessity if we are to formalize AHEAD's support for multiple kinds of artifacts.

---

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

<sup>1</sup>Note that at the very core, AHEAD is based on a theoretical model that is more generic than GRAFT (namely, nested functions and constants). But this does not currently show in practical applications which are completely describable in GRAFT and easier to analyze that way.

```

class Converter {
  void celsiusToFahrenheit(int c) {
    present("Fahrenheit", ((c * 9) / 5) + 32);
  }
  void fahrenheitToCelsius(int f) {
    present("Celsius", ((f - 32) * 5) / 9);
  }
  void present(String label, int value) {
    System.out.println(label + ": " + value);
  }
}

```

Figure 1: The source code of the monolithic version of the converter program. Method `celsiusToFahrenheit` takes an integer argument specifying a temperature in degrees Fahrenheit, converts it to degrees Celsius and presents the result on Standard Out using method `present`. Method `fahrenheitToCelsius` performs the same conversation in the opposite direction.

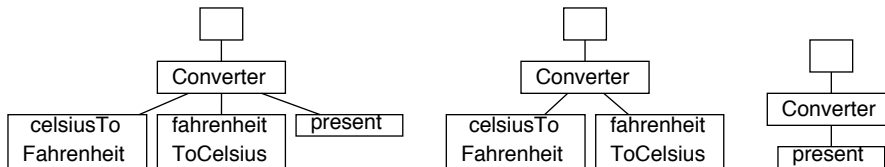


Figure 2: The tree of the monolithic version of `converter` (left) has been split into two features, `base` (middle) and `presentstdout` (right).

The next sections are structured as follows: In Sect. 2, we give a short introduction to AHEAD. Sect. 3 presents *design rules*, the current approach to type checking used in AHEAD. AHEAD data structures and design rules can be encoded in GRAFT, as shown in Sect. 4. With this foundation, GRAFT allows us to perform several analyses (Sect. 5). In Sect. 6 we sketch correctness proofs of some properties of GRAFT. We end by mentioning related work (Sect. 7) and by drawing conclusions about present and future work (Sect. 8).

## 2 AHEAD

AHEAD stores a software system as a tree and breaks its monolithic structure into *features*, modules implementing one specific functionality. Features can then be flexibly composed to generate a whole *family* of programs. We'll demonstrate that process with a small example. The program in Fig. 1, consisting of a single class, converts temperatures from degrees Celsius to Fahrenheit and vice versa. Fig. 2 (left) shows this code represented as a tree.

Now suppose we want to provide two alternate ways of presenting the result of the conversion: One version of the program should output the value on standard out, another one should display a Swing dialog as a graphical interface. These kinds of requirements are what led to programming with features and this is therefore a good example of how it can be put to use. The first step is to break up the monolithic program into two separate features called `base` and `presentstdout`. The two resulting trees are shown in Fig. 2. To get back the original monolithic version, all we need to do is to add feature `presentstdout` to the initial feature `base`. Informally, the algorithm for composition is as follows: We merge nodes with the same name and add nodes whose name does not yet appear in the original tree. This process starts at the root node and recursively proceeds to all of its descendants. The root node can either be viewed as nameless or we look at the names as being attached to the edges instead of the nodes. This prevents the root node from being a special case in the algorithm. With this preparation, we can introduce the new feature `presentswing` for Swing-based presentation. Fig. 3 shows the source

```

class Converter {
    void celsiusToFahrenheit(int c, int f) {
        present("Fahrenheit", ((c * 9) / 5) + 32);
    }
    void fahrenheitToCelsius(int f, int c) {
        present("Celsius", ((f - 32) * 5) / 9);
    }
}

```

(a) Feature `base`

```

refines class Converter {
    void present(String label, int value) {
        System.out.println(label + ": " + value);
    }
}

```

(b) Feature `presentstdout`

```

refines class Converter {
    void present(String label, int value) {
        javax.swing.JOptionPane.showMessageDialog(null,
            label + ": " + value);
    }
}

```

(c) Feature `presentswing`

Figure 3: The final version of the converter program has three features: Feature `base` is the foundation of the programs, features `presentstdout` and `presentswing` provide alternate ways of presenting the result of a conversion. The modifier `refines` in front of `class` is AHEAD’s way of indicating that we add to an existing class.

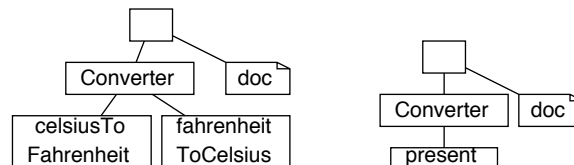


Figure 4: Features `base` and `presentstdout`, with one more kind of artifact in addition to Java source: documentation, residing in a node separate to the source code branches.

code of the whole software system.

As a final comment, note that in the following, we are only looking at AHEAD software that contains Java source code. But all our considerations apply just as well to any other artifact kind. Fig. 4 shows how we could add documentation to our system: Adding a branch for documentation and providing AHEAD with the means to compose documentation node would enable us to automatically generate documentation for either the swing or the standard out version of the program. Note that our name-based algorithm automatically does the right thing and that a composed tree contains one composed documentation node by the name “doc”.

### 3 Design Rule Checking

Even with only three features, like in our example, there are already many possible combinations, especially as we always compose a *sequence* of features. This means that order matters and there are AHEAD systems in use where the ability to add the same feature several times is very useful. How can we prevent combinations that don't make sense? This is where AHEAD uses *design rules*. Abstractly, a feature has a set of design rules associated with it. A design rule expresses the demand of the feature that its environment meet certain criteria and/or specifies the feature's contribution to the environment. "Environment" means the predecessors and successors in the feature sequence. Accordingly, design rule checking propagates properties and checks in two directions: From the first feature to the last and vice versa. This is similar to propagating inherited and synthesized properties up and down a parse tree in attribute grammars. One typical design rule is that feature  $g$  demands that another feature  $f$  has been previously introduced. This can be encoded as a boolean<sup>2</sup> property `gIsThere`:  $g$  sets the property to `true`,  $f$  performs the check `gIsThere = true` and this property is propagated first-to-last. The same pattern can be used for ensuring the existence of any kind of resource, be it a class, a method or something that is completely abstract. It is up to the programmer to give adequate names to properties and to set and check them correctly. This universality is a useful complement to purely syntactic checks. For more details on design rules refer to [Batory and Geraci, 1997] and the documentation delivered with the AHEAD distribution [Batory et al.].

### 4 Graft: Trees and Design Rules

In this chapter, we introduce GRAFT and show how we represent AHEAD trees and design in it. The next chapter shows what kind of analyses are possible with the new representation.

#### 4.1 Representing AHEAD trees in Graft

AHEAD has a slightly asymmetric view of composition: Composing means starting with a base feature and "applying" features as *increments* to it. If, for example, we start with base feature  $b$  and add the increments  $f$  and  $g$  to it, we write  $g(f(b))$ . AHEAD alternatively uses the traditional, *functional*, composition operator 'o' to express composition. The example becomes  $g \circ f(b)$  and is evaluated from right to left. In GRAFT, we prefer diagrammatic composition with ';' and write  $(b)f;g$  which is evaluated from left to right. We first show how GRAFT encodes increments and then get to representing base features.

We have already seen that applying an increment  $f$  to a base  $b$  is like taking  $b$  as a starting point and then looping over every node in  $f$ : If the node (which is identified by its path name) is already present in  $b$ , we combine it with the existing node (a process that is called *refinement* in AHEAD). Otherwise, this node is a new addition to  $b$ . We have thus encountered two operations that can be performed on a tree: refining and adding. But two more operations are typically used when manipulating trees: moving and deleting. In order to be able to freely use and combine any of these four operations, GRAFT represents AHEAD increments as a sequence of commands, each one performing an addition, a delete, a move or a refinement. The node to operate on is given as a path of node names, the commands `add` and `refine` additionally get an argument containing the node (or rather, its data) that is to be added to the original tree. The last argument of every command is of course the tree that is to be modified. Let's look at the `add` command. It is written as

`add p [code] + {d} t`

This command contains one additional construct: We want to express that certain nodes in an AHEAD tree *depend on* others: If node  $x$  depends on node  $y$ , node  $y$  has to exist whenever node  $x$  exists. The above pattern is to be read as: At the location denoted by path  $p$ , add the node given by `code` (which depends on another node whose path is  $d$ ) to the tree  $t$ .

To build sequences of commands, we take a set of them, instantiate every argument except the tree argument and thus get a set of functions mapping trees to trees (this technique is called *currying*). Now simple function composition with ';' allows us to combine this set sequentially into one composite

---

<sup>2</sup>Here, we only look at boolean properties, while integer values can also be used in design rule checking. Support for these necessitates only a slight generalization of the mechanisms we'll present.

```

add Converter [class Converter];
add Converter.celsiusToFahrenheit  $\dashv$  {Converter.present}
  [void celsiusToFahrenheit(...) {...}];
add Converter.fahrenheitToCelsius  $\dashv$  {Converter.present}
  [void fahrenheitToCelsius(...) {...}]

```

(a) Feature `base`

```

add Converter.present [void present(...) {...}]

```

(b) Feature `presentstdout`. It is not necessary to specify that class `Converter` is being refined, `add` handles this correctly.

```

add Converter.present [void present(...) {...}]

```

(c) Feature `presentswing`.

Figure 5: Our running example encoded in GRAFT.

function that again maps trees to trees. This makes it obvious that composing increments in GRAFT means composing blocks of code.

How about base features then? We would need other commands that create trees instead of just adding to them. But GRAFT avoids that by viewing even base features as increments of empty trees. Accordingly, compositions always implicitly start with the empty tree. We have therefore eliminated the asymmetry between base features and increments and, in our analyses, can always talk about GRAFT programs without the need to ever refer to trees. In Fig. 5 we use command sequences to “implement” the example AHEAD system from Fig. 3 in GRAFT.

## 4.2 Representing Design Rules in Graft

Integrating design rule checking in GRAFT means that we need to emulate the following aspects of design rules: first-to-last properties, first-to-last checks, last-to-first properties and last-to-first checks.

The first concept we introduce is that of a *state*, a set of bindings assigning values to properties. If a property does not appear at the right side of any of these bindings, we say that its value is *undefined*. GRAFT programs are evaluated first-to-last so that we can implement the first kind of checks by providing commands to change and check the state. Note that, for the time being, we only support equality in checks.

<code>set <math>p = v</math></code>	Assigns value $v$ to property $p$
<code>checknow <math>p = v</math></code>	Checks that property $p$ has value $v$

The main justification for last-to-first checks is to make sure certain properties (such as the existence of a resource) hold *after* everything has been composed, i. e., at the end of a GRAFT program. We therefore achieve a major simplification<sup>3</sup> of GRAFT evaluation by reusing the first-to-last state for last-to-first and only add one new command:

<code>checkfinal <math>p = v</math></code>	Checks that, at the end of the program, property $p$ has value $v$
--	--

This means that in order to perform the final checks, we first completely evaluate the GRAFT program and then perform the checks against the final “version” of the state. Design rules also have a `single` modifier for declaring that a feature only appear once in a sequence. We emulate this construct by first checking whether a property is defined and then setting it to `true`. To perform this check with an equation, we allow the symbol ‘ $\perp$ ’ denoting “undefined” to appear on the right side:

<sup>3</sup>This simplification implies that GRAFT is not a superset of design rules any more, but in usual practical applications, this is not an issue.

```

checknow baseIsThere =  $\perp$ ;
set baseIsThere = true;
checkfinal presentIsThere = true;

```

(a) Design rules for feature `base`.

```

checknow presentIsThere =  $\perp$ ;
set presentIsThere = true;
checknow baseIsThere = true;

```

(b) Features `presentstdout` and `presentswing` have the same design rules.

Figure 6: Design rules for the converter program expressed in GRAFT: Feature `base` has the following two demands: It must be single, i. e., it can only appear once. Additionally, one of its successors has to be a presentation feature, i. e., such a feature must be *finally* present. Similarly, only one of the presentation features is allowed and each one needs `base` to be a predecessor.

```

checknow  $p = \perp$     Checks that property  $p$  is undefined

```

With these preparations, we can give some example design rules expressed in GRAFT (Fig. 6).

## 5 Graft: Checking

We have already mentioned that we want to perform two kinds of checks: Checks internal to a feature (intra-checking) and checks between features (inter-checking). What kind of intra-checks do we need? It would be nice to have the semantic constraints provided by design rules at a local level, too. So we'll start with these (in a way, we already have). To keep book of our analysis, we store properties and their values as a set of bindings called `State`. And we store final checks in a set `Final`. Next are dependencies. By expressing that if one node is present another one must be present as well, we can, among other things, verify that a Java program is well typed after a modification. Otherwise, we might accidentally remove a method that is invoked by another one and the program would not compile any more. Dependencies are handled similarly to final checks, we only demand that a node that we ask for be introduced “somewhere”. If we view a dependency as a link between a source and a target node, we need to take care of a few special cases: The link disappears if the source is deleted and it must be updated if either source or target are moved. We therefore extend `Final` to also include *dependent checks* that remember the source of their existence. The last intra-check concerns the internal consistency of our GRAFT program: Before an operation can be executed, certain conditions have to be fulfilled. For example, before we can add a new node, we have to make sure that it isn't already there. The wrong sequence of commands can therefore lead to a feature that is internally inconsistent. As we shall see later, the presence of nodes can also be stored in `State` and the consistency (or executability) conditions encoded as checks on `State`. We also include consistency checks that make sure that we always produce well-formed trees.

It turns out that the result of our analysis provides a perfect description of a feature's interface: The declaration of what a feature exports is obviously to be found in `State`. Demands on subsequent features are all those checks in `Final` that have not yet been fulfilled. If we look closer, we notice that non-final checks that don't have a definite answer in `State` (because there is no binding of the property being checked, yet) are actually requirements for previous features. Therefore, we collect them in a set `Pre` which is the third component of the feature interface. It is interesting to note that for none of these analyses, we needed the actual node data. That suggests to provide a skeletal GRAFT program, stripped of the source code, as public information about a feature. One can then check dependencies, design rules and consistency before actually performing the composition; the GRAFT skeleton acts as a type system. Unfortunately, the feature interface *on its own*, while it can be used for checking for all other conflicts, is not sufficient for answering all questions about dependencies: an interface does not tell if a node a feature depends on can be provided by a predecessor or if it is deleted inside and thus can only

be added by a successor. In the future, we might classify these two kinds of dependencies differently in the interface.

## 5.1 Semantics of Checking

The result of analyzing a GRAFT program  $G$  is a *constraint store* that contains *pre-constraints* Pre that must be fulfilled before executing  $G$  and *final constraints* Final that must be fulfilled at the end of any sequence of commands that contains  $G$ . During the computation, constraints are always checked against a current state State that is thus also part of the store.

The analysis comprises the following steps:

1. Translate the GRAFT program  $G$  to a sequence  $S$  of commands in an intermediate *constraint store language* (CSL) which manipulates the constraint store.
2. Apply  $S$  to the empty constraint store.
3. Within the resulting store, the state is used to erase final constraints that are already fulfilled. This leads to the result of the analysis, a “normalized” constraint store. Due to space constraints, we do not give the details of this step here.

Note that each one of the steps mentioned above is a partial operation. Inability to perform any one of them can be seen as a “compilation error”, i. e., the type check has failed.

## 5.2 Constraint Store Language

We give an overview of concepts relevant to CSL.

*Paths:* A *path*  $p$  is a sequence of node names separated by the dot operator ‘.’ (which can also be used for concatenation). Large capital variable names such as  $X$  stand for single components. For example,  $r.X$  is a path that starts with the sequence  $r$  of node names and has  $X$  as the last path component.

*Constraint store:* A constraint store is a triple (Pre, State, Final) where

- Pre is the set of pre-constraints. These are conditions on properties that must be fulfilled prior to executing the program under analysis. A pre-constraint can be viewed as a guard which, if it holds, guarantees correct termination of the program. Pre is encoded as a set of equations, pairs written as  $(p = b)$  where  $p$  is a property name and  $b$  is a boolean value (either true or false).
- State stores the current state, the values of the properties. If a property is set, it appears as a pair  $(p = b) \in \text{State}$ . Undefined properties do not appear on the left side of a pair in State. We sometimes use State as a total function that returns a properties value if it is set and the special symbol  $\perp$ , otherwise.
- Final is the set of post-constraints that are checked against the final “version” of State. Here we find conditions that must hold after the termination of a GRAFT program, such as “make sure that resource  $x$  has been added”. Final generally contains *dependent checks*, an extension of the pair construct in Pre. These are encoded as a triple  $(p = v \text{ depby } q)$  that means that the check  $p = v$  depends on the presence of property  $q$ . If the check does not depend on any node, we write it as  $(p = v \text{ depby } \perp)$ .

## Checks

*$\epsilon$  check:* There is one special property, the so-called  $\epsilon$  property that has the empty name. Any check on this property (“ $\epsilon$  checks”) always and immediately succeeds. See the definition of add below for an example where an  $\epsilon$  check can occur.

*Residuating check:* The question mark operation checks that a binding in the state have a certain value. This operation can only be performed if the check does not contradict a binding in the state. If there is no corresponding binding then we keep the check as a Pre constraint (the check *residuates*).

$$?p = b \text{ (Pre, State, Final)} = \begin{cases} (\text{Pre, State, Final}) & \text{if State}(p) = b \\ (\text{Pre} \cup \{(p = b)\}, \text{State, Final}) & \\ & \text{if State}(p) = \perp \end{cases}$$

*Final check:* The same check as above can be performed “finally”, i. e., we have to add it to the final constraints. The condition means that Final must not become inconsistent by the addition.

$$\begin{aligned} \text{final?}p = b \text{ depby } q \text{ (Pre, State, Final)} = \\ (\text{Pre, State, Final} \cup \{(p = b \text{ depby } q)\}) \\ \text{if } \nexists x \neq b, r. (p = x \text{ depby } r) \in \text{Final} \end{aligned}$$

*Immediate check:* Then there is one last variety of check, the non-residuating “now” check. It allows one to find out if a property has a value yet. In order to do that, we introduce the symbol  $\perp$  to denote “undefined” and can check for undefinedness with the usual equality operator.

$$\begin{aligned} \text{now?}p = v \text{ (Pre, State, Final)} = (\text{Pre, State, Final}) \\ \text{if } \text{State}(p) = v \end{aligned}$$

## Destructive Commands

*Change state:* The exclamation mark operation changes the state in an imperative manner. No inconsistencies can arise, because we overwrite existing bindings.

$$!p = b' \text{ (Pre, State, Final)} = \begin{cases} (\text{Pre, (State} \setminus \{(p = b)\}) \cup \{(p = b')\}, \text{Final}) \\ \quad \text{if } b' \neq b \text{ where } b = \text{State}(p) \\ (\text{Pre, State, Final}) \\ \quad \text{if } b' = \text{State}(p) \end{cases}$$

*Deletion in Final:* As final constraints sometimes depend on a node  $p$  (are “attached” to it), we have to provide an operation that deletes these constraints when  $p$  is deleted.

$$\text{final}-p \text{ (Pre, State, Final)} = (\text{Pre, State, } \{(r = b \text{ depby } s) \in \text{Final} \mid s \neq p\})$$

*Renaming:* The following operation provides the means to rename a binding. To maintain consistency, we also have to change references to it in the final constraints.

$$\begin{aligned} \text{rename } p \rightarrow q \text{ (Pre, State, Final)} = ( & \text{Pre, } \{(m(r) = b) \mid (r = b) \in \text{State}\}, \\ & \{(m(r) = b \text{ depby } m(s)) \\ & \quad \mid (r = b \text{ depby } s) \in \text{Final}\}) \\ \text{where } m(r) = \begin{cases} q & \text{if } r = p \\ r & \text{otherwise} \end{cases} \end{aligned}$$

## Child Defaults

So far, we’ve made sure that **add** always produces well-formed trees. But we cannot make the same guarantee for **delete**, because we don’t have yet the means to express the condition “if we delete a node, it must not have descendants”. Child defaults are properties whose last path component is a wildcard. They allow us to make universally quantified assertions and checks about nodes. For example, when we delete a node  $p$  we have to check that all children are missing  $?p.* = \text{false}$  and potentially remember that as a constraint in Pre. Note that more specific assertions override the default (example: we first add a node  $p$  which guarantees that there are no children and then add one child  $p.X$ . Now the child default is still  $p.* = \text{false}$ , but in addition, we have the assertion  $p.X = \text{true}$ ). We see that inside a GRAFT program, child defaults are used by **add** and **delete** in a complementary fashion: **add** asserts that the new node has no children, **delete** requires its argument to be a leaf.

Here is how we can extend the constraint store to support child defaults:

- Pre has a new kind of entry  $(p.* = b)$ .
- State also contains child defaults  $(p.* = b)$ . Additionally, we redefine the use of State as a function:
- Final is unchanged, we only support child defaults for pre-checks.

$$\text{State}(p.X) = \begin{cases} b & \text{if } (p.X = b) \in \text{State} \\ c & \text{if } (\nexists (p.X = b) \in \text{State}) \wedge (p.* = c) \in \text{State} \\ \perp & \text{otherwise} \end{cases}$$

Operations are extended as follows:

- New residuating check:  $?p.* = b$ :

$$?p.* = b(\text{Pre}, \text{State}, \text{Final}) = \begin{cases} (\text{Pre}, \text{State}, \text{Final}) & \text{if } \forall X. \text{State}(p.X) = b \\ (\text{Pre} \cup \{(p.* = b)\}, \text{State}, \text{Final}) & \text{if } \forall X. \text{State}(p.X) \in \{b, \perp\} \end{cases}$$

- Change state: already works correctly for  $!p = b$  in all instances: if the set command really provides new information, it either overrides an existing binding or complements a default with more specific information. Otherwise, it leaves State untouched. A new variant of the exclamation mark operation can set defaults:  $!p.* = b$  clears all settings for children of  $p$ .
- Rename: Rename must now change the names of the defaults, too. We omit the details.

### 5.3 Translating from Graft to the Constraint Store Language

$$\llbracket \text{add } r.X \dashv \{d\} \rrbracket = ?r = \text{true}; ?r.X = \text{false}; !r.X = \text{true}; !r.X.* = \text{false}; \text{final}?d = \text{true} \text{ depby } r.X$$

Adding a node named  $X$  whose parent has the path  $r$  means that we check that the parent exists and that the node itself does not exist. Then we log that  $r.X$  now exists, has no children and that a check for the existence of  $d$  depends on it. If the node to be added has no father,  $r$  is empty and ascertaining its existence is an  $\epsilon$  check that always succeeds (see above).

$$\llbracket \text{refine } p \dashv \{d\} \rrbracket = ?p = \text{true}; \text{final}?d = \text{true} \text{ depby } p$$

Refinement is similar to adding, but has less side effects. It merges a new node with an existing one.

$$\llbracket \text{delete } p \rrbracket = ?p = \text{true}; ?p.* = \text{false}; !p = \text{false}; \text{final}-p$$

delete performs the usual checks and changes. Before deletion, we have to make sure that all children (whose path is  $p.*$ ) are gone. Additionally, we need to delete dependencies that have become obsolete by this operation.

$$\llbracket \text{move } p \ r.X \rrbracket = ?p = \text{true}; ?p.* = \text{false}; ?r = \text{true}; ?r.X = \text{false}; \text{rename } p \rightarrow r.X$$

Moving is much like first deleting and then adding, but renaming the bindings obviates the need for modifying State and Final. move has an application that is motivated by Bracha's Jigsaw [Bracha, 1992]: Refining a class is very similar to subclassing and move provides an alternative to `super()` calls from Java that is conceptually much cleaner: If one ever wants access to a method one overrides (refines), one just moves the old method out of the way and calls it from the new one. move makes sure that existing references (dependencies) to the old method are correctly updated.

$$\begin{aligned} \llbracket \text{set } e \rrbracket &= !e \\ \llbracket \text{check } e \rrbracket &= ?e \\ \llbracket \text{checknow } e \rrbracket &= \text{now}?e \\ \llbracket \text{checkfinal } e \rrbracket &= \text{final}?e \text{ depby } \perp \end{aligned}$$

Setting and checking translates directly into constraint store language commands.

## 5.4 Example Analysis

How design rule checking works is fairly obvious given the above definitions, we therefore only give an example of consistency and dependency checks. For illustration, we show  $\epsilon$  checks (involving the property whose name is the empty path) in the example when they first appear. Normally, they always succeed immediately. We abbreviate property names inside the constraint store as follows: Converter becomes C, celsiusToFahrenheit c2F, present p, fahrenheitToCelsius f2C. The first example computes the pre and post-constraints of feature `base` from Fig. 5(a).

```

add Converter;
  Pre = { $\epsilon = \text{true}, C = \text{false}$ }
  State = { $C = \text{true}, C.* = \text{false}$ }
  Final =  $\emptyset$ 
add Converter.celsiusToFahrenheit  $\neg$  {Converter.present};
  Pre = { $C = \text{false}, C.c2F = \text{false}$ }
  State = { $C = \text{true}, C.* = \text{false}, C.c2F = \text{true}, C.c2F.* = \text{false}$ }
  Final = { $C.p = \text{true depby } C.c2F$ }
add Converter.fahrenheitToCelsius  $\neg$  {Converter.present}
  Pre = { $C = \text{false}, C.c2F = \text{false}, C.f2C = \text{false}$ }
  State = { $C = \text{true}, C.* = \text{false}, C.c2F = \text{true}, C.c2F.* = \text{false},$ 
     $C.f2C = \text{true}, C.f2C.* = \text{false}$ }
  Final = { $C.p = \text{true depby } C.c2F, C.p = \text{true depby } C.f2C$ }

```

The last version of Final cannot be further reduced, all of these dependencies are open. Next, we want an analysis to find out that the composition `presentstdout;presentswing` (see Fig. 5(b) and 5(c)) fails, because both features add the same method `Converter.present`.

```

add Converter.present;
  Pre = { $C = \text{true}, C.p = \text{false}$ }, State = { $C.p = \text{true}, C.p.* = \text{false}$ }
  Final =  $\emptyset$ 
add Converter.present
  Check ?Converter.present = false fails, as it disagrees with State

```

## 6 Theorems

In this chapter, we briefly sketch what theoretical properties of GRAFT programs and typings can be proven and how. We first introduce a data structure for trees. It holds the tree data plus the state and the final constraints that were created during its construction. Then we define how GRAFT operations change this data structure (omitted here). Finally, we need to define type relations in order to make meaningful assertions in our theorems:

- The type of a GRAFT program  $f$  is the constraint store that we have inferred for it:  $f : (\text{Pre}, \text{State}, \text{Final})$
- Given a tree  $t$  and a set Pre of pre-constraints,  $t$  has type Pre, written  $t : \text{Pre}$ , if every constraint in Pre is fulfilled by the state data in  $t$ .
- If we view a State as a special case of pre-constraints, we can define a relation  $t : \text{State}$  just like in the previous item.

The following theorem asserts that if a tree  $t$  fulfills the pre-constraints of a program  $f$ , applying  $f$  to  $t$  will terminate and produce a tree whose type is the last state of  $f$ .

**Theorem 1 (Subject Reduction)** *Given a tree  $t$  and a GRAFT program  $f : (\text{Pre}, \text{State}, \text{Final})$ . If  $t : \text{Pre}$  then  $(t)f : \text{State}$ .*

Intuitively, the next theorem says that for any GRAFT program  $f : (\text{Pre}, \text{State}, \text{Final})$ , Final correctly describes what dependencies that were declared in  $f$  have not yet been fulfilled. The meta-function `deps` extracts all dependencies that are declared in a program. Set difference  $C \setminus \text{State}$  between a set  $C$  of constraints and a state is defined as removing all constraints from  $C$  that are fulfilled by State. The subset relation  $C \subset D$  between sets of constraints has the obvious definition and ignores the source of a constraint.

**Theorem 2 (Correctness of Final)** *If  $f : (\text{Pre}, \text{State}, \text{Final})$  then  $\text{deps}(f) \setminus \text{State} \subset \text{Final}$ .*

## 7 Related Work

*A Framework for the Detection and Resolution of Aspect Interactions* (DRAI): [Douence et al., 2002] take an approach to formalizing Aspect-Oriented Programming (AOP, [Kiczales, 2001]) that is similar *in structure* to how we formalized AHEAD: They first present a formal model for AOP and then perform analyses on programs that have either been directly expressed in that model or translated to it. But this is where the similarities end: DRAI models the dynamic execution of a program. Behavior that is to be added to the base program is kept separate from it, as a set of *aspects*. These are expressed in a special programming language as condition-action rules plus the means for expressing sequential composition, choice and recursion. Execution of the base program is observed by a *monitor* and drives the parallel evaluation of the aspects: Whenever the currently active condition in an aspect matches the present state of the base program, the action associated with it is executed. That is, the action is *woven* into the behavior of the base program. Analysis is only concerned with making sure that only one of the simultaneously active rules is applicable at a time, leading to deterministic execution, no matter how many aspects are present. In contrast, GRAFT formalizes a static approach to composition that focuses on scalability and support for many artifact kinds (some of whom cannot be executed). At a fundamental level, though, detection and resolution of conflicts *is* similar: Both DRAI and GRAFT search for operations that contradict each other and provide the means for manually eliminating the contradiction. Additionally, GRAFT's properties and checks permit enforcement of semantic constraints and dependencies that often cannot be inferred from the syntax and complement conflict prevention.

*A Calculus of Module Systems* (CMS): There are many module calculi out there, CMS is one of the most recent and general. It has been inspired by work from Bracha [Bracha, 1992] and Leroy [Leroy, 2000]. CMS is based on the observation of two commonalities between many module calculi: First, the language for manipulating modules including the inter-module namespace (the *module language*) and the actual implementation language (or *core language*) should be separate. Second, modules should correspond to compilation units and operators to extra-linguistic tools like a linker. These goals and the static view of modules are similar to GRAFT. Checking in CMS depends on a module interface that describes imports and exports. These correspond to GRAFT's consistency constraints and dependencies on the one hand and to the State construct on the other hand, which are more flexible. Resolving dependencies in CMS resembles how it is done in GRAFT: A *sum* operator combines two modules, a *freeze* operator performs the actual dependency resolution. There are also marked differences: The focus of CMS is theoretic, it is able to encode the Abadi-Cardelli object calculus [Abadi and Cardelli, 1996] and lambda calculus; GRAFT grew out of the desire to perform checking for AHEAD. The CMS way of composition is algebraic, GRAFT's ideas lean towards tree transformation and better suit the needs of generative programming (for example, CMS can only perform selection of sub-components on *concrete* modules, i. e. modules where all input components have been resolved). Finally, CMS is targeted at programming languages, while GRAFT needs to support many artifact kinds. CMS does not have an equivalent to design rules, either.

*Evolution Contracts*: Evolution contracts [Mens and D'Hondt, 2000] provide a formal foundation for automated software evolution, especially for applying and propagating changes to software systems. A central focus is on detecting evolution conflicts where either a change violates consistency rules or aggregated changes are incompatible. In order to support the various kinds of artifacts that arise at different steps of a software development process, evolution contracts are defined as an extension to the UML meta-model. Changes are applied in a stepwise fashion through *evolution contracts*, rules containing a *provider clause* that describes the nature of the element to be modified and a *modifier clause* specifying what changes are to be made. The possible changes are classified as the *contract types* called *Addition*, *Removal*, *Connection* and *Disconnection*. The function of the first two is obvious, the latter to add an remove relationships between model elements. Evolution contracts (and the artifacts to be evolved) are expressed in formalisms such as *specialisation interfaces* [Lamping, 1993], encoded as UML. As these contain, among other things, information about interdependencies between methods, various complicated conflict analyses can be performed. In order to scale up, evolution contracts employ techniques that are similar to AHEAD's (and are, in fact, general principles often used in computer science): Data is nested using the UML *Package* mechanism. Contracts operate at different levels of nesting (i. e. abstraction)

and are grouped to sequences. Finally, it is possible to define new contracts using UML meta-modeling. To summarize, evolution contracts are a very powerful formalism for supporting software evolution and detecting conflicts while doing so. The comprehensiveness of this approach is impressive [Mens, 2001], but its complexity can be daunting and evolution and analysis steps must often be defined and applied by hand. Contrarily, GRAFT keeps things simple, custom semantic constraints are easily implemented using design rule checks and the analysis is performed automatically. The focus of evolution contracts on software evolution is also slightly different from GRAFT's focus on generative programming.

*Design Rule Checking:* Design rule checking has already been treated in depth above. See [Batory and Geraci, 1997] for additional background on it.

## 8 Conclusions and Future Research

**Acknowledgements:** We thank Don Batory for many helpful comments.

We have presented GRAFT, a calculus that provides a theoretical foundation for the generator framework AHEAD. Our main contributions are: A clear and concise definition of the semantics of AHEAD, automated analyses for intra- and inter-checking of features and automatic inference of an interface describing what a feature needs and what it offers. In order to fulfill the requirements of real-world applications, our approach is also independent of the artifact language being used and has a delete operator whose non-monotonicity was an added challenge for the analyses.

Future research will enrich the GRAFT language with further features such as lambda abstraction, the ability to reflect on the contents of a node, loops, conditionals etc. Dependencies are also just a small first step in describing various relationships between and properties of artifacts. In this field there are many ideas from other works that are applicable, among others: Behavioral interface specification [Burdy et al., 2003], statechart diagrams [Prehofer, 2003], specialisation interfaces [Lamping, 1993] and refinement calculus [Büchi and Sekerinski, 1997]. Typing is another issue: Is the current mechanism enough or would we want more than just constraints for typing a GRAFT program? Tree types similar to XML Schema [xml, 2001] would also probably be a useful addition. Easy and logical minor enhancements include support for property types other than boolean and more checks than just equality (both things are already present in design rule checking). Lastly, we have implemented a previous version of GRAFT in Java and plan to update it to the status quo.

## References

- XML Schema Part 0: Primer. W3C Recommendation, May 2001. URL <http://www.w3.org/TR/xmlschema-0>.
- Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
- Don Batory et al. Homepage of the AHEAD Tool Suite. URL <http://www.cs.utexas.edu/users/schwartz/ATS.html>.
- 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. 25<sup>th</sup> IEEE Int. Conf. Software Engineering (ICSE)*. IEEE, 2003a.
- Don Batory, Jack Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. Submitted to journal publication, 2003b.
- Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
- Martin Büchi and Emil Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In *Proc. 2<sup>nd</sup> Workshop on Component-Oriented Programming (WCOP) held in conjunction with ECOOP, Jyväskylä*, June 1997.

- Lilian Burdy et al. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *8<sup>th</sup> International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, June 2003.
- Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.
- 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. 1<sup>st</sup> Conf. Generative Programming and Component Engineering (GPCE)*, volume 2487 of *Lect. Notes Comp. Sci.*, pages 173–188, 2002.
- Michel Jaring and Jan Bosch. Representing Variability in Software Product Lines: A Case Study. In *Proc. 2<sup>nd</sup> International Conf. on Software Product Lines (SPLC)*, Lecture Notes in Computer Science, pages 15–36. Springer, 2002.
- Gregor Kiczales. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.
- John Lamping. Typing the Specialisation Interface. In *Proc. 8<sup>th</sup> Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 28 of *ACM SIGPLAN Notices*, pages 201–214. ACM Press, 1993.
- Xavier Leroy. A Modular Module System. *Journal of Functional Programming*, 10(3):269–303, May 2000.
- 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.
- Christian Prehofer. Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams. *7<sup>th</sup> International Workshop on Feature Interactions in Telecommunications and Software Systems*, Ottawa, 2003.
- Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.