

Catching CATs

Obstacles on the path towards a fully typesafe Eiffel

Diploma Thesis

Markus Keller

Advised by Bernd Schoeller
and Prof. Dr. Bertrand Meyer

Chair of Software Engineering

ETH Zürich

2003-03-10 – 2003-07-09

Acknowledgements

First of all, I thank Prof. Bertrand Meyer for giving me the opportunity to study a central problem of object-oriented software construction. I'm indebted to Bernd Schoeller for his advice, guidance and encouragement in times of pressure, and for the fruitful discussions on and off topic.

I'd also like to thank all my fellow students for the interesting discussions we had in the students' lab during the summer. Last but not least I want to thank my parents and my brother for their personal support.

Contents

1	Introduction	4
1.1	Overview	4
1.2	Scope of the work	4
1.3	Introduction to object-oriented concepts	5
1.3.1	Static Typing	5
1.3.2	Polymorphism	5
1.3.3	Dynamic Binding	6
1.3.4	Constrained Genericity	6
1.3.5	Covariance	7
1.4	Problems with Covariance	8
1.4.1	Catcalls	9
1.4.2	Generic conformance	9
1.4.3	Three approaches to ensure strong typing	9
2	Solutions with dynamic typing	9
2.1	Run-time type checking	9
2.2	Eiffel “recast” proposal	11
2.3	Multimethods	11
2.4	Conclusion	12
3	System Validity	13
3.1	Description	13
3.2	Problems in the algorithm	14
3.3	Straightforward extensions to the algorithm	15
3.4	Manifest Arrays have no static type	16
3.5	Function result type anchored to an argument	18
3.5.1	Throwing everything into one bag	22
3.5.2	Clone Wars	24
3.6	Interfacing with external routines	26
3.6.1	Sources and sinks	28
3.7	The POINTER type	28
3.8	Dynamic class loading	29
3.9	Conclusion	30
4	Implementation of a System Validity Checker	30
4.1	Parsing Eiffel class files	30
4.2	Gobo Eiffel Tools	32
4.3	Flattening classes	33
4.3.1	Resolving called features and static types of expressions	35
4.4	Flattening types	36
4.4.1	Computing and storing dynamic type set information	38
4.5	A System Validity checker is a compiler (and more)	41
4.6	A language has only <i>one</i> typing policy	42

5 Conclusion	43
6 Glossary	44
7 References	45

1 Introduction

1.1 Overview

Object-oriented programming languages strive to be expressive and safe at the same time. There are situations where the standard ingredients of fully object-oriented languages interfere and expose a hard problem for language designers. Pierre America's conjecture [Ame90] was rather drastic: Of the three protagonists *static typing*, *substitutivity* (subtype polymorphism) and *covariance*, at most two can be on the stage at the same time.

Covariance is “the policy allowing a feature redeclaration to change the signature so that the new types of both arguments and result conform to the originals” [Mey97]. This flexibility is required by the proposed dual role of a class as type and reusable module. Furthermore, Eiffel allows changing the export status of features in descendant classes.

Unfortunately, it is easy to devise redefined features which cannot be used polymorphically in every context the original routine could (and thus break straightforward static typing). The term **CAT** describes features that **C**hange **A**vailability or **T**ype. *Catcalls* are calls to routines which are CAT in any redefinition. Calling them on a polymorphic entity breaks static typing and is a bug.

The Catcall problem potentially appears in every object-oriented language, but is only identified as such in advanced languages with a strong yet flexible static type system, which allows covariant redefinitions.

1.2 Scope of the work

[Section 1.3](#) will give a short introduction to object-oriented concepts as used in Eiffel and other languages. In [Section 1.4](#), problems of covariance are discussed. Covariant attribute and parameter redefinitions are nevertheless very useful mechanisms which allow for reuse and adaptations of classes. Still, they open up possibilities to break the static type system, which is not acceptable.

Dynamically typed systems are discussed in [Section 2](#). It is shown how they “solve” the problems discussed in [Section 1.4](#) by deferring certain validity checks until run time. [Section 3](#) elaborates on various examples of statically (i.e. compile time) safe type systems and explains the advantages and downsides of the various proposals.

System Validity tries to solve the static typing problem through a global analysis. In [Section 3](#), the approach is described and a number of issues with the proposed algorithm are presented. It is shown how System Validity interferes with other language mechanisms. Especially [Sections 3.6](#) through [3.8](#) demonstrate, that System Validity cannot be applied to open systems.

Section 4 reports on the encountered difficulties while trying to implement a System Validity checker. Approaches to parse Eiffel source files are discussed, and a possible design of a checker is sketched. **Section 4.5** explains why a complete implementation of a System Validity checker was out of range for this thesis. **Section 4.6** argues that the type checking policy is an integral part of a language, and that the initial intention of creating a configurable “testbed” for trying different typing policies on existing Eiffel source code is hardly achievable.

1.3 Introduction to object-oriented concepts

This section introduces various fundamental concepts of object-oriented programming, as presented in [Mey97]. The description is restricted to concepts of class-based languages and to mechanisms which will be shown to cause problems if used together.

1.3.1 Static Typing

Static typing is the ability to check, on the basis of the software text alone, that no execution of a system will ever try to apply to an object an operation that is not applicable to that object.

To enable an automated tool (a *static type checker*) to prove the above statement, the language must provide the necessary type annotations: Every entity (i.e. every identifier in the software text which can refer to a run-time object) must bear an explicit type annotation, called its static type. This includes formal routine arguments and function return types. A type (usually based on a class) defines all possible feature calls on an entity declared of that type.

Explicit static typing has a number of benefits, which will be quickly summarized here. For a more thorough discussion, see [Mey97], [PS94], or your favorite book on object-oriented programming.

Types enhance the readability of a program. They are a valuable form of documentation, since they reduce complexity by explicitly reducing the possible set of values which an entity can contain. At the same time, the static type of an entity restricts possible feature calls to those which are sensibly defined. Thus, types are also a tool for enhancing the safety of a program, since they ban invalid feature calls at compile time. If a program is statically type safe, we can avoid costly and unnecessary run-time checks, which yields a gain in efficiency without sacrificing safety.

Types are a very important tool for software design and construction. Since types can be organized in subtyping hierarchies, their relationships help in understanding the models and abstractions upon which a software is built.

The static type system can be seen as a restricted form of compile-time Design by Contract [Mey97]. It guarantees that for every feature call, a feature implementation is available. This implies a system-wide invariant: the absence of run-time errors due to undefined features.

1.3.2 Polymorphism

“Polymorphism is the ability for an entity to become attached to objects of various possible types” [Mey97]. Polymorphism, together with dynamic binding, is a key property to

build extensible object-oriented systems. For example, it allows to inject new functionality into an otherwise closed component:

```

class MENU_ITEM
...
  action: ACTION
...
  set_action (an_action: ACTION) is
    do
      action := an_action
    end
...
  engage is
    do
      action.execute
    end
...
end

class ACTION
...
  execute is
    deferred
  end
end

class QUIT_ACTION
...
  execute is
    do
      application.quit
    end
  end
class ...
...
  fill_menu is
    local
      an_item: MENU_ITEM
    do
      ...
      create an_item.make ("Quit")
      an_item.set_action (
        create {QUIT_ACTION}
      )
    end
  end
end

```

Here, the call to routine *set_action* takes a *QUIT_ACTION* as argument, where the declaration of the routine in class *MENU_ITEM* only knows about general *ACTIONS*.

Polymorphism needs to be restricted in order to ensure static type safety. A reattachment is only allowed when the type of the source object conforms to the type of the target entity.

1.3.3 Dynamic Binding

Dynamic Binding is the policy of selecting the concrete version of the called feature based on the *dynamic type* of the target. It is the second ingredient we need to implement flexible systems without touching closed modules.

In the above example, the call to *action.execute* must clearly select the version of routine *execute* based on the type of the object attached to *action*. Otherwise, a *deferred* routine would be called, which is certainly not the right thing to do! Instead, the version of *action*'s dynamic type should be executed.

1.3.4 Constrained Genericity

For some applications, e.g. collection types, it is essential to have a notation to *parameterize* a class. Constrained genericity is the object-oriented mechanism which allows to create classes with a generic parameter that is constrained by a given type.

Class *HASH_TABLE* from EiffelBase [ES03] is a good example to show how constrained genericity enables the definition of reusable data structures. The class declaration *HASH_TABLE [G, H -> HASHABLE]* has two open parameters: *G* is a placeholder

for the type of the values and $H \rightarrow HASHABLE$ restricts the allowed types of keys in the table to types conforming to $HASHABLE$.

When clients declare an entity for storing a hash table, they supply the concrete types: *people_by_name*: $HASH_TABLE [PERSON, STRING]$ is a hash table with instances of class $PERSON$ as values and instances of $STRING$ as keys.

1.3.5 Covariance

Covariance is the act of redeclaring the type of attributes or routine arguments into more specific types when inheriting from a parent class. This is often useful when parallel inheritance hierarchies are used. We will consider a simple example here, but the same problem also appears in real code, especially when complex frameworks are extended.

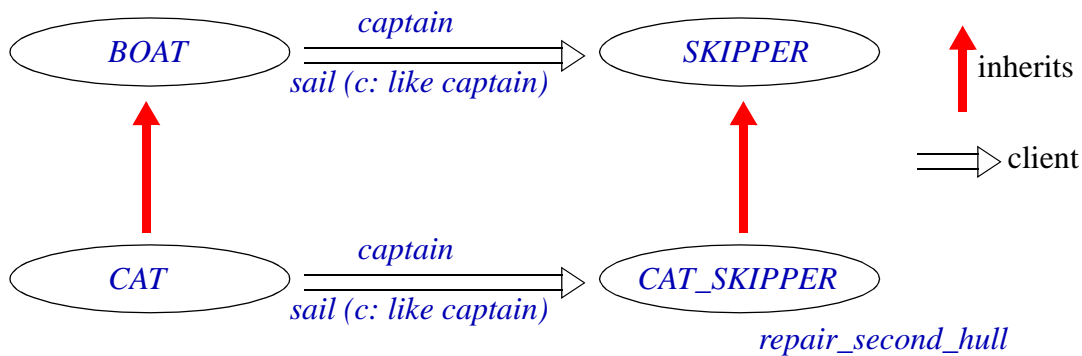


Figure 1: The cat-skipper example (from [HBM+03]).

Figure 1 is our working example. Objects of type $BOAT$ are normal boats, which can be operated by any instance of $SKIPPER$, whereas a CAT (standing for “catamaran”) can only be sailed by a $CAT_SKIPPER$, who knows how to *repair_second_hull*.

The classes can be implemented like this:

<pre>class BOAT feature captain: SKIPPER -- Skipper assigned to this boat sail (c: like captain) is -- Appoint c as captain of this boat. do captain := c end end</pre>	<pre>class CAT inherit BOAT redefine captain end feature captain: CAT_SKIPPER -- Catamaran skipper assigned to -- this catamaran end</pre>
--	---

$SKIPPER$ is not relevant for this example and can be assumed an empty class.

$CAT_SKIPPER$ inherits $SKIPPER$ and adds a routine *repair_second_hull*, whose implementation is not further relevant.

captain is redefined covariantly from $SKIPPER$ to $CAT_SKIPPER$. As the term *co-variant* suggests, the redefinition goes into the *same* direction as the inheritance hierarchy:

As *CAT* conforms to *BOAT*, the new type *CAT_SKIPPER* of *captain* conforms to *SKIPPER*.

A special mechanism of the Eiffel language is used here to avoid an explicit redefinition of routine *sail*: The *anchored type declaration* “*like captain*”. Anchored types are a useful tool in practice, when many types declarations would have to be adjusted only because one query changed its type. In our example, we could as well have written *sail* in *BOAT* as *sail (c: SKIPPER)*, but then we would have had to redefine *sail* in *CAT* explicitly to *sail (c: CAT_SKIPPER) is do captain := c end*. For now, we can think of anchored types as simple textual substitutions. However, [Section 3.5](#) will show cases where anchored types cannot be resolved as easily as here.

1.4 Problems with Covariance

[Mey97] and [HBM+03] describe problems caused by covariant redefinitions at full length. This section quickly summarize the issues.

The example in [Figure 1](#) is not as innocent as it looks on first sight. Using the classes as it was probably the supplier’s intention looks like this:

```
boat: BOAT; cat: CAT
skipper: SKIPPER; cat_skipper: CAT_SKIPPER
...
... create boat, cat, skipper, cat_skipper
boat.sail (skipper)
cat.sail (cat_skipper)
```

Everything’s fine up to now. There may also be situations, in which we want to assign a *CAT* to an entity of type *BOAT* (e.g. to dock it at a pier where concrete boat categories are of no importance):

```
docked_boat: BOAT
...
docked_boat := cat -- note: polymorphic reattachment
```

Still, nothing bad happens. Soon, the above *skipper* comes along and wants to sail a *BOAT*. She decides for *docked_boat*, unfastens the rope and heads for the open sea:

```
docked_boat.sail (skipper)
```

Whew! That calls for trouble. Remember that *docked_boat* was attached to a *CAT* a few lines ago. Even though *sail* is declared as *sail (s: SKIPPER)* for targets of static type *BOAT*, the call with an argument of type *SKIPPER* is not valid on targets of dynamic type *CAT*. *CATs* may only be sailed by trained *CAT_SKIPPERs*. Consequences of this typing error are severe if not detected immediately. If the second hull of the *CAT* leaks for any reason, a proper *CAT_SKIPPER* would execute the standard procedure to *repair_second_hull*, and solve the problem with ease. But our poor *skipper* doesn’t know how to *repair_second_hull*, since that is only taught in a *CAT_SKIPPER*’s training! Therefore, it is imperative to prevent invalid calls like *docked_boat.sail (skipper)*.

1.4.1 Catcalls

A routine is *CAT* (Changing Availability or Type), if some redefinition of the routine changes its export status or the type of any of its arguments. We will not further discuss the first cause here, since it is only useful in rare cases such as ostrich-oriented programming. Routine *sail* from class *BOAT* in our example is *CAT*, since the type of its argument is redefined in class *CAT* from *s: SKIPPER* to *s: CAT_SKIPPER*.

A *catcall* is a call which can lead to a typing error since the called routine is *CAT*. In our example, *docked_boat.sail (skipper)* is a *catcall*.

1.4.2 Generic conformance

Besides redefinitions, there's a second mechanism which can cause *catcalls*: Conformance between different generic derivations. In Eiffel, *LIST [CAT]* conforms to *LIST [BOAT]*, since *CAT* conforms to *BOAT*. Let's look at that in another example:

```
cat_list: LIST [CAT]
some_boat_list: LIST [BOAT]
...
... create a cat_list, possibly filled with instances of type CAT
some_boat_list := cat_list
some_boat_list.extend (boat)
```

Routine *extend* of class *LIST [G]* is declared as *extend (v: G)*. Therefore, we can say that *extend* in *LIST [BOAT]* is *CAT*, since in *LIST [CAT]*, it changes the type of argument *v* from *BOAT* to *CAT*. Thus, *some_boat_list.extend (boat)* is a *catcall*.

1.4.3 Three approaches to ensure strong typing

There are basically three ways to ensure that no typing errors will happen at run time:

- 1 • *Type system without covariant redefinitions*. Problematic *catcalls* are not possible any more. The price to pay is a huge loss of flexibility.
- 2 • *Dynamic type checks before each catcall*. [Section 2](#) describes dynamic approaches, which check for invalid *catcalls* at *run time* and raise an exception in case of an argument type mismatch.
- 3 • *Global static analysis*. If an analysis of the whole system can prove the absence of invalid *catcalls* at run time, the problem just disappears. With covariant redefinitions: global static analysis to ensure that no *catcall* can happen at run time. System Validity is such an approach and it is described in [Section 3](#).

2 Solutions with dynamic typing

2.1 Run-time type checking

Smalltalk [\[GR83\]](#) went its very own way to avoid all these problems with static type systems. That language just dropped any kind of static type checking in favor of run-time routine lookup. With that decision, all typing related problems disappear magically. At least they disappear at compile time. Because Smalltalk has no static types, it is not clear, what routines can be called on a certain target. Therefore, every routine call must be pre-

ceded by a dynamic check, which determines whether the routine is actually defined. If it is defined, the call is executed. If it is not defined, an exception is raised with the famous note: “Message Not Understood”.

Despite of the performance problems of such an approach, the solution is not really satisfactory: The detection of invalid calls is deferred until run time, which is relatively late. We would like to be warned about problems *before* they happen, and not after we have been bitten.

Java [Java] seems to have taken another path. All entities in Java have an explicit static type annotation, and routine calls are statically typesafe. But neither genericity nor any kind of type redefinitions are supported (genericity is being discussed, but not released yet). In practice, Java programmers simulate this missing functionality by explicit *dynamic casts*.

The cat-skipper example from [Figure 1 on page 7](#) would be written like this in Java:

```
public class Boat {
    protected Skipper captain;

    public Skipper getCaptain() {
        return captain;
    }

    public void sail(Skipper s) {
        captain = s;
    }
}

public class Cat extends Boat {
    public CatSkipper getCatCaptain() {
        return (CatSkipper) captain;
    }

    public void sail(Skipper s) {
        if (s instanceof CatSkipper) {
            captain = s;
        } else {
            throw new IllegalArgumentException(
                "s is not a CatSkipper");
        }
    }
}
```

Every time a client accesses the captain of the *Cat*, an explicit dynamic cast of the form *(CatSkipper) captain* must be executed to allow the contents of the attribute *captain* to be seen as a *CatSkipper*. On the other side, the routine *sail* must dynamically check, whether the given argument is really a *CatSkipper*. Otherwise, the routine cannot do anything but raising a run-time exception.

You can imagine that this practice of explicit dynamic type tests and dynamic casting quickly becomes fragile, since the static checks are replaced by run-time checks. The interface of routine *sail* in class *Cat* does not properly reflect the fact, that the implementation will only accept *CatSkippers*.

Things get even worse when collections of objects are used. Since class *List* cannot know what type of objects it will contain, it simply declares its interface with type *Object*, Java’s root type. Consequently, every call to *get(): Object* must be followed by a dynamic type check on the resulting object.

Since Java programs rely heavily on this dynamic cast mechanism, we cannot say that Java programs are statically typed (even though the language itself is).

2.2 Eiffel “recast” proposal

[HBM+03] is a recent proposal which aims at a fully typesafe Eiffel. Its main idea is to require a *recast* clause whenever an entity is redefined covariantly. In the cat-skipper example, we would have to rewrite the routine *sail* as:

```
sail (c: CAT_SKIPPER) is
  recast
    trained_as_cat_skipper
  do
    captain := c
  end
```

The idea is to call a transformation function *trained_as_cat_skipper* (*s*: *SKIPPER*): *CAT_SKIPPER*, whenever catcall with an argument of the original type *SKIPPER* happens at run time.

This proposal makes Eiffel theoretically typesafe, but its practicability is unknown. In my opinion, the solution doesn’t solve more real problems than the above solutions with run-time checks and exceptions. In real situations, catcalls are very often *not* resolvable by a transformation. In the common case of framework inheritance (see [MN96]), they are just plain errors.

When a manager puts together a sailing exercise group, he must take care of the fact that a “plain” skipper cannot sail a catamaran. If he wrongly declares a plain skipper to be captain of a catamaran, he’ll be out of luck: In a reasonable scenario, he doesn’t have the possibility to train an arbitrary plain skipper for sailing catamarans before the exercise begins. It’s not the scheduler’s task to train people in an ad-hoc fashion. What he’s got to do is a rewrite of the training plan, taking actual skills into account.

Coming back to the software example, the only remaining “solution” to catch the erroneous situation is then to raise a run-time exception. This dynamic approach to solve the typing problem is not helpful to protect from errors without relying on test runs.

2.3 Multimethods

[BCC+96], [Cecil] and others try to solve problems with covariant argument redefinitions by using multimethods. Multimethods are routines which not only look at the target to determine the actually executed routine body, but they also take the (dynamic) types of the routine’s arguments into account.

Multimethods can be seen as a generalization of the “recast” proposal from Section 2.2. Where “recast” enforces a sequential execution of transformation routine, followed by the redefined routine, multimethods are free to implement the different routine bodies independently.

[Kur00] views the covariantly redefined method as a partially overriding specialization of the original routine, since it only applies to particular argument types.

Although the approach looks nice at first sight, it fails to address the real problem. Cases where an additional dispatch on argument types are useful are rare. Far more often, calling the routine with the original argument type is not another case, which needs another

handling, but it is simply a bug. The only thing we can do for “illegal” combinations is again throwing an exception.

2.4 Conclusion

The dynamic solutions presented in this section are all variants of the same pattern. They give the programmer more or less sophisticated notations to express covariant argument redefinitions, but they fail to provide a semantically richer model than what is known from Java. The approaches work well for safe feature calls and handle possibly unsafe calls with differing syntactical devices. Nevertheless, they fail to detect erroneous calls at compile-time, which lessens their usefulness for raising the quality of reusable software.

The main problem is, that dynamic approaches defer the detection of erroneous calls until run-time. Type mismatches are only detectable at the place where an erroneous call is executed. That late, we have only 2 possible solutions at hand:

- Ignore the issue and execute the call without further precautions.
- Spread dynamic type checks all over the code to detect unsafe calls.

The former approach is what C-style static casts are about, and accounts for the majority of security and stability issues in current software. This path is clearly not to be taken, since it can cause arbitrary misbehavior and completely destroys predictability of software execution.

The latter solution works as follows: Whenever an unsafe call is performed, run-time checks reveal the problem and deal with it in a controlled and predictable manner: an *exception* is raised. Even though the type system fails to unhide the doomed call at compile time, it specifies at least what’s going to happen at run time in case of an erroneous call. That’s the best we can expect from a dynamic solution, but it comes at a price: *every* call which might fail at run time must be equipped with a type check. That’s a serious performance hit, which is evident in the dissatisfactory performance of such languages: Smalltalk, Java, and .NET, etc.

For concrete applications, dynamic approaches are mostly just not honest. They don’t rule out erroneous programs until the latest possible time – execution. The only “solution” to prevent arbitrary crashes is then to raise a run-time exception (of the well-known Smalltalkish “Message not understood”).

[Sha96] asks: “Isn’t that an ostrich-like policy? We cannot attempt to avoid the danger by refusing to face it.” Later, he continues: “Interface specification plays an important role in software systems. If a fact requires a precise interface, the language should be able to provide a suitable specification that does not mislead the interface user.”

Another serious issue is the lack of a protagonist to blame. Should the implementor of the called feature be blamed if a vicious caller supplies an inappropriate argument? Clearly not! So we have to blame the caller, who used an interface with statically perfectly reasonable arguments? That doesn’t seem to be the rightfully accused, too!

3 System Validity

3.1 Description

System Validity is a conservative global analysis of an Eiffel system, which should ensure static type safety. Bertrand Meyer posted a first version on `news:comp.lang.eiffel` with the subject “Static typing for Eiffel” in 1989. A revised version was published in the first Eiffel language reference [Mey92]. The latest refinement can be found in [Mey0?], which is a work in progress of the next version of the Eiffel language, which is on its way to become an official ECMA standard.

The goal of System Validity is to check static type safety of an entire Eiffel program (a system). This is done by computing the *dynamic type set* (DTS for short) of every expression and entity in every type used in the system:

“For every variable e of an entire program (a system), the static checker will compute a set of types, the *dynamic type set* of e , that includes the types of all objects to which e may become attached during any execution. Then to typecheck every call $x.r(a, \dots)$ it will consider that x and the arguments may take on not only their declared types but also every type of their dynamic type sets.

The dynamic type set of e doesn't have to be the *exact* set of run-time object types for e , which would be impossible or very hard to compute; it has to *include* all these types. System validity, then, takes a possibly pessimistic approach, in line with the earlier observation that static typing is by nature pessimistic. To avoid performing complicated control flow analysis, it considers that any assignment $e := f$ in the system text causes the dynamic type set of e to contain all the types in the dynamic type set of f .” [HBM+03]

Basically, a System Validity check proceeds in 3 phases:

- 1 • Collecting all types used in the system. This set of used types consists of
 - types directly based on a class
 - all used derivations of generic types

Note that the analysis requires that all possible run-time types are known at compile time. We will see later (in [Section 3.8](#)), how this requirement is an impediment for practical extensible systems (e.g. for a plug-in mechanism).

- 2 • Computing the dynamic type sets in an iterative process as defined in [Figure 2 on page 15](#).
- 3 • Checking calls with respect to all possible dynamic types of the target. Due to polymorphic binding, the target of a call can be an instance of every type of the corresponding DTS. That's why we have to typecheck the call on every possible dynamic type separately. A call to the feature would be an (invalid) catcall, iff the called feature was hidden in one of the descendants (i.e. its export status was restricted) or if a formal argument of the feature was redefined covariantly. If the dynamic type set of the target contains any type in which the call is CAT (changing availability or type), then we have found a system validity violation because the call's semantics is not specified. Export-restricted features should not be visible and calling them is therefore an error. Features which changed their arguments' types covariantly are not to be called with actual arguments of the original types. The feature is simply not

defined for such arguments and the effects of calling it with wrong argument types are implementation dependent and not predictable

Since the approach is somewhat pessimistic in that it doesn't take control flow into account, it might wrongly accuse a system of being invalid, even though a more thorough analysis could show that the system is statically typesafe.

Another problem is the late detection of system validity violations. The global approach makes it impossible for suppliers of a library to call their product "typesafe", since the notion of static type safety is only defined for complete systems (which include client code as well). Therefore, client developers can be faced with system validity violations at any time they add a reattachment anywhere in the system. In case of system validity violations, it is difficult to come up with sensible error messages. In the end, who is to be blamed for a violation? The best a system validity checker can do is to emit a *witness path*, which is a sequence of reattachments that could lead to an illegal routine call. Still, the user might be faced with code references to locations deep in a library, where he has no chance to "correct" an invalid system.

This lack of possibility to typecheck a library is a major drawback of the proposal. Static typing is widely recognized as an essential language mechanism for various reasons explained in [Section 1.3.1](#).

3.2 Problems in the algorithm

Apart from the usability problems mentioned above, the proposed algorithm for the computation of the dynamic type set is in fact not complete. For our convenience, the rele-

vant definition of dynamic type sets is reprinted here (with corrected rule numbering). The full mechanism is described in [Mey0?].

Dynamic type set (DTS)

The dynamic type sets of the expressions, entities and functions of a system result from performing all possible applications of the following rules to every Class_type T , of base class C , used in the system:

- 1 • If a routine of C contains a creation instruction, with target x and creation type U , the dynamic type set of x for T is $\{U_T\}$.
- 2 • The dynamic type set for T of an occurrence of *Current* in the text of a routine of C is $\{T\}$.
- 3 • For any entity or expression e of expanded type appearing in the text of C , if the type ET of e is expanded, the dynamic type set of e for T is $\{ET_T\}$. (Rules 4 to 7, when used to determine elements of the dynamic type set of some e , assume that e 's type is not expanded.)
- 4 • If a routine of C contains an Assignment of target x and source e , the dynamic type set of x for T includes (recursively) every member of the dynamic type set of e for T .
- 5 • If a routine of C contains an Assignment_attempt of target x , with type U , and source e , the dynamic type set of x for T includes (recursively) every type conforming to U_T which is also a member of the dynamic type set of e for T .
- 6 • If a routine of C contains a call h of target ta , U is (recursively) a member of the dynamic type set of ta for T , and tf is the version of the call's feature in the base class of U , then the dynamic type set for U of any formal argument of tf includes every member of the dynamic type set for U_T of the corresponding actual argument in h .
- 7 • If h , tf and U are as in case 6 and tf is an attribute or function, the dynamic type set of h for T includes (recursively) every member of the dynamic type set for U_T of the Result entity in tf .

Figure 2: Dynamic type set calculation ([Mey0?], §24)

Note that the algorithm leaves various Eiffel language constructs unspecified. Some of them are easy to fill in, but others eventually lead to situations where the dynamic type sets are not predictable in a meaningful way. We omit most recent additions to the Eiffel language (such as tuples and agents, see e.g. [Mey0?], §13 and §25) from our considerations. Their semantics is still in flux and different Eiffel compilers handle them in rather diverging ways. Moreover, it is unclear whether agents can (or should) be statically type safe or whether dynamic checks are deemed good enough to ensure their type safety.

3.3 Straightforward extensions to the algorithm

- The DTS of *Void* is $\{ \}$ (the empty type set).

- The DTS of manifest constant expressions should be defined for each manifest constant. This is clear for manifest strings ($\{STRING\}$), *True* and *False* ($\{BOOLEAN\}$), and manifest characters ($\{CHARACTER\}$). It is less clear for manifest numbers, but it seems to be reasonable to choose the smallest numeric type which can hold the manifest constant. The problems with manifest arrays follow in [Section 3.4](#).
- The DTS of a constant attribute is the declared type of the feature. The DTS of a unique attribute is $\{INTEGER\}$.
- The DTS of an equality expression (“=” and “/=”) is $\{BOOLEAN\}$.
- The DTS of an *old* expression is the DTS of the referred expression.
- Reattachments (rules 4–6) can involve conversions between convertible types (e.g. numeric promotion), and between expanded and reference types. If the target entity of the reattachment is of an expanded type, only conversions may take place. But in the other cases, possible conversions and clonings according to the semantics of direct reattachment ([Mey0?], §22.7) must all be taken into account when the algorithm is executed. Note that we will discuss problems with *clone* and *copy* in [Section 3.5 on page 18](#). These problems also apply to the implicit conversions during reattachments.
- The DTS of a creation *expression* with creation type *T* is $\{T\}$.
- Feature and class contracts must be taken into account. This requires a complete semantic definition of assertion execution, even in complex cases such as Precursor calls and contracts of features selected in inheritance clauses. All DTS calculations must be performed as if all assertion checks were enabled and executed at their specific locations.

3.4 Manifest Arrays have no static type

Eiffel arrays are nicely embedded into the object-oriented type system. The generic class *ARRAY [G]* forms the basis of array types. A generic derivation of *ARRAY* serves as type of typed array variables of a specific type. An array of strings is e.g. declared as

```
menu_names: ARRAY [STRING]
```

The array (which is a reference type) can be created by a creation instruction:

```
create menu_names.make (7)
```

Initializing an array by consecutive calls to *put* is a tedious and unnecessarily wordy job:

```
menu_names.put ("File", 1)
menu_names.put ("Edit", 2)
menu_names.put ("Format", 3)
menu_names.put ("View", 4)
menu_names.put ("Special", 5)
menu_names.put ("Window", 6)
menu_names.put ("Help", 7)
```

That’s why Eiffel offers a leaner way to accomplish the same task: Manifest Arrays:

```
menu_names := <<"File", "Edit", "Format", "View", "Special", "Window", "Help">>
```

So far, so good. We have an elegant notation to shorten a recurrent task. But is it suitable for static type checking? The manifest array does not state its type explicitly. But for the resolution of feature calls on a manifest array expression, we *need* to know the static type. Otherwise, the compiler couldn't even know what features are defined on the expression. Therefore, current Eiffel compilers try to *infer* the static type of a manifest array by inspection of the static types of its components.

Unfortunately, the static type of a manifest array expression is explicitly declared *undefined* by the Eiffel language standard. A simple example shows, that the static type of a manifest array is *ambiguous*: `<<"Hello", 42>>` could be `ARRAY [COMPARABLE]` or `ARRAY [HASHABLE]`. In general, the static type of an array is ambiguous if the types of its elements have multiple common ancestors which are mutually non-conforming.

[Mey0?], §27.9 says:

“Fortunately, this inability to settle for a single type will not cause any difficulty in the two situations which require obtaining type information about a Manifest_tuple *ma*:

The Manifest Array rule enables us to ascertain conformance of *ma* to a certain type (the type of the target in an assignment, or of a routine's formal argument) without any ambiguity, [...].

The dynamic type set of *ma*, needed to ascertain system-level call validity, is the set of all types of the form `ARRAY [T]` for every type *T* in the dynamic type set of any of the elements of *ma*.”

This is not true. There's a third situation where a compiler needs to know the type of a manifest array (at least the *static* type): resolution of calls on a manifest array expression. The expression

`(<<"Hello", 1>>).item (1).hash_code`

cannot be compiled without knowledge of the static type of `(<<"Hello", 1>>)`.

If the static type of an expression would be re-declared into a static type *set* (in this case: `{ARRAY [COMPARABLE], ARRAY [HASHABLE]}`), the feature name resolution algorithm would become very complex (and potentially ambiguous as well!). Also, if we could handle the problem mentioned in Section 3.5 (“like argument” result types), then we would clearly need to know the static type of actual argument expressions.

Having expressions with a *set* of static types would entail a plethora of language changes, since this affects every feature call. In addition, it would make Eiffel programs hardly understandable for developers, since *they* must have all these rules in mind too, when writing programs.

Current Eiffel compilers ([ES03], [SE02]) just give up when they encounter manifest arrays with mixed element types. They simply declare the expression's type as `ARRAY [ANY]`, which can be observed by looking at the result of the expression `(<<"Hello", 1>>).generating_type`. Consequently, they declare the call to `hash_code` in the above example as invalid.

But how do compilers check assignments of manifest arrays to entities? They have to implement some special rules for these assignments (which is very unfortunate for a language which wants to stay clean and without obscure built-in rules).

SmartEiffel [SE02] looks like it has a special assignment operation for manifest arrays, since it accepts direct assignments such as `an_array_of_hashable := (<<1, "Hello">>)`, but rejects assignment of an expression as in `an_array_of_hashable := (<<1, "Hello">>)`.

On the other hand, EiffelStudio [ES03] apparently tracks the type of every manifest array element, since it accepts `an_array_of_comparable := (<<1, "Hello">>)` as well as `an_array_of_hashable := (<<1, "Hello">>)`, but rejects things like `an_array_of_integer := (<<1, "Hello">>)` by moaning that source type `<<INTEGER, STRING>>` does not conform to target type `ARRAY [STRING]`. EiffelStudio seems to handle at least assignments correctly, but at the expense of a new form of type information for manifest array expressions.

Concluding, we must state that the definition of manifest array expressions needs more thought. In their current form, they're not statically checkable and not sufficiently specified in the general case.

The absence of a single static type for a manifest array expression makes it impossible to implement DTS rules 6 & 7, since a feature call with a manifest array expression as target is not resolvable without knowledge of the exact static type of the target. As long as the language definition allows such calls, System Validity is not implementable without far reaching changes to the algorithm.

There are two ways to overcome the lack of a static type for manifest arrays:

- *Disallow manifest arrays with heterogeneous elements.* Since the problems from this section only appear if not all array elements are of the same type, such a restriction would solve the problems all at once. The restriction could be alleviated for a new form of constant attributes for manifest arrays. There, the disambiguation can be performed by the explicitly declared type of the attribute.
- *Require an explicit type annotation for manifest arrays with heterogeneous elements.* A notation such as `an_array_of_hashable := (<<1, "Hello": HASH-ABLE>>)` could provide the necessary information in cases where elements of differing static types should go into the same array.

3.5 Function result type anchored to an argument

A peculiar mechanism of the Eiffel language is the notion of *anchored types*. The type of an entity can not only be based on a class or a derivation of a generic class. It can also be *linked to the type of another entity*. This section deals with typing problems in connection with types of the form *like argument*, where *argument* is not an query (function or attribute) of the class, but the name of an argument of the current routine. It's not only possible to declare a local variable of a type anchored to an argument, but also a later argument or the result of a routine may be declared as *like argument*. The mechanism is described in [Mey0?], §11.14, in the subsection titled "Anchoring to an argument".

Eiffel's root of the type system is class *ANY*. Two of the most fundamental methods in this class are *clone* and *equal* — and they both use types anchored to an argument. Their implementation is partly reprinted here:

```

frozen clone (other: ANY): like other is
  -- Void if `other' is void; otherwise new object
  -- equal to `other'
  --
  -- For non-void `other', `clone' calls `copy';
  -- to change copying/cloning semantics, redefine `copy'.
do
  if other /= Void then
    ...
    Result := feature {ISE_RUNTIME}.c_standard_clone ($other)
    Result.copy (other)
    ...
  end
ensure
  equal: equal (Result, other)
end



---


copy (other: like Current) is
  -- Update current object using fields of object attached
  -- to `other', so as to yield equal objects.
require
  other_not_void: other /= Void
  type_identity: same_type (other)
do
  feature {ISE_RUNTIME}.c_standard_copy ($other, $Current)
ensure
  is_equal: is_equal (other)
end



---


frozen equal (some: ANY; other: like some): BOOLEAN is
  -- Are `some' and `other' either both void or attached
  -- to objects considered equal?
do
  if some = Void then
    Result := other = Void
  else
    Result := other /= Void and then
      some.is_equal (other)
  end
ensure
  definition: Result = (some = Void and other = Void) or else
    ((some /= Void and other /= Void) and then
      some.is_equal (other))
end

```

Figure 3: Extracts from class *ANY*: *clone*, *copy*, *equal*, *is_equal* (source: [ES03])

```

is_equal (other: like Current): BOOLEAN is
  -- Is `other' attached to an object considered
  -- equal to current object?
  require
    other_not_void: other /= Void
  do
    Result := feature {ISE_RUNTIME}.c_standard_is_equal ($Current, $other)
  ensure
    symmetric: Result implies other.is_equal (Current)
    consistent: standard_is_equal (other) implies Result
  end

```

Figure 3: Extracts from class *ANY*: *clone*, *copy*, *equal*, *is_equal* (source: [ES03])

Since these routines are defined in class *ANY* (which is the root of the inheritance hierarchy), they are available in every routine body without further ado. One may criticize this design for its non-object-orientedness: *equal* and *clone* are in fact *global* functions, since they just operate on their *arguments* and have no connection whatsoever to the current object (they don't contain unqualified calls to other routines and they don't access attributes of the current object). But let's first have a look at what they promise to do.

ETL3 [Mey0?] contains a special validity rule on “Expression Conformance” (code CNCX), which tries to establish a meaningful conformance relation between a first routine argument and another argument (or between an argument and the result type of a function) in the presence of anchored types. But the rules give no hint as to how a compiler could ensure that these rules are actually met in every possible run-time situation. The corresponding chapter (11) argues that ensuring static type safety is indispensable for building reliable software, but it deliberately omits any statement about how this goal can be reached with the given language rules.

A closer inspection on the mechanism of “types anchored to an argument” reveals an astounding finding: the seemingly harmless function

equal (*some*: *ANY*; *other*: like *some*): *BOOLEAN*

is in fact *statically overloaded*. Let's consider a simple example:

<pre> class FILE feature name: STRING directory: DIRECTORY ... end </pre>	<pre> class USER feature same_file (file_1: FILE; file2: FILE):BOOLEAN is do Result := equal (file_1.name, file_2.name) and equal (file_1.directory, file_2.direcory) end ... end </pre>
---	--

Figure 4: A simple example using *equal*.

The similarity of the two calls to *equal* hides the fact that they must be typechecked differently. The first application uses two arguments of static type *STRING*, whereas the second application has both arguments of static type *DIRECTORY*.

The best a typechecker can do without global analysis of dynamic type sets is to check conformance between the static types of the two arguments. This results in *different* checks for the two calls. The first uses a version of *equal* looking like

equal (some: STRING; other: STRING): BOOLEAN

, whereas the second call can be seen as if it used a version like ...

equal (some: DIRECTORY; other: DIRECTORY): BOOLEAN.

Eiffel disallows overloaded features for good reasons (explained in [Mey97]). Unfortunately, the language itself breaks this rule with the overloaded behavior of routines which use types anchored to arguments. Whether type conformance is checked or not: a programmer who wants to test two objects for equality must have this whole story in mind when calling the function *equal*. And beware of programmers who really try to apply this mechanism of “types anchored to an argument” to more complicated things than equality checks.

But the story has just begun. The static type equivalence check on *equal* is by far not enough. Let’s have a look at the specification of *is_equal*, which is called from *equal*:

```
is_equal (other: like Current): BOOLEAN is
  require
    other_not_void: other /= Void
  ensure
    symmetric: Result implies other.is_equal (Current)
    consistent: standard_is_equal (other) implies Result
  end
```

The signature demands that *other* conforms to *Current*. On the other hand, the postcondition *symmetric* requires the inverse conformance relation: *Current* must conform to *other*. Since the Eiffel language uses name-based subtyping¹, this symmetric conformance is only fulfilled iff the *dynamic* types of *Current* and *other* coincide. Therefore, function *equal* must ensure statically, that both target and argument of the call to *is_equal* have the same dynamic type — otherwise, the call would not be valid! Note that unequal dynamic types would *not* just be a postcondition violation. It would be a statically invalid call to feature *is_equal* (either the original call or else the call in the postcondition).

1. *Name-based subtyping* is the rule that a type B is only a subtype of type A, iff A is explicitly declared as a parent of B or, recursively, A is a subtype of one of B’s parents. The contrary would be *structural subtyping*, where subtyping relationships are not declared explicitly, but *inferred* by comparison of all features declared in a class. Structural subtyping is rather dangerous, since it introduces conformance paths based solely on feature signatures (name, argument types, result types). Such implicit relationships are hard to track for humans and are therefore to be dismissed from a software engineering perspective.

If we go back to the implementation of *equal*, the problem becomes apparent: The call *some.is_equal (other)* is only valid if *other* has certainly the same dynamic type as *some*. But this information is not available for polymorphic entities — and in Eiffel, all entities are potentially polymorphic.

3.5.1 Throwing everything into one bag

The last observations showed that class-level type checks are not sufficiently expressive to guarantee static safety of calls to the routines *equal* and *is_equal*. Maybe a broader approach such as System Validity might help? Unfortunately not for any real system. The basic problem is the non-object-oriented nature of *equal*.

Remember that System Validity is based on dynamic typesets, which accumulate all possible dynamic types of objects to which an entity, expression or function of a system can refer. Let's come back to the example in [Figure 4 on page 20](#). There are two calls to *equal*: in the first one, both arguments are of declared type *STRING*, whereas in the second call, the arguments bear the type *DIRECTORY*.

Looking at the System Validity rules (see [Figure 2 on page 15](#)) to calculate the dynamic type set (DTS) of the arguments of function *equal*, we see that rule 6 deals with the actual-formal association in routine calls. Since *equal* is always used unqualified, the target type is in this case the current type *USER*. Therefore, the first call to *equal* adds the DTS of *file_1.name* to the DTS of the first argument of function *equal* and the DTS of

file_2.name to the DTS of the second argument. Analogously, the second call to *equal* proceeds with *file_*.directory*.

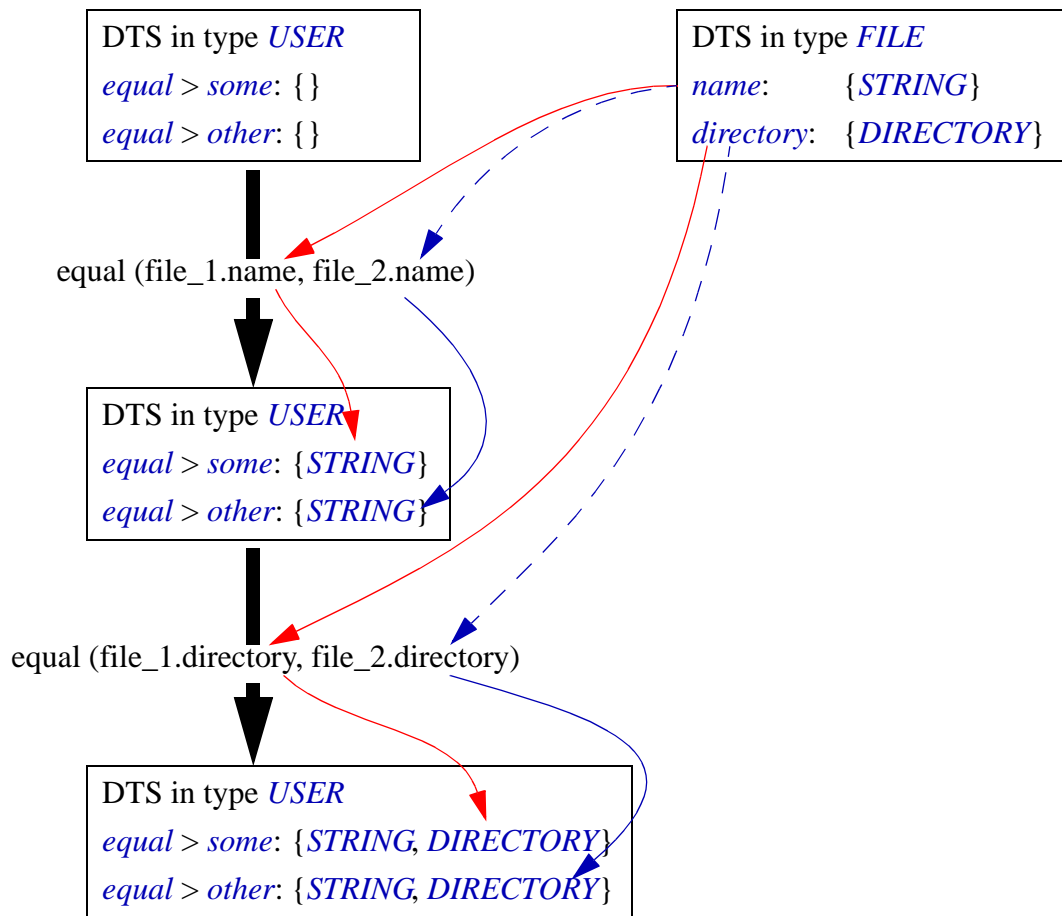


Figure 5: Evolution of dynamic type sets of *equal*.

Figure 5 illustrates how these two harmless and safe equality comparisons are enough to trigger a System Validity violation. The dynamic type sets of *equal*'s arguments are *independently* extended with the additional types. After the iterative phase, the typesets of the arguments *some* and *other* of routine *equal* in type *USER* both contain *STRING* as well as *DIRECTORY*. In phase 3, we finally have to check all calls on a target with respect to all types of the target's DTS. Let's do that for *some.is_equal (other)* in routine *equal* (see Figure 3 on page 19), in the simple system from Figure 4:

As we saw, the DTS of *some* is $\{ \textit{STRING}, \textit{DIRECTORY} \}$. First, we'll check with *some* of dynamic type *STRING*. Routine *is_equal (other: like Current)* in type *STRING* requires an argument of type *like Current*, which reduces to *STRING*. But now, a look at the DTS of *other* should turn you pale: *other* includes *DIRECTORY*, which is obviously *not* a valid argument to *is_equal*! A System Validity checker must flag an error here, which is very unfortunate, since we *know* that we never mixed *STRING*s with *DIRECTORY*s.

System Validity only considers single entities and expressions, but not *combinations* of them. Unfortunately, there's not much we can do to change the situation for the better. Tracking dynamic type combination sets across multiple entities would end in an algo-

rhythmically complex solution which is beyond the capabilities of today’s systems and users.

As already stated, one origin of the problem is the procedural style of function *equal*. The function is not applied to a sensible target, but is just called from everywhere. Other languages call such features “*global functions*”, and they define them outside of any class. Since Eiffel has no global features, they were put into class *ANY* (which is an ancestor of every Eiffel class), where they are accessible from code in any class.

To summarize: any two valid calls to different overloaded versions of *equal* (first arguments have different types) trigger a System Validity violation, even though the calls are totally safe. System Validity does not produce useful errors in this case.

3.5.2 Clone Wars

Even more hideous examples emerge when several “borderline” mechanisms are combined. A small program shows another subtle way to break Eiffel’s type system — and System Validity can’t help. The two concrete mechanisms which take part in this example are:

- Result type anchored to an argument
- create instruction applied to target of anchored type

<pre> class ROOT_CLASS create make, null feature make is local a, b, c: ROOT_CLASS do create a.null -- *1* b := nasty_clone (a) -- *2* c := nasty_clone (c) -- *3* b.null -- *4* c.null -- *5* end </pre>	<pre> nasty_clone (other: ANY): like other is do create Result -- *6* end null is do end end </pre>
--	--

Figure 6: *nasty_clone*, which is not type-checkable

Trying to apply the DTS algorithm (Figure 2 on page 15) to the example in Figure 6 leads to serious trouble. The following notation is used to explain the proceeding of the algorithm:

execution_step • (*rule*) at **instruction**: $dts[routine:local_var] := \{Members_of_dts\}$

At every execution step, the DTS of an entity is the last mentioned DTS of that entity. All entities start with an empty DTS. The system we look at is amazingly easy, since it has only two types: *ROOT_CLASS* and *ANY* (but no feature of *ANY* is used ...).

- 1 • (1) at *1*: $\text{dts}[\textit{make:a}] := \{\textit{ROOT_CLASS}\}$
- 2 • (1) at *6*: $\text{dts}[\textit{nasty_clone:Result}] := \{\langle\textit{unknown (ANY?)}\rangle\}$
- 3 • (4) at *2*: $\text{dts}[\textit{make:b}] := \{ \}$
- 4 • (4) at *3*: $\text{dts}[\textit{make:c}] := \{ \}$
- 5 • (7) at *2*: $\text{dts}[\textit{make:b}] := \{\langle\textit{unknown}\rangle\}$
- 6 • (7) at *3*: $\text{dts}[\textit{make:c}] := \{\langle\textit{unknown}\rangle\}$

At execution step 2, we're checking the function *nasty_clone* in type *ROOT_CLASS*. What is the (static) result type of this function? It's not just *ANY*, because that would make instruction *2* class-level invalid. But *2* must be class-level valid, since it is analogous to the routine *clone* of class *ANY*.

The misery is: inside *nasty_clone*, we don't know the static type of the actual argument to *other*. All we know inside of *nasty_clone* is

- that the current type doesn't matter at all,
- that *other* is of static type *ANY*,
- that the *Result* must conform to *ANY* (since the result type is declared *like other*).

This information is not sufficient to know of what type the result must be. In addition, the required result type might have a creation clause, which would necessitate a creation instruction of the form *Result.<a_creation_procedure>*. Since the required creation type is even unknown in a System Validity check, we cannot determine the dynamic type sets of *b* and *c* after the assignments in *2* and *3*. Therefore, instructions *4* and *5* cannot be typechecked, since the DTS of their targets is unknown.

A pessimistic solution could just declare the DTS of *clone*'s result as all possible result types. But since the result type is only known to be *like other* (which reduces to *ANY*), this would include all types in the system. Consequently, the application of any feature to the result of a *clone* operation is system-invalid (unless the feature is declared in *ANY* and doesn't change availability or type in any descendant). Every relevant application executes feature calls on cloned targets. If all these calls are marked as errors, System Validity has evidently failed to let useful system pass its check.

It might be interesting to know what actual Eiffel compilers do with the example in [Figure 6](#):

- ISE EiffelStudio 5.2 [\[ES03\]](#) compiles the system without any warning.
 - *create Result* in *nasty_clone* creates a *Result* with same *dynamic* type as the given argument. This hack sometimes gives the expected result.
 - After *2*, *b* is of type *ROOT_CLASS*. After *3*, *c* is of type *ANY*.
 - Instruction *4* executes nicely, since *b*'s type was *ROOT_CLASS*.
 - Instruction *5* crashes with a Segmentation Violation.
- SmartEiffel 1.0 [\[SE02\]](#) compiles the system without any warning (even if compiled with `-safety_check`, which should find such errors).

- *create Result* in *nasty_clone* always creates a *Result* of type *ANY*. This is the expected behavior if we look at feature *clone* in isolation.
- Instruction **4** is consequently an error, since feature *null* is not defined in *ANY*. Depending on the compilation mode, a run-time check ensures a safe termination:
- ***** Error at Run Time *** :**
 - Target is not valid (not the good type).
 - Expected: "ROOT_CLASS", Actual: "ANY".
 - [etc. ...]

Besides of the above hole in the type system, the function *clone* from class *ANY* faces the same problem as *equal*: arguments to *clone* are all thrown into the same abyss (to be specific: into *clone*'s argument *other*). You can easily make up an example similar to the one in [Figure 4](#), where a type contains multiple independent calls to *clone* with arguments of varying types. The *Result* entity of *clone* will accumulate all types which are fed into *other* and a System Validity checker will inevitably spit out spurious errors at call-sites which hold no problem.

There are three possible solutions to this problem:

- Remove *clone* and *copy* from class *ANY* and force users to write their own copy-features. This would have the advantageous side-effect that statically safe implementations for common patterns such as Singleton, Flyweight, Factory, etc. become possible in Eiffel. They are currently not possible, since the creation of new object instances can not be controlled by any construct. *copy* and *clone* could always be abused to create new objects (which circumvent the normal creation path and suddenly appear in a system). [\[AB03\]](#) describes difficulties with the implementation of singletons in Eiffel. Their solution finally has to rely on Design by Contract, which is a rather weak guarantee of program correctness, since violations are only detected as late as at run time. Languages such as Java, C++, Modula, etc. support the implementation of *statically* guaranteed singletons.
- Make *clone* and *copy* special language features (keywords) and introduce new conformance rules to typecheck constructs involving these two features.
- View all functions with a return type *like argument* as code macros which must be expanded and checked separately for each call. This solution could solve the problem of “throwing everything into one bag”, since calls to *equal* with different first argument types would not add the argument types to the same DTS. On the other hand, this solution might be difficult to understand, and it has certainly disadvantages in terms of code bloat.

3.6 Interfacing with external routines

Although most of the functionality of an Eiffel system is written in “pure” Eiffel, we sometimes face the need for “talking to strangers”. The object-oriented Eiffel environment is usually all we need to model and implement our applications. Nevertheless, there are two domains which require more than the object-oriented core language can provide:

- Executing code from other languages (e.g. for database bindings, user interface libraries, etc.).

- Implementing low-level facilities such as bit-wise object cloning or equality tests, serialization and persistence, setting and getting environment properties (such as controlling the garbage collector), and executing system calls.

Eiffel provides a construct called *external routines* ([Mey0?], §28), which constitutes the interface to features that are implemented in a foreign language. An external routine has no *do ... end* part, but a special body which specifies the external language and the routine to be called in that language.

When a client calls an external routine, all actual argument expressions are evaluated and their results passed to the foreign routine. The Eiffel compiler emits code to call the specified foreign routine according to the external language's calling conventions. If the external routine is a function, then its result is the result of the foreign routine.

Unfortunately, external routines can easily break the type system in unpredictable ways. E.g. unsafe casts in C code cannot be detected by any type checker. Therefore, we have to *trust* external code to a certain degree and make assumptions on the possible transformations a correct external routine can perform. To ensure class-level validity, it is enough to demand that external routines may only perform reattachments which are type-wise correct according to the usual rules of object reattachment.

But System Validity needs much more information about what's going on behind the scenes. Remember that the goal of System Validity is to track the *dynamic* type sets of entities and expressions. To accomplish this goal, a System Validity checker must know about every assignment and actual-formal association in the system (see the algorithm for calculating dynamic type sets in [Figure 2 on page 15](#)).

We could try to restrict admissible transformations a foreign routine may perform. For example, we could forbid foreign routines to create new objects and to change the dynamic type sets of object attributes. But that's clearly not reasonable. You could neither implement a low-level *clone* operation nor a persistence mechanism, since those obviously *have* to create new objects by other means than a creation instruction.

Objects returned by external functions can dynamically be of any type conforming to the declared type. The persistence mechanism of ISE EiffelBase [ES03] reveals the incompleteness of System Validity in an archetypical way. Class *FILE* offers a query *retrieved: ANY*, which returns a complete (possibly complex) object structure that was previously stored by another routine of class *FILE*. The routine *retrieved* relies on external code, since it must be able to create instances of arbitrary types to reproduce the stored run-time structure.

Now, what should a System Validity checker do with such a query? The dynamic type set of feature *retrieved: ANY* cannot be constrained in any way — it is the complete set of types in the system. This undermines the whole idea of System Validity checking, which tries to make validity statements on the basis of restricted dynamic type sets of polymorphic call targets.

The problem here is, that objects are created by an external routine. The System Validity checker has no access to the external language and therefore, it cannot track the creation of objects and their assignment to entities in the external routine. *In a nutshell: Dynamic*

type sets can not be calculated in the external language, which destroys the possibility of a global whole-system-analysis.

[Mey0?], §24.9 contains a special “Array type rule” for doing System Validity checks on class *ARRAY*:

“To study the effect of array manipulations on dynamic type sets, assume that in class *ARRAY* feature *item* is an attribute, and that *put (v, i)* and *force (v, i)* are both implemented as *item := v.*”

This rule just deals with one common usage of external features, but doesn’t address the general problem. Sidenote: In ISE EiffelBase [ES03], the class to which the “Array type rule” must be applied is not *ARRAY*, but *SPECIAL*. Unfortunately, the implementation of *put* and *item* in class *SPECIAL* is just an empty routine body with a comment -- *Built-in*, which are in reality a just disguised external routines. This forces a System Validity checker to literally know about that special class and handle it in a special way.

3.6.1 Sources and sinks

There’s only an unrealistic way to uphold System Validity in a realistic application: every external query must explicitly specify the complete dynamic typeset of its result. Taking up ideas of the “Array type rule”, we could try to devise a model of communicating “sources” and “sinks”. Sources are external queries which can bring new object instances into a running system. Sinks are external routines which take arguments and pass them to foreign code outside of the system. A custom specification language must then completely describe all possible connections introduced by the external routines “behind the scenes”. This information can be given to the System Validity checker, which must include that knowledge in an additional system validity rule.

The class *SPECIAL* for example, has a sink in feature *put* and a source in feature *item*. Those two must be connected by a manual annotation inside the code or in an additional file in the system. Although this model works in theory, it’s probably not scalable to include all external routines of a system. For more complex bindings (e.g. to a database or to an XML DOM), a model with just a single connection between a source and a sink (like in *SPECIAL*) is not sufficient. Expecting users to write a complete data-flow graph for all external features by hand is not very realistic, especially if they just wrote a binding to a library for which they don’t have access to the source code.

3.7 The POINTER type

External languages usually don’t know or care about the structure of Eiffel’s typed objects. All they are interested in is the *address* of an object. Object addresses are a low-level implementation detail, which is of little interest in object-oriented development. But for interfacing with external routines, the address of an object is often what is needed. To meet this requirement, Eiffel has a special notation to get an object’s address: the “\$” operator, which can be applied to arbitrary expressions and returns the address of the expression’s value as a *POINTER*.

We have already come across an application of the address operator in [Figure 3 on page 19](#). The implementation of the function *clone (other: ANY): like other* of class *ANY* includes the following line:

Result := feature {ISE_RUNTIME}.c_standard_clone (\$other)

This is a static call (an ISE-proprietary extension to Eiffel, which denotes a call without a target object) to feature *c_standard_clone* of class *ISE_RUNTIME*, whose signature is

c_standard_clone (other: POINTER): ANY

Entities and expressions of type *POINTER* render the sources & sinks approach even more difficult, since they conceal the objects to which they point. System Validity must track all possible assignment paths of objects to ensure the completeness of dynamic type sets. For entities of type *POINTER*, it is therefore not sufficient just to track the possible dynamic types of the entity. In addition, we have to track the dynamic type sets of all *instances* that could be *pointees* of a *POINTER*. This additional complexity is necessary, since *POINTERS* can be passed around and finally land in a “sink”. Even though the interface of the external routine doesn’t care about the object’s dynamic type, System Validity must care. To summarize: entities of type *POINTER* must carry the dynamic type set of all possible objects to which they could point in the running system. Note that the implementation of this special “set of pointees’ dynamic types” just for entities of type *POINTER* is very complicated and requires precautions all over the place.

As if that complex tracking of dynamic pointee type sets would not be enough, the old problem of “throwing everything into one bag” (see [page 22](#)) is also applicable here. Low-level external features for bit-wise copying or comparison tend to behave like black holes, which attract everything they can get from their vicinity. All tracking of paths between sinks and sources cannot help ensuring System Validity if objects of any type are passed to such general routines.

3.8 Dynamic class loading

The above observation also applies to all run-time environments which can dynamically load classes. E.g. the Microsoft .NET framework supports loading classes at run time and using them polymorphically in feature calls or assignments which expect an instance of a supertype of the newly loaded class. Such a late-loading of classes can introduce new types into a running system. Those types were not available at compile time and are therefore missing in the dynamic typeset calculation. Therefore, the System Validity approach is not applicable in such environments.

Late-loading of classes (and types) is an important mechanism for extensible and scalable architectures. It should be possible to load plug-ins at run time into a running system without prior knowledge of all available extensions. A typical application of late-loading is the integration of a downloaded viewer for a new data file format. The interface to such a viewer component is relatively lean and is usually implemented as a small collection of deferred classes. Each new data format has to inherit the given interface classes and implement their concrete behavior. At run time, an instance of the new viewer gets polymorphically assigned to a variable of the interface type — which introduces a possible new source of catcalls. Catcalls to features of the new viewer type cannot be detected at compile time of the original system. The only possible way to cope with such an error is raising a run-time exception, which throws us back to the dynamic solutions of [Section 2](#).

3.9 Conclusion

System Validity is violated in almost every program and can therefore (without dynamic type checks) not be used to ensure strong typing. Furthermore, the examples of this chapter have shown some serious problems, which make the feasibility of a complete System Validity checker questionable. Especially the problems with dynamically loaded types and with external routines are most probably not completely solvable.

System Validity was first presented by Bertrand Meyer in 1989 in a posting on *news:comp.lang.eiffel* [Mey89]. Since then, no successful complete implementation of System Validity has been reported (even though several declarations of intent were circling around). I am convinced that the reasons for the current situation are not just lack of interest, volition, time, or money. The real reason is that *System Validity cannot guarantee the soundness of a system and is therefore of little practical use*. An additional reason could be that the implementation presents more obstacles than obvious from the description. More about this in the next section.

4 Implementation of a System Validity Checker

One of the original goals of this thesis was building a reusable “testbed” for trying out different Eiffel typing policies. Unfortunately, it turned out that such a framework is hardly implementable. The main focus was then laid on the implementation of System Validity. This section sheds some light on the encountered difficulties.

4.1 Parsing Eiffel class files

One of the first decisions to be taken was the choice of a parsing infrastructure. A complete Eiffel system must be parsed and transformed into a form known as *Abstract Syntax Tree (AST)*. An AST is a structured representation of the input files. A scanner splits a flat text file into symbols, keywords and identifiers. Those tokens are passed to a parser, which builds an structured representation according to a *grammar* of the input language. The produced AST can be *decorated* by additional information, such as resolution of identifiers and type information.

Five approaches were initially considered:

- 1 • Writing a recursive descent parser by hand.
- 2 • EiffelLex and EiffelParse: Object-oriented parsing libraries from ISE [Mey94], [ES03], updated version from [But00].
- 3 • YOCC and YoocLa: Object-oriented parser generators [AMH95], [But00].
- 4 • Gobo gelex and geyacc: Scanner and parser generators in the tradition of lex and yacc [Bez].
- 5 • Gobo Eiffel Tools: Portable Eiffel class library, including a lace parser and a parser to build abstract syntax trees from Eiffel class files. Still under development [Bez].

The first approach was quickly dismissed for various reasons. Recursive descent parsers are notoriously slow. Since they only see their local part of the grammar, they have to backtrack fairly often if the chosen descent path was not the right one. Some parser gen-

erators (such as yacc, or its Eiffel-counterpart geyacc) can automatically perform a global analysis on the input grammar and eliminate unsuccessful descent paths. Other disadvantages of manually written recursive descent parsers are:

- frequent code repetition
- distribution of the grammar over many classes, which makes it difficult to keep the overview
- mainly due to the last two points: increased amount of work

The second option, EiffelParse [Mey94], is a reusable library of classes which implement some of the repetitious tasks of writing a recursive descent parser. A programmer implements a parser by writing *syntax classes*, where each syntax class describes one production of the grammar. Syntax classes inherit from descendants of a basic class *CONSTRUCT*. *CONSTRUCT* is basically a tree of *CONSTRUCT*s. Its descendants *AGGREGATE*, *CHOICE*, *REPETITION* and *TERMINAL* are templates for the concrete productions of the grammar. The effective syntax classes have to implement a deferred query *production: LINKED_LIST [CONSTRUCT]*, which contains the necessary information from the grammar in the form of an Eiffel list object. Additional *semantic classes* for each syntax class can implement the “semantics” of a tree node by specifying code to be executed during the traversal of a parse tree.

Unfortunately, EiffelParse shares the disadvantages of manually written recursive descent parsers, since its parsing strategy is in fact recursive descent. Furthermore, the approach of manually writing an Eiffel class for each production is only suitable for simple “document processors” (which is by the way the originally intended target audience of EiffelParse). Writing a parser for the whole Eiffel language in EiffelParse would be very time consuming, and it is highly questionable whether the complex semantics of a static type checker would fit into the given framework.

The third option, YOOCC [AMH95], tries to reduce the amount of work required to implement a parser in EiffelParse. Its name “Yes! An Object-Oriented Compiler Compiler” already suggests what it’s all about: eliminating the need to write syntax classes by hand. YOOCC takes a grammar file from which it creates syntax classes and skeleton semantic classes for EiffelParse. YOOCC slightly eases the task of writing syntax classes, but doesn’t address the other problems of EiffelParse at all. It even uses a very “seamful” technique: generating skeleton classes which should be changed by the user is a rather bad idea, since we all know that software evolves. Re-runs of the compiler produce new skeletons, into which the user must manually re-enter the implementation.

YoocLa (YOOCC with Left Attributes) [But00] builds on YOOCC and alleviates its problems with semantic definitions. YoocLa uses an augmented grammar description, which includes local names for production elements and semantic rules for each production. A special feature of YoocLa is the calculation of certain kinds of “attributes” of productions. Those attributes are passed up and down the parse tree and they are evaluated while parsing the input text. Still, since YoocLa builds upon YOOCC, the mentioned problems of recursive descent parsers apply. Furthermore, YoocLa’s main operational field is parsing and processing simple document formats and thereby com-

puting reports of low complexity — and not large scale applications such as complete language compilers or system-wide analysis tools.

The fourth of the considered approaches was building a parser by using *gelex* and *geyacc* [Bez], a pair of Eiffel scanner and parser generators in the tradition of the famous UNIX tools *lex* and *yacc*. They produce highly optimized bottom-up parsers, which build on the findings of a whole field in computer science: parsing theory. Yacc’s parsers don’t build an AST by default. Semantic actions embedded in the grammar allow the user to create up a custom AST, which can omit nodes that are only necessary for the grammar, but don’t represent a useful abstraction. Since the produced AST doesn’t have to map one-to-one to the grammar, its design can be more problem-tailored. It would for example not use a *CHOICE* construct to allow for different kinds of feature declarations such as attributes, internal functions, external procedures, Instead, it can introduce an object-oriented (multiple) inheritance hierarchy with abstractions such as *FEATURE*, *QUERY*, *ATTRIBUTE*, *FUNCTION*, *ROUTINE*, down to things like *INTERNAL_ONCE_PROCEDURE*, and implement the choice with polymorphism.

Geyacc would have been our choice, if there would not have been an even better starting point: the Gobo Eiffel Tools Library [Bez], our fifth option, was taken as a basis for the development of a System Validity Checker.

4.2 Gobo Eiffel Tools

Although there exists virtually no documentation about the project, the Gobo Library [Bez] includes a fairly complete Eiffel parser and flattener in the directory `#{GOBO}/library/tools`. All classes from Gobo Eiffel Tools begin with *ET_*. From the corresponding `Readme.txt`:

WARNING: THIS LIBRARY IS STILL UNDER DEVELOPMENT!

Gobo Eiffel Tools Library

The Gobo Eiffel Tools Library is a portable Eiffel class library to make easier the development of Eiffel tools such as Eiffel pretty-printers or flat-short tools. This library contains a LACE parser to analyze Ace files, an Eiffel parser which builds abstract syntax trees out of Eiffel source code, and routines to process these abstract syntax trees. The Gobo Eiffel Tools Library is primarily used to develop Gobo Eiffel Lint (*gelint*).

Still, the library is very cleanly designed and implemented, and it was not considered worth the effort to start a new Eiffel parsing project when such a great foundation is already available.

The fundamental notion of an Eiffel system¹ is a *universe*. Class *ET_UNIVERSE* models this container, which holds clusters, classes and features of the system being examined. Eiffel systems are usually specified in so-called *Ace* files (Assembly of Classes in Eiffel), which are written in the *Lace* syntax (Language for Ace) as defined in [Mey0?], Appendix B. An instance of *ET_UNIVERSE* can be obtained by processing the Ace file

1. In Eiffel, programs ready for compilation are called “systems”.

for a system with an *ET_LACE_PARSER*. The concrete code for this follows (without any error handling):

```

ace_filename: STRING
ace_file: KL_TEXT_INPUT_FILE

lace_parser: ET_LACE_PARSER
lace_error_handler: ET_LACE_ERROR_HANDLER
lace_ast_factory: ET_LACE_AST_FACTORY
eiffel_ast_factory: ET_DECORATED_AST_FACTORY
eiffel_universe: ET_UNIVERSE
...
create ace_file.make (ace_filename)
ace_file.open_read

create lace_error_handler.make_standard
create lace_ast_factory.make
create eiffel_ast_factory.make
lace_ast_factory.set_ast_factory (eiffel_ast_factory)
create lace_parser.make_with_factory (lace_ast_factory, lace_error_handler)

lace_parser.parse (ace_file)
eiffel_universe := lace_parser.last_universe

ace_file.close

```

This prepares the *eiffel_universe* for being parsed. The factory pattern [GHJV95] allows for customization of the created syntax tree nodes. It can also be used for other purposes, like e.g. displaying a progress monitor by printing some message when a new class or feature is created.

```
eiffel_universe.preparse_single
```

... examines all files in the system's clusters to find those which contain Eiffel classes.

```
eiffel_universe.parse_system
```

... finally parses all classes in the system, starting at the root class from the Ace file. After this call, all classes in the system are completely parsed and accessible by name through the hash table *eiffel_universe.classes*.

4.3 Flattening classes

Just parsing classes is often not enough. As we know, inheritance is a central mechanism of object-oriented programming to build hierarchies of classes. Therefore, a class interface contains not only the features a class defines explicitly, but it also includes all inherited features from its ancestors. Eiffel's powerful feature adaptation facilities offer many ways how features can undergo changes when they are inherited. It is not straightforward to determine the real set of features of a class. Therefore, an Eiffel language processor must be able to *flatten* classes. In Gobo Eiffel Tools, this is done by calling

```
eiffel_universe.compute_ancestors
```

This starts a traversal over all classes in the system, computing and registering the ancestors of each class and all inherited features. Thereby, a lot of complicated bookkeeping must be performed. Things would be easy, if inherited features could simply be copied and added to the list of features of a class. However, such a copy & paste approach does not retain information about the origin of the inherited feature. But this information is required for the correct resolution of the dynamic binding version of a polymorphic call.

Remember that an entity or expression has a declared static type (except for manifest arrays, but [Section 3.4](#) showed that this is an inadequacy in the language definition). Valid feature names for a call on an entity or expression are the names of features of the static type of the target. Due to polymorphic reattachments, the dynamic type of the target can be different from the static type. In such a case, dynamic binding requires a lookup in a dynamic dispatch table to get the right version of the feature to be called.

To determine the correct dynamic binding version of a feature, the notion of *feature seeds* has been introduced [\[Mey0?\]](#) (§10.25). A seed is a version of a feature from the most remote ancestor. Each feature (inherited and immediate) in a class should know its seeds, and each class should know which of its features has a given seed. With this information, the lookup of the dynamic binding version becomes easy: get a seed of the called feature and ask the target's dynamic type about its version of the seed of the feature called on the static type. The returned feature is the one to be executed.

This was a simplified explanation of the story behind determining seeds and dynamic binding versions of features, which omitted concrete details about more complicated cases involving feature joining, replication, sharing, and selection under repeated inheritance. A complete description would exceed the scope of this thesis. It shall suffice to say that Gobo Eiffel Tools seems to compute correct seeds even for hairy situations. Unfortunately, [\[Mey92\]](#) and [\[Mey0?\]](#) (§10.25) completely messed up the definition of seeds¹, which cost me quite some time to understand the intended underlying concept.

Now let's come back to the description of Gobo Eiffel Tools. As yet, their flattening of classes only takes feature declarations into account, but does not analyze routine implementations. Feature calls are simply left as identifiers or, for prefix and infix features, as operators. What's also not available is the static type of entities and expressions.

1. For the interested reader: Rule 3 from “Origin, Seeds” ([\[Mey0?\]](#), §10.25) states that joined features should lose their “history” and just take on a new seed. That's bogus and would miserably fail if implemented in a compiler. The correct solution is to give a feature a *set* of seeds, which is the union of all its precursors' seeds. This correction entails further changes in various other rules (e.g. the “dynamic binding version” from §15.13 can be defined much easier).

What's also worth noting is the observation, that feature sets are *monotonously accumulating*. This means that features never lose any of their seeds. Each seed from features of a class is guaranteed to resolve to exactly one feature in every descendant class. The same seed can only be inherited from different ancestors if *repeated inheritance* is involved. In that case, the “repeated inheritance consistency constraint” ensures that only the conforming path (or: the selected feature) obtains the seeds in question.

4.3.1 Resolving called features and static types of expressions

The AST implements the Visitor Pattern [GHJV95], which allows clients to implement new algorithms on the AST without changing the underlying classes. Visitors are called “processors” and must inherit from *ET_AST_PROCESSOR*.

A System Validity checking tool, which needs to know about feature calls and static types of entities and expressions, could implement such a processor to resolve the required information. But where should the additionally gathered information be stored? There are three possibilities of how this could be done:

- 1 • If AST nodes had an additional field such as *properties: DS_HASH_TABLE [ANY; STRING]*, then the information could directly be stored in the corresponding nodes (indexed by an identifying string). Since Gobo Eiffel Tools does not offer such a field and we don’t want to touch library classes, this solution remains hypothetical.
- 2 • If all AST nodes inherited *HASHABLE*, we could use an external hashtable to associate AST nodes with the additional information. Unfortunately, only selected AST nodes such as *ET_FEATURE* and *ET_CLASS* inherit *HASHABLE*, but the others don’t. Therefore, the idea to store static types of expressions in an ordinary hashtable is also not crowned with success.
- 3 • The third possibility is to store the supplementary attributes in a hashtable which accepts all *ET_AST_NODES*, and provides the hashing function externally. This is easily doable by inheriting *DS_ARRAYED_SPARSE_TABLE* (from the Gobo Structure Library) and implementing the *hash_position* query. The only information the hash code could be based on is the starting position of the node in the source file:

```
class CA_NODE_TABLE [G, K -> ET_AST_NODE] inherit
  DS_ARRAYED_SPARSE_TABLE [G, K]
  redefine new_cursor end

creation
  make_map, make_map_default
...

feature {NONE} -- Implementation
  hash_position (k: K): INTEGER is
    local
      a_position: ET_POSITION
    do
      if k /= Void then
        a_position := k.position
        Result := (256 * a_position.line + a_position.column) || modulus
      else
        Result := modulus
      end
    end
end
```

With this infrastructure, *CA_STATIC_TYPE_AND_FEATURE_RESOLVER* can inherit *ET_AST_PROCESSOR* and resolve the static type of every expression along with the

seed of the feature called in a feature call. Therefore, it must traverse all internal routines of every class and collect this information by visiting each and every instruction and expression in the routine bodies. Furthermore, all contracts (pre- & postconditions, class invariants) must be visited too.

Note that there are quite a number of different kinds of expressions and instructions which involve a feature call. *ET_QUALIFIED_CALL* and its descendants are the easiest to spot. But the creation procedures of *ET_CREATION_INSTRUCTION*s and *ET_CREATE_EXPRESSION*s are feature calls, too. Thereto come *ET_INFIX_EXPRESSION*s, *ET_PREFIX_EXPRESSION*s and *ET_INFIX_INSTRUCTION*s and *ET_PREFIX_INSTRUCTION*s, which bear an additional complication for call resolution: their feature names can have various forms, ranging from keywords (e.g. *and then*) over symbols (e.g. \leq) to free operator names (e.g. *infix "/./"*).

*ET_EQUALITY_EXPRESSION*s are the most perfidious ones: depending on the expansion status of their left expression, they can either be feature calls or not. If the left expression is of reference type, then reference equality (and hence no feature call) is applicable. Otherwise, feature *is_equal* is called on the (expanded) target, with the right-hand expression as argument.

There are two more issues which require our attention: precursor calls and inherited contracts. Precursor calls with explicit parents are easy: the called feature can just be looked up in the given parent. For precursor calls without an explicit parent, the single effective precursor must be found by inspection of all precursors of the routine.

Design by Contract [Mey97] prescribes that contracts from ancestors are inherited by all descendants. This applies to class invariants as well as to feature pre- and postconditions. To avoid a costly seeking of inherited contracts at every routine call, contracts from ancestors should be collected once and for all for every class and every feature.

In hindsight, I think that all this additional information should not be stored externally, since it strongly belongs to the respective AST nodes. The right solution would be to enhance the corresponding classes of Gobo Eiffel Tools by additional attributes for storing static expression types and seeds of called features, since this information is required for every tool which operates on routine bodies.

After having set up the described infrastructure, we could think of finally starting with the implementation of System Validity's dynamic type set algorithm. But once again, things turn out to be more complicated than apparent on first sight.

4.4 Flattening types

Looking closely at the definition of the DTS algorithm in [Figure 2 on page 15](#), we see that the algorithm operates on *Class_types*. This language construct stands for either types which are directly based on a class, or for *derivations of generic types*. That second part will give rise to headaches. What it implies is that flattening of classes as described above is not sufficient for System Validity. Generic classes do not just define one type, but *a family of types*. Concrete derivations of generic classes are created in an ad hoc fashion wherever an entity or function of that type is declared.

System Validity must handle each generic derivation separately, since violations can occur in only some of the derivations. If we would e.g. just calculate dynamic type sets on class *CELL [G]* (for arbitrary *G*), then the presence of two generic derivations of *CELL* would be enough to trigger an unjustified System Validity violation:

```
int: CELL [INTEGER]
string: CELL [STRING]
...
int.put (42)                -- dts [{CELL}.item] = {INTEGER}
string.put ("Brave new World") -- dts [{CELL}.item] = {INTEGER, STRING}
io.put_string (string.item)  -- error
```

put_string (s: STRING) is not valid for *s* of dynamic type *INTEGER*. If the two derivations are considered separately, the problem disappears:

```
int.put (42)                -- dts [{CELL [INTEGER]}.item] = {INTEGER}
string.put ("Brave new World") -- dts [{CELL [STRING]}.item] = {STRING}
io.put_string (string.item)  -- success
```

The System Validity checker must generate data structures to hold dynamic type set information for every entity, expression, and function in every type separately. This flattening of generic derivations into full-fledged types causes the type count to explode in every non-toy example. It leads to the same kind of problems, which people experience with C++'s template mechanism:

- The large amount of generated types causes high memory consumption and enormously slows down the compiler. Since the DTS calculation algorithm traverses the all types several times, the whole system must be kept in memory at the same time. For every entity, expression, and function in every type, the checker has to remember a dynamic type set.
- It is very difficult to understand type violations which only appear in certain generic derivations. This problem is abundant in C++ template programming and a major source of confusion and trouble with C++ libraries.

If we're not embarrassed enough by the exploding system size, we can implement the flattening of types by an additional visitor on the AST, which gathers all types used in the system. The visitor traverses all classes and features and adds every encountered type to the set of *run-types*, which accumulates all *Class_types* used in the system. As discussed in [Section 3.4 on page 16](#), manifest arrays in their current form have no type and therefore break the whole approach. But let's turn an eye blind for this shortage for the moment and assume that manifest array types are definite.

Anchored types must also be resolved at this phase, which is apparent from a small example:

```
thing: INTEGER
list_of_things: LIST [like thing]
```

The actual derivation of *LIST* is only known when the anchored type *like thing* is properly resolved. Similarly, formal generic types used inside a generic class must be resolved by their corresponding actual generic parameter:

```

class LIST [G]
...
feature
  item: G
...
  linear_representation: LINEAR [G]
...
end

```

Applying the visitor to *LIST [INTEGER]* yields type *LINEAR [INTEGER]*, which must be added to the set of run-types.

The type gathering visitor starts with an empty set of run-types and by creating the first known type of the system, the root type. The root type is based on the root class of the system, which is by definition not generic. Whenever it encounters a new run-type, that type is added to the set of run-types and then immediately visited. Thus, after terminating the visit of the root type, all types of the system should be known. Note, that traversing a type means traversing its base class while keeping the type context in mind (or in an attribute, if you like). Traversing a class means to traverse all features of that class (including inherited ones), as well as all routine bodies and all contracts. Do we have to visit expressions too? Yes, because there are two kinds of expressions which can introduce new types:

- Creation expressions of the form *create {CRATION_TYPE}.creation_procedure* can create instances of arbitrary types, which are only found if *CREATION_TYPE* is visited.
- Static function calls are an undocumented feature of ISE Eiffel [ES03]. Although they only appear in rare locations, a complete System Validity Checker still has to take them into account. We have already seen applications of static function calls in [Figure 3 on page 19](#), e.g. in function *clone* from class *ANY*:
Result := feature {ISE_RUNTIME}.c_standard_clone (\$other).

4.4.1 Computing and storing dynamic type set information

System Validity has rather huge demands on data structures to run on and to attach dynamic type set information. Since there's no one-to-one mapping between classes and types, the AST from Gobo Eiffel Tools cannot be used to attach DTS information. Instead, a parallel structure is required, which builds on *types*, rather than on classes.

For every *Class_type* in the system, a unique instance of type *CA_RUN_TYPE* is created.

```

class CA_RUN_TYPE
...
feature -- Access
  type: ET_CLASS_TYPE
  -- class type (fully resolved if generic type)

  run_routines: DS_HASH_TABLE [CA_RUN_ROUTINE, INTEGER]
  -- effective routines, indexed by feature id (in `type')

  attribute_dts: DS_HASH_TABLE [CA_DTS, INTEGER]

```

```

-- Dynamic type sets of attributes, indexed by feature id (in `type')
-- This includes all non-routine features: ET_ATTRIBUTE,
-- ET_CONSTANT_ATTRIBUTE, ET_UNIQUE_ATTRIBUTE

expressions: CA_NODE_TABLE [CA_DTS, ET_EXPRESSION]
  -- expressions in `type', indexed by expression
...
end

```

Some distinctions (e.g. between *once*-routines and *do*-routines) are not important for System Validity, but many constructs from the AST must be duplicated to be able to store dynamic type sets of entities, expressions, and functions.

CA_RUN_ROUTINE, for example nearly duplicates the hierarchy below *ET_ROUTINE*, since every kind of routine has its own combination of DTS-bearing attributes:

- formal arguments,
- local variables,
- Result,
- and possibly information on sources and sinks for external routines (see [Section 3.6.1](#)).

CA_DTS is the core abstraction to gather dynamic type set information. For the DTS algorithm, it would be sufficient to make *CA_DTS* just a set of *CA_RUN_TYPE*s. But as soon as the System Validity checker should report an invalid catcall, a problem becomes apparent: How to tell the user what's gone wrong?

Let's look at a simple example:

```

string: STRING
comp: COMPARABLE
greater: BOOLEAN
...
string := "Hello"      -- dts [string] = {STRING}
comp := string         -- dts [comp] = {STRING}
greater := comp < 1291

```

The last call to *{COMPARABLE}.infix "<" (other: COMPARABLE): BOOLEAN* is invalid, because the DTS of the target *comp* contains *STRING*, and *infix "<"* is covariantly redefined in *STRING* to *infix "<" (other: STRING): BOOLEAN*. Now, the argument of type *INTEGER* is invalid, since it doesn't conform to *STRING*.

The only error message that a could be emitted if *CA_DTS* would simply be a set of *CA_RUN_TYPE*s would look something like (from [\[SE02\]](#)):

*Warning: Unsafe covariant redefinition (type "COMPARABLE" redefined as "STRING").
Line 145 column 22 in STRING:*

```

infix "<" (other: like Current): BOOLEAN is
      ^

```

Line 29 column 22 in COMPARABLE:

```

infix "<" (other: like Current): BOOLEAN is
      ^

```

Line 6 column 19 in ROOT_CLASS:

```
greater := comp < 1291
      ^
```

What's missing here, is a *witness path* (a sequence of reattachments back to a creation operation), which shows how the offending type came into the DTS.

A more sophisticated version of *CA_DTS* would include a witness path for every type in the DTS. *add_type_set* is called for every reattachment, whereby *a_dts* is the *DTS* of the source and *a_reason* is the position of the reattachment. The routine returns a *BOOLEAN*, which tells the caller, whether the addition has changed the target DTS. This is a violation of the command-query separation principle, but it was not considered worthwhile to introduce an additional attribute to every *CA_DTS*, since this information has no longer-lasting value.

```
class
  CA_DTS
...
  dynamic_types: DS_HASH_TABLE [DS_LINKABLE [ET_FILE_POSITION],
                                CA_RUN_TYPE]
    -- path to creation, indexed by run type

  add_type_set (a_dts: CA_DTS; a_reason: ET_FILE_POSITION): BOOLEAN is
    -- Has the addition of `a_dts' to `dynamic_types' brought a new type into
    -- `dynamic_types'?
    -- Extend origin path of added type by origin `a_reason'.
    -- Retain old origin path if type already present in `dynamic_types'.
    -- (Warning: this is a side-effect function.)
    local
      a_dynamic_types_cursor: DS_HASH_TABLE_CURSOR
        [DS_LINKABLE [ET_FILE_POSITION], CA_RUN_TYPE]
      a_run_type: CA_RUN_TYPE
      an_origin_path: DS_LINKABLE [ET_FILE_POSITION]
    do
      a_dynamic_types_cursor := a_dts.dynamic_types.new_cursor
      from a_dynamic_types_cursor.start until a_dynamic_types_cursor.after loop
        a_run_type := a_dynamic_types_cursor.key
        if not dynamic_types.has (a_run_type) then
          Result := True
          create an_origin_path.make (a_reason)
          an_origin_path.put_right (a_dynamic_types_cursor.item)
          dynamic_types.force_new (an_origin_path, a_run_type)
        else
          -- There's another (shorter) path contributing `a_run_type'.
        end
      a_dynamic_types_cursor.forth
    end
  end
...
end
```

With the additional information in *CA_DTS*, the error message can be made much more expressive:

Warning: Unsafe covariant redefinition (type "COMPARABLE" redefined as "STRING").

Line 145 column 22 in STRING:

infix "<" (other: like Current {STRING}): BOOLEAN is

^

Line 29 column 22 in COMPARABLE:

infix "<" (other: like Current {COMPARABLE}): BOOLEAN is

^

Unsafe call (`other': type "INTEGER" does not conform to "STRING")

Line 6 column 19 in ROOT_CLASS:

greater := comp < 1291

^

** Witness path: `comp' is of type "STRING" because of*

Line 4 column 10 in ROOT_CLASS:

string := "Hello"

^

Line 5 column 8 in ROOT_CLASS:

comp := string

^

** Witness path: 1291 is of type "INTEGER" because of*

Line 6 column 22 in ROOT_CLASS:

greater := comp < 1291

^

With this infrastructure in place, the real work of implementing the DTS calculation algorithm can begin. That task turned out to be much more time-consuming than expected and could not be finished in the timeframe of this thesis. Reasons for this include the general complexity of the task (see also [Section 4.5](#)) and the amount of work to build up correct and complete mappings between the class-based AST of Gobo Eiffel Tools and the types-based data structures of the System Validity checker. Especially tricky are e.g. inherited features, whose implementation refers to a formal parameter of their originating class.

4.5 A System Validity checker is a compiler (and more)

As the previous sections already indicated, building a System Validity checker is not just a matter of having a parser and performing a minimal analysis on the AST. In the end, a System Validity checker must incorporate a fairly complete compiler (without the code-generating backend). This is an enormous task, and in hindsight, I must admit that it was a too audacious venture for a master's thesis.

Although the Eiffel language has a relatively clean syntax, its semantics are far more complicated than those of most other programming languages (except for C++, of course...). The interplay of polymorphism, static typing and dynamic binding are far from trivial, and a System Validity checker must know about all these details and implement them correctly ([Section 4.3](#) already dipped into that subject).

The tiniest detail of the language is important. Here are some areas where uncertainties were encountered:

- What is the execution model of precursor calls? Are pre- and postconditions evaluated for precursor calls? And what about class invariants? (If they are evaluated,

which invariants: those of the precursor's implementation class or also those of the current type?) What is the type of *Current* during the execution of a precursor's body?

- What about inherited contracts in case of repeated inheritance with a *select* clause? Are contracts inherited from all precursors, or only from the *selected* branch?
- What are the exact built-in conversion rules between compiler-known basic types? E.g. can a small manifest integer be used where an `INTEGER_64_REF` is expected? If yes, does the resulting object have type `INTEGER_64_REF` or can it have a smaller size?
- Are *expanded INTEGER_REF* and *INTEGER* the same type? Or does one conform to the other (and in the reverse direction)? Or are they converted into each other?
- Where are types of the form *like Identifier* allowed? Can such types be the declared type of a creation operation? E.g. in a routine *nasty_clone (other: ANY): like other*, what is the type of an object created by *create {like other}*? And is that the same as *create Result*?
- Can an entity of a reference type be redefined into an entity of an expanded type and vice versa?

All those details *do* matter, since System Validity is only worthwhile if it is complete, i.e. if it ensures static type safety by excluding all possible catcalls.

4.6 A language has only *one* typing policy

Static typing is a global business. The following example shows how a design in a language quickly becomes invalid if conformance relations are made illegal by a change in the typing policy of a language. Two important Eiffel conformance relations are used here:

- Conformance between different generic derivations of the same generic class: If all generic parameters of a derivation conform to their counterparts in an other generic derivation, then the first type conforms to the second (i.e. `LIST [LINE]` conforms to `LIST [FIGURE]`, since `LINE` conforms to `FIGURE`).
- Covariant attribute redefinitions: The type of an attribute can be redefined in a descendant to a type conforming to the original type (i.e. an attribute of type `FIGURE` can be redefined into type `LINE` or an attribute of type `LIST [FIGURE]` can be redefined into type `LIST [LINE]`).

<pre> class FIGURE ... feature draw is deferred end ... end </pre>	<pre> class LINE inherit FIGURE ... feature draw is do ... end ... end </pre>
--	---

Figure 7: A figurative Eiffel design, using redefinitions and generic conformance.

<pre> class COMPOSITE_FIGURE inherit FIGURE ... feature components: LIST [FIGURE] draw is do -- foreach c in components: c.draw end ... end </pre>	<pre> class POLYGON inherit COMPOSITE_FIGURE redefine components end ... feature components: LIST [LINE] ... end </pre>
--	---

Figure 7: A figurative Eiffel design, using redefinitions and generic conformance.

Note that this code is perfectly valid in current Eiffel and examples with a similar structure are abundant in Eiffel code. Discarding one of these rules would immediately make large parts of the base libraries invalid. This is enough to change the face of the language in such a heavy way, that the changed language could hardly be identified with the original one. A change in (typing) semantics usually creates a new language rather than modifying an existing one. This is always the case if the new semantics *restrict* the original language in any way, since previously valid programs could become invalid.

Real enhancements in static type checking require radical changes to the language, which includes additional and removed language constructs and rules at unpredictable locations and with unpredictable impact to the semantics of the type system. Therefore it's very difficult (if not impossible) to provide a flexible testbed framework which could be reused for different “typing policies”. At least with today's programming languages, there seems to be little possibility to achieve the goal of an extensible typechecking testbed framework which could be used for any but trivial language changes.

The best we can do is to provide a clean and understandable parser, analyzer and type checker for one language. If we want to build a checker for a different language, we can then take a copy of the original checker and reuse its implementation as a basis for the required changes. Typing is intimately coupled with the whole language design and cannot be considered separate. Details of the chosen type system require adaptations to nearly every part of the type checking algorithm and have a direct impact on the required data structures (as exemplified by the necessity of flattening types in [Section 4.4](#)).

5 Conclusion

None of the examined solutions was able to completely solve to problems from [Section 1.4](#). Covariant redefinitions remain an unsafe construct as long as we cannot come up with statically enforceable rules which exclude error-prone programs, but let safely usable system pass. [Section 3](#) uncovers a bunch of problems which prove that a global approach like System Validity is not successful.

The initial goal of a statically safe type system with covariance has (once again) not been reached. But the discussions shed light on why some promising attempts won't do in practice. The quest for powerful statically safe type systems has not found an end yet.

Research is still active in various language communities, and progress can be expected, among others, from the work on *Variance* performed in connection with the introduction of generics into the Java language [VGJ03], or from the ECMA Committee for the Standardization of the Eiffel language, TC39-TG4 [ECMA].

6 Glossary

object type

A type specifies the externally observable interface and behavior of an object. It is a set of signatures which defines all features applicable on the object.

typing error

A feature is applied to an object, but the object's type does not include that feature.

dynamic typing

Typing errors can occur at run-time, but they are caught in a predictable way (e.g. by throwing an exception). Prototype of a completely dynamically typed language: Smalltalk.

static typing

“The ability to check, on the basis of the software text alone, that no execution of a system will ever try to apply to an object an operation that is not applicable to that object” [Mey97]. The guaranteed absence of typing errors in compiled programs.

strong typing

All typing errors are correctly reported and handled in a predictable way. Note that strong typing is possible with static typing as well as with dynamic typing. Strong typing is imperative for every serious high-level language.

weak typing

Typing errors are not always detected. An example are unchecked C-style casts, which may break the type system without notice and lead to unpredictable behavior.

subclassing

“A mechanism whereby a class is defined in reference to others, adding all their features to its own” [Mey97]. Sometimes, the term “inheritance” is used for subclassing.

subtyping

Subtyping is a substitutability relationship between object behaviors, describing when an object of one type may be type-safely substituted for an object of another type. [Kur00]

subtype

Let S be a subtype of T (i.e. $S <: T$). Every object attached to an entity of type S must understand all features which can be applied to an object of type T.

polymorphism, substitutability

An object of one type is substitutable for an object of another type in any context if no type error (mainly of the "method not found" variety) can result from the substitution. Note that an object of a subtype can always safely be used in place of an object of the original type.

7 References

- [AB03] Karine Arnout, Éric Bezault: *How to get a Singleton in Eiffel*. To be published in *Journal of Object Technology*, vol. 2, no. 5, Special Issue: TOOLS USA 2003. http://se.inf.ethz.ch/people/arnout/arnout_bezault_singleton.pdf (July 2003).
- [Ame90] Pierre America: *The America Conjecture*. Unpublished Lecture Slides by Bertrand Meyer, citing dictum at TOOLS Europe, 1990.
- [AMH95] Jon Avotins, Christine Mingins, Heinz Schmidt: *Yes! An Object-Oriented Compiler Compiler (YOOCC)*. Technical report TR95-2, Department of Software Development, Monash University, Australia, 1995. <http://www.csse.monash.edu.au/publications/1995/tr-sd95-2.ps.gz> (July 2003).
- [Bez] Eric Bezault and others: *Gobo Eiffel Project*. Release 3.1. <http://www.gobosoft.com/> (April 2003).
- [BCC+96] Kim Bruce, Luca Cardelli, Giuseppe Castagna, et al.: *On Binary Methods. Theory and Practice of Object Systems*, 1(1995), pp. 221-242, <http://research.microsoft.com/Users/luca/Papers/Binary.A4.ps> (July 2003).
- [But00] Gerry Butler: *Attribute Grammars as a Means of Improving the Accessibility of Parser Generators to Software Engineers*. Master's Thesis, Monash University, 2000. <http://www.csse.monash.edu.au/~gabutler/projects/> (July 2003).
- [Cas95] Giuseppe Castagna: *Covariance and Contravariance: Conflict without a Cause*. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17 (3):431–447, 1995. ISSN 0164-0925.
- [Cecil] Craig Chambers et al.: *The Cecil Language – Specification and Rationale*. Dept. of Computer Science and Engineering, University of Washington, Seattle, December 2002. <http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-cecil-lang/cecil-spec.ps> (April 2003).
- [ES03] Eiffel Software: *Eiffel Studio 5.3*. Eiffel development environment, freely available for non-commercial purposes. <http://www.eiffel.com/> (July 2003).
- [ECMA] *ECMA International — An industrial association for the standardization of information and communication systems*. <http://www.ecma-international.org/> (July 2003).
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*. Addison-Wesley, 1995.
- [GR83] Adele J. Goldberg, David Robson: *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983

- [HBM+03] Mark Howard, Eric Bezault, Bertrand Meyer, Dominique Colnet, Emmanuel Stauf, Karine Arnout, Markus Keller: *Type-safe covariance: Competent compilers can catch all catcalls*. Unpublished.
<http://www.inf.ethz.ch/~meyer/ongoing/covariance/recast.pdf> (July 2003).
- [Java] *The Java Programming Language*. <http://java.sun.com/> (July 2003).
- [Kur00] Stoyan Kurtev: *Subtyping and Inheritance in Object-Oriented Programming*. Diploma Thesis, Institut für Computersprachen, Technische Universität Wien, 2002-02-07. <http://www.complang.tuwien.ac.at/Diplomarbeiten/kurtev00.ps.gz> (May 2003).
- [Mey89] Bertrand Meyer: *Static typing for Eiffel*. Posting on news:comp.lang.eiffel, draft as of 1989-07-10, Message-ID: 176@eiffel.UUCP.
- [Mey92] Bertrand Meyer: *Eiffel: The Language, second printing*. Prentice Hall, 1992.
- [Mey94] Bertrand Meyer: *Reusable Software: The Base Object-Oriented Libraries*. Prentice Hall, 1994.
- [Mey97] Bertrand Meyer: *Object-Oriented Software Construction, second edition*. Prentice Hall, 1997.
- [Mey0?] Bertrand Meyer: *Eiffel: The Language, third edition* (“ETL3”), work in progress, <http://www.inf.ethz.ch/~meyer/ongoing/etl/> (July 2003), user name “Talkitover”, password “etl3”.
- [MN96] Gail C. Murphy, David Notkin: *On the use of static typing to support operations on frameworks*. In *Object-Oriented Systems 3*, 1996, pp. 197-213.
<http://compscinet.dcs.kcl.ac.uk/OO/oo030402.abs.html> (May 2003).
- [PS94] Jens Palsberg, Michael I. Schwarzbach: *Object-oriented type systems*. John Wiley and Sons, 1994, ISBN 0-471-94128-X.
- [SE02] Dominique Colnet and Suzanne Collin: *SmartEiffel: The GNU Eiffel Compiler*. LORIA, UHP, INRIA, FRANCE, Release 1.0, 2002-12-06. <http://SmartEiffel.loria.fr> (May 2003).
- [Sha96] David Lujun Shang: *Are Cows Animals?* In *Object Currents*, monthly online magazine (ceased), SIGS Publications, Volume 1, Issue 1 - January 1996. Originally at www.sigs.com/publications/docs/oc/9601/oc9601.c.shang.html. Archived at <http://web.archive.org/web/19970723054531/www.sigs.com/publications/docs/oc/9601/oc9601.c.shang.html> (May 2003).
- [TT99] Kresten Krab Thorup, Mads Torgersen: *Unifying Genericity – Combining the Benefits of Virtual Types and Parameterized Classes*. In ECOOP’99 Proceedings, LNCS 1628, 1999, pp.186-204.
<http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/1628/16280186.pdf> (May 2003).

- [VGJ03] *Variant Generic Java*. Prototype Java compiler with generics and variance. <http://www.daimi.au.dk/~plesner/variance/> (July 2003). Connected with Sun's JSR-14: *Adding Generics to the Java(TM) Programming Language*. http://developer.java.sun.com/developer/earlyAccess/adding_generics/ (July 2003)