Diss. ETH No. 24129

Void Safety

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
ALEXANDER KOGTENKOV
Diploma with Honors in Applied Mathematics
Moscow State Engineering Physics Institute (Technical University)
Russian Federation

born on
November 18th, 1970

citizen of
Russian Federation

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Carlo A. Furia, co-examiner
Prof. Dr. Manuel Mazzara, co-examiner
Prof. Dr. Erik Meijer, co-examiner
Prof. Dr. Lothar Thiele, co-examiner

2017

Alexander Kogtenkov
Void Safety
Avoiding Null Pointer Dereferencing in an Object-Oriented Language

DOI:10.3929/ethz-b-000000135

Acknowledgments

My research work would be impossible without continuous support of many people and I am very grateful for overall appreciation of the idea to make yet another step towards more reliable software.

I learned about **void safety** from *Bertrand Meyer* who coined the term to reflect absence of access on void target in Eiffel. He introduced me to the basic elements of the mechanism and permanently promoted it bringing us to a solid theoretical and practical solution. Many issues discussed in this work came out as a result of recurring discussions with *Bertrand Meyer* about language evolution and his strong position about intransigent correctness and usability of proposed language rules.

The ideas seemed to be around for some time at the beginning of 2000's and were discussed by top-level scientists. I was lucky to meet *Eric Meijer* who told me about the early communications on null dereference avoidance techniques that looked rather simple at that time, but turned out to be a nightmare when it came to implementation.

Indeed, the implementation part was not easy, especially because there were tons of void-unsafe code around, and I am very thankful to *Eiffel Software* that was generous enough to go through multiple releases of the product featuring varying maturity of void safety, finally reaching its firm position. We had innumerable discussions about technical details with *Emmanuel Stapf* who shared a lot of important ideas. In particular, he and *Mark Howard* came up with the thought to have several void safety levels when migrating from void-unsafe to void-safe source code. Similarly, *David Hollenberg* became an early tester of the technology and found several border-line overlooked cases incorrectly handled by the compiler. The whole *Eiffel Software* team was involved in adapting existing libraries to the new rules of the game, including *Jocelyn Fiat* who wholeheartedly embraced the migration, and *Ian King* who worked on the migration of a cross-platform GUI library and proposed the object initialization scheme that was

compatible with the solution I had implemented in the compiler at that time. We also discussed so called "design mode" that allowed for developing incomplete systems. At a summer school I was able to share the ideas with *Tony Hoare* and to get a quick feedback on the theoretical side of the approach.

Many language-specific decisions came from hot discussions at the *ECMA TC49-TG4 standardization committee*. One of its constant members, *Eric Bezault* thoroughly reviewed the language standard that allowed me to avoid missing cases in the implementation. He also persuaded me to have a closer look at certified attachment patterns that finally resulted in a much more generalized version of the rules. His comments also forced me to support more flexible object initialization. Talking about object initialization, I should mention a pretty lengthy discussion on how this could be done with *Bertrand Meyer* and *Emmanuel Stapf*, resulted in a very practical solution compared to the one we originally devised.

An opportunity to attend a *IFIP WG 2.3* meeting, especially to hear straight comments of *Jayadev Misra* and to observe a detailed analysis of problems by *Rustan Leino* as well as later short conversations with both of them, taught me to look for rigorous and thorough theoretical background of a model, and urged me to look in the direction of proof assistants. I am very thankful to *Andreas Lochbihler* and *Christoph Sprenger* who introduced me to the world of Isabelle/HOL and showed me how to use the tool. I am also grateful to anonymous reviewers of my papers for their helpful comments on my work.

The *Chair of Software Engineering* at ETH Zürich collected a fantastic group of people, such as *Sebastian Nanz*, *Carlo A. Furia*, *Martin Nordio* that provided general support though the course of my endeavors, as well as *Chris Poskitt*, *Roman Schmocker*, *Jiwon Shin*, *Christian Estler*, *Max Pei*, *Marco Trudel*, *Marco Piccioni*, *Chandrakana Nandi*, *Đurica Nikolić*, *Georgiana Caltais*, *Julian Tschannen*, *Nadia Polikarpova*, *Scott West*, *Benjamin Morandi*, *Jason Wei*, *Piotr Nienaltowski*, *Alexey Kolesnichenko*, *Andrey Rusakov* who always welcomed me during my brief visits to ETH. Also, many thanks to *Mischael Schill* for helping me with translation into German. On the organizational side *Denise Spicher* all the time helped me to find appropriate answers to administrative questions I occasionally had. And *Claudia Günthart* was an incredible savior in multiple predictable and unforeseen situations immediately re-

acting to my (sometimes crazy) requests from the very beginning of my application as a PhD student.

I give my special thanks to the examiners who took their time to read and evaluate my work: *Carlo A. Furia*, *Manuel Mazzara*, *Erik Meijer* and *Lothar Thiele*.

Contents

List of Figures

List of Tables

Abstract

Null pointer dereferencing is a well-known issue in object-oriented programming, and can be avoided by adding special validity rules to the programming language. However, just introducing a single rule is not enough: the whole language infrastructure has to be considered instead. The resulting guarantees are called *void safety*.

The thesis reviews, in detail, *engineering solutions* and *migration efforts* that enabled the transition from classic to *void safe* code of multiple libraries and projects with lines of code ranging in the millions. Experience with the tiny details of the implementation can be an invaluable source of insight for researcher looking into making a language void safe.

The void safety rules can be divided into three major categories. The first one is the extension of a regular type system with attached (non-null) and detachable (possibly-null) types. Generic programming opens a door to different interpretations. The thesis defines some base void safety *properties for formal generic types* and specifies void-safety-aware conformance rules.

The second category of rules ensures that newly created objects reach a stable state maintaining the type system guarantees. The thesis proposes two solutions for this *object initialization issue* and compares them to previous work. It formalizes the rules in the Isabelle/HOL proof assistant and establishes some of their properties. To ensure safety at the end of object life cycle it also specifies *validity rules for finalizers*. A number of examples are used to demonstrate that the proposed solutions are of practical use and do not suffer from limited expressiveness caused by the lack of additional annotations describing intermediate object states.

The third category of void safety rules covers a practical need to bridge the gap between attached and detachable types. The thesis proposes *formal void safety rules for local variables* in the context of an object-oriented language that do not require any marks to distinguish between attached and detachable types. It demonstrates advantages of the annotation-free approach with bench-

marks based on open source code, discusses implementation decisions and how they are reflected in the formal model.

The thesis concludes with a *machine-checkable soundness proof* for the rules involving local variables using the Isabelle/HOL proof assistant.

ZUSAMMENFASSUNG

Nullzeigerdereferenzierung ist eine bekannte Problematik in der objektorientierten Programmierung und kann durch spezifische Gültigkeitsregeln der Programmiersprache vermieden werden. Eine einzelne Regel macht dies nicht möglich, stattdessen muss die gesamte Sprachinfrastruktur einbezogen werden. Die dadurch erreichten Garantien werden *Void Safety* genannt.

Diese Dissertation untersucht detailliert einige technische Lösungen und Migrationsansätze, welche den Übergang von klassischem zu *Void Safe* Eiffel-Quelltext mehrerer Bibliotheken und Projekte mit millionen Programmzeilen ermöglichten. Erfahrung mit den winzigen Details der Implementation kann eine wertvolle Quelle der Erkenntnis für Forscher sein, welche eine Sprache um Void Safety erweitern wollen. Die Regeln für Void Safety können in drei Kategorien eingeteilt werden. Die erste umfasst die Erweiterung des Typsystems mit befestigten (attached, garantiert nicht Null) und lösbaren (detachable, möglicherweise Null). Generische Programmierung öffnet dabei aber die Türe für verschiedene Interpretationen. Diese Dissertation definiert grundlegende *Eigenschaften formaler generischer Typen* und spezifiziert diesbezüglich Konformitätsregeln für Void Safety.

Die zweite Kategorie der Regeln stellt sicher, dass neue Objekte einen stabilen Zustand erreichen, der die Typsystemgarantieren sicherstellt. Die Dissertation schlägt zwei Lösungen für diese Objektinitialisierungsproblematik vor und vergleicht sie mit vorherigen Ansätzen. Sie formalisiert die Regeln für den Isabelle/HOL Beweisassistenten und zeigt einige der Eigenschaften. Um Void Safety auch am Ende des Objektlebenszyklus sicherzustellen spezifiziert die Dissertation auch *Gültigkeitsregeln für Finalisierer*. Eine Anzahl von Beispielen demonstriert dass die vorgeschlagenen Lösungen praktikabel sind und nicht an einer, durch den Mangel an zusätzlichen, zwischenzeitliche Objektzustände beschreibenden, Annotationen, limitierten Ausdrucksfähigkeit leiden.

Die dritte Kategorie der Regeln für Void Safety befasst sich mit der praktischen Notwendigkeit die Brücke zwischen befestigten

und lösbaren Typen zu schlagen. Die Dissertation präsentiert dafür *formale Void Safety Regeln für lokale Variablen* im Kontext einer objektorientierten Sprache, welche keine Markierungen benötigt um zwischen befestigten und lösbaren Typen zu unterscheiden. Sie demonstriert die Vorteile des annotationsfreien Ansatzes mit Benchmarks basierend auf offenem Quelltext, diskutiert Implementationsentscheidungen und wie sie im formalen Modell reflektiert werden.

Die Thesis schliesst mit einem, mit dem Isabelle/HOL Beweisassistenten durchgeführten, *maschinenüberprüfbaren Zuverlässigkeitsbeweis* der Regeln bezüglich lokaler Variablen.

1 Introduction

1.1 Motivation and goal

In his talk at a conference in 2009 Tony Hoare called his invention of the null reference in 1965 a "billion-dollar mistake" ([24]). The reason is simple: most object-oriented languages suffer from a problem of null pointer dereferencing. What does it mean in practice? It is possible that at run-time some variables (or expressions in general) do not reference any existing object, or are *null*. On the other hand the core of object-oriented languages is in the ability to send a message to a particular object or, in other terms, to make a call on an object. Given that there is no object when the reference is null, the run-time should signal to the program about the issue. Different languages deal with the problem in different ways, the most popular are as follows:

- throw an exception (Java [21, 22], C# [26], JavaScript [13], Python [63], Ruby [28])
- consider the behavior as undefined (C++ [29])
- return default values for basic types or structures, consider the behavior as undefined otherwise (Objective C [4])
- trigger a fatal error that can be caught by a user-specified error handler (PHP [23])

In particular, it turns out that in Java or C# an object call

```
expr.something (args)
```

denotes not just a call, but also a check whether the target is `null`, i. e., is interpreted similar to the following code snippet:

```
tmp = expr;
if (tmp == null)
      throw new NullPointerException ();
else
      tmp.something (args);
```

This hidden semantics requires additional static analysis and testing for production software due to the need to reveal the cases when `NullPointerException` may be triggered, and still not guaranteeing 100% absence of bugs.

Solutions to this problem fall into two major groups depending on whether the absence of null pointer dereferencing bugs can be guaranteed for a self-contained set of classes or requires a whole system analysis. Solutions from the first group ([17, 51, 62, 70]) equip types with additional information that tells whether the corresponding run-time value is never null or may be null at run-time. The type system conformance rules are used to specify what operations are permitted or not depending on the new nullness information associated with types. Though in a stable execution, when all objects have finished their initialization, the rules are simple, object initialization introduces certain challenges for language designers because some variables that are normally not null, may be null at this stage.

The obvious approach to distinguish between never-null and possibly-null references is to introduce a type mark [17, 51, 70] to indicate this property. Java-like languages (including C#) use their annotation mechanism with marks `[NotNull]` and `[MaybeNull]` to indicate nullness status of a type, Eiffel uses type marks **attached** and **detachable** for the same purpose. Masked types proposed in [62] are essentially different from this convention and operate on a much higher level of granularity allowing for identifying specific object parts that are initialized or not. In particular, they go far beyond null safety at the cost of amount of additional annotations. This approach might be justified for critical system development but seems too elaborate for everyday programming.

Solutions from the second group, with the most influential [67], do not use any annotations and infer nullness status of expressions from the program itself. This requires whole system analysis and cannot be used as a basis for language rules specification as safety properties of a class may be affected by other classes, not directly reachable from the current one. The main advantage of the approach is that no source code changes are required. Unfortunately, without knowing whether a type of an expression should be never-null or maybe-null, it is unclear how to report issues detected in a program that does not pass all checks. The issue might be in the client that does not respect supplier's expectations, or in the supplier that is buggy.

An ideal solution should be able to take advantage of both approaches to be:

- modular: there should be no requirement to do whole system analysis to see that a particular class, or, better, a self-contained library, is free of null pointer dereferencing;
- simple: the annotations should be easy to grasp and to use, preferably without special notions for rare cases;
- naturally expressive: there should be no artificial restrictions caused by the type system and language rules.

This work focuses on specification of void safety rules on a language level, so that a program written in this language, if it compiles successfully, never causes a null dereferencing bug at runtime.

1.2 Effect in industry

Null-pointer dereference being known for a long time (at least from the time mentioned by Tony Hoare in [24]) remains one of the day-to-day issue discovered in open source and private software. About 20 years ago software industry realized that even though different platforms have different score for software bugs due to different paradigms, tools, design and coding style, similar issues tend to repeat in different environments leading to similar vulnerabilities in software products. One of such initiatives was launched in 1999 and led to creation of a dictionary of common names for publicly known cybersecurity vulnerabilities known as Common Vulnerabilities and Exposures (CVE®) [8] funded by *CERT* (*Computer emergency response teams*, [10]). As of December 2016, the database has 727 entries mentioning null pointer dereference bugs explicitly. The distribution by years is shown in figure 1.1.

The statistics if strongly incomplete, because there are other bugs related to null-pointer safety. For example,

CVE-2016-2519
: `ntpq` and `ntpdc` can be used to store and retrieve information in `ntpd`. It is possible to store a data value that is larger than the size of the buffer that the `ctl_getitem()` function of `ntpd` uses to report the return value. If the length of the requested data value returned by `ctl_getitem()` is too

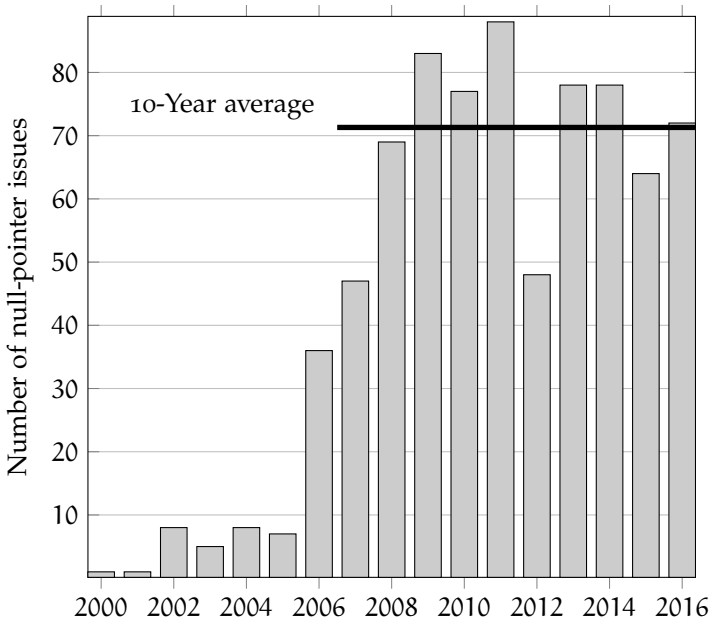

Figure 1.1. Null pointer issues (such as null pointer dereferencing) in Common Vulnerabilities and Exposures database.

> large, the value `NULL` is returned instead. . . . if one has permission to store values and one stores a value that is "too large", then `ntpd` will abort if an attempt is made to read that oversized value.

The description of the issue is taken from the vulnerability note VU#718152 titled *"NTP.org ntpd contains multiple vulnerabilities"* released on 28th of April 2016 by *CERT* and affects almost every computer on the Earth. Among vendors the bulletin mentions *Apple*, *AT&T*, *Cisco*, *Google*, *IBM Corporation*, and *Microsoft Corporation* to name a few. All planet is explicitly or implicitly involved in updating operating systems that uses `ntpd` program, because the issue is discovered in the reference implementation.

Other examples from the database:

CVE-2015-4444
: Adobe Reader and Acrobat 10.x before 10.1.15 and 11.x before 11.0.12, Acrobat and Acrobat Reader DC Classic before 2015.006.30060, and Acrobat and Acrobat Reader DC Continuous before 2015.008.20082 on Windows and OS X allow attackers to cause a denial of service (NULL pointer dereference) via unspecified vectors.

CVE-2015-8746
: `fs/nfs/nfs4proc.c` in the NFS client in the Linux kernel before 4.2.2 does not properly initialize memory for migration recovery operations, which allows remote NFS servers to cause a denial of service (NULL pointer dereference and panic) via crafted network traffic.

CVE-2015-8835
: The `make_http_soap_request` function in `ext/soap/php_http.c` in PHP before 5.4.44, 5.5.x before 5.5.28, and 5.6.x before 5.6.12 does not properly retrieve keys, which allows remote attackers to cause a denial of service (NULL pointer dereference, type confusion, and application crash) or possibly execute arbitrary code via crafted serialized data representing a numerically indexed `_cookies` array, related to the `SoapClient::__call` method in `ext/soap/soap.c`.

CVE-2016-1756
: The kernel in Apple iOS before 9.3 and OS X before 10.11.4 allows attackers to execute arbitrary code in a privileged context or cause a denial of service (NULL pointer dereference) via a crafted app. (Fix provided by Apple

(see https://support.apple.com/en-us/HT206167): A null pointer dereference was addressed through improved input validation.)

CVE-2016-2168

The req_check_access function in the mod_authz_svn module in the httpd server in Apache Subversion before 1.8.16 and 1.9.x before 1.9.4 allows remote authenticated users to cause a denial of service (NULL pointer dereference and crash) via a crafted header in a (1) MOVE or (2) COPY request, involving an authorization check.

Here an extract from the bug database [60] (https://bugs.php.net/bug.php?id=70081) that describes **CVE-2015-8835**:

> The first problem lies how zend_hash_get_current_key is called in php_http.c:826
>
> ```
> zend_hash_get_current_key_ex(Z_ARRVAL_PP(cookies),
>                              &key, &key_len, NULL, 0, NULL);
> ```
>
> here a wrong assumption is made about key always being a **char, in fact, this is not true when unserializing a SoapClient object crafted with a numerically indexed array as _cookies. The scenario mentioned above would then result in a null pointer dereference occurring in zend_hash_get_current_key(), zend_hash.c, line 1088.
>
> ```
> *num_index = p->h;
> ```
>
> where num_index is the NULL passed as 4th argument, and p->h a user-controlled value. While remotely this will always lead to a crash attempting to dereference 0x0, locally, if memory mapping is possible, this could be used to get arbitrary memory write and most likely code execution.

As described in the bug report, the null pointer dereference is just one step before crashing an application (best case scenario) or gaining privileged access to the computer by an attacker.

The references above are only the tip of the iceberg because they give an overview of what has been discovered. The real issue is about how much is going to be discovered in the future. The international community-driven dictionary of software weakness types *Common Weakness Enumeration* (*CWE™*, [9]) mentions the following consequences of *CWE-476 NULL Pointer Dereference*:

- NULL pointer dereferences usually result in the failure of the process unless exception handling (on some platforms)

is available and implemented. Even when exception handling is being used, it can still be very difficult to return the software to a safe state of operation.

- In very rare circumstances and environments, code execution is possible.

It suggests the following potential mitigation depending on the software development phase:

Requirements

Use a language that is not susceptible to these issues.

Architecture and Design

Identify all variables and data stores that receive information from external sources, and apply input validation to make sure that they are only initialized to expected values.

Implementation

- Sanity-check all pointers previous to use.
- Check the results of all functions that return a value and verify that the value is non-null before acting upon it.
- Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage.

Testing

Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.

The weakness is described as an indicator of poor code quality. Using this terminology all software that we use every day is of a poor quality as demonstrated by the statistics and specific examples presented above. Other categories with this weakness include:

- *OWASP* [58] Top Ten 2004 Category A9 – Denial of Service
- *CERT* [10] C and C++ Secure coding (Expressions and Memory Management sections)
- *CWE* [9] Nominee of 2010 and 2011 CWE/SANS Top 25 Most Dangerous Programming Errors

So, although the null-safety problem is known for a long time, it is still quite relevant in practice.

1.3 The keys to void safety

A void-safe language distinguishes between types of expressions that always produce an object and types of expressions that may produce null. The type system rules ensure that a variable marked as non-null is never assigned a null value. With such rules it is possible to go from never-null values to maybe-null values, because the latter is a superset of the former, or to stay within the same kind of nullness category. This introduces two stable groups of expressions that may be exchanged – never-null and maybe-null – and unidirectional flow from never-null to maybe-null values. The opposite direction – from maybe-null to never-null values – is not covered by the typing rules to preserve soundness.

However, this opposite direction is essential in at least two cases. Firstly, when a new object is created, if it has fields declared as never-null, they have to be initialized properly before the object can be safely used. This problem is known as object initialization. Secondly, even in a stable system, an expression of a maybe-null type can be checked for nullness at run-time. As soon as its value is not null, it should be safe to use it even in place of a never-null expression. The associated code structures are known as certified attachment patterns (CAP) ([12, 27]).

Therefore, the complete solution comes as a combination of

- a null-reference-aware type system
- language rules for object initialization
- certified attachment patterns

1.4 Challenges

One role of type systems in modern statically typed object-oriented languages is to enable reasoning about execution-time properties of objects at compile time. It effectively serves as a mechanism to limit potential aliasing when two variables of different static types can reference the same run-time object. Direct application of the type system approach to null safety quickly becomes too complicated because unlike regular typing rules, variables can become null and non-null during execution. Moreover, step-by-step execution principle does not allow for setting mu-

tually dependent variables simultaneously. In other words, an object of the same type can be seen as having different nullness properties at run-time. A very detailed specification of these properties is possible at compile time, but introduces too much annotations in the language with little benefit for the end user, who, looking at the code, can say "Isn't it obvious from the code? Why do I have to annotate it?"

Removal of annotations is possible, but should not lead to another extreme: the need to recompile the whole system from scratch to make sure after every change it still remains null-safe. This actually would prevent programmers from development of reusable libraries that can be checked independently of each other.

These observations lead to the following challenges when designing null safety rules:

- Avoid additional annotations in the source code to keep language syntax simple and to abstain from special notions for rarely used cases.
- Support human-understandable reasoning for obvious code (in particular, overcome restrictions introduced by a type system that are too strong).
- Make sure the checks enable modular software development, so that if a library is checked, there should not be any reason to recheck it if it is used in a specific project.
- Allow for fast re-compilation cycle to keep incremental re-compilation time minimal.

1.5 Role of Isabelle

Programming language rules ensure no inconsistencies arise at compile time or at run-time. The desire to make a language less restrictive and still preserving its guarantees, such as void safety, leads to the need to introduce more sophisticated rules to the language specification and to have a more complex implementation. At some point the rules and their interaction become too complicated to rely just on intuition. Moreover, the intuition may induce false believes in soundness of a mechanism. Carrying out soundness proofs on paper improves the situation, but does not guarantee absence of implicit assumptions, missing cases or some de-

tails that can be simply overlooked. Fortunately, current achievements in software tools assisting with the development of formal proofs enables their use in a casual setting. In my work I used one of such proof assistants – Isabelle/HOL– to formalize language rules and to prove their soundness. Without this tool, evolution of the theory would take much longer and would be error-prone. Sometimes an addition of a new concept to the theory triggered a significant rewrite that was immediately caught by the mechanical assistant and would be difficult to track manually. Also, the tool does not allow for performing proofs "by parity of reasoning", every case should be considered. This is especially important for a language specification where every tiny detail should be in place.

Isabelle/HOL was successfully used in different projects starting from algebraic topology to verification of an operating system micro-kernel ([25]). It is built on top of a logic-neutral core called *Pure* with a specialized formalism of Higher-Order Logic (*HOL*). Talking about safety properties it was used to verify type soundness of JinjaThreads using operational semantics for concurrent execution of Java-like programs ([32, 33, 41]). Some decisions used in that formalization are adopted in the current work, some are new.

Even though selection of Isabelle/HOL is both voluntary (I knew it better) and traditional (it was used to formalize and prove type safety of Jinja) – both look like obvious advantages to me, there are some other features that make it more attractive compared to other proof assistants:

- ability to write forward proofs in *Isar* language that makes reasoning closer to conventional textbooks;
- proof automation allowing for finding direct (i. e., not involving case analysis or induction) proofs automatically without diving into low-level details;
- built-in document preparation system enabling to type set all formulas (e. g., in this paper) directly from verified lemmas and preventing from using them for unfinished or failed proof scripts.

All formal specifications of functions and predicates presented in the text, as well as associated lemmas and theorems have corresponding machine-checked code written in Isabelle/HOL and available in appendix B.

1.6 Terminology

Different languages use different terminology for language constructs and notions. Given that the research has been done using programming language Eiffel ([12, 27]), here is a summary of terms that are used throughout the text and are known somewhere else by different terms:

Access on void target – an exception raised by the run-time when it detects that a target of a call is **Void**. This is similar to `NullPointerException` in Java.

Anchor – an entity referenced in an anchored type.

Anchored type – a type described in terms of some entity rather than explicitly. Most used anchored types are **like Current** denoting a type of the current class and **like** *query* where *query* is a feature that returns a value. The type declared in an ancestor class is automatically recomputed in a descendant class to take into account its context.

Attribute – a data member of a class that does not involve computation and whose value can be retrieved from memory. A variable attribute is represented by a field of an object at run-time. A constant attribute has a fixed value. When the qualification is omitted, the term **attribute** usually means a variable attribute.

Conformance – a binary relation on types that tells if an expression of one type can be used in place of an expression of another type. In mathematical notation *A conforms to B* is denoted as $A \leqslant B$.

Creation procedure – a constructor, a feature used to initialize an object after it is allocated in memory; unlike Java-like languages, creation procedures in Eiffel have different names, so it is possible to have two different creation procedures with the same signature.

Current – an entity used to denote a current object on which all unqualified calls are implicitly applied, often denoted by **this** in other languages.

Expanded class – a class whose instances have copy semantics, i. e., are passed as objects rather than references to objects. An expanded class is similar to a value type in C#, but un-

like a value type it should provide a default creation procedure that satisfies all requirements to creation procedures.

Feature – a member of a class, i. e., a routine or an attribute.

Root class – a class that is used to start system execution, together with a dedicated creation procedure called **root creation procedure**, indicates an entry point to the program that is similar to a function **main** in other languages.

Routine – a method. Routines returning values are called functions, those that do not return a value are called procedures.

Routine body – a sequence of instructions associated with a given routine.

System – a program.

Void – a special entity that evaluates to an absent reference, also denoted by **null** in some languages.

Void safety – a compile-time guarantee that a system never causes access on void target at run-time, also known as null safety.

1.7 Contributions

The effort to address challenges from section 1.4 has led to the development of new language mechanisms and engineering decisions. The list of novelties can be classified by the associated subject:

- Type system
  - Extension of conformance rules for formal generic types with respect to void safety.
  - Specification of attachment and self-initialization properties of formal generic types.
- Application to real projects
  - Implementation of all proposed mechanisms in a production compiler environment with the ability to test them on a large code base with millions lines of code.
  - Gathering statistics for and analyzing publicly available libraries and projects, confirming or disproving certain proposals.

  - Introduction of a notion of void safety levels and other engineering decisions to ease transition from void-unsafe to void-safe code.
- Object initialization
  - Support for creating object structures with circular references without any additional annotations, maintaining independent component development.
  - Ability to call regular methods from class constructors without loosing safety guarantees.
- Certified attachment patterns
  - Generalization of scope rules for arbitrary conditional expressions.
  - Relaxing declaration and reattachment rules for local variables.
  - Addition of "design mode" to enable gradual development of large systems.
- Mechanically checked formal proofs
  - Specification of language validity rules in a proof assistant environment.
  - Drawing essential properties of transfer functions and validity predicates.
  - Proving soundness of proposed certified attachment patterns with respect to a big-step operational semantics.

2

Overview

2.1 Research area

Void safety, also known as null safety, introduced in Eiffel, guarantees absence access on void target, analogous to `NullPointerException`, at run-time in void-safe programs. The issue with such exceptions remains one of the most critical ones in modern object-oriented software development. High interest in the area can be seen in the syntactic sugar added to existing popular languages to deal with nullity such as null coalescing and safe navigation operators as well as claimed null safety in relatively young languages such as Kotlin that, according to the online documentation [30], may exhibit "some data inconsistency with regard to initialization (an uninitialized **`this`** available in a constructor is used somewhere)", and its specification [31] mentions "null safety" next to "possible violations" in a *TODO* list.

The most obvious language-based solution to the problem is distinction between possibly-null and non-null types corresponding to expressions that can and cannot return **`null`** respectively. As soon as possibly-null expressions cannot be assigned to variables of non-null types, all the guarantees should be there. This statement constitutes the basis of a void-safe type system. Data flow compatible with this statement is schematically shown in figure 2.1 in solid lines: an expression with a specific attachment property (denoted *Attached* for non-null types and *Detachable* for possibly-null types) can be assigned to a variable with the same attachment type and an expression of an attached type can be assigned to a variable of a detachable type.

Given that objects are created with all attributes unset and taking into account the type system rules, it is easy to realize that very soon everything in the program will get a detachable type. To avoid this collapse, some mechanisms are required to go in the opposite direction – from detachable expressions to attached variables. There are two such mechanisms: object initialization rules

ensuring that soon after object creation all fields of attached types become indeed attached to some objects, and so called certified attachment patterns that guarantee that in certain conditions results of expressions of detachable types are in fact attached to existing objects. These two mechanisms attract the most research interest.

2.1.1 *State of the art*

Instruments to remedy the famous `NullPointerException` can be divided into two large groups. The first group relies on language mechanisms to prevent access on **`null`**. They enable modular development and favor not only code reuse, but also reuse of the safety guarantees. Once a class is compiled with all checks on, it will be complied with the same result in a different project. The second group relies on pure code analysis not requiring any support from the language. They do not force a programmer to provide any hints in a form of type specifications or extra annotations – a program can be compiled "as is". Consequently, it is not known in advance whether a particular expression may produce **`null`** or not, therefore, requiring whole system analysis that is usually performed on compiled byte-code using abstract interpretation techniques to figure out an attachment status of the expression, and reporting errors or warnings depending on whether 100% safety guarantees are requested or not. The price to pay for such a posteriori analysis is longer compile-check-update cycle and lack of support for modular component-based development. In the ideal world the best of both approaches should be combined to support development of reusable components without compromising automatic detection of safe code snippets.

Language-based solutions tend to present a proof of concept without diving into "not that important" details of specific language features. One of them is genericity enabling creation of type-safe classes not using run-time type checks to make sure expressions are of a specific type. Consequently, language rules for generic parameters of a class are either ignored or considered as not requiring special attention. In practice, however, it becomes important to know properties of generic types to decide whether a variable of a generic type needs initialization or not and whether its attachment status can be guaranteed.

The issue with object initialization was identified in early days of research in the area of null safety. Most proposed schemes deal with it by employing techniques known from the theory of type systems: they propose to specify whether a particular expression corresponds to a completely initialized object or not. This is similar to specifying that an expression has one type rather than another one. However, the main difference between class types and attachment types is that the same reference can have different initialization status during execution and, therefore, different attachment types. On the other hand, in a class-based type system a reference has just one, possibly abstract, type. The solutions try to identify suitable rules when one attachment type can be turned into another one without compromising void safety. In order to mark such intermediate states in the code they need more type annotations than just non-null and possibly-null. Moreover, the rules for such types differ from those of regular class-based types, because the marks are temporary, i. e., do not apply to the whole object life cycle, and, therefore, cause confusion for developers.

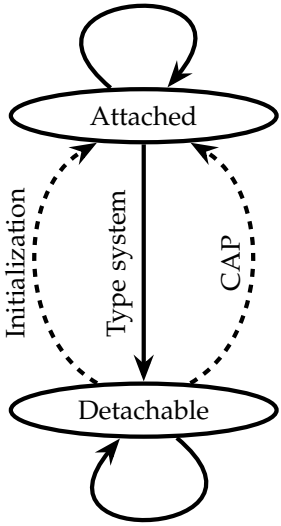

Figure 2.1. Key elements of void safety.

In most cases proposals to make a language null-safe are accompanied with the proofs confirming their soundness against some formally specified language semantics. Unfortunately, sometimes authors ignore some important features of a real programming language such as branching expressions with different outcomes in different branches or exceptions. This leads to incomplete understanding of whether the proof is sufficient and can be extrapolated to the full language, or whether the rules need further fine-tuning to make them realistic. The proofs are carried out on paper and are potentially subject to mistakes and missing details. Current state of proof checking environments allows for verifying proofs by software tools rather than by hand. Some subtle issues have been found earlier when performing such automated

checks for type systems soundness proofs. The same can be done for null safety.

Finally, development of null safety theories is often decoupled from a real code base. The prototypes allow for analyzing a limited set of classes and for reporting how many annotations are required or how many errors or warnings are reported with the suggested checks. While this gives a feeling whether the proposed solution is good enough, it does not permit to see all the consequences and how well the solution scales on real-life code.

2.1.2 *Goals*

Usability of a particular safety mechanism depends on how well it can be integrated in the development cycle. For this reason and based on the analysis of different approaches to guarantee void safety, it should rely on language rules allowing for *quick recompilation* and for *creation of reusable components*.

To this end a void-safe language needs to distinguish between non-null and possibly-null types. Being effective for the object life cycle starting from the moment when an object reaches a stable state after its creation, they cannot be used to describe a transition period when some object fields marked as non-null may be `null`. Fortunately, the cases when such an object is passed around are rare. Therefore, it would be nice to *avoid any additional annotations* describing the temporary object state, but to keep the guarantees intact.

Although purely type-based language rules for void safety are sound, for software developers they are often too strict and trigger too many false positives, rejecting programs that are provably safe. This issue is addressed by certified attachment patterns (CAPs). If it would be possible to *improve CAPs* and to allow for more code to be accepted by the language rules, it would improve users perception of the mechanism and back up their intuition.

As a set of language rules becomes more involved, it is essential to provide evidence of their soundness. To avoid any misinterpretations and omissions, *formal specification of void safety rules* can be done in some computed-aided environment rather than on paper. This opens the road to *mechanically checked proofs* of soundness in a proof assistant environment.

2.1.3 *Novelty*

The research touches almost all fields of the target domain:

- On the engineering front, it reviews practical decisions that made Eiffel a void-safe language, and presents detailed statistics of migration from void-unsafe to void-safe code.
- On the type system front, it clarifies properties and validity rules for formal generic parameters of generic classes.
- On the object initialization issue, it proposes two possible solutions that do not require any new language constructs and cover all practical needs, preserving modular software development.
- On the usability front, it makes treatment of local variables flow-sensitive, introduces a concept of positive and negative scopes and frees users from the need to specify attachment marks for local variables explicitly.
- On the formalization front, it is the first work that uses a proof assistant environment to specify void safety properties and validity predicates in a machine-checkable way.
- On the soundness proofs front, it is the first time when they are carried out and mechanically verified by the proof checker.

2.2 Achievements

2.2.1 *Methodology*

One of the main objectives used when performing this work was to ensure usability and practical feasibility of the proposed solutions. This led to the following methodological principles:

- *Graduality of changes.* New language mechanisms can break existing code. This situation is unsatisfactory for large projects and should be alleviated by ensuring backward compatibility when
  - particular guarantees are not requested, e. g., by allowing for side-by-side compilation of the same code in void-safe and void-unsafe modes;

  - new language rules are claimed to be "better" than existing ones, e. g., by compiling already void-safe code with new rules without any changes.
- *Practical acceptance.* New safety guarantees require more restrictions to be placed on code. A satisfactory solution should show its usability on a large code base developed by different users with varied level of experience as well as on new large projects where not all classes can have ready-to-use non-abstract implementations.
- *Human-factor avoidance.* New language rules with a lot of technical details can be difficult to grasp and to specify correctly by hand. Human intuition should be replaced by automated tools for specifying the rules and for verifying expected outcomes.

2.2.2 *Value of the work*

The work has the following merits:

- All proposed validity rules that ensure safety of object initialization and improve CAPs are specified formally in Isabelle/HOL. This leaves no space for misinterpretation and can be used as a reference to update the language standard.
- Comparison to examples from the previous work and reported statistics demonstrate practical superiority of the proposed solutions.
- Usage of tool-assisted technique to prove soundness of the mechanisms lifts confidence of their correctness to the new level.

2.2.3 *Proposed solutions*

In my work I propose to discriminate on the void-safety related properties of formal generic types and to specify their attachment status and whether they are self-initializing. Both properties are used in the language validity rules ensuring void safety. Moreover, based on these definitions I propose new conformance rules for formal generic types in Eiffel that are less restrictive than the ones specified in the current reading of the standard.

For the issue with object initialization I propose two solutions: a stricter one that may be of interest for other languages aiming to provide void safety, and a less demanding one requiring slightly more sophisticated source code analysis. Application of the rules is shown on several examples taken from real code. I describe coding guidelines and report what changes are required to adapt existing code. In the first solution it is required to set all attributes of a class before a reference to the current object can be safely used. This is ensured by the proposed language rules and does not require any additional annotations in source code. Analysis of public libraries with more than a million lines of code shows that this solution works for 98% of code.

The remaining 2% of cases can be dealt with the second solution. It permits passing a reference to the current object before all attributes of a class are properly set, but prohibits qualified feature calls until all attributes are set. This enables creation of object structures with mutual references to each other stored in the fields of attached types. I show that all examples from the previous work can be expressed without violation of the proposed rules and unlike the previous work do not require new annotations.

In addition, I formally specify the proposed rules as predicates in Isabelle/HOL and prove some of their properties such as monotonicity. Also, I show that the first solution is a more restrictive version of the second one.

To ensure completeness of void safety guarantees for object initialization I review finalizers and identify issues that can lead to access on void target. Then I propose to eliminate the issues with a special validity rule similar to the rule for object initialization: qualified calls on reference targets should not be allowed in a finalizer.

Next I present deficiencies of using a type-based approach to govern void-safety reattachment rules for local variables. Instead, I analyze safe uses of the variables in void-safe programs and introduce a notion of positive and negative scopes. Combined with proposed conversion for boolean connectives this removes the need to do a case-by-case analysis for each boolean operator and simplifies further theory development. I propose a way to develop a void-safe system when not all classes can have concrete implementations, and reflect a required concept in the for-

mal model. Then I define a transfer function for a subset of an object-oriented language that tells what variables are definitely attached at a particular execution point. I use it to specify a validity predicate ensuring that code is void-safe. By analyzing public libraries I reveal that the proposed predicate triggers about 33% fewer spurious errors than the one using attachment marks for local variables. At the same time all existing void-safe code compiles without an issue.

In order to prove soundness of the proposed certified attachment patterns, I introduce a notion of a valid state that connects compile-time information with a run-time state. In a valid state every variable known to be attached at compile time should have an existing object value at run-time. I specify language semantics in a big-step style for two different cases: when the language is considered void-safe and when – not. After that I prove a preservation theorem for the void-safe version of the semantics. Then I perform comparison of program execution using void-safe and void-unsafe versions of the semantics and prove that for a void-safe program there is no difference. This signifies absence of null pointer exceptions at run-time and soundness of the validity predicate.

All formalization and proofs are performed using the general proof assistant Isabelle/HOL. This ensures there are no missing cases in the proofs or implicit assumptions in premises of lemmas and theorems. I provide the complete code of the proofs in appendices.

2.2.4 *Practical effect*

The work demonstrates close integration of theoretical approaches with practical needs:

- Collected statistics about changes required to make void-unsafe code void-safe can be used to estimate efforts required to do the conversion for existing projects.
- Measured increase of compilation time to check object initialization rules shows its marginally low influence on total compilation time.
- Case study for the object initialization issue presents qualitative measure of achieved results by using examples from previous work and suggesting new ones.

- Certified attachment patterns for local variables are tested on a large code base against previously implemented type-based validity rules to estimate their superiority for real code.
- All massive experiments are performed on 100% live code from production releases of different open-source libraries.

2.2.5 *Relevant publications*

My publications on the subject of the thesis include (chapter and section numbers in the comments indicate appropriate sections of the thesis):

[50] Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. "Void Safety: Putting an End To the Plague of Null Dereferencing." In: *Dr.Dobbs Journal* online (Sept. 1, 2009). URL: http://drdobbs.com/architecture-and-design/219500827

Introduces the concept of void safety with some base rules covering basic type system rules (chapter 4), simple certified attachment patterns (chapter 6) and issues with creating array objects (section 3.2.2).

[51] Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. "Avoid a Void: The Eradication of Null Dereferencing." In: *Reflections on the Work of C.A.R. Hoare*. Ed. by A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood. History of Computing. Springer London, 2010, pp. 189–211. ISBN: 978-1-84882-912-1. DOI: 10.1007/978-1-84882-912-1_9

Expands on the void safety rules and explains new constructs added to the language founded on practical experience such as stable attributes (section 3.2.4) and mandatory check instruction (section 3.4.2), provides the first statistics about migration (sections 3.3.2, 5.1.6 and 5.2.4 and appendix A).

[37] A.V. Kogtenkov. "Mechanically Proved Practical Local Null Safety." In: *Proceedings of the Institute for System Programming of the RAS* 28.5 (Dec. 2016), pp. 27–54. ISSN: 2079-8156 (Print), 2220-6426 (Online). DOI: 10.15514/ISPRAS-2016-28(5)-2

Formalizes certified attachment patterns in Isabelle/HOL for read-only and read-write local variables (chapter 6) and provides mechanically-checked soundness proofs for the validity rules (chapter 7).

Some ideas, in particular, for definitions and notations used for the transfer functions in chapter 5 and chapter 6 are taken from the following articles:

[35] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. "Alias and Change Calculi, Applied to Frame Inference." In: *CoRR* abs/1307.3189 (2013). URL: http://arxiv.org/abs/1307.3189

[36] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. "Alias calculus, change calculus and frame inference." In: *Science of Computer Programming* 97, Part 1 (2015). Special Issue on New Ideas and Emerging Results in Understanding Software, pp. 163–172. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.11.006

Selected on-line resources on void-safety I added when working on void safety mechanism as an author or as a co-author include:

[34] Alexander Kogtenkov. *Void-safety: tag info*. Stackoverflow, July 15, 2016. URL: http://stackoverflow.com/tags/void-safety/info (visited on 2016-12-30)

[14] *Eiffel compatibility options*. Community portal for Eiffel, 2016. URL: https://www.eiffel.org/doc/eiffelstudio/Eiffel%20compatibility%20options (visited on 2016-12-30)

[11] *Creating a new void-safe project*. Community portal for Eiffel, 2016. URL: https://www.eiffel.org/doc/eiffel/Creating%20a%20new%20void-safe%20project (visited on 2016-12-30)

2.3 Outline

The rest of the dissertation is organized as follows:

Chapter 3 reports overall experience with building void-safe Eiffel infrastructure, including particular engineering solutions that made this possible.

Chapter 4 explains conformance rules in a void-safe type system and associated type properties like attachment

status and self-initialization. Then it considers how they are applied to formal generic types and clarifies corresponding definitions and validity rules.

Chapter 5 describes the issue with object initialization and proposes two solutions. The informal validity rules are accompanied by formal specifications of transfer functions and validity predicates in Isabelle/HOL. The solutions are then compared between each other and against different examples on the subject proposed by other authors.

Chapter 6 focuses on formalization of validity rules for local variables without using any attachment information. The section is concluded with a report about practical results of using this scheme instead of the type-based one.

Chapter 7 introduces big-step semantics for the subset of a language described in the previous chapter and proves soundness of the void safety predicate using the Isabelle/HOL proof assistant.

Appendix A provides detailed statistics about migration from void-unsafe to void-safe code in *EiffelStudio* environment.

Appendix B contains the complete theories code with proofs in Isabelle/HOL.

2.4 Conclusion

2.4.1 *Main results*

The outcome of the work can be summarized as follows:

- Clarification of attachment properties and conformance rules for formal generic types.
- Demonstration of practical feasibility of annotation-free safe object initialization on existing and new examples as well as on large open-source libraries.
- Replacement of type-based rules for local variables with certified attachment patterns.

- Formalization of void safety rules for object initialization and for certified attachment patterns in Isabelle/HOL.
- Mechanically-checked soundness proofs for certified attachment patterns.
- Complete implementation of all proposed solutions in a production environment confirming their usability.

2.4.2 *Future work*

Immediate continuation of the current work could be its extension to

- Improve object initialization rules to enable certain assertion checks involving back references.
- Move certain run-time assertion checks such as whether a given type is self-initializing to compile time to enable static detection of void-safety related errors.
- Carry out mechanically-checked soundness proofs for object initialization.

The second item from the list above could also be useful in other scenarios and could become the first step to make software verification available for casual use. Research areas affecting void safety guarantees include:

- Checks for invariants of objects being constructed. Until the construction is finished, some attributes may be unset and class invariant checks may be unsafe. Solving this issue would be an important event for object-oriented programming in general.
- Covariant redeclarations that are needed to support *Design by Contract*™. Such redeclarations are known to be type-unsafe and for the same reason they can ruin void safety. One of the challenges here is to avoid breaking too much existing code while keeping language complexity moderate.

3

From Theory to Practice

3.1 First steps

For everyone who is just introduced to the realm of void-safe programming, the rules of the game seem pretty simple: variables and functions may be declared to have an attached type, meaning that they always produce an existing object at run-time, or to have a detachable type, meaning that they may produce either an object or **Void**. In order to preserve soundness, i. e., to make sure an attached expression never returns **Void** at run-time, it is permitted to assign expressions of attached types to variables of attached or detachable types, but expressions of detachable types can be assigned only to variables of detachable types. If it were allowed to assign an expression of a detachable type to a variable of an attached type, it would be possible to set this attached variable to **Void** breaking all the guarantees. (The term "to assign" is used here in a broader meaning and also includes argument passing involving actual and formal arguments as source expressions and target variables respectively.)

Of course, it should be possible to check that an expression is attached and to use its value in the context that requires an attached type:

```
if x /= Void then
    ... -- Use 'x' as if it is of an attached
           type.
end
```

Also, attributes of attached types should be initialized prior to their use, e. g., they all should be set by the corresponding creation procedure. Then a validity rule that ensures there is no access on void target at run-time would be that a qualified call is valid only if its target is of an attached type. All this has been de-

scribed in more details by Bertrand Meyer in [47]. And it would be the entire story if there were no generic parameters, if objects could be created in an atomic operation and if users did not want to forget about attachment status of variables in "obviously" safe cases. These important elements of the mechanism are described in greater details in the following chapters. The current one focuses on less fundamental but still important aspects of the language design and tool support.

3.2 Language conventions

Assuming there is a distinction between attached and detachable types, a number of other language-related issues need to be sorted out to make the language void-safe.

For Eiffel, additional difficulties came from the fact that void safety was added for an existing language with a lot of existing libraries and projects. Therefore, tools such as the compiler or the IDE had to handle the old void-unsafe code and the new void-safe code simultaneously. This was supported by using Eiffel Configuration Files (ecf) that specified whether a particular library or project is void-safe or not.

3.2.1 *Default attachment status*

At the very beginning of transition to void-safe code in Eiffel it seemed obvious that old code was written without any void-safety rules in mind and all reference types should be considered detachable by default because void-safety added new rules to the existing language. To let users play with the new rules without having to rewrite all the code at once an attempt was made to have a compiler option `is_attached_by_default` to treat class types attached or detachable depending on its value. This was even mentioned in an informative part of the language [12, 27]. Fortunately the standard itself specified that class types without any special annotation are considered attached.

In practice the decision to have such an option caused more trouble than benefit, in particular,

- the option caused confusion for users as some of them treated `attached-by-default` setting as an alias to the void safety option;
- the requirement to add **attached** keyword in front of attached class types and omit **detachable** keyword in front of detachable class types complicated further migration to standard rules;
- even non-void-safe code uses type declarations to denote attached types more frequently than detachable ones.

After 10 years upon introduction of the option that controls default attachment status of class types, it was dropped in [16].

3.2.2 *Array items*

The original creation procedure of a class *ARRAY* [*G*] has the signature

make (*min_index*, *max_index*: *INTEGER*)

to create an array object with item indexes in the specified range. This works perfectly in a void-unsafe program. If the formal generic parameter *G* is instantiated with an expanded type, all items in the range get their default values, if with a reference type, all items are set to **Void**. The feature to retrieve items from an array has the signature

item (*index*: *INTEGER*): *G*

where *index* is a requested item index. The return type is the formal generic of the array class.

The same declarations do not work for void-safe programs when the formal generic is instantiated with an attached reference type. A program then can simply do

```
my_array: ARRAY [FOO]
value: FOO -- This type is attached by default.
...
create my_array.make (1, 5)
value := my_array [3] -- Problem here!
```

and expect to get an attached value at the last line. But no value has been stored at this index yet!

Several solutions are possible:

1. Change the signature of the feature *item* to return **detachable** *G* instead of *G*. Would it work? Of course, now the code would be safe: for a freshly created array the feature would return **Void** that would be legal. A client would then check the return value whether it is **Void** or not and use it accordingly. Would it be convenient? No. Every access to an array of attached elements would be followed by a voidness test for the returned value. The same effect could be achieved by declaring the array as *ARRAY* [**detachable** *FOO*]. In other words, for daily programming, if an array is declared with an attached actual generic parameter, any item at a valid index is expected to be attached.
2. Add a precondition to the feature *make* that will ensure the actual generic type has a default value that can be used to fill the array elements:

```
make (min_index, max_index: INTEGER)
  require
    ({G}).has_default
```

   The syntax {*G*} is used to denote a type object for the type *G*. The query *has_default* tells if this type has a default value. This is true for every type except for an attached reference one. The creation procedure can be used to create arrays with elements of expanded or detachable types. But the precondition prevents from allocating memory for a whole array object when the actual generic parameter is an attached reference type. In this case the array should be created using a creation procedure *make_empty* and updated by adding new elements one-by-one that is pretty inefficient. Moreover, the precondition check is done at run-time. A static analyzer could catch creation of an array with a type that has no default value, but at the moment the language standard does not support validity rules that involve static checks like this one. (It would be a good addition.)
3. Provide a new feature that is passed an explicit value to fill a newly created array with:

```
make_filled (default_value: G; min_index, max_index:
    INTEGER)
```

   This solution has no issues with efficiency and is completely safe.

Solution 1 was dropped as very inconvenient for void-safe programming, solution 2 is taken as a temporary one for a transition period. The feature *make* of the class *ARRAY* is marked as obsolete and will be removed as soon as all code is updated to use *make_empty* or *make_filled*. Solution 3 is used as the permanent one.

Apart from the class *ARRAY* there are other classes providing array-like functionality. They all have been updated to have creation procedures that take a default element value. Adaptation of the low-level class *SPECIAL* was more sophisticated. The class allows for allocation of memory for arbitrary number of elements, but distinguishes between the current element count and the capacity of the storage. The capacity is passed as an argument to the creation procedure *make_empty* for creating a new storage. But the number of elements is set to zero by this creation procedure. When new elements are put to the storage, the number of current elements is updated accordingly. All items are stored starting from a fixed index and no gaps between two items are permitted. This approach removes the requirement to fill the whole array area with a default value at creation time, but preserves efficiency of preallocated memory. Only elements in the range between the first and last indexes can be accessed, and storing elements is permitted within the same range or at the index `last_index + 1` provided that the new total number of elements does not exceed the storage capacity. When a new element is stored at `last_index + 1`, the value of `last_index` is incremented by one, otherwise it remains unchanged.

3.2.3 *Once functions*

Unlike Java-like programming languages, Eiffel [12, 27] does not support static fields and static initializers. The similar functionality is encapsulated in a form of a function that is computed only once. Any subsequent calls to the function yield the result of the previous evaluation. A validity rule for a result of a once function is identical to the validity rule of a regular function: if its type is attached, **Result** should be properly set at the end of the function. This works perfectly together with void safety except for recursive calls.

According to the semantics of a once feature call, if this feature is called for the second time, its value is the current value of the special entity **Result**. While we know that it will be set at the end of the function, during execution of the function body the value may still be unset. The simplest example demonstrating the problem is:

```
f: FOO
    once
        f.something
        create Result.make
    end
```

The special entity **Result** is indeed properly set the end of the feature *f*, but the very same feature is accessed earlier. In real life scenario a recursive call to a once function can occur indirectly and may involve dynamic binding, making it infeasible to detect such cases at compile time without analyzing the whole system.

It might be possible to add a new validity rule for once functions requiring that **Result** should be set before any qualified feature call (this would be somewhat similar to object initialization techniques discussed in chapter 5). Unfortunately there is a pretty common pattern to relay creation of the result to some other query: **Result** := *other.something*. A target of the qualified call is usually another once function used to retrieve a required value. Disallowing qualified feature calls before **Result** of a once function is properly set would lead to disallowing this pattern that in turn could break the principle of information hiding.

So far, no better solution is found than to check whether the result of a once function of an attached type on the second call is actually set. If it is not set, an exception is raised, signaling an error before any attempt to perform access of this unset value, and preserving void safety guarantees of the language type system. In order to maintain semantics of once functions, the exception is recorded and if it is caught before returning to the place where the once function was called first, it is re-raised there, so that both second and first accesses to the once function produce exactly the same (exceptional) result.

Strictly speaking, access to a result of a once function that has been already called, but has not exited yet, is not necessary specific to void safety. The same approach to detect a recursive call and to raise an exception can be used regardless of the result type

of the function. This ensures the same program behaves identically when compiled as void-safe and as void-unsafe.

3.2.4 *Scopes for attributes*

Chapter 5 formally specifies rules to deal with the cases when attachment status of an expression can be derived from the code itself. For example, for a local variable *local_var* of a detachable type the following code is absolutely valid:

```
if local_var /= Void then
    local_var.something
end
```

This code falls in the category of certified attachment patterns that enable the call in **then**-branch of the conditional instruction. What if *x* were an attribute, would it work too?

Unfortunately, no. Firstly, if there is any intermediate call *some_call* after the voidness test and before the call on *attr* (remember, it is assumed that *attr* is of a detachable type) like in

```
if attr /= Void then
    some_call
    attr.something
end
```

then *some_call* can set the attribute *attr* to **Void** and the instruction *attr.something* will trigger an exception.

What if the intermediate call is removed like in the following code?

```
if attr /= Void then
    attr.something
end
```

It turns out that the code could still be unsafe depending on used concurrency mode. If thread-based concurrency is enabled, another thread can set *attr* to **Void** after the voidness test and before the call on this attribute. Therefore, even such simple code would not be safe if certified attachment patterns were applied to attributes rather than only to arguments and to local variables.

If the program runs in SCOOP mode (see [56]), the value of the attribute *attr* cannot be modified asynchronously from this code.

Therefore, the last example becomes safe. However, the previous one does not. So, in general case, extension of CAPs to attributes brings more complexity to the language definition than benefits.

But what if we know that an attribute is never assigned **Void** explicitly? In that case once a non-void value is attached to the attribute, it remains non-void all the time. Due to the property to keep their attached state forever, such attributes are called **stable**. As of now, a note clause syntax is used to mark such attributes in the code (here the note clause serves the same purpose as annotation attributes in C# [26]):

```
attr: detachable FOO
    note
        option: stable
    attribute
    end
```

Stable attributes have a detachable type, but a validity rule tells that they can be assigned only an expression of an attached type. This guarantees that as soon as we know that the attribute is not void, we can rely on its attachment status, because it cannot become detachable anymore. As a result all certified attachment patterns specified in chapter 6 for local variables are applicable to stable attributes, and are even simpler, because, unlike local variables, a stable attribute cannot be assigned a detachable expression.

The notion of attached state stability turned out to be convenient for assigner procedures as well. Eiffel allows for associating a procedure of an appropriate signature with a query. This procedure is then called when the query is used in a qualified call on the left-hand side of an assignment. E. g., the class *ARRAY* declares a feature *item* with an assigner *put*

```
item alias "[]" (i: INTEGER): G assign put ...
put (v: G; i: INTEGER) ...
```

This allows writing

```
some_array.item (3) := "A string"
```

or, in a more conventional syntax,

```
some_array [3] := "A string"
```

Both variants are equivalent to

```
some_array.put ("A string", 3)
```

This works excellent for an array where access to elements is performed by an index that is easy to check whether it fits the index range or not. The story is different for tables with arbitrary keys. Assume that both actual generic parameters of a class *HASH_TABLE* denoting an item type and a key type are attached types. The feature *put* has the signature

```
put (value: G; key: K)
```

If we want the feature *item* to be of type *G* (the type of elements) like in *ARRAY*, there should be a precondition to make sure the element is present in the table and that the return value is not **Void** when **Void** elements are not permitted:

```
item alias "[]" (key: K): G assign put
    require
        has (key)
    ...
```

The function *has* returns **True** if and only if the table has an element with this key.

Then, before a client could call the feature *item*, it would have to check whether the element is present in the table, and valid code would look like

```
if some_table.has (my_key) then
    value := some_table [my_key]
...
```

In other words, the element would be looked up by the feature *has* and then its value would be retrieved by the feature *item*. This rings the bell about efficiency: wouldn't it be better to return a found element immediately or to return **Void** if the element is not found? At least most of existing code relied on the code pattern that a value returned by a table lookup function is tested against **Void** and used afterwards. To support this behavior the precondition *has* is removed and the return type is changed to be **detachable** *G* instead of just *G*. Now we expect to get **Void** even if all elements in the table are attached because we do not check beforehand that a given key is present in the table:

```
item alias "[]" (key: K): detachable G assign put
```

Unfortunately, the modified code does not compile anymore: the result of the query and the argument of the assigner are now of different types – **detachable** *G* and *G* respectively – and a validity rule says that they should be the same. In particular, this rule is essential in making sure that types of elements that are put to an array and that are retrieved from the array are the same.

The trick here is similar to the one with stable attributes. Even though the query type is detachable, it is allowed to "assign" to it only attached values. Therefore, the signature of the assigner does not have a mark **detachable**. However, retrieved values are of a detachable type, because some keys may have no associated elements. On the source code side, the feature *item* is marked in *HASH_TABLE* as stable:

```
item alias "[]" (key: K): detachable G assign put
    note
        option: stable
...
```

This convention relaxes the validity rule for assigners of **stable** queries: attachment-mark-free types used in a query and in the associated assigner should be the same and the query type should conform to the type used in the assigner, thus enabling the declaration above. Now the items from a hash table can be retrieved in one step:

```
if attached some_table [my_key] as value then
    ... -- The element is found and attached to 'value',
           use it.
else
    ... -- There is no element with this key.
end
```

3.3 Adapting legacy code

3.3.1 *Void safety levels*

The best way to write void-safe systems is to ensure all validity rules are checked from the beginning of the development. Unfortunately, if a void-unsafe system has to be adapted to void safety, it becomes difficult for developers to identify steps for achieving

the final goal. The validity rules could serve as a good basis to split migration of the code into stages, called **void safety level**:

1. **None:** The code is void-unsafe, no void-safety rules are checked.
2. **Conformance:** Attachment status is taken into account in conformance checks (section 4.2, validity rule 4.1).
3. **Initialization: conformance +** Variables should be set before use (section 4.2, validity rule 4.2).
4. **Transitional: initialization +** Feature calls are allowed only on attached targets (section 4.2, validity rule 4.3).
5. **Complete:** All avoid safety rules are checked, including object initialization (chapter 5).

The void safety levels allow for performing the transition from void-unsafe code gradually. The levels are ordered as follows:

$$\textit{None} < \textit{Conformance} < \textit{Initialization} < \textit{Transitional} < \textit{Complete}$$

The levels can be specified for a class using a compiler option `void_safety`.

Inheritance and client relationships between classes introduce relations *is_descendant* and *is_client*.

Definition 3.1. *A class A uses a class B if it is a descendant of B, a client, or both: uses A B = is_descendant A B* $\vee$ *is_client A B.*

A reflexive transitive closure of the predicate *uses* allows telling what classes are reachable from the given one.

Definition 3.2. *A reflexive transitive closure of the predicate* `uses` *is called* `reachable`*.*

Definition 3.3 (Void safety level validity)**.** *A class has a particular void safety level if it satisfies all validity rules of that level and all classes it* `uses` *have the same or higher level of void safety.*

This validity rule ensures that all classes a given one depends on satisfy at least the same void safety level:

Lemma 3.1. *If a class has a particular void safety level then all classes reachable from it have the same or higher level of void safety.*

Proof. From the definition of reachable classes and definition 3.3. □

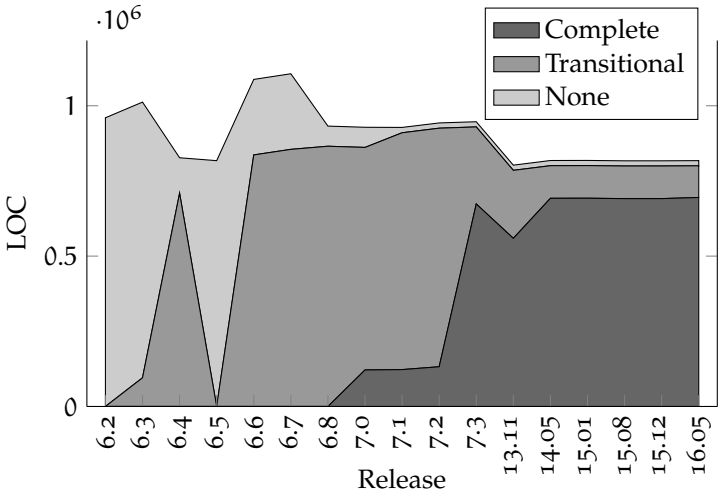

Figure 3.1. Number of lines (LOC) with different void safety levels in different releases.

Corollary 3.1. *If a root class of a system is completely void-safe, then all classes in the system are completely void-safe, i. e., all classes in the system satisfy all void-safety validity rules.*

3.3.2 *Migration statistics*

While source code conventions and coding guidelines are great to write software from scratch, sometimes they are not immediately applicable to existing software. First void safety checks were added to *EiffelStudio 6.3* at the time when public libraries almost reached a million lines of code. Rewriting them from scratch was not an option because of code size and the requirement to preserve backward compatibility for customers so that they can use the same libraries in their projects without any changes, just by compiling in non-void-safe mode. That migration took several releases and is still an ongoing work for few remaining libraries that are not completely void-safe.

Although the compiler supported several levels (section 3.3.1), an updated version of a library was not released until it reached a top level of void safety available at that time. As a result there were no public releases with `Conformance` and `Initialization`

levels (figure 3.1). In practice all levels are used. Initially the compiler did not check a possible issue with passing an incompletely initialized object to another creation procedure, therefore many libraries were updated to match a transitional level without creation-specific checks and only later updated to complete void safety.

Figure 3.1 also demonstrates a critical void-safety problem that was discovered in a library class *ARRAY* (or, more precisely in a low-level library class *SPECIAL*) in version *6.5* and is not specific to the language rules. It was allowed to allocate an array for objects of an attached type without providing an object to fill this array with. The fix is described in section 3.2.2. Taking this issue into account, the figure reflects presumed void-safety levels: version *6.5* was not worse than *6.4*, the issue was not realized at that time instead.

Figure 3.2 shows how much code had to be changed to lift non-void-safe libraries to the transitional level in terms of relative number of updated lines (absolute numbers can be found in appendix A). Either the libraries code had to be modified or some additional checks had to added, very little code had to be deleted. There is no obvious dependency between amount of changes and library size. On average, about 15–16% of code had to be updated. Note, that the figure does not take into account improvements of code analysis that became possible thanks to my work, such as significantly lower rate of false positives reported for local variables (section 6.4).

Another metric – the number of classes affected by the migration – is shown in figure 3.3. Compared to the previous one, the percentage is pretty high: about 76% of classes needed to be touched. This affirms that void safety does not come for free and cannot be seen as a minor code update.

The statistics reflecting migration from non-void-safe code to transitional void safety and, as a second step, from the transitional level to the complete one is summarized in table 3.1 (the detailed data can be found in tables A.3 and A.4). According to the table, migration was not a complete rewrite. Also, adaptation of code to make it safe with respect to object initialization ("transitional to complete" part, reviewed in section 5.1.6) involved much fewer changes than the step to make it void-safe without taking object initialization into account.

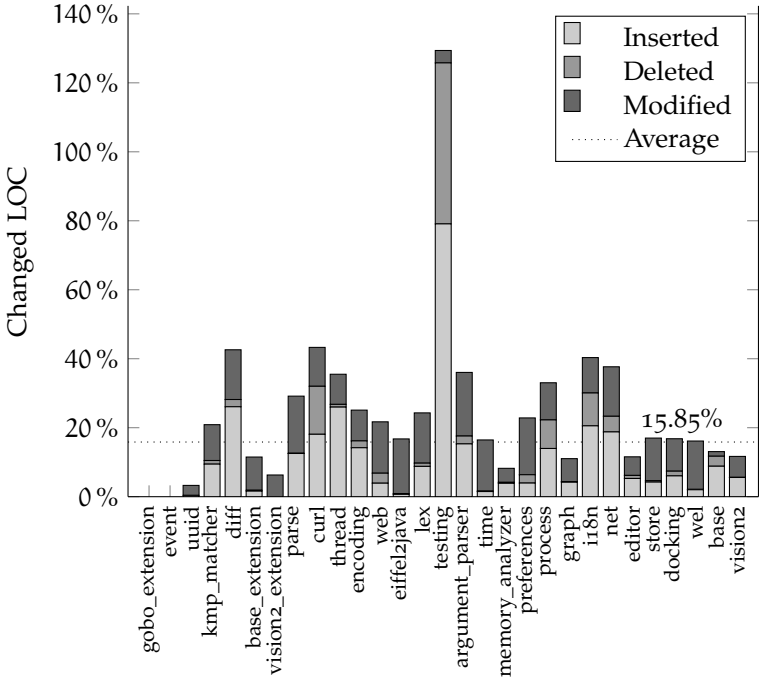

Figure 3.2. Changes in non-void-safe public libraries in lines of code to bring them to transitional void safety level relative to their size.

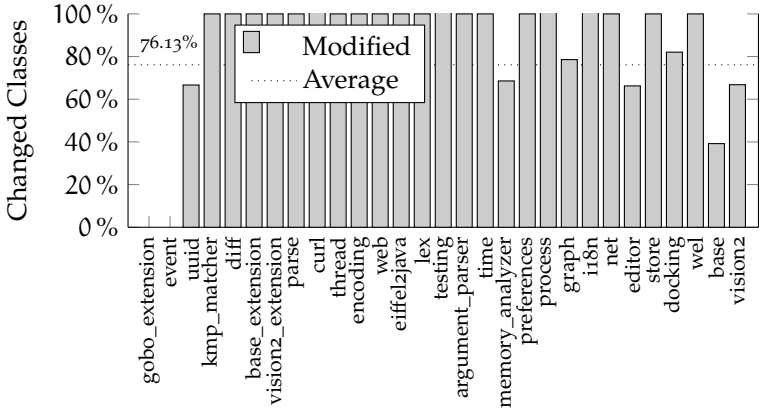

Figure 3.3. Changes in non-void-safe public libraries in number of classes to bring them to transitional void safety level relative to their size.

Table 3.1. Summary of changes in public libraries.

Measurement		Migration to a particular void safety level			
		None → Transitional		Transitional → Complete	
Classes		2912 of 3825	**76.13%**	1867 of 4254	**43.89%**
LOC	Inserted	53867	6.83%	16306	1.98%
	Deleted	10659 of 788984	1.35%	12794 of 822487	1.56%
	Modified	60525	7.67%	9209	1.12%
	All	125051	**15.85%**	38309	**4.66%**

3.4 Controversial issues

Sometimes it is difficult to draw a strict line between alternatives with a similar outcome. Below are the examples that may need deeper discussion or rely on some policy decisions in particular projects.

3.4.1 *Self-initializing attributes*

Attributes in Eiffel can have an associated body similar to a function:

```
attr: FOO
    attribute
        ... -- Some code that sets 'Result'.
    end
```

Such attributes need not be properly set at the end of a creation procedure, i. e., in advance. They are set "on demand" when they are accessed before they have been set, hence the name – self-initializing. If the attribute type is detachable, no initialization is needed because the type has a default value **Void**. If the attribute is of an attached reference type, it has to be initialized prior to its use. When the attribute is accessed before it is explicitly set, its body is executed and a value of the special entity **Result** at the end of the body is used to set the attribute.

This works perfectly when void safety rules are respected all the time. If the same code has to work in both void-safe and void-unsafe mode, there is a problem. In void-unsafe code any refer-

ence type has a default value **Void**. Therefore, there is no reason to execute the attribute body in void-unsafe code. Consequently, the same code behaves differently depending on a selected void-safety compilation mode. In a void-safe mode it calls the attribute body, in a void-unsafe mode it does not.

One of the following approaches can resolve the issue:

- Discard attributes with bodies from the language standard. All attribute initialization would have to be moved to the corresponding creation procedures. This may require code refactoring to do the object initialization properly.
- Respect attachment marks even when code is compiled as void-unsafe. Access to attributes with associated bodies would be performed with a check like in void-safe mode.

Either solution will remove users confusion about the semantics of self-initializing attributes in void-unsafe mode and honor the principle of least surprise.

3.4.2 *Assertion checks*

One of the most problematic cases during migration from void-unsafe to void-safe code is adaptation of routine contracts and routine bodies that rely on contracts. For example, class *TABLE* provides a feature *item* that returns an element corresponding to a given key. The feature has a precondition *valid_key*. In the descendant class *HASH_TABLE* the feature *valid_key* does not perform any checks and just returns **True**. It means that the feature *item* can return either a found value or a default value when no corresponding value exists. Because attached types have no default values, the type of *item* is detachable. As described later in chapter 4, the result in this case is properly set even if it is not assigned a value explicitly. This is what we need when the key is not found.

Contrast to this, a class *CHAIN* defines *valid_key* as a predicate that tells whether a specified index is in the range of valid index values that includes elements from 1 to *count*. Therefore, as soon as a chain index is within the range, there should be an element with this index, and the type of the feature *item* is a formal generic without any attachment marks. Roughly speaking, the class *CHAIN* models an array-like container without any missing elements between indexes 1 and *count*. On the implementation

side the feature *i_th* (this is a renamed version of the feature *item* of the class *TABLE*) positions a cursor to the specified index and retrieves a value using a cursor-based feature *item*. For example, in a descendant class *LINKED_LIST*, if all the assertions could be statically verified, one could prove that there would be indeed a linked list cell with the value. The feature *i_th* would set an attribute *active* to point to that found cell. Then the item value could be retrieved via *active* that would be known to be attached. Unfortunately, the technology is not there yet and in order to preserve soundness of the type system we have to be pessimistic. Because of that, the type of the feature *active* is detachable. The implementation of {*LINKED_LIST*}.*item* looks like

```
check attached active as a then
    Result := a.item
end
```

Here the assertion **check** is mandatory (it is called later a **guard** instruction), i. e., cannot be disabled at run-time like most other assertion checks. It guarantees that the local variable *a* is attached inside the **then** part. If at run-time the value of *active* is not attached, an exception is raised. This is similar to raising an exception `ClassCastException` by the Java virtual machine performing a type cast when a source object type does not match a target type, with the difference that here we are checking whether a value is attached or not.

Although the guard instruction can be used to quickly make code "void-safe", its use should be strictly controlled and allowed only in cases when there no other way to make sure a particular variable is properly set. Otherwise, making code void-safe would mean replacing access on void target exceptions with assertion violations and defeat the whole idea of void safety.

3.5 Related work

Raymie Stata proposes to stick to the convention that class types are detachable by default in [68] with the rationale that existing programs are null-unsafe and allow for `null` to be assigned to a variable of any reference type. Early experiments [17] carried out by Manuel Fähndrich and Rustan Leino attested that it is better to use a non-null type as the default for a reference. They

found that this choice requires fewer annotations and mentioned it would make sense to allow alternative class-wide or module-wide defaults. The experience with Eiffel showed that it is better stick to just a single default with attached types, though the implementation allowed for selecting the default on different levels of hierarchy, including project, library and class. In the standard [12, 27] the class types are attached by default. The choice with a single default, where class types without any marks are attached, is confirmed by a group of authors reporting about their experience with Spec# in [6]. Some tools, e. g., *the Checker Framework* [72], use non-null type annotations by default, but allow for explicit setting of default annotations because their values are not part of the language standard.

In [17] Manuel Fähndrich and Rustan Leino also identified an issue with initialization of array types. They propose to instantiate arrays as having a partially initialized type. Then, as soon as all elements of the array are set, users will cast the temporary array type to the regular one that expects all elements to be set. This should be sound, but requires a new notion of "raw" types (see section 5.4) and a run-time type cast. In particular, it is not clear what a program should do if the type cast fails. The Checker Framework [72] follows the similar route except that the run-time cast is replaced with an annotation suppressing a warning when an array of nullable values is assigned to an array of a non-null values. Bertrand Mayer in [47] proposes to introduce a notion of self-initializing formal generic types that would make sure that every array instance can create a default element value whenever it is required to fill the array. This preserves soundness, but limits the use of *ARRAY* only to self-initializing types as actual generics. A better approach with using a new creation procedure *make_filled* instead of *make* is described by Bertrand Meyer, Alexander Kogtenkov and Emmanuel Stapf in [50]. Alexander Summers and Peter Müller in [70] suggest to use special marker methods to identify places where an array is completely constructed. This is similar to the already mentioned solution proposed in [17] and later extended by Manuel Fähndrich and Songtao Xia in the framework with delayed types [18], but forbids initialization of objects with circular references involving array types. The same run-time cast is used by Chris Male et al. in a tool [42] that verifies Java byte code. The authors clarify that such a cast essentially replaces `NullPointerException` with

`ClassCastException` thus impairing the effect of null safety. All the authors try to find a solution preserving the limitation of the Java virtual machine [40] that accepts just a total number of elements for every array dimension when creating a new array object. On the other hand, the solution with two index boundaries – one for maximum number of elements and one for the current number of elements – described at the end of section 3.2.2 seems to be far superior in terms of convenience and compile-time safety guarantees. The changes applied to classes *ARRAY* and *SPECIAL* are described in more details in [73].

A notion similar to stable attributes is supported by the Checker Framework [72] with an annotation `@MonotonicNonNull`.

Cyclic dependencies between static initializers are considered by Manuel Fähndrich and Rustan Leino as symptomatic of a design problem that could be caught by a static analyzer. Given that in general there are no special restrictions on the code in such initializers it remains unclear whether and how such an analyzer can be used as part of a language specification. They also propose to mark all methods (directly or indirectly) called by initializers in a special way to indicate that static fields might not be set yet. Alexander Summers and Peter Müller confirm that static initializers cannot be checked modularly and therefore insist that a static field should be of a possibly-null type. This is required for two reasons: to make sure no uninitialized object can reach a commitment point where all reachable objects have all non-null fields set and to avoid escaping uninitialized objects from one thread into another. In both cases static fields would serve as a carrier of incompletely initialized objects. Chris Male et al. argue that marking all static fields as possibly-null is impractical. Indeed, a lot of code uses `System.out` or similar static fields without checking them for nullity. The idea would be then to keep `NullPointerException`, but limit its used to static fields only. This is close to the decision about raising an exception in recursive feature calls to once functions. Unlike static fields and static initializers, a once function connects both a value and a piece of code to compute it, so it might be possible to find a better solution in the future.

In [51] Bertrand Meyer, Emmanuel Stapf and I collected some statistics about first experiments with adapting existing libraries to void safety rules. Like in [17] the statistics was limited to a few libraries. In this work I report migration results for a much

larger set of libraries. None of the existing publications discussed engineering decisions to do the migration. They are an important mechanism to move from void-unsafe to void-safe code for a large code base. In particular, one of them – void safety levels – seems to be a successful technology for software developers performing the migration.

4

A Type System for Void Safety

4.1 Attachment status

A type system usually associates with a class hierarchy that enables inheritance and polymorphism in object-oriented languages. But class-based type conformance is irrelevant for void safety. The only interesting property here is whether a particular expression is always attached at run-time to an object or it can be **Void**. In the first case the type is called **attached**, in the second case it is called **detachable**. The property of a type to be in one of these groups is called an **attachment status**. In Eiffel source code it is indicated by specifying an attachment mark **attached** or **detachable** ([12, 27]). From practical experience of developing void-safe systems, class types are considered as attached by default when they are declared without any attachment marks.

Expanded types are always attached regardless of specified attachment marks because the corresponding expressions always have associated values at run-time. Anchored types have attachment status of their anchors if there are no attachment marks. Otherwise, the status is explicitly specified by the marks. Dropping attachment marks from an anchor would break the rule of least surprise because the resulting type might become different from the type of the anchor.

It is allowed to assign an expression to a variable of an attached type only when the expression is of an attached type. This guarantees that the variable cannot become **Void** and thus cause a call on void target at a later time at run-time. Therefore, the relation on attached and detachable types is reflexive, antisymmetric and transitive with the inequality

$$\textit{Attached} < \textit{Detachable}$$

Given that class types of a system form a lattice, the elements *Attached* and *Detachable* can be seen as a bottom and as a top of the lattice *Attachment* reflecting attachment status, the product $ClassType \times Attachment$ forms a lattice too. The conformance relation is defined component-wise:

$$(C_1,\ A_1) \leqslant (C_2, A_2) \iff C_1 \leqslant C_2 \wedge (\mathit{is\_expanded}\ C_1 \vee A_1 \leqslant A_2) \quad (1)$$

According to the language specification [12, 27] the type of the special entity **Void** is *NONE* that conforms to any detachable reference type. Taking attachment marks into account, the type of **Void** should be **detachable** *NONE*. According to the definition (1) the type with a different attachment mark – **attached** *NONE* – should conform to any attached type. Even though such a type makes not much sense in the source code and there could be a validity rule that forbids such type declarations, it can appear indirectly via anchored or generic types as soon as the type declaration **detachable** *NONE* is allowed. The type **attached** *NONE* can be seen as the lowest bound for all attached reference types. Instantiation of objects of such a type is impossible.

As with class-based type conformance, inherited routine assertions remain usable only if attachment status of routine arguments and function result follows covariant rules, i. e., if an argument or a function are attached, their counterparts in the redeclaration are attached as well. For example,

```
search (n: STRING)
    require
        n.is_ascii
    do ... end
```

can be safely redeclared into

```
search (n: STRING)
    require else
        n.is_unicode
    do ... end
```

Class type	*self_initializing*	*is_attached*
Expanded	True	True
Attached Reference	False	True
Detachable Reference	True	False

Figure 4.1. Self-initializing and attached types.

but not into

```
search (n: detachable STRING)
    require else
        attached n and then n.is_unicode
    do ... end
```

In the last case, the inherited assertion *n.is_ascii* would cause access on void target if the contravariant redeclaration would be allowed and the redeclared version of the feature would be called with **Void**. Covariant redeclarations cause other issues but the corresponding solutions go beyond this work.

A variable of an expanded or of a detachable reference type is immediately usable right after its declaration. In the first case it is initialized by executing a standard creation procedure *default_create* without arguments. In the second case the variables are initialized with **Void**. These types are called **self-initializing**.

Depending on whether a class type is expanded or reference and has an (implicit or explicit) attached or detachable mark, it can be self-initializing or not, attached or detachable. This can be expressed with 2 functions *self_initializing* and *is_attached* as shown in figure 4.1.

4.2 General validity rules

Validity rules in a void-safe language serve several goals:

- make sure a variable of an attached type is indeed attached to an object at run-time;
- make sure the variable is initialized prior to its use;
- disallow calling a feature on an expression unless it is attached.

The first validity rule prevents from assigning **Void** to a variable of an attached type:

Validity rule 4.1. *A reattachment of an expression* e *to a variable* v *is valid if the type of* e *conforms to the type of* v.

The notion of reattachment stands either for an assignment instruction or an association between an actual and a formal argument of a routine. Even though the rule does not mention attachment status of involved types explicitly, the type conformance takes it into account according to the equation (1).

As discussed in chapter 6 it is not necessary that the declared types of the expression e and of the variable v coincide with the types used for checks. The actual type may depend on the context.

Some types are self-initializing, i. e., do not require explicit initialization of variables of these types:

Definition 4.1. *A type is self-initializing if and only if any of the following is true:*

- *it is an expanded class type;*
- *it is a detachable reference class type (i. e., a reference class type with a mark* ***detachable****);*
- *it is a self-initializing formal generic type.*

In the first case, a variable is initialized automatically with a default value of the corresponding expanded type. In particular, standard basic class types such as *BOOLEAN* and *NATURAL_64* are expanded and are initialized with associated default values (**False** and 0 (zero) in this case).

Definition 4.2. *A variable is properly set if and only if any of the following is true:*

- *it is of a self-initializing type;*
- *it has been assigned a value in all preceding blocks of code;*
- *it is an attribute of an attached type that is properly set at the end of every creation procedure.*

The preceding blocks of code are defined canonically ([55]). In the first case the variable is automatically initialized either to **Void** or by creating an expanded object, in the second and third cases it is initialized by an assignment instruction, by a creation instruction,

or as a result of associating an actual argument with a formal one. The requirements to initialize a variable prior to its use is guaranteed by:

Validity rule 4.2. *Access to a variable is valid if the variable is properly set.*

Finally, the third goal of a void-safe system is achieved by making sure all calls are performed on existing objects:

Validity rule 4.3. *A feature call with a target* e *is valid if* e *is of an attached type.*

Generally speaking, these rules do not guarantee absence of feature calls on void target because even though a current target of a call may be attached, some attribute of an attached type of this target may be uninitialized yet. This issue is addressed by additional validity rules in chapter 5.

4.3 Formal generics

Even though conformance rules for formal generics specified in [12, 27] are sound, they are too restrictive and rule out some safe cases. With attachment marks a formal generic type has three variants:

- *G* – a formal generic type without an attachment mark
- **attached** *G* – a formal generic type with an explicit **attached** mark
- **detachable** *G* – a formal generic type with an explicit **detachable** mark

Allowed attachment status of an actual generic type that can be substituted for a formal generic parameter depends on the formal generic constraints. In the void-unsafe world an unconstrained formal generic conforms to *ANY*. Introduction of void safety brings two possibilities for default constraints: **attached** *ANY* and **detachable** *ANY*. It turns out that formal generics are mostly used in container classes where a type of elements does not matter and could be attached or detachable. Given that an attached type of a constraint forbids specifying detachable actual generic parameter, it is more convenient to have the default **detachable** *ANY*. To be more specific, out of 245 generic classes of all standard library

classes included in *EiffelStudio 16.11* [16], 141 (58%) use unconstrained genericity with the default **detachable** *ANY* and 18 use constrained genericity with a detachable constraint type. In other words 65% of all generic parameters can be substituted with attached or detachable actual generic types and, therefore, the selection of **detachable** *ANY* as the default formal generic constraint is well-justified.

4.3.1 *Attachment property*

An attachment status of a class type – whether the type is attached or detachable – is known from its declaration. This is not true for formal generic types. Even more: for a formal generic type we never know if it can be detachable, i. e., whether the type is detachable for any substituted actual generic parameter. Indeed, substituting an expanded type as an actual parameter for a formal generic parameter *G*, the actual type of the formal generic type **detachable** *G* is still the original expanded type and is therefore attached. In other words, for any attachment mark, a formal generic parameter may be substituted so that the resulting type will be attached.

Fortunately, there are definitive cases when a formal generic type is attached. This might be important to know when the type is used as a source of a reattachment (validity rule 4.1) or as a target of a call (validity rule 4.3). Whether a formal generic is attached depends on two factors: what attachment mark is used and what are its constraints. For example, if there is an attachment type among constraints, any valid actual generic should be attached and, hence, the formal generic can be safely considered as attached too. Formally, the attachment status of a formal generic can be expressed with the following lemma that denotes constraints of a given formal generic as *Cs*:

B.26 p. 290

Lemma 4.1. *All substitution of a formal generic type with an actual one conforming to all its constraints Cs and taking into account its attachment mark m (if any) are attached if and only if one the constraints is expanded, or one of the constraints is attached and the formal generic attachment mark m is not Detachable, or the mark m is Attached:*

$$Cs \neq [] \implies (\forall A.\ (\forall C.\ C \in Cs \longrightarrow A \leqslant C) \longrightarrow is\_attached\ ((A, m))) \longleftrightarrow (\exists C.\ C \in Cs \wedge C = Expanded) \vee (\exists C.\ C \in Cs \wedge C = AttachedReference \wedge m \neq Detachable) \vee m = Attached$$

Proof. The proof follows from a lemma that considers just a single constraint and is proved by case analysis. □

The left-hand side specifies the requirement for a formal generic to be safely considered attached: for any actual generic type that conforms to all formal generic constraints the resulting type is attached. The premise that the list of constraining class types is not empty is always true because when there are no class types in the constraints, the default **detachable** *ANY* is used. The right-hand side provides the exact conditions when the formal generic can be safely treated as attached. Lemma 4.1 shows that the following definition of an attachment status of a formal generic type is sound:

Definition 4.3 (Formal generic attachment status)**.** *A formal generic type is attached if and only if any of the following is true:*

- *one of its constraints is an expanded type*
- *it does not have a mark* ***detachable*** *and one of its constraints is an attached type*
- *it has a mark* ***attached***

As explained above, in contrast to class types, there is no way to tell if a formal generic is detachable.

4.3.2 *Self-initialization status*

Self-initializing types enable automatic initialization of variables. In certain conditions a formal generic type can also be proved to be self-initializing:

Lemma 4.2. *All substitutions of a formal generic type with an actual one conforming to all its constraints Cs and taking into account its* *B.26 p. 290*

attachment mark m (if any) are self-initializing if and only if one the constraints is expanded or the mark m is Detachable:

$$Cs \neq [] \implies$$
$$(\forall A.\ (\forall C.\ C \in Cs \longrightarrow A \leqslant C) \longrightarrow \mathit{self\_initializing}\ ((A, m)))$$
$$\longleftrightarrow\ (\exists C.\ C \in Cs \wedge C = \mathit{Expanded}) \vee m = \mathit{Detachable}$$

As before, *Cs* stands for a list of given formal generic parameter constraints. The left-hand side specifies the requirement that for any actual generic type as soon as it conforms to all formal generic constraints, the resulting type is self-initializing. The right hand side specifies the necessary and sufficient conditions for this requirement. The premise that the list of constraints is not empty is guaranteed by the language rule to use a default class type constraint if none is given explicitly. This proves soundness of the following definition.

Definition 4.4 (Self-initializing formal generic)**.** *A formal generic type is self-initializing if and only if either of the following is true:*

- *it has a mark* ***detachable***
- *one of its constraints is an expanded type*

4.3.3 *Conformance*

An actual generic type should conform to all formal generic type constraints of the corresponding generic parameter. Therefore, in order to deduce conformance rules for formal generic types it is sufficient to analyze all possible variants of constraining types.

A formal generic with an expanded type constraint. According to the conformance rules, the only actual generic that can be used is the corresponding expanded type. Because attachment marks do not change attachment status of expanded types, the formal generic with or without any attachment mark conforms to the same formal generic with or without any attachment mark (here and in the following tables "+" indicates that a source type with the specified mark conforms to a target type with the specified mark, while "−" indicates that they do not conform):

Target mark	Source mark		
	–	**attached**	**detachable**
–	+	+	+
attached	+	+	+
detachable	+	+	+

A formal generic without an expanded type constraint but with an attached type constraint. The only actual generic type that can be substituted for this formal generic is attached. There are two cases. If the actual generic is an expanded type, there are no restrictions on attachment marks as discussed in the paragraph above. If the actual generic is an attached reference type, having no attachment mark or having the mark **attached** is equivalent. Therefore, all reattachments are safe except when the source has the mark **detachable** and the target does not:

Target mark	Source mark		
	–	**attached**	**detachable**
–	+	+	–
attached	+	+	–
detachable	+	+	+

A formal generic without an attached type constraint. The actual generic can be any suitable type. The conformance rules for the formal generic should preserve properties for the actual generics, namely, if the target type is not self-initializing, so has to be the source type, and if the target type is attached, so has to be the source type (figure 4.1). The following conformance table satisfies these properties:

Target mark	Source mark		
	–	**attached**	**detachable**
–	+	+	–
attached	–	+	–
detachable	+	+	+

If we denote the function that corresponds to the tables above as $\cdot \leqslant_G \cdot$, then it preserves conformance rules for actual generics:

B.26 p. 291 **Lemma 4.3.** $m_S \leqslant_G m_T \Longrightarrow (T, m_S) \leqslant (T, m_T)$

Moreover, this is the maximum function that preserves actual generic properties *self_initializing* and *is_attached*:

B.26 p. 291 **Lemma 4.4.**
$(\bigwedge T\ m_T\ m_S.\ f\ m_S\ m_T \wedge \mathit{is\_attached}\ ((T, m_T)) \wedge \neg\ \mathit{self\_initializing}\ ((T, m_T)) \Longrightarrow \mathit{is\_attached}\ ((T, m_S)) \wedge \neg\ \mathit{self\_initializing}\ ((T, m_S))) \Longrightarrow f\ m_S\ m_T \leqslant (m_S \leqslant_G m_T)$

Proof. By case analysis. □

This leads to the following conformance rules for formal generics.

Definition 4.5. *A formal generic type* G *conforms to a formal generic type* F *if their attachment-mark-free forms are the same and any of the following is true:*

- G *and* F *are the same type*
- G *is constrained by an expanded type*
- G *is constrained by an attached type and* G *has no* ***detachable*** *mark*
- G *has an* ***attached*** *mark*
- F *has a* ***detachable*** *mark*

The similar definition can be used for specifying conformance rules between formal generic types and their constraints if the formal generic type F is replaced with a constraint of the formal generic type G.

4.4 Related work

The idea of distinguishing between different types of expressions that always return an object and that may return **null** is not new. In the early days of Java in 1995 Raymie Stata proposed ([68]) to use a notation `T ?` in the spirit of the C language notation `T *` to denote that the value may be **null**. In all other cases the value could not be `null`. In other words, a convention adopted in functional languages should be used: a special type `Maybe [T]` would indicate that a value might be absent. Because all existing code was implicitly relying on the default that permitted to use `null` instead of an object value, there was also a proposal to introduce `T !` for types that guarantee presence of object values.

Certain experiments aimed to perform static analysis of Java and to report software defects at early stages in the development cycle led to introduction of the *Extended Static Checked for Java* (ESC/Java). A group of its authors reported ([19]) that the tool relies on `/*@non_null*/` annotations for declarations of non-null types. The checker did not guarantee absence of errors but rather warned a user about potential issues. Authors of the Checkers Framework [72] mention that most static analysis tools for Java use similar annotations: `@Nullable` and `@NonNull`, though from different packages depending on the associated tool.

Manuel Fähndrich and Rustan Leino attempted to construct a sound null-safe type system in [17]. They used C# mechanism to annotate types. Unlike ESC/Java that had to use comments for describing properties outside of the language standard, C# has a built-in extension mechanism. The authors used C# attributes `[NotNull]` and `[MayBeNull]` for non-null and maybe-null types respectively. They also briefly reviewed introduction of generic types to C# and Java and came to the conclusion that it might be problematic to instantiate formal generics with actual types of arbitrary attachment status in the cases when null annotations are allowed for formal generics. My work gives a thorough analysis of all combinations of attachment marks for both formal and actual generic parameters, including expanded types, and clarifies the conformance rules. It also establishes conditions when formal generic types are considered attached and when – self-initializing. An expression of a formal generic type is permitted to be used as a target of a call only if it is known to be attached. When the

type system fails to capture this property, the code should rely on other means, discussed in subsequent chapters. And when a formal generic type is not self-initializing, a variable of this type should be set explicitly (section 4.2).

Erik Meijer and Wolfram Schulte in [45] explore a more advanced type system to work with SQL and XML in an object-oriented language directly. Because SQL features nullable types, there is a need to reflect this in the type system together with other domain-specific type facets like streams. In this type system non-null, non-stream and possibly-null values have types with the following subtyping relation: `T! <: T <: T?`. For reference types, the special entity `null` of type `0?` is still allowed as a valid value in all cases: `0? <: T`. Therefore, reference types without any marks effectively behave like maybe-null types. Because the rules are expected to model SQL, it is allowed to make a call on a target of a maybe-null type. The result of such a call is also of a maybe-null type regardless of the message type. This is different from validity rule 4.3 where only calls on attached targets are permitted.

The term **attached type** was introduced by Bertrand Meyer in [47] where the notations ! *T* and ? *T* were used for attached and detachable types respectively. The work covered most issues of void-safe programming, including required validity rules and certified attachment patterns (discussed in chapter 6). The rule for self-initializing formal generics was also given, but it was different from the definition I use in this work. Particularly, the self-initialization status of a formal generic parameter was explicitly specified in the class declaration. Then clients of the class were obliged to provide a self-initializing type as the corresponding actual generic parameter. The rules proposed in my work do not put any restrictions on this aspect of actual generics. Still, formal generics are considered as self-initializing in certain cases. The keyword-based syntax that replaced question and exclamation marks in type declarations in Eiffel was mentioned by Bertrand Meyer, Alexander Kogtenkov and Emmanuel Stapf in [50].

A completely different approach to describe whether a variable is attached or not is used by Xin Qi and Andrew C. Myers in [62]. Instead of talking about attachment status, they proposed to specify whether an object field is initialized or not and claimed that in that case the use of `null` can become rare and can be replaced with values of types `Option` and `Maybe`. All the examples they use to demonstrate the approach do not rely on run-time

tests for `null`. However, the experiments show that in some cases `null` is a legitimate value. Thorough review of the accompanying soundness proof of the type system in [61] reveals that `null` references are not taken into account in the proofs. Thus, there is a significant dissonance between the theory and real programs. The work covers a very important aspect of object initialization, but it seems that there are no void safety guarantees if use of `null` is allowed.

Finally, no additional type information is required for a static null-pointer analysis like the one described by Fausto Spoto in [67]. Whether a particular expression can produce `null` at run-time is computed using an abstract interpretation technique. Then calls on expressions that can produce `null` are flagged by the tool as unsafe. This permits to analyze any program compiled into Java byte code, but requires whole system analysis and is not suitable for a void safe language specification.

4.5 Conclusion

Almost all languages or their dialects that allow for **Void** references and support some sort of void safety distinguish between attached and detachable types (probably, using different names, such as “non-null” and “possibly-null”). However, publications usually focus on regular class types and do not specify precise rules for formal generics. Similarly, the rules for formal generics specified in the language standard [12, 27] are sound, but are too pessimistic. Another aspect of developing void-safe programs – support for gradual migration from existing void-unsafe to void-safe code – is also not covered in publications. The main contributions of my work include:

Clarification of formal generic properties Precise conditions when a formal generic type has a specific void-safety-related property are defined for:

- Attachment status – tells if a formal generic is attached.
- Self-initialization – tells if the type is self-initializing.
- Conformance – tells if two formal generic types conform to each other or if a formal generic type conforms to the corresponding constraint.

Decription of void safety levels An engineering approach to facilitate transition from void-unsafe to void-safe code is described with the details how to do the gradual update.

Migration statistics Statistics summarizing changes required to update existing code is collected for a large open source code base with more than a million lines of code.

5 The Object Initialization Issue

General validity rules described in section 4.2 are sufficient for making a program void-safe except for the rare case when **Current** object escapes to the rest of the program before all attributes are properly set. Indeed, according to the rules, the attributes have to be properly set at the end of the corresponding creation procedure. However, it is quite possible that **Current** is passed as an argument of a call to some feature that incorrectly accesses an attribute that is not set yet. This may lead to accessing an uninitialized field with unexpected value (**Void** for reference types) as if it was initialized and cause access on void target. Additional rules applied for creation procedures address this issue.

5.1 Attribute access safety

5.1.1 *Motivating example*

Let's consider an application that uses a GUI library and has a class *PARENT_DIALOG* that allows creating and showing some dialog to a user (figure 5.1).

The dialog is very simple. In addition to the standard controls such as `OK` and `Cancel` buttons it adds a reset button to put all user controls to some default state. This button is referred in the code via an attribute *reset_button*. The feature *create_implementation* is inherited from the parent class *DIALOG* and implements the underlying window toolkit functionality to initialize the dialog.

Suppose that a child dialog adds a text area that a user can fill in. There is also some default text that is put to this area. A user can change the text area, and when the button `OK` is pressed, the text is saved using an agent *set_text*. But if a user decides to start over,

```
class PARENT_DIALOG inherit DIALOG create make
feature {NONE} -- Creation
    make
        do
            create reset_button
            create_implementation (Current)
            reset_button.select_actions.extend (agent
                on_reset)
        end
feature -- Access
    reset_button: BUTTON
feature {NONE} -- Actions
    on_reset
        do
        end
end
```

Figure 5.1. A parent dialog class.

(s)he can press the button `Reset` available from the parent dialog. In order to avoid code duplication, the procedure that resets all the data is also called in the creation procedure.

Let's look at how the child class creation procedure is executed. Firstly, it calls the parent's creation procedure to initialize attributes declared there. Then it creates a text area, records the default text, calls the feature to set all widgets to the default values and registers an action to be performed when a user presses a button `OK`. All the conditions specified in section 4.2 are fulfilled. However, calling a parent creation procedure causes a call to *create_implementation* that in turn calls *on_create*. The latter is supposed to be redefined in a descendant if some actions need to be performed right after the dialog is created. The dynamic type of the current object is *CHILD_A*, so the version from the class *CHILD_A* is called and it executes *text.put (default_text)*. But at this point the field *text* is not set yet that causes access on void target.

```
class CHILD_A inherit
    PARENT_DIALOG
        rename make as make_parent
        redefine on_create, on_reset
        end
create
    make
feature {NONE} -- Creation
    make (original_text: STRING; set_text: PROCEDURE
          [STRING])
        require
            not original_text.is_empty
        do
            make_parent
            create text
            default_text := original_text
            ok_actions.extend (agent do set_text (text.item) end)
        end
feature -- Access: user interface
    text: TEXT_AREA
feature -- Access: default values
    default_text: STRING
feature {NONE} -- Actions
    on_create, on_reset
        do
            text.put (default_text)
        end
end
```

Figure 5.2. A child dialog class (A).

```
class CHILD_A
...
    make (original_text: STRING; set_text: PROCEDURE
          [STRING])
        require
            not original_text.is_empty
        do
            create text
            default_text := original_text
            make_parent
            ok_actions.extend (agent do set_text (text.item) end)
        end
...
end
```

Figure 5.3. A child dialog class (A) – corrected version.

5.1.2 *Solution*

The issue in the example from section 5.1.1 arises because **Current** object is passed before all attributes of this object are properly set (definition 4.2). The simplest rule would be to forbid using **Current** until all attributes are properly set:

Validity rule 5.1 (Creation procedure; see validity rule 5.2 for a weaker version)**.** *An expression **Current** is valid in a creation procedure or in an unqualified feature it (directly or indirectly) calls if all attributes of the current class are properly set at the execution point of the expression.*

The clarification about calling unqualified features is important to make sure that access on void target does not happen in a called feature that accesses **Current**. In particular, this clarification ensures that for *DIALOG_A* not only the creation procedure *make* is checked, but also the parent's creation procedure *make_parent* that passes **Current** to the window toolkit.

The corrected version of the class (only changed code) is shown in figure 5.3.

The core difference is the order of initialization. The attributes of the child class are set before calling a parent's creation procedure.

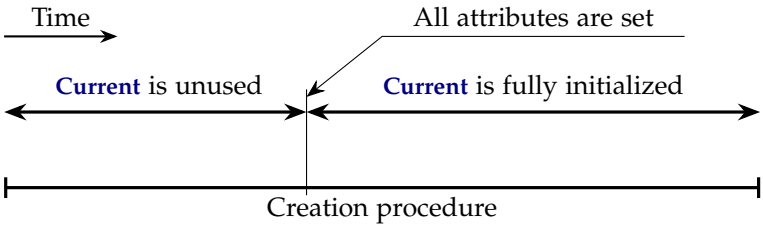

Figure 5.4. Timeline during object initialization according to validity rule 5.1.

This ensures that at the time **Current** is used, all attributes are properly set and are safe to access.

The rule introduced by validity rule 5.1 could be depicted with a time line that goes from left to right (figure 5.4). During execution, object fields are initialized with proper values, but until they all are properly set, the reference to the current object cannot be used. And when they all are set, **Current** is fully initialized and can be freely used alongside with any other object references without any special restriction.

The main benefit in this scheme is that all used objects are completely initialized throughout program execution. So, all features can be freely called regardless of the fact that an object is being created: the inconsistent state when not all attributes are properly set is simply kept inaccessible due to validity rule 5.1.

The picture in figure 5.4 should not be interpreted literally. Due to branching instructions such as conditionals, loops, etc., all attributes could be set in one branch so that **Current** can be safely used in this branch. However, it is possible that not all attributes are set in another branch. Therefore, for the whole expression both conditions "there are unset attributes" and "**Current** is used" could be true at the same time even if the expression passes the validity check.

The validity rules in this chapter are formalized using a simplified version of an Eiffel-like abstract syntax shown in figure 5.5. It assumes that the source code is parsed and the compiler distinguishes between local variables and attribute variables. Even though the language does not provide direct means to raise exceptions, this is an important concept and is reflected in the syntax productions. This allows for modeling some missing lan-

B.25 p. 280

```
expr =
    Value value                          | – Value (constant)
    Current                              | – Current object
    Local name                           | – Local variable
    Attribute name                       | – Attribute variable
    expr ;; expr                         | – Sequence
    name :=L expr                        | – Assignment to local
    name :=A expr                        | – Assignment to attribute
    create {type} name (expr list)       | – Creation expression
    expr.name (expr list)                | – Feature call
    if expr then expr else expr end      | – Conditional
    until expr loop expr end             | – Loop
    attached type expr as name           | – Object test
    Exception                              – Exception
```

Figure 5.5. Simplified abstract expression syntax.

guage constructs. For example, the mandatory check instruction (guard) **check** ... **then** ... **end** could be translated into the abstract form *if . . . then . . . else Exception end*.

In order to see if all attributes are properly set at a particular execution point, we need to know how every expression affects the set of properly-set attributes. As soon as all attributes are set, it is safe to use **Current**. Because comparing the set of currently set attributes to the set of all attributes of the current class would require passing the set of class attributes all the time, it is easier to track if there are any unset attributes left and to check if this set is empty. As soon as the set of unattached attributes is empty, all attributes are properly set. The computation is done by the transfer function $\cdot \gg \cdot$. The equations for the function are specified in figure 5.6 (here _ is a placeholder for any expression different from all explicitly specified ones and $\gg$ has higher precedence than any set-theoretic operations like set difference or union).

The transfer function takes 2 arguments – an expression and a set of attributes V that may be unattached before the expression – and returns a set of attributes that may be unattached after

$$
\begin{aligned}
V \gg e_1 \;;;\; e_2 &= V \gg e_1 \gg e_2 \\
V \gg n :=_{\mathrm{L}} e &= V \gg e \\
V \gg n :=_{\mathrm{A}} e &= V \gg e - \{n\} \\
V \gg \mathit{create}\ \{t\}\ .\ n\ (es) &= V \gg es \\
V \gg e\ .\ n\ (es) &= V \gg e \gg es \\
V \gg \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} &= V \gg c \gg e_1 \cup V \gg c \gg e_2 \\
V \gg \mathit{until}\ c\ \mathit{loop}\ b\ \mathit{end} &= V \gg c \\
V \gg \mathit{attached}\ t\ e\ \mathit{as}\ n &= V \gg e \\
V \gg \mathit{Exception} &= \varnothing \\
V \gg \_ &= V \\
V \gg [] &= V \\
V \gg (e \cdot es) &= V \gg e \gg es
\end{aligned}
$$

B.25.1 p. 280

Figure 5.6. A function to compute a set of unattached attributes.

the expression. At the beginning of a creation procedure the set of unattached attributes is a set of all current class attributes of attached reference types (attributes of expanded and detachable types are initialized automatically).

If the expression is a sequence or an argument list, the set of unset attributes for a subsequent expression is computed starting from the set computed for the first expression. For assignment to an attribute, firstly the set is computed for the source expression, then the attribute name is removed from the set of unset attributes, because after the assignment it is set. Removal of the attribute from the set is sound because the validity of the assignment is checked according to conformance rules discussed in section 4.2. Because the attribute is of an attached type, the source expression is also attached. Therefore, after the assignment the attribute is attached.

For a creation expression, the set is computed using associated list of actual arguments starting from the initial set. For a qualified call the rule is similar except that the set computed for the target of the call is used as a starting one.

For a conditional expression, the computation is done for both branches like for a sequence and then their union is used as a

result. The rationale is that even though every branch can set some attributes, if these attributes are not set in the other branch, the attributes should not be considered set because it is unknown which branch is going to be taken at execution time. For a loop only an exit condition is taken into account and a loop body is completely ignored. It is possible that the loop body is not going to be executed at all, so whatever attributes it sets cannot be used to reduce the set of unattached attributes.

For an exception the set of unattached attributes is empty because execution never goes after this point, so any assumptions are valid. Using an empty set signals to the compiler that no more attributes have to be initialized and effectively triggers "design mode" discussed in section 6.2.4.

For the rest of expressions, if there are subexpressions, the function returns sets for these subexpressions, otherwise it returns the initial set.

The function is monotone, i. e., the more attributes are set before an expression, the more are set after the expression. This is true thanks to the rule that attached attributes, once initialized, cannot be uninitialized back.

B.25.1 p. 281 **Lemma 5.1** (Monotonicity of $\gg$). $A \subseteq B \Longrightarrow A \gg e \subseteq B \gg e$

A predicate $V \vdash e \surd_{\mathrm{c}}{}' $ tells if an expression e satisfies validity rule 5.1 in the context with unset attributes V. (The subscript letter c indicates that the predicate applies to creation procedures only, not to arbitrary features, and the apostrophe $'$ is used to distinguish this set of rules from a more sophisticated one discussed in section 5.2.2.3.) Because the set of unattached attributes changes from one expression to another, it uses the transfer function for a set of unattached attributes to update the context of nested expressions. The rules of the predicate are shown in figure 5.7.

Access to a constant and to a local variable (rules Value and Local) is valid in all cases. Access to the special entity **Current** that denotes a current object is valid only if all attributes are initialized (Current). Access to an attribute is valid only when it is not in the set of unattached attributes (Attr).

A sequence of two expressions is valid if and only if the first expression is valid in the initial context and the second expression is valid in the context obtained by replacing a set of unattached

$$\frac{}{V \vdash \mathit{Value}\ v\ \surd_c\,'}\ \textsc{Value} \qquad \frac{V = \varnothing}{V \vdash \mathit{Current}\ \surd_c\,'}\ \textsc{Current}$$

B.25.2 p. 282

$$\frac{}{V \vdash \mathit{Local}\ n\ \surd_c\,'}\ \textsc{Local} \qquad \frac{n \notin V}{V \vdash \mathit{Attribute}\ n\ \surd_c\,'}\ \textsc{Attr}$$

$$\frac{V \vdash e\ \surd_c\,'}{V \vdash n :=_L e\ \surd_c\,'}\ \textsc{AssignLocal} \qquad \frac{V \vdash e\ \surd_c\,'}{V \vdash n :=_A e\ \surd_c\,'}\ \textsc{AssignAttr}$$

$$\frac{V \vdash es\ [\surd_c]\,'}{V \vdash \mathit{create}\ \{t\}\ .\ n\ (es)\ \surd_c\,'}\ \textsc{Create}$$

$$\frac{V \vdash e\ \surd_c\,' \wedge V \gg e \vdash es\ [\surd_c]\,'}{V \vdash e\ .\ n\ (es)\ \surd_c\,'}\ \textsc{Call}$$

$$\frac{V \vdash e_1\ \surd_c\,' \wedge V \gg e_1 \vdash e_2\ \surd_c\,'}{V \vdash e_1\ ;;\ e_2\ \surd_c\,'}\ \textsc{Seq}$$

$$\frac{V \vdash e\ \surd_c\,' \wedge V \gg e \vdash b\ \surd_c\,'}{V \vdash \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}\ \surd_c\,'}\ \textsc{Loop}$$

$$\frac{V \vdash c\ \surd_c\,' \wedge V \gg c \vdash e_1\ \surd_c\,' \wedge V \gg c \vdash e_2\ \surd_c\,'}{V \vdash \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end}\ \surd_c\,'}\ \textsc{If}$$

$$\frac{V \vdash e\ \surd_c\,'}{V \vdash \mathit{attached}\ t\ e\ \mathit{as}\ n\ \surd_c\,'}\ \textsc{Test} \qquad \frac{}{V \vdash \mathit{Exception}\ \surd_c\,'}\ \textsc{Exception}$$

$$\frac{}{V \vdash []\ [\surd_c]\,'}\ \textsc{Arg}_{\mathtt{Nil}} \qquad \frac{V \vdash e\ \surd_c\,' \wedge V \gg e \vdash es\ [\surd_c]\,'}{V \vdash e \cdot es\ [\surd_c]\,'}\ \textsc{Arg}_{\mathtt{Cons}}$$

Figure 5.7. A predicate that tells if an expression is valid in a creation procedure of a system S in the context with a set of unattached attributes V.

```
class COUNTER_AREA inherit TEXT_AREA
create
    make
feature {NONE} -- Creation
    make (other: TEXT_AREA)
        do
            default_create
            other.change_actions.extend (agent do put
                (other.item.count) end)
        end
end
```

Figure 5.8. A counter widget.

attributes with a new set computed for the first expression (SEQ). The similar technique applies to other expressions with several subexpressions: CREATE, CALL, IF, LOOP. ARG_{Cons}. There is no context change if an expression or an instruction has only one subexpression: ASSIGNLOCAL, ASSIGNATTR, TEST. Finally, the context is irrelevant for exceptions (EXCEPTION) and an empty list of arguments (ARG_{Nil}).

The next section demonstrates that the required initialization order sometimes cannot be achieved straightaway. But it is feasible if dynamic binding is exploited.

5.1.3 *Initialization order in presence of inheritance*

Suppose, the widget *text* of type *TEXT_AREA* from the example of section 5.1.1 may have a limit on the number of entered characters. The limit is enforced by an application domain, and it is important to show to a user how many characters the text area contains. Assuming this functionality is common, we want to add a new widget type *COUNTER_AREA* that can be associated with a text area and show a current number of entered characters (figure 5.8).

```
class CHILD_B inherit CHILD_A redefine make end
create
    make
feature {NONE} -- Creation
    make (original_text: STRING; set_text: PROCEDURE
        [STRING])
        require
            not original_text.is_empty
        do
                -- Too early: 'text' from CHILD_A is not
                    created yet.
            create counter.make (text)
            make (original_text, set_text)
                -- Too late: 'counter' should have been set
                -- before calling 'create_implementation'
                    by 'make'.
            create counter.make (text)
        end
feature -- Access: user interface
    counter: COUNTER_AREA
end
```

Figure 5.9. A child dialog class (B).

5.1.3.1 *Coding guidelines*

Unfortunately, it is impossible to create the new widget before all other widgets like it was done for *DIALOG_A* because it requires the text area widget as an argument of the creation procedure, but creating the widget after calling parent's creation procedure is too late, validity rule 5.1 would be violated (figure 5.9).

In order to fulfill the conditions specified in validity rule 5.1, the code has to follow certain coding guidelines:

1. Create all required objects and set all class attributes.
2. Call features that need **Current**.
3. Operate in the context were all objects are completely initialized.

One particular source of accesses to **Current** is indirectly via unqualified agents. An unqualified agent of the form **agent** *foo*

(*arguments*) or an inline agent **agent** (*arguments*) **do end** implicitly passes **Current** to create the agent object. This happens because such an agent can access attributes of the current class. In case of mutual dependencies and if the agent is attached to an attribute it might be problematic to satisfy validity rule 5.1. Given that agents are frequently used for callbacks, means of a subscriber pattern and alike, it is more flexible to use a collection of agent objects instead of a single attribute of an attached agent type. This way it is easier to implement subscribe/unsubscribe interface, providing for more agile interface to clients. Moreover, in terms of void safety it becomes sufficient to create empty containers at object creation time. The required agent objects can be registered later at any suitable moment.

5.1.3.2 *GUI library pattern*

Applying specified coding principles to the design of a GUI library, every window or widget is initialized and starts its operation by implementing 3 features:

- *create_interface_objects* – Creates all nested objects and initializes all attributes.
- *create_implementation* – Creates a bridge between the interface classes and underlying window toolkit, this is the first time **Current** is used.
- *initialize* – Initializes previously created widgets, e. g., by registering event handlers.

The top-level GUI library class implements a creation procedure *default_create* with exactly the following definition:

```
default_create
    do
        create_interface_objects
        create_implementation
        initialize
    end
```

Then the corrected modular version of the dialog classes would look like in figure 5.10.

```
class CHILD_A inherit PARENT_DIALOG
    redefine create_interface_objects, initialize, on_reset end
create make feature {NONE} -- Creation
    make (original_text: STRING; set_text: PROCEDURE
          [STRING])
        require not original_text.is_empty
        do
            default_text := original_text
            default_create
        end
    create_interface_objects
        do
            Precursor
            create text
        end
    initialize
        do
            Precursor
            ok_actions.extend (agent do set_text (text.item) end)
        end
    ...
end

class CHILD_B inherit CHILD_A
    redefine create_interface_objects end
create make feature {NONE} -- Creation
    create_interface_objects
        do
            Precursor
            create counter.make (text)
        end
feature -- Access: user interface
    counter: COUNTER_AREA
end
```

Figure 5.10. Child dialog classes (A) and (B).

5.1.4 *Modification of existing structures*

Convenience of the ability to invoke regular procedures inside a creation procedure can be demonstrated with a mediator pattern ([20]). The purpose of the pattern is to decouple objects so that they do not know about each other, but still can communicate using an intermediate object, called **mediator**. The concrete setup can be different and the set of participating objects can be fixed or can change dynamically. In the first case creation of objects structure can be fixed and all objects can be recursively created in a single creation procedure. In the second case this is scenario is impossible. Some objects can still be created at the beginning, but the rest will be added as program runs.

Because all interacting objects should have access to the mediator object, its type is attached. The number of communicating objects is unknown and references to them are stored in a container that is extended with new items as required. In general, concrete types of the communicating objects are unknown to the mediator. Therefore, creation of these objects cannot be specified in the mediator class. It is done in some client's code.

On the other hand, communicating objects know about the mediator and can register in it according to their role. The registration can be conveniently done inside their creation procedures, so that clients do not need to clutter the code with a call to a special feature *register* every time they create a new communicating object:

```
create communicating_object.make (mediator)
communicating_object.register (mediator)
```

For clarity and to avoid errors with setting up object communication, it would be much cleaner to write just

```
create communicating_object.make (mediator)
```

To be more specific, consider a simplified example of a chat room adapted from [44]. The room of type *ROOM* may have a number of participants that can register in the chat by calling a feature *join* (see figure 5.11). Participants are abstract communicating agents of type *USER*. Their creation procedure records a reference to the chat room for future use and registers the user in the room.

When feature *join* is called, all attributes of the class *USER* or the corresponding descendant class should be set. Feature *join* of the class *ROOM* adds a new participant to the list of the registered

```
class ROOM create make feature
    users: ARRAYED_LIST [USER]
    make
        do
            create users.make (0)
        end
    join (a: USER)
        do
            users.extend (a)
        end
    send (s: STRING)
        do
            across users as u loop
                u.item.receive (s)
            end
        end
end
```

```
class USER create make
    feature
    room: ROOM
    make (r: ROOM)
        do
            room := r
            r.join (Current)
        end
    send (s: STRING)
        do
            room.send (s)
        end
    receive (s: STRING)
        do
            io.put_string (s)
            io.put_new_line
        end
end
```

Figure 5.11. An example of a mediator pattern.

users. Because this is done only after the user object is completely initialized, there are no issues with leaking an uninitialized object. As discussed in section 5.4 most existing type-system-based proposals fail to detect that the code is safe. They report it as erroneous instead. The difficulty for those methods is in the fact that the new object registers in the existing one before its creation procedure finishes. Approaches based on type declarations fail to capture that at some point the new object is completely initialized and can be safely used in the context that does not expect uninitialized objects. Only with masked types ([62]) it is possible to make the code compile at the expense of sophisticated type annotations.

5.1.5 *Implementation*

Evaluation of the transfer function as well as of the validity predicate is done in one pass over an abstract syntax tree. Because sets of currently attached attributes computed by the transfer

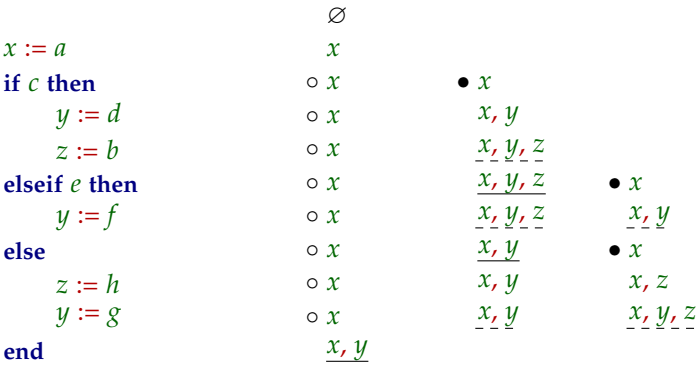

Figure 5.12. An example of using a stack to compute currently set attributes.

function are used immediately, there is no need to store them. This allows for a very efficient implementation with low memory footprint and little computational overhead. The sets are represented by bit vectors with a single bit per an attribute indicating whether the attribute is set or not. When a total number of attributes in a class is below a certain value, the bit vectors are mapped to the built-in integer numbers with fast bitwise operations. This is the case almost all the time, because there are not that many classes with more than 60 attributes. Usually the sets fit into *NATURAL_64*.

For code without branching instructions it is sufficient to have just one value representing a set of properly-set attributes. When there are branching instructions, the sets for different branches have to be computed separately. An example how this is done is shown in figure 5.12.

At the beginning of a conditional instruction the current set of attached attributes is marked as an outer for the instruction ($\circ\ x$). A new copy of this value is used every time a new branch is an-

alyzed. The copy is marked in the picture with a bullet (• x). Inner sequential blocks are processed as usual, updaing a set of attached attributes as required. When a block is finished, the value computed so far is merged with values of other branches. If this is the first branch, its value is just kept unchanged ($\underline{x, y, z}$). If this is the second or any subsequent branch the value is merged with the one computed for previous branches. The result replaces the current inner value ($\underline{x, y}$). Because there is just a single "merged" value and a single value for a branch being processed, at most two values for inner scopes have to be stored in addition to the outer scope value.

A natural choice to support this is a stack. In figure 5.12 the top of the stack corresponds to the right-most non-empty column. Nested branching instructions treat the top element of the stack as an outer scope and proceed as described above. After completing analysis of the last branch of a conditional instruction, the "merged" value is used as a result for the instruction as a whole and is propagated to the higher level replacing the current value of the outer scope (the last line in the example).

In addition to instructions and expressions that are always part of a program, Eiffel has instructions and expressions that are executed optionally, depending on some external settings. This includes assertion monitoring and debug instructions. In order to make sure void safety is not compromised by these instructions, they are also subject to validity checks. However, for such optional instructions a scheme similar to branching instructions is used. A current set of attached attributes is duplicated on the stack and the analysis is done using this new stack element. But unlike branching instructions, on completion, the value computed for an optional instruction is simply discarded and the analysis proceeds with the value as if the optional instruction had no effect. This ensures void safety guarantees do not depend on whether assertion monitoring and debug instructions are on or off.

Memory required to preform the analysis can be estimated if the number of nested branching and optional instructions is known. As we have seen, a branching instruction uses at most two additional slots on the stack. A nested optional instruction uses one additional slot. So, if there are N_b nested branching instructions and N_o nested optional instructions, the total number of required slots is limited by $1 + N_o + 2N_b$. Provided that creation proce-

dures do not use many branching instructions, the real number remains very small and the whole initialization analysis can be performed using preallocated memory with fast bitwise operations very quickly.

5.1.6 *Practical experience*

Even though validity rule 5.1 seems to be pretty restrictive, 4254 classes of public libraries (all libraries listed in table A.4) have been successfully converted (some – with modifications, some – without) to complete void safety relying on this rule. This comprises 822487 lines of code and 3194 explicit creation procedures (some classes do not declare any explicit creation procedure and use a default one instead). 59% of these creation procedures (1894 in absolute numbers) perform regular direct or indirect qualified calls and might be in danger if not all attributes were set before **Current** was used. However, it was possible to refactor all the classes to satisfy the rule.

This refactoring involved almost 44% of all classes (figure 5.13) that indicates that the original code did not follow the ordering of initialization required by validity rule 5.1. This number is twice smaller than the total number of classes that needed to be changed to add attachment marks to the source code. In terms of affected lines of code the change is even smaller – only 4.66% (figure 5.14), where about 2% corresponds to inserted lines, 1.6% – to deleted lines and 1.1% to modified lines (table 3.1). This is less than one third of code changes caused by applying conformance rules. The amount of changes was smaller because only creation procedures were involved.

On average, the number of inserted lines is slightly higher than the number of deleted lines due to the refactoring that addresses application of validity rule 5.1 to classes that reuse initialization code from parents (libraries *WEL* and *Vision2*) as explained in section 5.1.3.2.

There seems to be no strong correlation between library size and required changes. However, relatively more changes were needed in the libraries that work with text (*diff*, *parse*, *preferences*, *i18n*, *lex*) as well as in the database library *store* because the original versions did not care about proper order of attribute initialization and allowed for using **Current** before all attributes have been set.

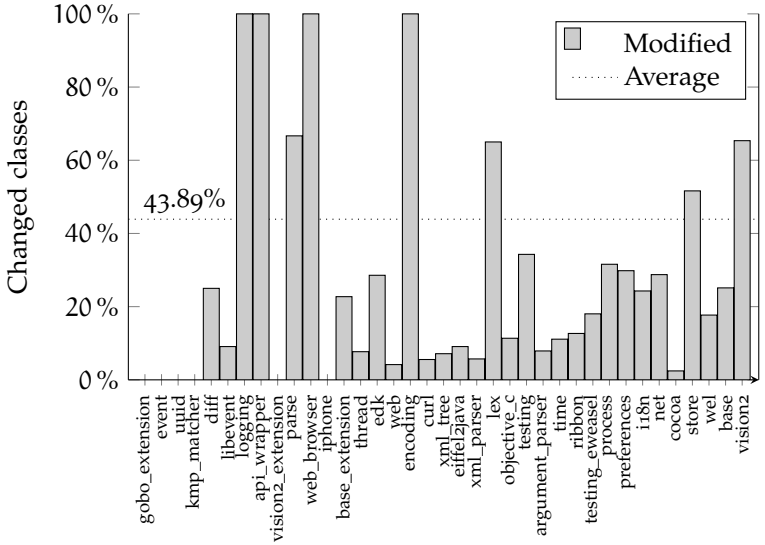

Figure 5.13. Changes in transitionally void-safe public libraries to make them completely void-safe ordered by library size.

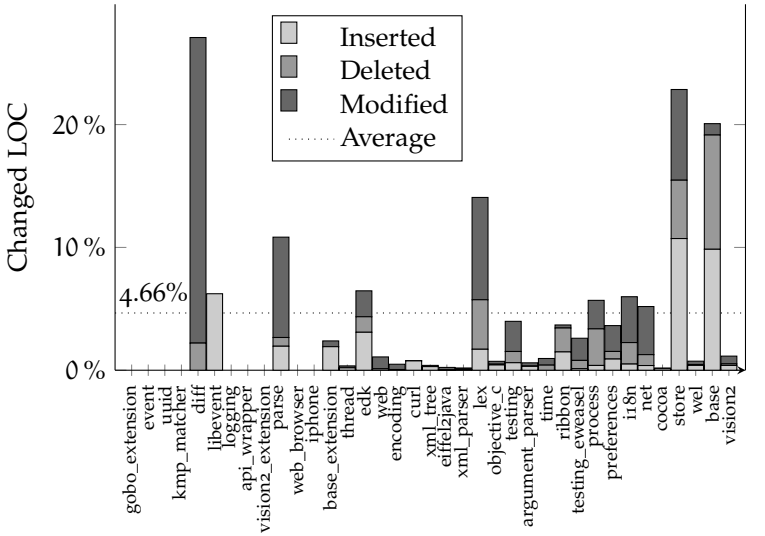

Figure 5.14. Changes in transitionally void-safe public libraries to make them completely void-safe ordered by library size.

The rule was too restrictive for two more advanced libraries though in most cases the source code could have been refactored to meet the requirements. To be more specific, only 77 creation procedures (less than 2%) of all libraries code took advantage of a more sophisticated and therefore more permissive rule discussed in the next section. Still, breaking some design guidelines, it should be possible to refactor all code by exploiting multiple inheritance and discarding separation of concern principle, to meet the requirements of validity rule 5.1. This indicates that in practice the code does not use all potential possibilities, and even simple rules can be practically useful to achieve safety.

5.1.7 *Conclusion*

Validity rule 5.1 introduced in this section has the following benefits:

No annotations. It does not require any additional annotations for both method signatures and types of arguments except for attachment marks used in type declarations.

Modularity. Type checking is modular, i. e., no additional information is required when analyzing creation procedures of a class. Because Eiffel does not have dedicated notion of a constructor and any procedure can be used as a creation procedure, access to parent code is required to perform the checks of inherited creation procedures or features that are called by the creation procedures with unqualified feature calls.

Simplicity. The analysis requires only tracking for attributes that are not properly set and for use of **Current**.

High coverage. In practice, the rule is suitable for 98% of code. Much higher acceptance rate can be achieved if some code patterns can be ignored and a group of classes is allowed to be replaced with a single class with more responsibilities. Alternatively, the responsibilities could be moved to parent classes if a language supports multiple inheritance. Then a common descendant would inherit from all of these classes and would be used instead of them when required. This may go against some design patterns though ([20]).

The main drawbacks of the rule are:

Certain coding pattern. The code has to follow certain coding guidelines in terms of initialization order. This enforces some structure on the code. Fortunately, it is easy to follow even in presence of (multiple) inheritance with the requirement of code reuse. On the other hand, the requirement to initialize all attributes in a creation procedure seems to be unavoidable. Therefore, some kind of code discipline would be required anyway, and this pitfall is not necessary specific to the proposed rule.

Insufficient flexibility. The rule does not allow for creating objects with mutual references of attached types. Indeed, when one object is created, it cannot initialize a reference to another object before that other object is created. Unless some stub objects are used, there is no way to satisfy the rule. This issue is addressed in the next section.

5.2 Circular references

5.2.1 *Motivating example*

According to the XML specification [7] an XML document has exactly one XML element that could have nested elements. It is called a root element. On the other hand, every XML element has a parent. A parent of a root element is the document it is contained in. It would be convenient to have an attribute *root* of an attached type in the class *DOCUMENT* to refer to the associated root element, and, conversely, to have an attribute *parent* of an attached type in the class *ELEMENT* to refer to the parent element, the document for the root element. In order to set both attributes *root* and *parent*, creation of a document should come along with creation of a root element. Ideally it should be done in a single step, so that there are no inconsistencies. Unfortunately modern languages do not provision for simultaneous object creation and initialization of their fields via an atomic operation. The creation and initialization is done step-by-step as shown in figure 5.15:

1. The document object is created (figure 5.15a). All fields are set by the run-time to respective default values. Because there are no meaningful values for fields of attached types, they are initialized with **Void** like any other reference field.

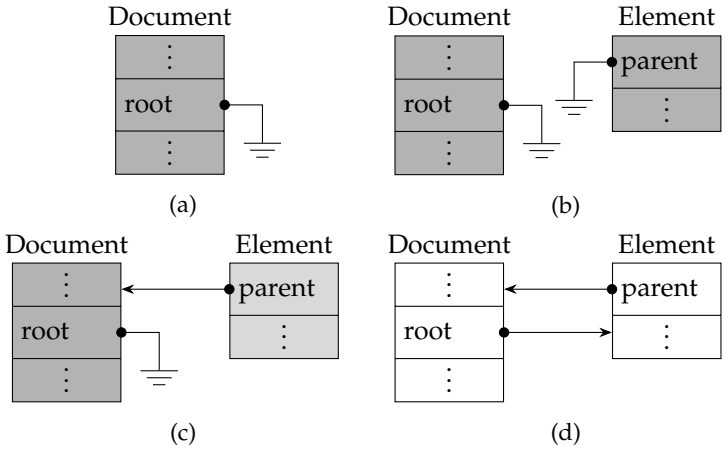

Figure 5.15. Stages of creating objects with circular references.

Legend:
– not all attributes are properly set
– all attributes are properly set, but they can (recursively) refer to an object with not all attributes properly set
– all attributes of all reachable objects are properly set .

These fields should not be accessed, or, at least, their type should not be considered as attached until they are actually set.

2. The creation procedure of the document object executes a creation procedure for the root element object and passes the document object **Current** as an argument. The element object is created (figure 5.15b) like in the previous step with all fields of attached types not in a valid state. In particular, the field *parent* is **Void**.
3. The attribute *parent* of the root element object is set to reference the document object passed as an argument (figure 5.15c). It is now acceptable to use the field *parent* and to rely on the fact that it is of an attached type. What is not acceptable is to access the fields of the object accessible by reference *parent*. E. g., *parent.root* would yield **Void** although the type of the expression is attached. Any other fields of the element object should be properly set before completion of the creation procedure.

4. On return from the creation procedure of the root element, the document creation procedure sets the attribute *root* to point to the newly created element object. As soon as all other fields of the document object are properly initialized, both objects become fully usable and any attribute can be accessed without a problem.

This scenario violates validity rule 5.1 in step 2: the current object **Current** is used before all attributes are initialized. Indeed, it is passed to the creation procedure of the root element before its attribute *root* is set. Moreover, the attribute cannot be set because the root element object is not created at this point yet. So, there should be some way to deal with objects before all their fields are completely initialized.

5.2.2 *Solution*

When an object of a particular type does not guarantee properties of this type, there is a type mismatch. In other words, the type of the object is different from what is declared. This is exactly the case with an incompletely initialized object, in particular, with objects `Document` and `Element` from the previous section in steps 1–3. Solutions proposed before (see [17, 62, 70]) introduce a notion of incompletely initialized types to a void-safe language. However, as noted in section 5.1.6 a lot of libraries can be made void-safe without any need for additional type annotations. So, in order to keep annotation burden limited it would be reasonable to find validity rules that would prevent from unsafe use of objects that are not completely set up.

The dangerous part is access to object's fields that are not attached to objects yet. As soon as a reference to an incompletely initialized object is released (i. e., reattached to some variable), the task to identify such an object becomes almost intractable not only in theory, but also due to complexity of implementing alias analysis correctly (see [75]). Use of explicit type annotations serves as a way to simplify the analysis and to move the detection of incompletely initialized objects from static analysis methods to a type system.

In this work I propose to avoid performing alias analysis and extending the type system in favor of preventing the use of incompletely initialized objects in the first place. As we have seen

before, access to such objects is fine inside associated creation procedures: their attributes can be read and written according to the corresponding validity rules. It is even fine to pass such objects as arguments provided that access to them is still controlled, i. e., field reads should either be prohibited or be processed without an assumption that they are properly set (definition 4.2).

The key source of obscurity in an object-oriented environment is polymorphism when execution of a feature call depends on the run-time type of the target object which is not statically known. Because creation procedures are associated with specific classes, no polymorphism is involved when they are checked. It is even possible to perform checks of unqualified features they call because the classes checked for creation validity are known. The checks will make sure that class fields are not accessed before they are set and **Current** is completely initialized. The only issue is with qualified calls:

- a call on an incompletely initialized object cannot assume all attributes are properly set
- a qualified call does not allow seeing what operations on an incompletely initialized object are performed

The solution is to prevent making qualified calls when some objects are not completely uninitialized:

Validity rule 5.2 (Creation procedure)**.** *A creation procedure is valid if any of the following is false at the same execution point:*

- ***Current** is used before all attributes have been properly set and not all attributes are properly set after that.*
- *The expression at the execution point is one of*
  - *a qualified feature call;*
  - *a creation expression that makes a qualified call.*

The difference from validity rule 5.1 is that it allows for using **Current** before all attributes are properly set. But this can be done only when no qualified calls are performed in these conditions.

The timeline of creation procedure execution corresponding to validity rule 5.2 is depicted in figure 5.16. Comparing to figure 5.4, there is now a gray area between the moment **Current** is used and the moment all attributes are properly set. There are no restrictions on making qualified calls before of after these moments. But qualified calls are prohibited between them, i. e., in the gray area.

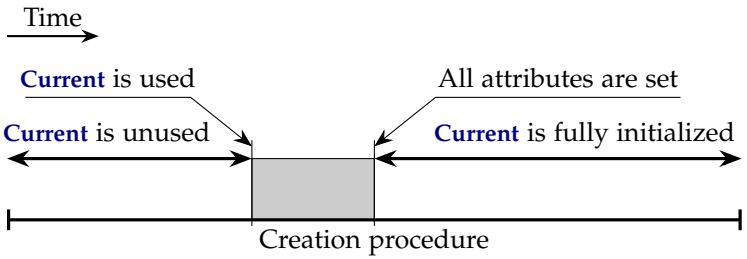

Figure 5.16. Timeline during object initialization according to validity rule 5.2.

To define the validity rule formally, in addition to the previously defined transfer function that computes a set of unattached attributes, several new are required:

- to see if **Current** is referenced before all attributes are set
- to determine if a routine body has any qualified calls
- to collect all creation procedures that can be called when not all attributes are set and when **Current** is used

Unlike validity rule 5.1 this assumes that there is information about other classes, in particular whether their creation procedures make direct or indirect qualified feature calls. This information could be explicitly or implicitly specified in creation procedure signatures. However, given that in most cases the type used in a creation expression is known statically, this additional annotation seems to be unnecessary, and the information can be inferred from the code.

5.2.2.1 *Safe use of Current*

Two conditions have to be checked to make sure expressions use **Current** safely:

- if a reference to **Current** is obtained before or after all attributes of the current class are properly set
- if all attributes of the current class are properly set for a given expression

If **Current** is never referenced, there are no problems because the incompletely initialized object is never passed to program parts that are unaware about its initialization status. If **Current** is referenced when all attributes are set, there is no problem as well:

B.25.4
p. 285

$$
\begin{aligned}
&\mathit{safe}\ \mathit{Current}\ V && = V = \varnothing \\
&\mathit{safe}\ (e_1\ ;;\ e_2)\ V && = \mathit{safe}\ [e_1, e_2]\ V \\
&\mathit{safe}\ (n :=_{\mathrm{L}}\ e)\ V && = \mathit{safe}\ e\ V \\
&\mathit{safe}\ (n :=_{\mathrm{A}}\ e)\ V && = \mathit{safe}\ e\ V \vee V \gg n :=_{\mathrm{A}}\ e = \varnothing \\
&\mathit{safe}\ (\mathit{create}\ \{t\}\ .\ n\ (es))\ V && = \mathit{safe}\ es\ V \\
&\mathit{safe}\ (e\ .\ n\ (es))\ V && = \mathit{safe}\ (e \cdot es)\ V \\
&\mathit{safe}\ (\mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end})\ V && = \mathit{safe}\ [c, e_1]\ V \wedge \mathit{safe}\ [c, e_2]\ V \\
&\mathit{safe}\ (\mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end})\ V && = \mathit{safe}\ [e, b]\ V \\
&\mathit{safe}\ (\mathit{attached}\ t\ e\ \mathit{as}\ n)\ V && = \mathit{safe}\ e\ V \\
&\mathit{safe}\ \_\ V && = \mathit{True}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{safe}\ []\ V && = \mathit{True} \\
&\mathit{safe}\ (e \cdot es)\ V && = \mathit{safe}\ e\ V \wedge \mathit{safe}\ es\ (V \gg e) \vee V \gg (e \cdot es) = \varnothing
\end{aligned}
$$

Figure 5.17. A function that tells if uses of **Current** (if any) are safe.

once an object is completely initialized, it remains completely initialized and can be freely used. Finally, if **Current** is referenced when not all attributes of the current class are set, but can escape only at the current execution point (i. e., all previous expressions do not make any qualified calls, thus excluding the possibility to access this incompletely initialized object), it is possible that all attributes are set now and therefore the object is completely initialized regardless of its status when the reference to it was used. The formal function equations following these informal observations are specified in figure 5.17.

An expression *Current* is safe if and only if all attributes are properly set (i. e., there are no unattached attributes). If the expression is an assignment to an attribute, it is possible that the attribute would be the last one to initialize, so the result will be true if either the source expression is safe or there are no unattached attributes left after the assignment.

For a list of expressions, the first expression in the initial context should be safe as well as the subsequent expressions in the context of unattached attributes obtained for the first expression should be true. Alternatively all attributes should be set for the

whole list. Similar rules are used for a sequence of expressions, a feature call and a creation expression.

For a conditional expression the checks should be done for both branches, and the use of **Current** is safe if it is safe in both branches. For a loop the check is done for its exit condition and for the loop body like for a sequence of expressions to make sure that if **Current** escapes from the loop, all attributes should be set when it may reach code that expects only completely initialized objects.

For expressions that have a single subexpression, this subexpression is checked for safety. And for the remaining expressions the function returns *True*.

If after an expression a set of unattached attributes is empty, the use of **Current** is safe because the corresponding object will be completely initialized:

Lemma 5.2. $V \gg e = \varnothing \Longrightarrow \mathit{safe}\ e\ V$ *B.25.4 p. 285*

Similarly, if an expression does not refer to **Current**, there is no way to access an incompletely initialized object, and therefore the expression is safe:

Lemma 5.3. $\neg\ \mathit{has\_current}\ e \Longrightarrow \mathit{safe}\ e\ V$ *B.25.4 p. 286*

Here the function *has_current* tells if a given expression references **Current**. (The formal definition can be found in appendix B.) *B.25.3 p. 283*

Finally, if an expression satisfies the validity predicate from section 5.1.2, then it is safe:

Lemma 5.4. $V \vdash e\ \surd_{\mathrm{c}}\ ' \Longrightarrow \mathit{safe}\ e\ V$ *B.25.4 p. 286*

All the proofs are carried out by induction.

In general, the reverse statement of lemma 5.4 is incorrect. Consider the following code snippet:

```
if c then
    b := Current
    a := b
else
    b := d
end
```

Assume that a is an attribute variable, b is a local variable and before the conditional instruction the set of unattached attributes

is $\{a\}$. Then according to the definition of the validity predicate from section 5.1.2 it will give *False* because **Current** is used before all attributes are initialized. On the other hand, the assignment instruction $a := b$ finishes initialization of the current object and therefore **Current** assigned to b can be safely used without any specific restrictions.

Also, note that the function *safe* cannot be obtained as a combination of the function *has_current* and the transfer function $\gg$. Indeed, for the code snippet above, *has_current* will evaluate to *True* because **Current** is referenced in the first branch, and the set of unattached attributes will not be empty because the second branch does not set attribute a. So, using their combination would render the code as unsafe that differs from the result of the function *safe*. In other words, this function is more permissive than the validity predicate used earlier. The restrictions come from the other condition that ensures that uninitialized objects do not reach features that do not expect them.

Function *safe* is monotone: if more attributes are set for a given expressions, the chances to use **Current** unsafely are lower:

B.25.4 p. 287

Lemma 5.5 (Monotonicity of function *safe*)**.** *antimono* (*safe e*)

Proof. By induction relying on monotonicity of transfer function $\gg$ (lemma 5.1). □

Even though formally the function is anti-monotone (the predicates *mono* and *antimono* differ by the conditions when they are true – for monotonically non-descreasing and for monotonically non-increasing functions respectively), using a set of attached attributes instead of unattached ones reverses the order, hence the lemma name.

5.2.2.2 *Detection of qualified feature calls*

For telling if a feature makes a qualified feature call, it is sufficient to analyze the corresponding abstract syntax tree as shown in figure 5.18.

It evaluates to *True* if any construct has an immediate qualified call in at least one of its subexpressions. Otherwise, it evaluates to *False*. In the real implementation it also takes care about qualified feature calls present in the features that are called from a current creation procedure using an unqualified feature call. This is done

$$
\begin{array}{ll}
\mathcal{Q}\ (e_1\ ;;\ e_2) & = \mathcal{Q}\ e_1 \vee \mathcal{Q}\ e_2 \\
\mathcal{Q}\ (n :=_{\mathrm{L}}\ e) & = \mathcal{Q}\ e \\
\mathcal{Q}\ (n :=_{\mathrm{A}}\ e) & = \mathcal{Q}\ e \\
\mathcal{Q}\ (\mathit{create}\ \{t\}\ .\ n\ (es)) & = \mathcal{Q}s\ es \\
\mathcal{Q}\ (e\ .\ n\ (es)) & = \mathit{True} \\
\mathcal{Q}\ (\mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end}) & = \mathcal{Q}\ c \vee \mathcal{Q}\ e_1 \vee \mathcal{Q}\ e_2 \\
\mathcal{Q}\ (\mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}) & = \mathcal{Q}\ e \vee \mathcal{Q}\ b \\
\mathcal{Q}\ (\mathit{attached}\ t\ e\ \mathit{as}\ n) & = \mathcal{Q}\ e \\
\mathcal{Q}\ \_ & = \mathit{False} \\
\mathcal{Q}s\ [] & = \mathit{False} \\
\mathcal{Q}s\ (e \cdot es) & = \mathcal{Q}\ e \vee \mathcal{Q}s\ es
\end{array}
$$

B.25.4 p. 284

Figure 5.18. A function to detect presence of an immediate qualified call.

recursively with registering features that have been processed to avoid infinite recursion at compile time.

In a given system S the check whether a creation procedure f in a class c makes an immediate qualified feature call can be done with a function:

$$
\begin{array}{l}
\mathit{has\_immediate\_qualified\_in\_routine}\ S\ (c, f) = \\
\qquad \mathsf{case}\ \mathit{routine\_body}\ S\ c\ f\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{False} \mid \lfloor b \rfloor \Rightarrow \mathcal{Q}\ b
\end{array}
$$

B.25.4 p. 285

Here the function *routine_body* gives an optional routine body for a feature of name f in a class c of a system S. When the body is found, the function $\mathcal{Q}$ is called.

Qualified feature calls can potentially lead to execution of arbitrary code, including accesses to incompletely initialized objects if that were permitted and causing access on void target for attributes that were not properly set. So, when **Current** is used, all class attributes should be initialized, or qualified feature calls should not be allowed. Such qualified calls can appear not only in the current creation procedure, but also in creation procedures called from the current one to create other objects. The creation procedure calls could lead to execution of other creation procedures recursively.

B.25.4 p. 283

$$
\begin{aligned}
&\mathcal{S}\ (e_1\ ;;\ e_2) && = \mathcal{S}\ e_1 \cup \mathcal{S}\ e_2\\
&\mathcal{S}\ (n :=_{\mathrm{L}}\ e) && = \mathcal{S}\ e\\
&\mathcal{S}\ (n :=_{\mathrm{A}}\ e) && = \mathcal{S}\ e\\
&\mathcal{S}\ (\mathit{create}\ \{c\}\ .\ n\ (es)) && = \{(c, n)\} \cup \mathcal{S}s\ es\\
&\mathcal{S}\ (e\ .\ n\ (es)) && = \mathcal{S}\ e \cup \mathcal{S}s\ es\\
&\mathcal{S}\ (\mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end}) && = \mathcal{S}\ c \cup \mathcal{S}\ e_1 \cup \mathcal{S}\ e_2\\
&\mathcal{S}\ (\mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}) && = \mathcal{S}\ e \cup \mathcal{S}\ b\\
&\mathcal{S}\ (\mathit{attached}\ t\ e\ \mathit{as}\ n) && = \mathcal{S}\ e\\
&\mathcal{S}\ \_ && = \varnothing\\
&\mathcal{S}s\ [] && = \varnothing\\
&\mathcal{S}s\ (e \cdot es) && = \mathcal{S}\ e \cup \mathcal{S}s\ es
\end{aligned}
$$

Figure 5.19. A function that collects all creation procedure calls.

A set of creation procedures that can be called by the current one is computed by the function $\mathcal{S}$ (figure 5.19). It returns a set of pairs *class_type* × *feature_name* for all creation expressions with the corresponding class type and creation procedure name.

Using current system information it is possible to compute a set of creation procedures that can be called from a creation procedure of a given name from a given class:

B.25.4 p. 283

$$
\begin{aligned}
&\mathit{creation\_reachable}_1\ S\ (c, f) =\\
&\qquad \mathsf{case}\ \mathit{routine\_body}\ S\ c\ f\ \mathsf{of}\ \mathit{None} \Rightarrow \varnothing \mid \lfloor x \rfloor \Rightarrow \mathcal{S}\ x
\end{aligned}
$$

Because the set of classes is known at compile time and is bounded, all recursively reachable creation procedures can be computed as a least fixed point using the previous function:

B.25.4 p. 284

$$
\begin{aligned}
&\mathit{creation\_reachable}\ S\ (c, f) =\\
&\qquad \mathit{lfp}\ (\lambda x.\ \{(c, f)\} \cup x \cup (\bigcup\nolimits_{y \in x} \mathit{creation\_reachable}_1\ S\ y))
\end{aligned}
$$

Together with the function that tells whether a creation procedure has immediate qualified calls the function *has_qualified* tells if a creation procedure can (indirectly) lead to a qualified call:

B.25.4 p. 285

has_qualified S c =

$\exists x \in$ *creation_reachable S c*. *has_immediate_qualified_in_routine S x*

5.2.2.3 *Validity predicate*

With the function *safe* that tells whether **Current** is used only when all attributes are properly set and the function *has_qualified* that tells whether a creation procedure may (directly or indirectly) make a qualified call, it is now possible to define a formal predicate for validity rule 5.2. The rules are shown in figure 5.20. They are quite similar to the ones of the stronger predicate described earlier (figure 5.7).

The differences are:

- The rule for an expression *Current* is now an axiom, i. e., this expression is always valid.
- The rule for a creation expression has an additional premise that usage of **Current** should be safe or, alternatively, the called creation procedure should not make any qualified calls.
- The rule for a qualified call has an additional premise that **Current** is used safely.

Here is a detailed analysis of an example using the specified predicate. The first column shows a set of unattached attributes before the expression. The second column contains the source code. The third column reports a value of the function *safe* for the given expression in the given context. And the last column indicates if the validity predicate is *True* or *False*. It is assumed that the variable *a* is an attribute and the variables *b*, *c* and *d* are locals and that **Current** is not referenced in the code before this code snippet:

B.25.5
p. 287

$$\frac{}{S, V \vdash \mathit{Value}\ v\ \surd_c}\ \textsc{Value} \qquad \frac{}{S, V \vdash \mathit{Current}\ \surd_c}\ \textsc{Current}$$

$$\frac{}{S, V \vdash \mathit{Local}\ n\ \surd_c}\ \textsc{Local} \qquad \frac{n \notin V}{S, V \vdash \mathit{Attribute}\ n\ \surd_c}\ \textsc{Attr}$$

$$\frac{S, V \vdash e_1\ \surd_c \wedge S, V \gg e_1 \vdash e_2\ \surd_c}{S, V \vdash e_1\ ;;\ e_2\ \surd_c}\ \textsc{Seq}$$

$$\frac{S, V \vdash e\ \surd_c}{S, V \vdash n :=_L e\ \surd_c}\ \textsc{AssignLocal} \qquad \frac{S, V \vdash e\ \surd_c}{S, V \vdash n :=_A e\ \surd_c}\ \textsc{AssignAttr}$$

$$\frac{S, V \vdash es\ [\surd_c] \wedge (\mathit{safe}\ es\ V \vee \neg\ \mathit{has\_qualified}\ S\ (c, n))}{S, V \vdash \mathit{create}\ \{c\}\ .\ n\ (es)\ \surd_c}\ \textsc{Create}$$

$$\frac{S, V \vdash e\ \surd_c \wedge S, V \gg e \vdash es\ [\surd_c] \wedge \mathit{safe}\ (e \cdot es)\ V}{S, V \vdash e\ .\ n\ (es)\ \surd_c}\ \textsc{Call}$$

$$\frac{S, V \vdash c\ \surd_c \wedge S, V \gg c \vdash e_1\ \surd_c \wedge S, V \gg c \vdash e_2\ \surd_c}{S, V \vdash \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end}\ \surd_c}\ \textsc{If}$$

$$\frac{S, V \vdash e\ \surd_c \wedge S, V \gg e \vdash b\ \surd_c}{S, V \vdash \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}\ \surd_c}\ \textsc{Loop}$$

$$\frac{S, V \vdash e\ \surd_c}{S, V \vdash \mathit{attached}\ t\ e\ \mathit{as}\ n\ \surd_c}\ \textsc{Test} \qquad \frac{}{S, V \vdash \mathit{Exception}\ \surd_c}\ \textsc{Exception}$$

$$\frac{}{S, V \vdash []\ [\surd_c]}\ \textsc{Arg}_{\mathtt{Nil}}$$

$$\frac{S, V \vdash e\ \surd_c \wedge S, V \gg e \vdash es\ [\surd_c]}{S, V \vdash e \cdot es\ [\surd_c]}\ \textsc{Arg}_{\mathtt{Cons}}$$

Figure 5.20. A predicate that tells if an expression is valid in a creation procedure of a system S in the context with a set of unattached attributes V.

Unset attr.	Source code	Value of *safe*	Is valid?
{a}	**if** *c* **then**	*True*	Yes
{a}	*b* := **Current**	*False*	Yes: no qualified calls
{a}	**create** *a.make* (*b*)	*True*	Yes if ¬ *has_qualified ... make*
			No otherwise
∅	**else**	*True*	Yes: all attributes are now set
{a}	*b* := *d*	*True*	Yes: **Current** is not referenced
{a}	**end**	*True*	Yes: *safe* for both branches
{a}	*b.foo* (*d*)	*True*	Yes: *safe*

Note that the last instruction can be reached at run-time in two states: when all fields of the current object are completely set and when they are not. However, in the second case there is no reference to the incompletely initialized object, because there is no reference to **Current** in the "else" branch, therefore the code remains safe.

It turns out that validity rule 5.1 implies validity rule 5.2:

Lemma 5.6. $V \vdash e \surd_c{}' \Longrightarrow S, V \vdash e \surd_c$ *B.25.5 p. 288*

Proof. By induction on the definition of the predicate used in the premise. □

The reverse implication does not hold when V is non-empty and e is *Current*. Therefore, validity rule 5.1 is less permissive than validity rule 5.2.

Coming back to the motivating example, the code of the creation procedure of class *DOCUMENT* would look like

```
... -- Initialize some attributes.
create root.make (Current)
... -- Initialize some other attributes if
       required.
```

and the creation procedure of class *ELEMENT* would look like

```
make (p: like parent)
    do
        parent := p
        ... -- Initialize other attributes if required
        ... -- but do not make qualified calls.
    end
```

As soon as the creation procedure of class *ELEMENT* does not make any qualified calls, the code satisfies the validity predicate and compiles successfully.

For formal proofs it is important to know if the predicate is monotone. Then it is sufficient to analyze loops and unqualified feature calls just once, because any subsequent iterations or recursive feature calls would be analyzed with a larger set of properly-set attributes. Therefore, there is no need to perform the analysis iteratively and recursively. The property is indeed true for this predicate:

B.25.5 p. 288

Lemma 5.7 (Validity predicate monotonicity)**.**

$$A \subseteq B \wedge S, B \vdash e \surd_c \implies S, A \vdash e \surd_c$$

Proof. By induction using monotonicity of transfer function $\gg$ (lemma 5.1). In inductive cases for qualified calls and creation procedures, monotonicity of function *safe* (lemma 5.5) is used as well. □

5.2.2.4 *Remaining matters*

Formal generics. Validity rule 5.2 implies that for every creation procedure call the corresponding creation procedure is accessible for analysis. The analysis has not to be performed every time. But as soon as **Current** has been referenced before all attributes have been set, it should be feasible. This is a valid assumption when the target type of a creation expression is a class type. When the type is a formal generic parameter of the current class, only creation procedures of the formal generic constraint classes can be analyzed. In general this is insufficient. For example, if the creation procedures of the constraint classes are deferred or do not make any qualified calls, descendant classes can redefine the procedures and introduce qualified calls in the redeclarations.

Possible solutions to the issue include

- Changing validity rule 5.2 by rephrasing the condition "*a creation expression that makes a qualified call*" into "*a creation expression that makes a qualified call or when its creation type is a formal generic*". This is the approach taken in the current implementation.
- Introducing a convention that if the formal generic constraint creation procedure does not make a qualified call, the corresponding actual generic class creation procedure should not make a qualified call. This rule might be too restrictive because it could forbid generic instantiations not related to formal generic creation.

Class invariants. A standard class invariant of class *ANY* (see [16])

reflexive_equality: *standard_is_equal* (**Current**)

makes a call to the function *standard_is_equal* with the following postcondition

symmetric: **Result implies** *other.standard_is_equal* (**Current**)

If a rule of checking for class invariant is applied according to *Design by Contract*™, validity rule 5.2 should take this into account and require that class invariants of the corresponding classes are also checked. This would inevitably lead to detection of the qualified call in the postcondition `symmetric`. If the creation procedure is called in unsafe conditions, such as creation with the target of type *ELEMENT* during XML document creation, the class invariant will be invalid due to the qualified call.

The issue can be solved by

- Noticing that the feature *standard_is_equal* does not make any qualified calls itself, i. e., cannot lead to access on void target. So, it can be ignored when detecting qualified feature calls.
- Realizing that this is an instance of a general issue with class invariants described in [48] when a class invariant may be violated if there are mutual references between two objects and the class invariant is checked when at least one of the objects is in an unstable state. One possible solution is

to delay evaluation of class invariants until all attributes of all objects are properly set. On the language specification side it means that class invariants should not be subject to static checks when first two conditions of validity rule 5.2 are *True*.

Once features. Validity rule 5.2 guarantees no qualified calls can ever receive an incompletely initialized object either as a target of a call or as an argument. This ensures incompletely initialized objects cannot be accessed as a part of regular program execution, but only during object creation. Are there any other ways for incompletely initialized objects to become accessible outside of object initialization process? Yes, this can happen if they can be shared via some non-object variables. In Java and C# these are static fields, in Eiffel – once features, or, to be more precise, once functions.

For example, if the following once function is called when not all attributes of a class *FOO* are set, it still can be saved for the next time the function is called:

```
f (arg: FOO): FOO
    once
        Result := arg
    end
```

If, for some reason, the creation procedure that called *f* passing **Current**, when not all attributes have been set, fails before it sets all attributes, the reference to the object is still there: any subsequent call to the function *f* will yield this same object. Because not all callers of the feature *f* expect to get an incompletely initialized object, void safety can be compromised. Of course, real-life scenarios could be more complicated. E. g., a passed argument could be used to create another object rather than to assign the argument to the result. But the idea remains the same: as soon as an incompletely initialized object is passed to a once function, its result may depend on it and can make it reachable.

The following restriction on once functions prevents this from happening:

Validity rule 5.3 (Once function in a creation procedure)**.** *An unqualified call to a once function is valid in a creation procedure if the creation procedure remains valid according to validity rule 5.2 after*

adding a fictitious qualified call to an identity function on the once function result.

(Instead of introducing the new validity rule, validity rule 5.2 can be amended by replacing all occurrences of *a qualified (feature) call* with: *a qualified (feature) call or an unqualified call to a once function*.)

This validity rule mimics the situation when a qualified feature call on a result of a once function occurs in some code outside of a particular creation procedure. If the resulting object is not completely initialized, such a call could compromise void safety guarantees.

Like with qualified feature calls, calls to once functions could happen indirectly, inside other creation procedure calls. Therefore, the technique to detect such situations is similar to the one used to detect qualified calls, i. e., relies on computation of a transitive closure.

A consequence of the rule is that once functions can return only completely initialized objects.

5.2.3 *Implementation*

Evaluation of the functions *safe* and Ω is similar to evaluation of the functions described earlier for a simpler solution in section 5.1.5 and can be done in one pass over an abstract syntax tree. What becomes problematic is a function *has_qualified* that tells if a particular creation procedure of a particular class can directly or indirectly make a qualified call. It might look like a potential reason to perform whole system analysis and to break the requirement of modularity. Fortunately, the proposed scheme does not trigger massive recompilation of everything when small changes are applied to a large system. Let's see how this is achieved.

Every class is compiled separately as before. If, when checking a creation procedure, a qualified call appears in the AST at the position where the function *safe* gives *False*, the compiler triggers an error. Otherwise, the qualified feature call is allowed. Similarly, when *safe* is *True*, the corresponding creation expression is safe. What remains to check is that, when the function *safe* gives *False* and there is a creation expression, the function *has_qualified* for the corresponding creation procedure gives *False*. To this end every creation expression **create** *c*.*n* (*es*) is recorded for the current

creation procedure as a pair *(c, n)*, where *c* and *n* are the creation expression class name and the creation procedure name respectively, in either of the following tables:

- **safe** – when the creation expression appears at the position where the function *safe* gives *True*;
- **unsafe** – otherwise.

Also, information whether the current creation procedure makes any qualified calls is recorded together with other meta-information of the class for future use. During recompilation, if there are any changes to the class, this information together with the entries in the tables **safe** and **unsafe** is recomputed.

As soon as the current class and all classes it depends on are analyzed, the compiler checks if there are any entries in the table **unsafe**. If there are none, the class is void-safe.

If the table **unsafe** is not empty, all its entries are checked one-by-one to see if *has_qualified* for each entry is *False*. Instead of computing the transitive closure directly, the tables **safe** and **unsafe** are used to figure out what creation procedures are called directly or by some other reachable creation procedure. If at least one of them makes a qualified call, an error is reported explaining what attributes are not set in the initial creation procedure from the table **unsafe**. Otherwise, the initial creation procedure passes the check.

Whether some intermediate checks use entries from a particular table does not matter, but from the programmer's perspective accesses to the table **unsafe** correspond to nested creations of mutually dependent objects, allowing for making more complicated structures than just two objects referencing each other.

5.2.4 *Empirical results*

As of time of writing only two libraries take advantage of the more relaxed rule introduced in this section: *Gobo* and an unreleased void-safe version of a library *Docking*. From the perspective of the validity rule definition the following metrics are of interest:

- How many creation procedures have qualified calls?
- How often is **Current** passed before all attributes of a class are properly set?

Table 5.1. Creation procedures classified by use of qualified calls and incompletely initialized objects.

Library	Creation	Qualified		Uninitialized	
		abs.	rel.	abs.	rel.
Docking	2062	1365	66.2%	16	0.8%
Gobo	1258	726	57.7%	61	4.8%
Others	3194	1894	59.3%	0	0.0%
Total (cumulative)	4045	2442	**60.4%**	77	**1.9%**

Legend:
Creation – total number of explicit creation procedures (including supplier libraries).
Qualified – total number and percentage of creation procedures that have qualified calls.
Uninitialized – total number and percentage of creation procedures that pass **Current** before all attributes are initialized.

- What is a relation between classes that pass **Current** before all attributes are set?

On average the total number of creation procedures that make qualified calls is about 60% (see table 5.1). It means that the remaining 40% do not use any qualified calls and set attributes using supplied arguments or by creating new objects. In theory these 40% of creation procedures could be unconditionally (and even automatically) marked with annotations as safe for use with incompletely initialized objects.

In contrast to this, just a tiny fraction of all creation procedures – less than 2% – do pass uninitialized objects. In other words, if the specific annotations were used, at most 5% (remember that more than 40% of creation procedures can be marked as not relying on whether the arguments are completely initialized or not) of the annotations would be useful, the rest would just clutter the code.

Closer look at the cases when **Current** is passed before all attributes are set reveals the following major families of uses:

Internal cursors. 59 of 61 cases in *Gobo* library. Internal cursors are associated with containers to perform traversal abstracting away particular container implementation and are available through the whole container life cycle without the need to create new objects for traversal. In practice, they

cause programming errors if the same container needs to be traversed multiple times and one of the traversals is performed while some other traversal is still active. To avoid unexpected behavior the cursor state has to be recorded before doing any other traversal. The main benefit of using internal cursors is that unlike external cursors they can work when the underlying container changes, e. g., by inserting or removing elements ([5]). In *Gobo* external cursors are aware about changes in associated containers, so they are more robust and can replace internal cursors altogether. Also, the use of internal cursors implemented in separate classes is a design decision. The same functionality is provided in the library *EiffelBase* without any additional classes, directly in the containers.

Domain structure. 2 of 61 cases in *Gobo*. These cases correspond to the example with XML document and root element classes described in this section. The structure is suggested by the XML standard specifying a logical composition of an XML document. In theory the root element can be a descendant of a general XML element class with additional properties specific for a document, i. e., removing separation of concern can remove the need for two different classes.

Helper classes. All cases in *Docking* library. The library deals with many aspects of user interface, allows for storing and retrieving layout, animates placeholders, etc. All this functionality could be moved to just one class, but for maintainability it was distributed among different classes. Because interaction between classes is bi-directional, they need to refer to each other circularly. The classes are tightly coupled and could be replaced by larger monolithic classes if the principle of separation of concerns was not an issue.

To summarize, the need to pass **Current** before all attributes are properly set arises only for classes that are closely related. It could even be avoided altogether if the code is allowed to be refactored discarding the principle of separation of concern. The rate of cases when qualified features are used in creation procedures and when they are not used is pretty high, therefore relying solely on additional annotations describing when objects are passed incompletely initialized would be problematic because of high annotation burden.

Table 5.2. Compilation time increase due to additional checks for object initialization.

Library	Compilation time		Relative slowdown
	w/o checks	with checks	
Docking	42.89	43.52	1.5%
Gobo	19.14	19.35	1.1%
Others	53.32	52.85	-0.9%
Total (cumulative)	74.06	74.54	**0.7%**

Legend:
Compilation time – time to compile all classes without generating code in seconds
Relative slowdown – relative increase of compilation time when comparing compilation with and without checks.

The additional check for the validity rule for creation procedures is pretty light. In order to measure it, the libraries were compiled with two void safety levels (section 3.3.1): transitional and complete. The main difference between them is that in transitional mode validity rule 5.2 is not checked and in complete mode it is checked. The compilation involved only parsing and type checking, including void safety checks, but not code generation. It was done on a machine with 64-bit *Windows 10 Pro*, *Intel® Core™ i7-3720QM*, 16GB of RAM and SSD hard drive using *EiffelStudio 16.11 rev.99675*. The first run was ignored, from the subsequent 10 runs, 2 with the longest time were discarded to get rid of the runs when the operating system started its own tasks that caused general slowdown. There was also a 20-second interval between runs for cooling down the system. Under these conditions relative corrected sample standard deviation was below 2.4%. The relative compilation time increase is listed in table 5.2. Its maximum value is 1.5% for *Docking* library. The compilation time for libraries that do not trigger additional checks (*Other*) is even lower in this case. Most probably this is caused by different cache state or other accidental factors and does not reflect any real difference. For all libraries the slowdown is just 0.7% that seems to be more than acceptable.

5.3 Object disposal

Many object-oriented languages provide a hook for developers to catch an event when object's memory is about to be reclaimed. This happens when there are no more references to an object or when a system terminates. In Java [21, 22] and C# [26] this is done via methods `finalize` and `Finalize` of the class `Object`. In Eiffel the same functionality is available via a procedure *dispose* of the class *DISPOSABLE*.

If a class redefines the feature *dispose*, a run-time memory manager registers the corresponding object for custom disposal. This is done before an associated creation procedure is executed. If the creation procedure terminates with an exception, or if the object creation is initiated by another creation procedure that passes **Current** to the current one, does not complete initialization and terminates exceptionally, the object can be in any of the following states:

- All fields of the current object and of all objects reachable from it are properly set.
- All fields of the current object are properly set, but some fields of objects reachable from the current one are not properly set.
- Some fields of the current object are not properly set.

Unfortunately, there is no way to determine, which stage the current object is in, when *dispose* is called, unless some special compile-time and run-time machinery is used to track the object's state. Nevertheless, any unsafe operations should be prevented. And the unsafe operations include all accesses to reference fields regardless of their attachment status, because the state of objects they may refer to is also unknown. This leads to the following rule for the feature *dispose* (fields in the discussion above are run-time memory locations, but the rule is specified in terms of compile-time abstractions – attributes in this case):

Validity rule 5.4. *A redeclaration of a feature* dispose *of a class* DISPOSABLE *is valid if this redeclaration or any feature called from it by an unqualified call does not access any attribute except, possibly, for attributes of expanded types that (recursively) have no attributes of reference types.*

I believe this is the first time the issue of void safety in class finalizers is considered. Formalizations used to prove soundness of void-safe type systems usually abstract away garbage collector part of automatic memory management and take into account only object creation. But for the language specification it is important to cover all parts of object's life cycle.

5.4 Related work

Null pointer dereferencing is long-standing issue in object-oriented programs, but it attracted attention only in early 2000's with increasing adoption of popular object-oriented languages Java and C#. A bit earlier researchers started to look at type safety properties of languages, including formalization of their type systems and formal safety proofs. Therefore, several proposals were made to extend existing type systems with a new notion of types to identify objects that are not completely initialized.

Raw types. Manuel Fähndrich and K. Rustan M. Leino in [17] denote attached types with T^- and detachable types with T^+ and propose to add raw types T^{raw-} to be used for partially initialized objects. For an object of that type there is no guarantee that a field of an attached type will always have a non-null value. In other words, if class C has an attribute of a type T and some entity has a type C^{raw-} then a qualified call to this attribute has a type T^+ regardless of original attachment status of that attribute. The other rule is that an assignment to an entity of a raw type accepts only a source expression of a non-raw non-null type. This restriction is introduced to make sure that if an object becomes fully initialized, it cannot be uninitialized. It takes care about potential aliasing. If during object initialization the same reference is attached to two variables, one of type T^- and another of type T^{raw-}, the rule makes sure the variable of type T^- cannot always have non-null values in the associated fields of attached types. The other rule is that by the end of every constructor, every non-null field should be assigned.

Then they further refine the definition of the raw types by introducing class frames. Every frame corresponds to a superclass. A constructor is responsible for initialization of fields declared in this class only because all the fields from the parent classes are

initialized by super-class constructors that are called automatically on object creation according to the language rules of C# and Java ([21, 26]). So, inside a constructor of a class C, the special entity **this** has type C^{raw-}, and when the constructor finishes, the type becomes C^{-}. In a constructor of a super-class A the type of **this** is $C^{raw(A)-}$. The conformance rules in this type system for a subclass S of a class R (i. e., $S <: R$) are:

- $S^{+} <: R^{+}$
- $S^{-} <: R^{-}$
- $T^{raw(S)-} <: T^{raw(R)-}$
- $T^{raw(S)-} <: T^{-}$
- $T^{-} <: T^{raw-}$
- $T- <: T+$

The rules for super-class constructors seem to be tightly coupled with the design of Java and C# supporting only single class inheritance and may not be directly applicable to the languages with multiple class inheritance like Eiffel. At least the super-class types $T^{raw(R)-}$ would be useless.

An implementation of the proposal demonstrated that further extensions are required to deal with real code, in particular to access fields that have been initialized (`[Raw(except="field names")]`) and to indicate that a method initializes certain fields (`[Inits("field names")]`). Compared to the solution proposed in this chapter, the raw type approach shows the following differences:

- It requires 2 additional annotations (`[Raw]` and `[Init]`), possibly extended with additional data to indicate when fields are set and what methods set them.
- Conformance rules for raw types do not allow for creation of circular references.

The authors also identified several language-specific issues that prevent from efficient support of null-safe object initialization. One of these problems is value types that do not have default user-defined constructors. As a result, it is impossible to declare a value type with attached fields. They propose to add a new notion of a value type with an invariant that allows to define a default constructor. Because Eiffel requires expanded classes to define a default creation procedure *default_create* that is called to

```
class A create make_a feature
    name: attached STRING
    make_a (s: STRING)
        do
            name := s
            m (55)
        end
    m (x: INTEGER)
        do
        end
end
```

```
class B inherit A redefine m end
create make_b feature
    path: attached STRING
    make_b (p: STRING; s:
          STRING)
        do
            path := p
            make_a (s)
        end
    m (x: INTEGER)
        do
            io.put_string (path)
        end
end
```

Figure 5.21. A corrected example from [17].

initialize them, there is no such an issue with initialization of attributes in expanded classes.

The example used to demonstrate usefulness of raw types seems to be specific to the languages with strict order of constructor calls. Simple change in the order of assignment in the constructor solves the issue without requiring any annotations. The corrected version of the example in Eiffel is shown in figure 5.21.

Masked types. Xin Qi and Andrew C. Myers in [62] address a more general problem, not just object initialization: the complete object life cycle. They propose to instrument the type system with so called "masks" representing sets of fields that are not currently initialized. The masks are general enough to express a property that a field is initialized as soon as some other field of another object is initialized. This allows to state naturally when an object becomes completely initialized by specifying conditional masks that indicate what fields complete object initialization. For example, the notation `Node\parent!\Node.sub[l.parent] -> *[this.parent]` for an argument `l` tells that it has a type `Node` and on entry requires that its field `parent` is not set and at the same time fields declared in subclasses of `Node` are not set unless `l.parent` is initialized. On exit the actual argument conforms to

the type `Node\*[this.parent]` that indicates that the node object will be completely initialized as soon as its field `parent` is set.

The notation is very powerful and goes far beyond null safety. Even though the notation mentioned above is too complicated for daily use, authors explain that it is not sufficient for real programs. One of the issues is information hiding, when mask conditions cannot be expressed because concrete fields may differ from one concrete implementation of a general interface to another. For this reason they propose to use abstract masks that are updated in descendant classes as required. The idea seems to be similar to the data groups approach proposed by Rustan Leino in [39]. To allow for modular processing of abstract masks, subclass masks and mask constraints are introduced. In order to express initialization state abstractly, union and difference operations are defined on masks. The code in this dialect of Java called *J\Mask* is full of annotations equipped with non-trivial rules to ensure soundness. While this approach might be useful for rigorous checks of object initialization, their use seems to be problematic.

However, masked types capture one of the ideas employed in this section: flow-sensitive type system. Like with masked types, validity rule 5.2 depends on what class attributes are properly set and whether a reference to **Current** object escapes before all attributes have been set. In both cases flow-sensitive type analysis is performed without special annotations on the source code side. However, with masked types the results are finally checked against provided specifications, while in the approach proposed in my work the results are used to check whether all conditions of the validity rule are satisfied.

The work discusses two examples. Eiffel version of the first one is shown in figure 5.22. It demonstrates an initialization bug. The issue is easily detected with validity rule 5.2 and is reported for the line marked with `(*)` with the explanation that the attribute *color* is not properly set.

The second example applies masked annotations to a binary tree. The original version does not set the attribute `parent` in constructors of classes `Leaf` and `Binary`. For leafs, it is set in the constructor of the class `Binary`. For binary nodes it is set outside the constructors, demonstrating flexibility of the mechanism. With validity

```
class POINT
create make feature
    x, y: INTEGER
    make (xo, yo:
          INTEGER)
        do
            x := xo
            y := yo
            display
        end
    display
        do
            io.put_string
                (x.out + " " +
                     y.out)
        end
end
```

```
class COLORED_POINT inherit
      POINT redefine display end
create make_colored feature
    color: COLOR
    make_colored (xo, yo: INTEGER; c:
          COLOR)
        do
            make (xo, yo)
            color := c
        end
    display
        do
            io.put_string
                (x.out + " " + y.out + " " +
                color.name) -- (*)
        end
end
```

Figure 5.22. Eiffel version of buggy code from [62].

rule 5.2 all attributes have to be set in a creation procedure. The adapted version of the example is shown in figure 5.23.

Depending on whether it is allowed to have non-parented leaf nodes or not, the implementation provides two creation procedures in class *BINARY*. Creation procedure *make_root_1* supports the convention that leaf nodes should always be parented. This closely resembles behavior of the original code. Creation procedure *make_root_2* allows for creating leaf nodes before creating a binary root node. This makes the client code look similar to the original one at an expense of moving an assignment to the attribute *parent* inside the creation procedure with a different order: this attribute is now set before parent attributes of leaf nodes are updated.

Free and committed types. Alexander J. Summers and Peter Müller address 4 design goals of safe object initialization in [70]:

- Modularity – each class can be checked separately from its subclasses and clients.

```
deferred class NODE feature
    parent: NODE assign
          set_parent
    set_parent (p: NODE)
        do
            parent := p
        end
end

class LEAF inherit NODE create
    make, set_parent
feature
    make
        do
            parent := Current
        end
end

class TEST create make
feature
    make
        local
            root: BINARY
        do
            create root.make_root_1
            create root.make_root_2
                (create
                      {LEAF}.make,
                 create
                      {LEAF}.make)
        end
end
```

```
class BINARY inherit NODE
      create
    make_root_1, make_root_2
feature
    left, right: NODE
    make_root_1
        do
            create {LEAF}
                left.set_parent
                      (Current)
            create {LEAF}
                right.set_parent
                      (Current)
            parent := Current
        end
    make_root_2 (l, r: NODE)
        do
            left := l
            right := r
            parent := Current
            l.parent := Current
            r.parent := Current
        end
end
```

Figure 5.23. An adapted example of a binary tree from [62].

- Soundness – objects considered fully initialized do exhibit this property at run-time.
- Expressiveness – creation of cyclic data structures is supported.
- Simplicity – the type system is simple and has little annotation overhead.

Unlike the solution with masked types they propose to distinguish just two object states: under initialization and completely initialized. So, the object life cycle is divided into 2 stages. A newly allocated object has a so called "free" type meaning that not all object fields might be set, or, if they are set, might reference only completely initialized objects. When an object is deeply initialized, i. e., all its fields are set to deeply initialized objects, it is said to have a "committed" type. When it is unknown whether the object has free or committed type, it is considered "unclassified". By default, all variables are committed. Variables of free and unclassified types are marked with `[Free]` and `[Unclassified]` annotations respectively. The annotations do not apply to class attributes. Instead, the actual field type in a qualified call is computed using the type of the qualifier. If the qualifier has a committed type, the type of the whole expression matches the type of the field. Otherwise, it is treated as an unclassified detachable type. It is permitted to assign any expression to an attribute of a free type and to assign an expression of a committed type to an attribute of any type. In all cases the attachment status of the target should be respected.

The commitment point that logically changes the type of an object from free to committed is defined as the end of a constructor that takes only committed arguments. The soundness of this definition is based on the rule that all non-null class fields should be initialized at the end of the corresponding constructor. Because all fields of all created objects are set and the only reachable objects are the new ones and those accessible through the passed arguments, all newly created objects are deeply initialized. According to the definition of a commitment point, the special entity **`this`** has a free type until the end of the constructor where it is used. The possible aliasing between free and committed types is prevented by not having a subtyping relation between free and committed types. This differs from the convention for raw types described in [17].

The convention that **this** has a free type until the end of a constructor may require to annotate all methods that are called from the constructor using unqualified calls or are passed **this** or any of the class fields with [Free] and [Unclassified] annotations. The empirical data collected for libraries converted to complete void safety as well as results reported in [17] show that a good number of features are called from inside a constructor. (Side note: in Eiffel a program execution starts by creating an object of a root class and executing a specified root creation procedure. Therefore, the solution with free and committed types would render the root object as free all the time. All the objects that have a reference to the root object would be "infected" as well and will be considered as having an unclassified type. So, all features of the root class would have to be marked as [Unclassified], the same applies to all features where the root object is passed. This goes against the goal of the proposal to avoid annotation overhead as much as possible.)

Validity rule 5.2 is very close in spirit to the idea of free and committed types. However, it relies on a flow-sensitive analysis for class attributes and does not allow for propagating the free type status beyond the point when all attributes are set. This allows for creating cyclic data structures if needed. Lack of additional annotations does not permit sophisticated code when features can be called on incompletely initialized objects. Therefore, it might be the best to combine both approaches: use the flow sensitive type analysis to detect when **Current** becomes completely initialized and allow passing it as a free type before this moment. Fortunately, so far it was possible to refactor all code to avoid the need for handling free types in real programs. Moreover, the experimental results reported in [70] confirm that even with the proposed annotations some code refactoring is required and some feature calls have to be moves outside of constructors.

A slight variation of committed and free types is implemented in the Checker Framework [72] using annotations `@UnknownInitialization` and `@UnderInitialization` that also support type frames like `@UnderInitialization (A.class)` to tell that all fields specified in a (super-)class `A` have been initialized. In particular, the annotations are required because the tool performs only intraprocedural analysis and, unlike my solution proposed for Eiffel, fails to detect whether all attributes of a class are set in its super-class constructor. For the same reason there is no

way to use `this` in a class constructor as `@Initialized`. Authors of the Checker Framework mention that using `this` as `@Initialized` would be sound at least for final classes, but at the time of writing it is not implemented. Another deviation from the original work on free and committed types comes in adding an annotation `@NotOnlyInitialized` that should be used when a target field is assigned a value that may reference a non-completely initialized object. This makes it possible to distinguish between completely initialized and non-completely initialized fields at the cost of more annotation overhead.

The translated version of the buggy example from [70] of incorrect access to a field, performed before current object is completely initialized, is shown in figure 5.24. As expected, it is invalid according to validity rule 5.2. The compiler reports an error at line (*) telling that an attribute *g* is not set yet.

```
class C create make feature
    f, g: C
    set_f (q: C)
        do
            f := q
        end
    make (p: C)
        do
            set_f (p)
            f.set_f (Current) -- (*)
            g := p.f.g.f
        end
end
```

Figure 5.24. An Eiffel version of a buggy example from [70].

Alexander J. Summers and Peter Müller also discuss a doubly-linked list example that requires an annotation for the constructor in class `Node` used to create a sentinel. The same example in the Checker Framework additionally requires annotations `@NotOnlyInitialized` for all fields except `data`. Figure 5.25 shows the same code rewritten in Eiffel that passes validity rule 5.2 and demonstrates the ability to instantiate structures with cyclic references without any annotations.

In addition, they consider the binary tree example from [62]. The proposed version of the example is adjusted to initialize all attributes inside creation procedures like in one of the already discussed forms in figure 5.23. It corresponds to the variant that uses creation procedure *make_root_2*.

According to the authors the idea with free and committed types does not extend nicely to array types in constructors. The corresponding array objects could not be filled in with objects depend-

```
class NODE
create make, make_with_data
      feature
    parent: LIST
    prev, next: NODE
    data: detachable ANY
    make (list: LIST)
        do
            parent := list
            prev := Current
            next := Current
        end
    make_with_data (p, n:
          NODE;
        v: detachable ANY)
        do
            parent := p.parent
            prev := p
            next := n
            data := v
        end
    insert_after (v: detachable
          ANY)
        local
            n: NODE
        do
            create
                  n.make_with_data
                (Current, next, v)
            next.set_prev (n)
            next := n
        end
    set_prev (n: NODE)
        do
            prev := n
        end
end
```

```
class LIST create make feature
    sentinel: NODE
    make
        do
            create sentinel.make
                  (Current)
        end
    insert (v: detachable ANY)
        do
            sentinel.insert_after (v)
        end
end
```

Figure 5.25. A doubly-linked list example from [70] without any initialization-specific type annotations.

ing on **this** because its type inside a constructor is [Free] and only committed objects can be stored in an array. This breaks the whole scheme described in section 5.1.3.2 and used in the GUI library *EiffelVision* to create and setup widgets because widget agents are registered as event handlers in the last part of initialization process. The underlying storage for efficiency reasons is usually based on an array-like structure, and agents depend on the widget objects or the current object being constructed. The similar issue arises with the example of a mediator pattern described in section 5.1.4: use of free and committed types requires moving registration of participants outside of constructors and may lead to programming mistakes. A possible remedy of the limitation would be to take the proposed analysis of object initialization state into account and to treat the object as committed before the end of a constructor when possible.

It is worth to note that all the goals enunciated in the work about free and committed types seem to be achieved by validity rule 5.2: there is no need to check descendants and clients of a class to ensure its safety, the mechanism is sound, cyclic data structures can be created and no annotations are required. Of course, the expressiveness does not come for free: the code may need to follow certain patterns and require some refactoring to satisfy the validity rule. Fortunately, such cases are rare.

Other approaches. A different mechanism relying on so called "targeted expressions" was explored by Bertrand Meyer in [49]. Any additional type annotations are avoided by introducing a notion of creation-involved features. A feature is creation-involved if it is called from a creation procedure or from another creation-involved feature. This includes not only features declared in the classes that correspond to targets of feature calls, but also their redeclarations to take polymorphism into account. The analysis then becomes somewhat similar to the abstract interpretation approach used in [67] and should be applied to the system as a whole, thus sacrificing modularity. In other words, adding new classes to the system may break safety guarantees of previously considered safe classes, making it difficult to develop self-contained libraries. The advantage of using "targeted expressions" is in selective detection of variables that are not completely initialized. Validity rule 5.2 is too strong in this respect: it forbids any qualified calls when an incompletely initialized reference es-

capes during object creation. Targeted expressions, on the other hand, track if incompletely initialized references may be actually used as targets of qualified feature calls. The validity rule specifies that targeted expressions, i. e., expressions that could be used as targets of qualified calls, are never aliased with incompletely initialized **Current**. Compared to raw types and free-committed types, now the information sticks to all variables, not only to locals and arguments. Moreover, now every attribute can be considered targeted or not, not just the whole object. So, it is like with masked types, but with inferring annotations from the code. This should allow for more precise analysis. Unfortunately, the requirement to perform whole system analysis makes the approach difficult to apply in practice.

Rules for dispose. None of the publications discuss object reclamation that triggers a call to a finalizer and may cause all the issues usually coming out at object initialization time. Formal models do not usually cover garbage collection and therefore cannot make use of the finalizers. It is surprising that even informal discussions of various approaches in papers do not mention this problem.

5.5 Conclusion

Validity rule 5.2 has the following benefits:

No annotations. Like validity rule 5.1 it does not require any other type annotations in addition to attachment marks **attached** and **detachable**.

Flexibility. It enables creation of objects with fields of attached types mutually referencing each other.

Simplicity. The analysis requires only tracking for attributes that are not properly set, for use of **Current** and for checking whether certain conditions are satisfied when (direct or indirect) qualified feature calls are performed.

Coverage. It was possible to refactor all libraries to meed the requirements of the rule without changing design decisions, in particular without breaking separation of concerns between classes in complex cases. Also, all examples from [17, 62, 70] were rewritten in Eiffel and passed the checks

against the validity rule with the expected compilation results, including both validity error reports and successful compilations. Moreover, among reviewed initialization rules, only the proposed one and masked types [62] (with a complicated annotation system) can be used to express the design pattern from section 5.1.4 without the need to introduce helper features.

Modularity. The rule depends on properties of creation procedures from other classes. Because these creation procedures are known at compile time, the checks do not depend on classes that are not directly reachable from the one being checked. As a result it is possible to check a library as a standalone entity without the need to recheck it after inclusion in some other project.

Performance. Experiments demonstrate very moderate increase of total compilation time, below 1% on sample libraries with more than 2 millions lines of code.

Incrementality. Fast recompilation can be supported if information about reachable creation procedures and whether they perform qualified calls is recorded for every class. Provided the number of cases when **Current** is passed before all attributes are set is low, the checks for violations of the rule are very quick.

The main drawbacks of the rule are:

Certain coding pattern. Like with validity rule 5.1 the code still has to follow certain coding guidelines in terms of initialization order. In particular, the rule may forbid qualified calls even on completely initialized objects when the current object is not completely initialized. This may require to refactor code by moving these calls before or after initialization phase depending on whether the results of the calls are used for initialization itself.

Disallowing legitimate qualified calls. It is unnecessary for a qualified call to leak a reference to an uninitialized object. Lack of additional annotations does not allow for distinguishing between legitimate and non-legitimate qualified calls. In order to preserve soundness all qualified calls are considered as potentially risky.

Special convention for formal generics. As described in section 5.2.2.4 if a target type of a creation expression is a

formal generic parameter, special convention should be used to indicate whether a creation procedure of an actual generic parameter satisfies the validity rule requirements. It is unclear whether the observation that passing incompletely initialized **Current** indicates tight coupling between classes implies that use of formal generics in such situations is improbable. At least this was not an issue so far.

None of the previous work considered object disposal. This is now covered:

Validity rule for finalizers. Validity rule 5.4 specifies the conditions that ensure void safety at object's disposal time.

Further improvements. The issue with disallowing legitimate qualified calls in validity rule 5.2 can be alleviated by noticing that the issue with an incompletely initialized object realizes only when the reference is passed somewhere else where it is not controlled anymore. The current rule assumes that access to **Current** implies leaking a reference to it. Similarly, if **Current** is passed to some creation procedure before all attributes are properly set, it is assumed that the newly created object cannot be used in a qualified call because it can leak a previously passed reference to current object. The rule makes sure no aliases to the current object exist. However, it does not take into account the fact that not every access to **Current** leads to creation of an alias. More specifically, not every access to **Current** is involved in object reattachment. In particular, reference equality does not create new aliases. So, the rule can be relaxed by making sure **Current** or a qualified call are not used in object reattachment and are not part of a nested qualified call.

More qualified calls can be allowed by introducing a notion of **immediately-initialized** types that are similar to self-initializing types in terms of guarantees of their initialization. These are expanded or attached frozen types that may rely on explicit creation procedure calls to initialize associated class attributes, but none of these attributes are of a reference type or have (recursively) attributes of a reference type. For example, all basic types like *INTEGER* and *BOOLEAN* fall in this category. Because objects of immediately-initialized types do not reference any other objects, there is also no way for them to reference an incompletely initialized object. As a result, a qualified call that involves only

immediately-initialized types for its target and arguments cannot access an incompletely initialized object and is safe.

6

Certified Attachment Patterns

6.1 Overview

Existing proposals ([6, 51, 70]) that address void safety issues in languages supporting null references use a type system extended with a notion of detachable and attached types for expressions that may and may not produce a null value. This extended type system is then uniformly applied to the language (e. g., [12]) meaning that types of variables are specified explicitly. The type information is then used to check reattachment validity rules. For example, an assignment $x := y$ would be valid only when the type of y conforms to the type of x. If the type of x is attached, the type of y should be attached as well.

This rule makes perfect sense for class attributes that can be accessed in different features. Type information is essential in that case because objects can be aliased at run-time and it would be impossible to do type checks at compile time modularly. However, for local variables there is no aliasing, or, more precisely, locals can be changed only in a current feature. As a result it should be possible to get rid of type annotations altogether: certified attachment patterns (CAPs) are absolutely sufficient for local variables and attachment annotations can be safely discarded.

Moreover, CAPs allow for accepting code that is rejected by type-based rules. For example, consider the following declaration of a feature *color*:

```
color: COLOR
        -- Color used for painting.
    do
        Result := user_selected_color
        if not attached Result then
            Result := default_color
        end
    end

user_selected_color: detachable COLOR
        -- Color selected by a user (if any).

default_color: COLOR
        -- Color used when user did not select a
            color.
    once
        ...
    end
```

The type of **Result** in *color* is attached (there is always some color to be used for painting). However, a user could have picked a color (*user_selected_color* is attached) or not yet (*user_selected_color* is **Void**). The assignment

```
Result := user_selected_color
```

violates type rules because a detachable type (declared for *user_selected_color*) does not conform to an attached type (declared for **Result**). On the other hand, the code is absolutely void-safe, because after the assignment it checks whether **Result** is attached or not. If it is not attached, a default color is used with a value known to be attached. There are similar examples for local variables of an attached type. It is possible to write some boilerplate code to make the code null-safe relying only on type rules. For example, the code above can be changed into

```
color: COLOR
        -- Color used for painting.
    do
        if attached user_selected_color as c then
            Result := c
        else
            Result := default_color
        end
    end
```

This introduces a new variable *c* and therefore increases the overall code complexity. With several detachable expressions the number of temporary variables will grow proportionally. The increased complexity can lead to errors, not to mention that programming with void safety in mind becomes tedious, requires more code and prevents from keeping some legacy code unchanged just to please a type checker.

A certified attachment pattern addresses this issue by bridging the gap between detachable and attached types with a set of rules that permit expressions of a detachable type to be used in the context when only attached types are allowed. The rules for them were introduced in [12, 27] for read-only entities and had limited expressiveness.

For example, certified attachment patterns described in [12, 27] treat every boolean connective separately from each other: their combinations or nesting are not supported. Even though it might be a good practice to avoid complex expressions and to replace them with short and simple ones, when the first version of the compiler supporting void-safety was released, some users complained about missing cases. Moreover, complex expressions might be useful when code is not written by a human but is generated automatically – then it can be arbitrary complex. This work addresses the demand by replacing arbitrary boolean connectives with conditional expressions and specifying CAPs in terms of these expressions. Consequently, expressions of any complexity or nesting can be supported.

Although simple branches and loop conditions were taken care of by the original CAPs, the rules did not cover loop bodies. It was impossible to say whether a variable was attached or not after a loop and therefore it was considered detachable to preserve soundness. Similar-looking analyses like definite assignment re-

quired by Java [21, 22] are specified in a form that can be implemented in one pass over source code because on every iteration a variable can only become assigned, not the reverse. This is not the case for void safety. A local variable can become attached or void on different iterations, or even to flip-flop on every iteration from attached state to detached and back as shown in the example in figure 6.1.

```
from
    x := something_attached
    y := Void
until
    whatever
loop
    tmp := y
    y := x
    x := tmp
end
x.foo
```

Figure 6.1. An example of an issue with loop CAPs.

If safety checks rely only on type declarations and x is of a detachable type, there are no guarantees that it will be attached after the loop (the original rules are quite pessimistic). As demonstrated by this work, the rules for loops could be based on fixed-point computation to meet program developer's expectations. At the same time thanks to monotonicity properties they remain computable and efficiently verifiable by a compiler.

A set of CAPs specified in [12, 27] ensures void safety, but cannot be used in practice for any large scale application without provisioning for rules to escape void safety checks. It is just practically impossible to write 285 (or any other number of) classes in one go without intermediate compilation and testing. If at some point a feature is returning a value of a deferred class (an abstract class or an interface in Java/C# terms) and there are no effective descendants of this class yet, the program will not compile. The solution adopted in [15] is to rely on exceptions, including forced checks of assertions. This triggers so called "design mode" when the compiler ignores attachment status in the type checks.

Among other novelties of the thesis is specification of void safety rules and program semantics with attachment properties in a proof assistant environment. The model and its soundness is mechanically checked with *Isabelle 2016*. Such proofs do not leave a space for missing cases or fuzzy specifications and are an important step towards safe programming languages and safe software in general.

All presented safety rules, though in a form, adapted to specifics of Eiffel, have been implemented in a validity checker that allowed for comparing quantitatively type-based and CAP-based implementations. Benchmarks demonstrate 33% lower error rate when using CAP-based rules. This serves as an indicator of their superiority against type-based rules because they allow for more code to be accepted without compromising void safety.

6.2 Attachment state

At every execution point a variable may be in one of two states: attached or detachable. This applies to both compile time and run time of a program. In order to avoid voidness checks at run-time, a language should specify rules that, when satisfied at compile time, guarantee that an expression considered attached at compile time always produces an attached value at run-time. The values produced by expressions depend solely on a state of a system. The state, in turn, consists of states of all variables. Therefore, attachment state of a program can be modeled by a set-like structure that allows telling for every variable in a current scope of program analysis whether it is attached or not.

The reason why sets cannot be used directly for this purpose is explained in section 6.2.4 where the detailed specifications of the operations are provided. For the time being we can look at the attachment status of variables as on a set with the following operations:

- Union: $A \sqcup B$
- Intersection: $A \sqcap B$
- Insertion: $A \oplus x$
- Removal: $A \ominus x$
- Membership test: $x \in^{\top} A$

Union and intersection are familiar set-theoretical operations. Insertion and removal include and exclude a specific variable from a set of attached variables. And membership test is used to test whether a variable is attached in the given attachment state.

6.2.1 *Abstract syntax*

Some languages, including C# [26], Eiffel [12, 27], Java [21, 22] distinguish between constructs that produce a value at runtime, known as expressions, and that do not produce a value at runtime, known as statements or instructions. This division is convenient for language specification and for human readers but breaks uniformity of language structure and complicates description of the language in formal systems such as Isabelle/HOL.

Both can be described uniformly using a special return value `unit` from functional languages and denoting a singleton used for constructs without a value. This is a standard technique used for formalization of OOP languages (e. g., see Java-like language formalizations in [32, 41, 57]).

The abstract language syntax is modeled in Isabelle/HOL with appropriate constructors of a datatype *expression* (figure 6.2). In most cases there is one-to-one relation between source language and Isabelle/HOL terms with two important points of divergence: one for voidness tests and the other one for operator expressions.

6.2.2 *Scopes*

6.2.2.1 *Voidness tests*

The notion of an attached variable scope is orthogonal to the notion of attachment status in a sense that it does not depend on the value of a variable known at compile time but on the execution pattern that involves an explicit check whether the variable is attached or not. In simple cases the test is a comparison to a predefined value **Void** signaling that there is no associated object. They are called **voidness tests**. Some variants of the tests are listed in table 6.1.

The first one is based on a regular equality operation and a predefined constant denoting absence of an object. Given that there are multiple forms of equality ([46] mentions reference equality, shallow equality, deep equality for non-concurrent programs, [56] discusses other forms that take into account object placement with respect to active execution units for programs with structured concurrency), any form of equality tests falls into a category of

B.6.1
p. 182

expression =	
Value value |	– Value (constant)
Local name |	– Local variable
expression ;; expression |	– Sequence
name ::= expression |	– Assignment
create name |	– Creation instruction
expression.name (expression list) |	– Feature call
if expression	
then expression	
else expression *end* |	– Conditional
until expression *loop* expression *end* |	– Loop
attached type expression *as* name |	– Object test
Exception	– Exception

Figure 6.2. Datatype *expression*.

Table 6.1. Voidness tests.

Meaning	Eiffel Syntax		
	Equality		Object test
	Reference	Value	
Is expression attached?	x /= **Void**	x /~ **Void**	**attached** x
Is expression detached?	x = **Void**	x ~ **Void**	**not attached** x

suitable voidness tests as soon as they can tell whether a given expression is associated with an object or not. Therefore, possible voidness tests from table 6.1 can be extended with the standard library calls:

deep_equal (*x*, **Void**)	*deep_equal* (**Void**, *x*)
equal (*x*, **Void**)	*equal* (**Void**, *x*)
standard_equal (*x*, **Void**)	*standard_equal* (**Void**, *x*)

They are called **library voidness tests**.

Contemporary Eiffel specification [12, 27] defines certified attachment patterns using voidness tests listed in table 6.1. They are direct, i. e., compare expressions with **Void**. An alternative approach is to compare expressions with values that are known to be attached. They could be constant expressions (numbers, string literals, truth values) or arbitrary expressions that are definitely attached at run-time. Such **non-voidness tests** are less obvious and rare, but they can still appear in code and can be used to deduce attachment status of a variable. One of the most useful tests of this kind are equality tests for strings, e. g.:

$$s \sim \text{"-option\_a"} \textbf{ or } s \sim \text{"-option\_b"}$$

If the test above passes, then string s is attached.

Unlike comparisons to **Void**, non-voidness tests are limited to equality operators because the second operand has an associated value and inequality to this value does not mean the first operand is detached, it can just have a different value. Therefore, non-voidness tests are equivalent in terms of attachment status to voidness tests of the form *"Is expression attached?"* from table 6.1.

The key difference between the two forms of tests can be demonstrated by the following two statements:

$$\begin{aligned} &\forall x.(\mathbf{V}\ x \iff \textbf{attached}\ x) \wedge \forall x.(\textbf{not attached}\ x \iff \textbf{not V}\ x) \\ \vee\ &\forall x.(\mathbf{V}\ x \iff \textbf{not attached}\ x) \wedge \forall x.(\textbf{attached}\ x \iff \textbf{not V}\ x) \end{aligned}$$

$$\forall x.(\mathbf{N}\ x \implies \textbf{attached}\ x) \wedge \forall x.(\textbf{not attached}\ x \implies \textbf{not N}\ x)$$

where **V** stands for a voidness test and **N** stands for a non-voidness test. Non-voidness tests are weaker compared to voidness tests because they give only half of the guarantees. However, they still can be used in CAPs.

Similar to regular voidness tests, non-voidness tests can be expressed with library calls where one of the operands is known to be attached while the other one is checked for equality to its counterpart, for example, *equal* (*x*, *attached_expression*). The corresponding CAP is again limited to the case when equality is established. If the test evaluates to **False** there is no information whether this happens because there is no object attached to *x* or because this object is different from the value returned by *attached_expression*.

Non-voidness tests are not covered explicitly in my Isabelle/HOL formalization for clarity. However, they are similar to the case of object tests with explicit type specification (see section 6.2.2.2).

6.2.2.2 *Object test*

The most general form of an object test has 3 parts: a type, an expression and an object test variable:

attached {*SOME_TYPE*} *expression* **as** *my_variable*

The type is used to determine whether an expression is attached to an object of a type that conforms to the given one. If this is the case then the expression value is attached to the variable *my_variable* and the object test evaluates to **True**. Otherwise, it evaluates to **False**. The key observation here is that if the object test succeeds, both *expression* and *my_variable* are attached. Therefore, the type part of object test is irrelevant in most of the following discussion. However, it is worth to note that when the type part is present, object test behaves like a non-voidness test described in section 6.2.2.1, and when the type part is absent, it behaves like a regular voidness test. So the test *expression* /= **Void** is translated into

attached None expression *as* unique_variable

where *unique_variable* is a unique name not used anywhere else in the code.

The optional type part is reflected in the formal semantics of the object test expression (see section 7.3).

The object test local declaration is optional and can be omitted. The omission is useful when the value of the expression is not used, or when the expression itself is a variable. While the variable is unchanged, it can be safely used without the need for an

additional object test local. Talking about attachment status of variables the cases ([12, 27]) that affect it include

- only *Expression* part is present and it is a variable
- *Object_test_local* part is present and *Expression* is not a variable
- *Object_test_local* part is present and *Expression* is a variable

In the first two cases the set of attached variables is extended with a single element. In the last case it is extended with two elements.

Because it would be cumbersome to formalize all possible forms of the object test construct, the most general form is used in the model. Then the optional object test local part can be modeled by using an identifier that does not appear anywhere in source code.

6.2.2.3 *Scope kinds*

To get some feeling of a variable scope, consider the following code fragment:

```
if attached x then
    ... -- (1)
else
    ... -- (2)
end
```

Regardless of a state of variable x, after the voidness test, it is known to be attached in compound (1) and detached in compound (2).

The effect of voidness tests is not limited to constructs that expect a boolean value used to decide which execution path to take at run-time. They can influence attachment status of a variable inside an expression when used together with semistrict operators. These boolean operators lead to evaluation of only first operand and can perform or skip evaluation of the second operand depending on the value of the first one. In Eiffel these operators are **and then**, **or else** and **implies**.

For example, in the expression

```
attached x and then expression
```

1	**if** **attached** *x*	**and**	...	**then** ...	**else**	...	**end**
	if ...	**and** [**then**]	**attached** *x*	**then** ...	**else**	...	**end**
2	**if** **attached** *x*	**and then**	...	**then** ...	**else**	...	**end**
3	**if** **not attached** *x*	**or**	...	**then** ...	**else**	...	**end**
	if ...	**or** [**else**]	**not attached** *x*	**then** ...	**else**	...	**end**
	if ...	**implies**	**not attached** *x*	**then** ...	**else**	...	**end**
4	**if** **not attached** *x*	**or else**	...	**then** ...	**else**	...	**end**
	if **attached** *x*	**implies**	...	**then** ...	**else**	...	**end**

Figure 6.3. Scope combinations (code fragments where variable *x* is considered attached are marked with ...).

the variable *x* can be safely used in *expression* because if the first operand of the operator **and then** evaluates to **True** the variable *x* is attached. On the other hand, if the first operand gives **False**, the value of the expression as a whole is **False** as well and the second operand is never evaluated. Therefore, even if the second operand depends on *x*, it is safe to reference it. Moreover, if the second operand is evaluated, the variable *x* inside it is always attached.

A source code snippet where a variable is considered attached because of an associated voidness test for this variable is called an attachment scope of this variable. The two examples above correspond to two mechanisms to make scopes:

1. **Control flow scope** – an attachment scope based on language constructs that change execution flow.
2. **Operator scope** – an attachment scope based on semistrict operators.

In practice both kinds of attachment scopes are applied together. An exhaustive list of scope combinations involving at most one unary and at most one binary boolean operator in a conditional instruction is given in figure 6.3.

The third variant of a scope – **assertion scope** – is used in assertions where assertions clauses of the same assertion are logically combined using **and then** operator.

6.2.2.4 *Generalization of scopes*

The language standard [12, 27] specifies scopes of object test locals in terms of instructions and boolean operators, there are 8 clauses in total: 3 for expressions, 2 for conditional instructions and expressions, 1 for loops and 2 for assertion clauses. It might be tempting to mimic the rules in the logical framework and then to prove that they are sound. But this approach has several drawbacks:

- The formalization would be limited to the selected set of boolean operators. Applying results to another language with a different set of boolean operators would not be straightforward if some operators of that other language are not covered.
- There are 3 semistrict boolean operators, 2 regular operators and one unary operator. Adding them to the formalization would mean either addition of 6 new constructors to the datatype *expression* (figure 6.2) or addition of *Binary* and *Unary* constructors with accompanying datatypes for operators (like in [33]). In both cases all induction-based proofs would have to be performed for the new constructors.
- There is already some redundancy in the current operators because some of them can be expressed in terms of others using, for example, properly adapted De Morgan's laws.
- The rules as specified in the standard are not general enough and do not allow for deeper analysis of expressions. For example, they do not cover code like **if not not attached** *x* **then** ... **end** but cover its equivalent **if attached** *x* **then** ... **end**.

Generalization can be done with just 3 variants of expression: truth constants, conditional expressions and sequences. It is based on the observation that every boolean expression can be translated into a conditional expression with nested boolean constants and optional sequences. It is tightly coupled to semantics of the operators. Consider conjunction as an example in both forms, semistrict and regular.

Informally a semistrict conjunction *e1* **and then** *e2* is evaluated as follows:

1. Evaluate *e1*.

2. If result of this evaluation is **False**, use it as a value of the expression.
3. Otherwise, evaluate *e2* to get the value.

This algorithm can be written as

```
if e1 then
    e2
else
    False
end
```

With such a replacement the following two code fragments are equivalent:

```
if
    attached x and then
    attached y
then
    x.foo
    y.bar
end
```

```
if
    if attached x then
        attached y
    else
        False
    end
then
    x.foo
    y.bar
end
```

The algorithm for a regular conjunction *e1* **and** *e2* is a bit more complicated:

1. Evaluate *e1*.
2. Evaluate *e2*.
3. If either result is **False**, use it as a value of the expression.
4. Otherwise, use **True** as a result.

In the current formulation the algorithm requires temporary variables to be used in a conditional expression. In order to avoid it, it can be recast as follows:

1. Evaluate *e1*.
2. If result of this evaluation is **False**, evaluate *e2*, but use **False** as a value of the expression.
3. Otherwise evaluate *e2* to get the value.

This can be written as

```
if e1 then
    e2
else
    e2.do_nothing; False
end
```

Here *do_nothing* is a standard library feature that has no effect and serves as a no-op to consume a value produced by *e2*. In the languages that do not have a rule that elements of a sequence should be instructions (that should not produce any value) rather than expressions (that produce some value) this call is redundant and can be omitted. Also, note that even if a language does not allow for compound expressions (i. e., expressions formed from a sequence of nested expressions), it can be easily achieved by using specially crafted functions. In the example above the function would return a value of its second argument. The convention to omit explicit calls to such functions that simply return value of their last argument is used throughout the later discussion unless specified otherwise.

Applying the described conversion the following code fragments are equivalent:

```
if
    attached x and
    attached y
then
    x.foo
    y.bar
end
```

```
if
    if attached x then
        attached y
    else
        attached y; False
    end
then
    x.foo
    y.bar
end
```

Conversions for the boolean operators mentioned earlier and some others added for completeness are listed in figure 6.4. Following the terminology used in [12, 27] they are called **unfolded forms of boolean operators**.

Operator	Original expression	Translation	
Negation	**not** *e*	**if** *e* **then False else True end**	(2)
Conjunction	*e1* **and then** *e2*	**if** *e1* **then** *e2* **else False end**	(3)
	e1 **and** *e2*	**if** *e1* **then** *e2* **else** *e2*; **False end**	(4)
Disjunction	*e1* **or else** *e2*	**if** *e1* **then True else** *e2* **end**	(5)
	e1 **or** *e2*	**if** *e1* **then** *e2*; **True else** *e2* **end**	(6)
Implication	*e1* **implies** *e2*	**if** *e1* **then** *e2* **else True end**	(7)
	not *e1* **or** *e2*	**if** *e1* **then** *e2* **else** *e2*; **True end**	(8)
Converse nonimplication	**not** *e1* **and then** *e2*	**if** *e1* **then False else** *e2* **end**	(9)
	not *e1* **and** *e2*	**if** *e1* **then** *e2*; **False else** *e2* **end**	(10)

Figure 6.4. Unfolded forms of boolean operators.

It turns out that all unfolded forms of boolean operators are variants of the following patterns:

if *x* **then** *y*; *Const* **else** *z* **end**
if *x* **then** *y* **else** *z*; *Const* **end**

where *Const* is either **True** or **False**. So, instead of reasoning in terms of various forms of boolean operators and their combinations it is sufficient to reason in terms of special forms of conditional expressions. This approach does not only go beyond single-level scope definitions, but also allows for ternary operations in addition to unary and binary ones.

The special form of the branches ending with a boolean constant is captured by two functions defined in Isabelle/HOL as:

$$\begin{aligned} is\_false\ (c\ ;;\ False_c) &= True & \qquad is\_true\ (c\ ;;\ True_c) &= True \\ is\_false\ \_ &= False & is\_true\ \_ &= False \end{aligned}$$

B.6.3 p. 183

The cases from figure 6.4 when there is no expression followed by a constant, but just a constant **False** or **True** in a branch are represented by sequences *unit ;; False*$_c$ or *unit ;; True*$_c$ respectively. It would be possible to handle constants **False** and **True** directly, however it would just add one more case in the function definitions without any additional benefit.

The functions *is_false* and *is_true* can be also generalized by adding other variants of expressions that knowingly produce fixed boolean constants, for example

$$\begin{aligned} &\mathit{is\_true}\ (\mathit{if}\ \mathtt{b}\ \mathit{then}\ \mathtt{e}_1\ \mathit{else}\ \mathtt{e}_2\ \mathit{end}) \\ &\qquad = (\mathit{if}\ (\mathit{is\_true}\ \mathtt{b})\ \mathit{then}\ \mathit{is\_true}\ \mathtt{e}_1) \\ &\qquad \vee (\mathit{if}\ (\mathit{is\_false}\ \mathtt{b})\ \mathit{then}\ \mathit{is\_true}\ \mathtt{e}_2) \\ &\qquad \vee (\mathit{is\_true}\ \mathtt{e}_1 \wedge \mathit{is\_true}\ \mathtt{e}_2) \end{aligned}$$

This and other more complicated cases, however, are covered by optimization and code transformation techniques familiar from compiler technology, such as common sub-expression elimination, constant propagation, invariant code motion and others ([1, 53, 54]).

6.2.3 *Transfer function*

A variable can be attached to an object at run-time or be `null`. To track its status at compile time, we need to compute sets of variables that are always attached at particular execution points. This is done with a transfer function $A \rhd e$ that gives a set of attached variables for an expression e from the set of variables A that are known to be attached before e. It is defined inductively by 5 mutually recursive functions:

- $\cdot \rhd \cdot$ the transfer function itself (figure 6.5)
- $\cdot \rhd + \cdot$ $\cdot \rhd - \cdot$ computes a set of attached variables with an assumption that the expression evaluates to true/false (positive/negative scope) (figure 6.8)
- $\cdot \rhd\rhd \cdot$ computes a set of attached variables for a given list of expressions (used to model arguments in feature calls) (figure 6.7)
- $\cdot \hookrightarrow \cdot$ tells if a given expression is attached (figure 6.6)

The first argument of all functions has a type *topset* that is similar to a conventional set, but has an additional value describing an unreachable state (section 6.2.4). It corresponds to a set of attached variables before an expression e is evaluated. The second argument for all functions has a type *expression*, except for the function $\rhd\rhd$ that takes a list of expressions *expression list* instead. All functions return sets of attached variables of the type *topset*

$$
\begin{aligned}
A \rhd \mathit{Value}\ v &= A \\
A \rhd \mathit{Local}\ n &= A \\
A \rhd e_1\ ;;\ e_2 &= A \rhd e_1 \rhd e_2 \\
A \rhd \mathit{create}\ n &= A \oplus n \\
A \rhd \mathit{attached}\ t\ e\ \mathit{as}\ n &= A \rhd e \\
A \rhd \mathit{Exception} &= \top \\
A \rhd n ::= e &= \begin{cases} (A \rhd e) \oplus n & \text{if } A \hookrightarrow e \\ (A \rhd e) \ominus n & \text{otherwise} \end{cases} \\
A \rhd e\ .\ f\ (a) &= A \rhd e \rhd\rhd a \\
A \rhd \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} &= A \rhd{+}\ c \rhd e_1 \sqcap A \rhd{-}\ c \rhd e_2 \\
A \rhd \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end} &= A \rhd{*}\ (-\ e \rhd b) \rhd{+}\ e
\end{aligned}
$$

B.18.2 p. 217

Figure 6.5. Transfer function.

after the expression or the expression list are evaluated, except for the last function $\hookrightarrow$ returning a boolean value.

6.2.3.1 *Regular cases*

Accessing a value or a local variable does not change a set of attached variables (figure 6.5).

For a sequence, the result is computed as an application of the transfer function to the second expression with a set of attached variables obtained for the first expression (the operator $\rhd$ is left-associative).

For a creation instruction it always adds an associated variable to the set of attached variables because the instruction attaches a newly created object to the variable at run-time.

For an object test used in a non-branching construct, the attachment set is the one obtained for its expression. A different transfer function is used in a context when an object test value is known to be *True* (section 6.2.3.2).

For an assignment, a variable is added to a set of attached variables after the assignment if the source expression is attached and is removed from the set otherwise.

B.18.2 p. 217

$$
\begin{aligned}
A \hookrightarrow \mathit{Value}\ v &= v \neq \mathit{Void}_v \\
A \hookrightarrow \mathit{Local}\ n &= n \in^{\top} A \\
A \hookrightarrow \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} &= A \rhd\!+ c \hookrightarrow e_1 \wedge A \rhd\!- c \hookrightarrow e_2 \\
A \hookrightarrow \_ &= \mathit{True}
\end{aligned}
$$

Figure 6.6. Attachment status function.

B.18.2 p. 217

$$
\begin{aligned}
A \rhd\!\rhd [] &= A \\
A \rhd\!\rhd (e \cdot es) &= A \rhd e \rhd\!\rhd es
\end{aligned}
$$

Figure 6.7. Transfer functions for argument lists.

The definition for an assignment instruction uses a function computing an attachment status of an expression. It returns *True* if its argument is a value other than *Void*, a local in the set of attached variables, or a conditional expression with both branches attached (figure 6.6). Note that an attachment status of a conditional branch takes into account whether the branch is positive or negative. In other words the attachment status of the positive branch e_1 is computed with an assumption that the condition c evaluates to *True*, the attachment for the negative branch e_2 – with the assumption that c evaluates to *False*, and then their conjunction is used to determine whether the whole expression is attached.

Arguments of a call are subject to chained processing (figure 6.7). The transfer function for an argument is evaluated in the context of a previous one or in the context of a target (for the first argument).

A simplified version of the function that does not chain arguments would fail to capture attachment properties for the following code:

```
x.foo (if attached a then x else failed end, a.bar)
```

where *failed* has a postcondition **False**, and thus never returns normally. If the argument *a.bar* would be checked without taking attachment status of *a* into account, it would be flagged as

erroneous. Given that evaluation of *a.bar* is possible only when previous argument evaluated successfully (arguments in Eiffel are computed left-to-right), the target *a* is attached according to the current transfer function definition and does not cause any rule violations.

The rules for an expression list (figure 6.7) model arguments. Arguments of a call are subject to chained processing even though it might seem unnecessary. The reason is that an attachment status of an object test local could be affected by the rules for "design mode" (section 6.2.4). The transfer function for an argument is evaluated in the context of a previous one or in the context of a target (for the first argument). The same effect can be achieved by using the Isabelle/HOL function *fold*:

$$A \rhd\rhd es = \mathtt{fold}\ (\lambda\ e\ X.\ X \rhd e)\ es\ A$$

According to [33] for subsequent proofs it is more convenient to use the direct definition of the transfer function rather than the one based on *fold*. This work adopts the same approach.

6.2.3.2 *Branching*

Transfer functions for positive and negative scopes (figure 6.8) differ from the regular transfer function by taking into account a boolean value produced by the expression used in the condition. The only expressions of the model language that can produce a boolean value and affect resulting set of attached variables are object tests and conditional expressions.

If an object test evaluates to *True* (positive scope function), the corresponding object test variable n is attached. Moreover, if the object test expression is itself a variable n', it is also known to be attached. So, the resulting set includes n and, in the second case, n'.

Two other definitions cover general conditional expressions. The positive and negative scope functions recursively depend on each other. If a conditional expression does not evaluate to a boolean constant in at least one of its branches, nothing can be said about attachment status of nested object test locals. The reason is that the information whether an object test succeeded, be it a conditional expression c or one of the branch expressions e_1 or e_2, is lost in that case.

B.18.2 p. 217

$$A \rhd+ \textit{attached } T \textit{ Local } n' \textit{ as } n = A \oplus n' \oplus n \tag{11}$$

$$A \rhd+ \textit{attached } T\ e \textit{ as } n = A \rhd e \oplus n \tag{12}$$

if e is not a local

$$A \rhd+ \textit{if } c \textit{ then } e_1 \textit{ else } e_2 \textit{ end} = \tag{13}$$

$$= \begin{cases} A \rhd- c \rhd+ e_2 & \text{if } \textit{is\_false } e_1 \\ A \rhd+ c \rhd+ e_1 & \text{if } \textit{is\_false } e_2 \\ A \rhd \textit{if } c \textit{ then } e_1 \textit{ else } e_2 \textit{ end} & \text{otherwise} \end{cases}$$

$$A \rhd- \textit{if } c \textit{ then } e_1 \textit{ else } e_2 \textit{ end} = \tag{14}$$

$$= \begin{cases} A \rhd- c \rhd- e_2 & \text{if } \textit{is\_true } e_1 \\ A \rhd+ c \rhd- e_1 & \text{if } \textit{is\_true } e_2 \\ A \rhd \textit{if } c \textit{ then } e_1 \textit{ else } e_2 \textit{ end} & \text{otherwise} \end{cases}$$

$$A \rhd+ e = \begin{cases} \top & \text{if } \textit{is\_false } e \\ A \rhd e & \text{otherwise} \end{cases} \tag{15}$$

$$A \rhd- e = \begin{cases} \top & \text{if } \textit{is\_true } e \\ A \rhd e & \text{otherwise} \end{cases} \tag{16}$$

Figure 6.8. Transfer functions for positive and negative scopes.

Let's consider one of the cases when a branch expression meets a condition to produce a known constant value, for example, *is_false* e_1. This is a rule for the positive scope function $A \triangleright+$ *if* c *then* e_1 *else* e_2 *end*. The resulting set corresponds to the case when the conditional expression evaluates to *True*. From the condition *is_false* e_1 we know that c could not have been evaluated to *True*. Also, we know that the only case to get *True* for the whole expression is to get *True* for e_2. Therefore, the set of attached variables in this case is a result of application of the positive scope function for e_2 to a result of application of the negative scope function for c to the initial set A.

Other cases can be explained similarly. As an example let's see how the rules work for double negation:

$$
\begin{aligned}
& A \triangleright+ \textbf{not not attached } x \\
= {} & A \triangleright+ \textbf{if not attached } x \textbf{ then False else True end} && \text{by (2)} \\
= {} & A \triangleright- \textbf{not attached } x \sqcup A \triangleright+ \textbf{True} && \text{by (13)} \\
= {} & A \triangleright- \textbf{not attached } x && \text{by (15)} \\
= {} & A \triangleright- \textbf{if attached } x \textbf{ then False else True end} && \text{by (2)} \\
= {} & A \triangleright+ \textbf{attached } x \sqcup A \triangleright- \textbf{False} && \text{by (14)} \\
= {} & A \triangleright+ \textbf{attached } x && \text{by (16)} \\
= {} & A \oplus x && \text{by (11)}
\end{aligned}
$$

What if for a given conditional expression both *is_false* e_1 and *is_false* e_2 would give *True*? Would the positive scope function yield a consistent result? From *is_false* e_1 and *is_false* e_2 it follows that the whole conditional expression evaluates to *False*. This contradicts the assumption that the positive scope function is computed with the assumption that the whole expression evaluates to *True*. Therefore, such inconsistency is impossible.

6.2.3.3 *Loops*

For a loop the transfer function is specified using a loop operator. The loop body is evaluated in the negative branch of the exit condition and the effect of the loop as a whole is evaluated in the positive branch of the same condition. The loop operator for a loop with an exit condition e and a loop body b is defined as a greatest fixed point:

$$A \triangleright* (- e \triangleright b) \equiv gfp\ (\lambda X.\ A \sqcap X \triangleright- e \triangleright b) \qquad \begin{matrix} B.17 \\ p.\ 213 \end{matrix}$$

It is known that a greatest fixed point of a monotone function defined on a complete lattice exists (Tarski proved it in [71]) and can be computed by iterating over the result until it stabilizes (the theorem is often referred to as *Kleene's fixed point theorem*, but the exact authorship is unclear [38]). This ensures termination of the function in a static analyzer. The type *topset* forms a complete lattice (section 6.2.4.1), so it remains to prove monotonicity of a loop function. Instead of proving lemmas with a specific loop function, a generalized version is used: $loop\_operator\ f\ A \equiv gfp\ (\lambda x.\ A \sqcap f\ x)$. The loop function $\lambda\ A\ x.\ A \sqcap f\ x$ is monotone on both arguments, but we need monotonicity only on the last one:

B.17 p. 213 **Lemma 6.1** (Loop function monotonicity)**.**
$mono\ f \implies mono\ (\lambda X.\ A \sqcap f\ X)$

The loop operator is monotone and idempotent on both arguments:

B.17 p. 213 **Lemma 6.2** (Loop operator monotonicity)**.** $mono\ (loop\_operator\ f)$

Proof. From monotonicity of greatest fixed point. □

B.17 p. 213 **Lemma 6.3** (Loop operator unfolding)**.**
$mono\ f \implies loop\_operator\ f\ A = loop\_function\ f\ A\ (loop\_operator\ f\ A)$

B.17 p. 213 **Lemma 6.4** (Loop operator idempotence)**.**
$mono\ f \implies loop\_operator\ f\ (loop\_operator\ f\ x) = loop\_operator\ f\ x$

The shape of the loop function is derived from the following algorithm that can be used to compute sets of attached variables:

$$
\begin{aligned}
&A_{prev} := \top;\\
&\textbf{while}\ (A_{prev} \neq A)\\
&\{\\
&\quad A_{prev} := A_{prev} \sqcap A;\\
&\quad A := A_{prev} \rhd\!\!- e \rhd b;\\
&\}
\end{aligned}
$$

On every iteration the current set of attached variables is compared to the previous one. If they are the same, the greatest fixed point is reached and can be used for subsequent analysis. If they are different, the intersection of two sets is used as a previous attachment set and a new attachment set is computed for a single loop iteration. This generates a monotone chain A, $A \sqcap f\ A$,

$A \sqcap f\,A \sqcap f\,(A \sqcap f\,A)$, ..., where $f = \lambda\, X.\ X \rhd\!\!- e \rhd b$. The chain corresponds to the loop function $\lambda\, A\ x.\ A \sqcap f\,x$.

Detection of an infinite loop that indicates unreachable code (section 6.2.4) is captured indirectly by (15) where the loop exit condition meets *is_false* predicate. In this case, the transfer function for the loop gives $\top$.

As one would expect, an application of a loop operator produces a smaller set of attached variables:

Lemma 6.5. $mono\ f \implies loop\_operator\ f\ A \leqslant A$ *B.17 p. 215*

$loop\_operator\ f\ A \leqslant loop\_operator\ f\ (f\ A)$ *B.17 p. 215*

Using monotonicity of the loop operator and the loop function itself (lemmas 6.1 and 6.2) and the definition of the transfer function, the latter can be proved to be monotone:

Lemma 6.6 (Transfer function monotonicity)**.** $mono\ (\lambda X.\ X \rhd e)$ *B.18.2 p. 223*

Intuitively this means that the more variables are attached before an expression, the more are attached after the expression.

6.2.4 *Design mode*

As mentioned in section 6.1, there should be means to develop void-safe applications gradually. The most important issue is with features that take or return values of attached types. If there are no suitable effective classes yet, one cannot call such features or properly initialize their results. The idea to address such a need is to treat some code as unreachable. If the code is unreachable, there is no harm to skip void safety checks. In [15] the following constructs are used as indicators of unreachable code:

- enforced check: **check False then end**
- infinite loop: **from ... until False loop ... end**
- false postcondition: **ensure False**

Note that in general assertion checks are optional at run-time. However, to preserve soundness of void safety rules the assertion **ensure False** is always checked at run-time and triggers an exception. As a result, clients calling a feature with such a postcondition can rely on the fact that it never returns normally.

B.16 p. 198

$$
\begin{aligned}
X \sqsubseteq \top &= \mathit{True} \\
\top \sqsubseteq \lceil A \rceil &= \mathit{False} \\
\lceil A \rceil \sqsubseteq \lceil B \rceil &= A \subseteq B
\end{aligned}
\qquad
\begin{aligned}
X \sqcup \top &= \top \\
\top \sqcup X &= \top \\
\lceil A \rceil \sqcup \lceil B \rceil &= \lceil A \cup B \rceil
\end{aligned}
$$

B.16 p. 203

$$
\begin{aligned}
x \in^{\top} \top &= \mathit{True} \\
x \in^{\top} \lceil A \rceil &= x \in A
\end{aligned}
\qquad
\begin{aligned}
X \sqcap \top &= X \\
\top \sqcap X &= X \\
\lceil A \rceil \sqcap \lceil B \rceil &= \lceil A \cap B \rceil
\end{aligned}
$$

Figure 6.9. Operations on *topset*.

6.2.4.1 *Modeling unreachable code*

In [33] Gerwin Klein proposed to model definite assignment property of Jinja, a Java-like programming language, with formally specified semantics using type *set option*. A value *None* corresponds to an exceptional state and a value *Some x* – to a normal state. *x* is then a set of names of local variables that are definitely assigned.

The idea behind definite assignment in Java [21, 22] is that a (local) variable should be assigned prior to its use. This is a weaker notion compared to attachment status of a variable because once assigned, a variable remains assigned in all subsequent instructions and nested blocks of the same block. On the contrary, assigning **Void** to a variable changes its attachment status to *detachable* or *unset* in terms of definite assignment property in cases when subsequent instructions expect an attached type of the variable. Moreover, nested blocks can also change the attachment property in a similar way.

However, I found the idea quite suitable for modeling exceptional cases and unreachable code during attachment analysis. But instead of using somewhat ad hoc rules to handle *set option* I introduced a new type *topset* that is obtained from a regular *set* type by adding a new top (see theory *TopSet*). The operations on *topset* are defined as shown in figure 6.9.

B.16 p. 198

The *topset* type is proved to be a complete lattice and a distributive lattice. However unlike regular *set* it is not a boolean algebra because it does not provide the minus operator with the properties required by a boolean algebra.

Transitions of a local variable status from detachable to attached and back is modeled by two operations similar to insertion to a set and removal from a set:

$$A \oplus x = A \sqcup \lceil \{x\} \rceil$$
$$A \ominus x = A \sqcap \lceil \overline{\{x\}} \rceil$$

B.16 p. 207

The key difference is the case of a top element $\top$. Neither insertion nor removal changes it:

$$\top \oplus x = \top$$
$$\top \ominus x = \top$$

B.16 p. 208

An example of a diagram that shows all possible transitions between different states of *topset* for three variables *a*, *b* and *c* is shown in figure 6.10.

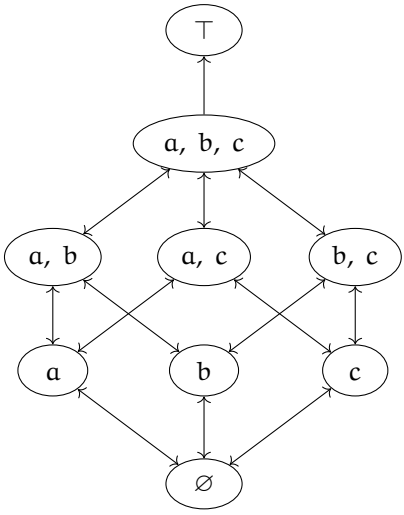

Figure 6.10. Insertion and removal in *topset*.

If the analysis identifies some code as unreachable, there is no reason to enforce void safety rules on variables. In particular, they can be assumed to have arbitrary attachment status. From practical point of view it is convenient to see all variables as being attached in that case. This enables writing code that compiles without an error even when some variables cannot be initialized with actual objects, e. g., when there are no suitable effective classes yet in the system being designed. Indeed, in such cases, execution points, where a variable is expected to be attached according to validity rules, are not reached and the expectations need not be fulfilled.

A common scenario is a function of an attached deferred type that cannot be instantiated with an effective one yet. According to void safety validity rules at the end of the function a special

local entity **Result** should be attached to an existing object. To deal with that a programmer can put an instruction guaranteed to raise an exception or to never terminate, e. g., a call to a feature (used to mark a place for further refactoring) with a postcondition **False**, before the function end. Execution points following this instruction are never reached at run-time. In particular, the function never returns normally and its result is never accessed. So, it is irrelevant if **Result** is attached at the end of the function or not. However, given that the function has an attached type, **Result** should be attached according to validity rules.

There are at least two ways to satisfy this requirement. One is to augment the rules with special treatment of unreachable code, another one is to consider all variables in this situation as attached. I chose the second variant because, on the one hand, it keeps validity rules simple, and on the other hand, it decouples detection of unreachable code from void safety rules. As a result, detection of unreachable code can be based on other mechanisms, not necessary related to void safety.

To summarize, proposed static analysis injects the special top for "design mode" that enables development of incomplete systems, making the whole framework practically useful. The formal proofs in chapter 7 rely solely on *topset* to track attached variables rather than on regular sets to ensure the "design mode" does not break soundness.

6.2.4.2 *Reachability properties*

The top element that models an attachment state for unreachable code is called an **unreachable attachment state**. This corresponds to the case when some piece of code is unreachable, i. e., the code is never executed at run-time. If evaluation of the attachment state resulted in a top element, then no subsequent computations can ever be performed at run-time. Therefore, the top element should be preserved by any subsequent computation:

B.18.2 p. 224

Lemma 6.7 (Preservation of unreachable attachment state)**.** *An unreachable attachment state induces itself by any computation:*

$$\top \rhd e = \top$$

There are few cases when the unreachable attachment state comes into play during analysis:

- An explicit instruction that triggers an exception, like `throw new MyException();` in Java.
- An implicit code pattern that leads to an exception at runtime, such as a missing *Else_part* in a multi-branch instruction in Eiffel.
- Specially crafted code that is known as a certified attachment pattern (CAP) and that is known to never trigger a subsequent computation, such as an infinite loop or an assertion with a constant value **False**.

For all such cases the analysis should produce an unreachable attachment state, e. g.: $A \rhd \mathit{Exception} = \top$. Note that the state before such a computation does not matter. *B.18.2 p. 217*

What if attachment analysis of an expression does not always produce a top element? Are subsequent expressions still reachable with some other initial state? Intuitively, if a new state is more defined, some potential execution paths may become impossible and the analysis may detect an unreachable state. On the other hand, if the initial state is smaller, i. e., we know less about attachment status of variables, the same or larger set of executions paths is possible and therefore the analysis cannot result in the unreachable attachment state. This brings us to the lemma complementary to lemma 6.7:

Lemma 6.8 (Preservation of reachable attachment state)**.** *If* $B \rhd e \neq \top \wedge A \leqslant B$ *then* $A \rhd e \neq \top$. *B.18.2 p. 225*

Proof. Follows from monotonicity of attachment transfer function (lemma 6.6). □

6.3 Validity rules

6.3.1 *Expression validity*

An expression e is void safe for an attachment set A if it satisfies an inductive predicate

$$A \vdash e : T$$ *B.19.1 p. 226*

where T – either *Attached* or *Detachable* – is an attachment type of the expression e, with inductive cases formally specified in figure 6.11. If the predicate is true, the expression satisfies void safety rules, otherwise an error is reported.

B.19.1
p. 226

$$\frac{v \neq \mathit{Void}_v}{A \vdash \mathit{Value}\ v : \mathit{Attached}}\ \textsc{Value}_{\mathtt{att}}$$

$$\frac{v = \mathit{Void}_v}{A \vdash \mathit{Value}\ v : \mathit{Detachable}}\ \textsc{Value}_{\mathtt{det}}$$

$$\frac{n \in^{\top} A}{A \vdash \mathit{Local}\ n : \mathit{Attached}}\ \textsc{Local}_{\mathtt{att}}$$

$$\frac{\neg\ n \in^{\top} A}{A \vdash \mathit{Local}\ n : \mathit{Detachable}}\ \textsc{Local}_{\mathtt{det}}$$

$$\frac{}{A \vdash \mathit{Exception} : \mathit{Attached}}\ \textsc{Exception}$$

$$\frac{}{A \vdash \mathit{create}\ n : \mathit{Attached}}\ \textsc{Create}$$

$$\frac{A \vdash e : T}{A \vdash n ::= e : \mathit{Attached}}\ \textsc{Assign}$$

$$\frac{A \vdash e : T}{A \vdash \mathit{attached}\ t\ e\ \mathit{as}\ n : \mathit{Attached}}\ \textsc{Test}$$

$$\frac{A \vdash e_1 : \mathit{Attached} \wedge A \rhd e_1 \vdash e_2 : \mathit{Attached}}{A \vdash e_1\ ;;\ e_2 : \mathit{Attached}}\ \textsc{Seq}$$

$$\frac{A \vdash e : \mathit{Attached} \wedge A \rhd e \vdash a\ [:]\ \mathit{Ts}}{A \vdash e\ .\ f\ (a) : \mathit{Attached}}\ \textsc{Call}$$

$$\frac{}{A \vdash []\ [:]\ []}\ \textsc{Arg}_{\mathtt{Nil}}$$

$$\frac{A \vdash e : T \wedge A \rhd e \vdash es\ [:]\ \mathit{Ts}}{A \vdash e \cdot es\ [:]\ T \cdot \mathit{Ts}}\ \textsc{Arg}_{\mathtt{Cons}}$$

$$\frac{\begin{array}{c}A \vdash b : \mathit{Attached} \wedge \\ A \rhd{+}\ b \vdash e_1 : T_1\ \wedge\ A \rhd{-}\ b \vdash e_2 : T_2\end{array}}{A \vdash \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} : \mathit{sup}\ T_1\ T_2}\ \textsc{If}$$

$$\frac{\begin{array}{c}A \rhd{*}\ (-\ e \rhd b) \vdash e : \mathit{Attached} \wedge \\ A \rhd{*}\ (-\ e \rhd b) \rhd{-}\ e \vdash b : \mathit{Attached}\end{array}}{A \vdash \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end} : \mathit{Attached}}\ \textsc{Loop}$$

Figure 6.11. Void safety rules.

If an expression is a value, it is void safe and detachable if the value is $\mathtt{Void}_v$ ($\textsc{Value}_{det}$) and is void safe and attached otherwise ($\textsc{Value}_{att}$).

If an expression is a local variable, it is void safe and its attachment type depends on whether the variable name is in the set of attached variables. When it is there, the corresponding expression is of an attached type ($\textsc{Local}_{att}$), otherwise it is of a detachable type ($\textsc{Local}_{det}$).

An attachment type of a conditional expression is computed as an upper bound of attachment types of both positive and negative branches (If). The upper bound is *Detachable* if any of the operands is *Detachable*, and *Attached* otherwise.

Validity of a loop exit condition and of a loop body is checked in the context of an attachment set obtained by applying a loop operator that reflects minimal set of attached variables after executing the loop any number of times (Loop).

All other language constructs are void safe as soon as all their components are void safe.

The validity predicate is properly defined, i. e., it cannot be true for attached and detachable types at the same time:

Lemma 6.9 (Attachment type uniqueness)**.** *A valid expression has one attachment type:* $A \vdash e : T \wedge A \vdash e : T' \Longrightarrow T = T'$ *B.19.2 p. 227*

If an input set of attached variables becomes larger, computed attachment type for an expression may only become "more attached". Therefore, an attachment type computed for a larger attachment set conforms to the attachment type for a smaller one.

Lemma 6.10 (Attachment type anti-monotonicity)**.** $A \leqslant B \wedge A \vdash e : T_A \implies \exists T_B . B \vdash e : T_B \wedge T_B \leqslant T_A$ *B.19.2 p. 229*

Proof. The proof is done by structural induction on the predicate definition. It relies on monotonicity of the transfer function (lemma 6.6) for compound expressions such as sequences and calls. For a conditional expression, types of both branches can be obtained thanks to lemma 6.9 and the resulting type will be computed as their upper bound, preserving monotonicity property. Validity of a loop expression follows from monotonicity of a loop operator (lemma 6.2). □

If a loop is valid in a given context, it is valid in the context obtained by a single or multiple application of the loop exit condition and the loop body:

B.19.2 p. 232 **Lemma 6.11.** *A loop remains valid after applying its transfer function to a set of attached variables one or any number of times:*

$$A \vdash \textit{until } e \textit{ loop } c \textit{ end} : T \implies A \rhd - e \rhd c \vdash \textit{until } e \textit{ loop } c \textit{ end} : T$$
$$A \vdash \textit{until } e \textit{ loop } c \textit{ end} : T \implies A \rhd * (- e \rhd c) \vdash \textit{until } e \textit{ loop } c \textit{ end} : T$$

Proof. Follows from monotonicity of the transfer function (lemma 6.6), anti-monotonicity of the expression validity predicate (lemma 6.10), idempotence of loop operator (lemma 6.4) and loop operator inequalities (lemma 6.5). □

A notion of void-safe expressions is defined using the expression validity predicate with or without an associated context:

Definition 6.1 (Void-safe expression)**.** *An expression e is void-safe with type T iff there is type that satisfies expression validity predicate with an empty set of attached variables:*

B.19.2 p. 233
$$\vdash e : T \equiv \lceil \varnothing \rceil \vdash e : T$$

An expression e is void-safe iff there is a type T with which e is void-safe:

B.19.2 p. 233
$$\vdash e \surd_e \equiv \exists T. \vdash e : T$$

6.3.2 *Beyond void safety*

A dual check that an expression is always attached to an object at run-time is a check that an expression is always **Void**. Of course, such static analysis has fewer applications compared to attachment analysis because there are fewer operations that may be performed on "no object" value. However, it is quite suitable in the following scenarios:

1. **Evaluation optimization.** Access to an expression that is always detachable can be replaced with **Void** provided that the expression has no side effect. Such a simplification may trigger further optimizations.
2. **Unreachable code removal.** If at the same code point a variable is detected to be both attached and detached, this code is unreachable and can be safely removed.

3. **Logical error detection.** When detected in non-inherited and non-inlined code the situation described in item 2 may indicate a logical error because the code contains two mutually exclusive conditions.

6.3.3 *Implementation*

The core part of the local code analysis described in section 6.3.1 is implemented in 19 Eiffel classes of about 2.5KLOC in total. Branching instructions, such as loops, and conditional instructions and expressions, share the same code that explains a form of the loop transfer function (section 6.2.3). Sets of attached local variables are computed similarly to sets of attached attributes in creation procedures described in section 5.1.5 with the new operations to remove variables from the set when they become detachable. In addition, the rules for attributes do not distinguish between positive and negative scopes. Therefore, for local variables there are dedicated classes that take care about scopes. All code is open source and is available at `https://dev.eiffel.com/Source_Code`. The proposed rules to compute attachment status of local variables are in production starting from *EiffelStudio 16.05* official release.

6.3.3.1 *Type checks*

In section 6.2.2 boolean operators were described as given in source code. This is indeed the case for some languages. E.g., in Java [21, 22] boolean operators `&&` and `||` are dedicated for logical operations only. This is not true in general case. Even though there is a restriction on redeclaring conditional operators `&&` and `||` in C# [26] and semistrict operators in Eiffel [12, 27], other kinds of operators operating on boolean values such as logical operators `&`, `|` and `!` in C# and boolean operators **and**, **or** and **not** in Eiffel can be redefined by a user.

Therefore, attachment rules involving scopes have to be applied after the code is analyzed and types of operands are computed. Then the scope rules are applied only when the first operand is of a *BOOLEAN* type. In all other cases the attachment algorithm still works but is cannot take advantage of the additional knowledge provided by these rules.

6.3.3.2 *Error reporting*

When reporting errors related to attachment status, it becomes important to avoid chains of similar errors for the same variable. For example, if there are two subsequent calls with a target variable *foo* and the first call turns out to be invalid because the variable is not attached, in addition to the error report it makes perfect sense to include the variable in the set of attached variables. Clearly, the developer assumed that the variable is non-void due it its type declaration or because of a particular CAP, but this assumption turned out to be wrong. There is no way to make this assumption correct for the second call. As soon as the cause of the error for the first call is fixed, it is very likely to be fixed simultaneously for the second one, so there is no reason to report the second error in the first place.

6.4 Practical experience

In order to benchmark validity rules based on certified attachment patterns rather than on type declarations two experiments were performed. In the first experiment a large code base (the complete *EiffelStudio* suite, including libraries, IDE, user and development tools, examples – several millions lines of code) known to be void-safe according to type-based validity rules was checked by the analyzer that applied CAP-based validity rules for local variables. All the code was compiled without an issue. The experiment demonstrated that CAP-based validity rules are as permissive as type-based ones.

In the second experiment several open-source void-unsafe libraries were used to see what kind of validity rules is more permissive. A general-purpose *POSIX*-based library *ePosix* was not void-safe at the moment of this writing, so, the most recent version *3.2.1* was used. Other libraries have been adapted for void safety already, so the versions that predated void safety changes were used: *Gobo 3.8* and *EiffelStudio 6.2*. The measurements were performed by checking only conformance rules and initialization rules that are relevant to this work using `Initialization` as a configuration setting for void safety. Results are shown in table 6.2.

Table 6.2. Number of void-safety errors reported for void-unsafe open-source libraries depending on used validity rules.

Library	Lines of code	Number of reported errors				Relative difference	
		Type rules		CAP rules			
		Total	Inner	Total	Inner	Total	Inner
base	89 648	144	144	118	118	18.1%	18.1%
base_extension	1 157	157	13	126	8	19.7%	38.5%
com	14 391	608	464	406	288	33.2%	37.9%
diff	795	150	6	122	4	18.7%	33.3%
docking	57 757	12 185	1 257	8 166	434	33.0%	65.5%
editor	20 126	11 309	157	7 942	106	29.8%	32.5%
eiffel2java	5 233	158	14	131	13	17.1%	7.1%
encoding	1 391	150	6	124	6	17.3%	0.0%
eposix	92 714	8 839	601	6 923	501	21.7%	16.6%
event	384	2 825	0	1 422	0	49.7%	0.0%
gobo	803 826	8 238	8 070	6 422	6 286	22.0%	22.1%
gobo_extension	84	8 238	0	6 422	0	22.0%	0.0%
graph	8 976	11 124	205	7 812	86	29.8%	58.0%
i18n	11 543	8 269	31	6 447	25	22.0%	19.4%
lex	5 495	170	26	142	24	16.5%	7.7%
mel	49 037	395	251	291	173	26.3%	31.1%
memory_analyzer	6 958	11 180	56	7 852	40	29.8%	28.6%
net	12 769	577	92	485	73	15.9%	20.7%
parse	1 453	174	4	146	4	16.1%	0.0%
patterns	459	148	4	121	3	18.2%	25.0%
preferences_reg	8 726	11 126	207	7 812	86	29.8%	58.5%
process	8 024	492	33	418	30	15.0%	9.1%
store	22 806	354	160	236	76	33.3%	52.5%
thread	1 982	144	0	118	0	18.1%	0.0%
time	7 146	168	24	136	18	19.0%	25.0%
uuid	394	168	0	136	0	19.0%	0.0%
vision	88 178	1 827	1 368	1 332	944	27.1%	31.0%
vision2	366 384	2 825	2 366	1 422	1 034	49.7%	56.3%
vision2_extension	1 303	2 834	9	1 428	6	49.6%	33.3%
web	3 210	161	17	132	14	18.0%	17.6%
wel	90 989	459	315	388	270	15.5%	14.3%
wizard	2 767	10 971	45	7 769	36	29.2%	20.0%
Summary	1786105	–	15945	–	10706	–	32.9%

- *Total* denotes errors reported by the validity checker.
- *Inner* is obtained as *Total* reduced by the number of dependent library errors.
- *Relative difference* is computed as $\frac{\mathrm{Errors(Type\ rules)} - \mathrm{Errors(CAP\ rules)}}{\mathrm{Errors(Type\ rules)}}$

The validity checkers reported errors for all classes, not just those discovered in the library itself. In order to get results only for the current library classes, the total number of errors was decreased by the number of errors reported for all (recursively) dependent libraries. To avoid duplicated errors, checks for inherited features were disabled, i. e., only immediate features were subject to validity checks. In total the code base had more than 1.7 millions of lines with library sizes varying from a few hundred to a few hundred thousand lines of code.

For all libraries the analysis based on type rules reported larger number of errors compared to the analysis based on CAPs. In order to see the effect, the relative difference was computed between the number of reported errors for two kinds of analyses. It ranges from 15% to 50% for aggregate code and from 0% to 65% for individual libraries. On average the CAP-based analysis triggers about 33% fewer spurious errors than the type-based one. This includes all types of errors, i. e., void safety errors for attributes, arguments and feature calls in addition to errors related to local variables. Therefore, the net effect for local variables is even higher.

The results were normalized and sorted by code size (figure 6.12) to see if there is any correlation between library size and number of detected errors as well as between library size and improvement demonstrated by CAP-based analysis. It appears to be none. However, there seems to be a correlation with an abstraction level of a library. The effect of CAP-based rules is much higher for more abstract libraries such as a general graph processing library (graph), multi-platform GUI toolkit (vision and vision2), preferences management (preference_reg), multi-platform persistent library (store) or GUI docking (docking). The libraries that are closer to an underlying platform – such as a general threading library (thread), Java connector (eiffel2java), external process launching library (process), Windows API wrapper (wel), POSIX API wrapper (eposix) – exhibit much less effect. The reason is that they operate on lower level that requires expanded rather than reference types and therefore are more immune to null pointer dereferencing at the library level.

The results demonstrate significant improvements in terms of accepted code that translates into lower cost required to adapt an existing library to void safety or to develop a new library due to less demanding validity rules. Consequently, the CAP-based

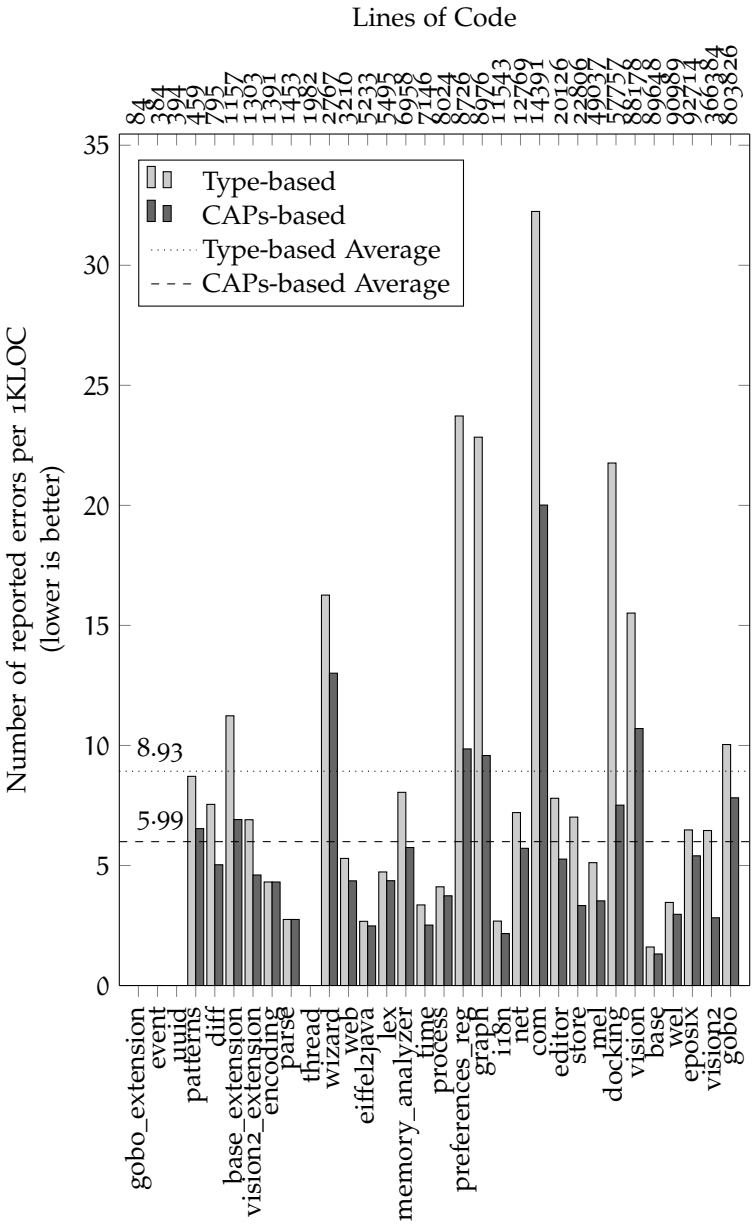

Figure 6.12. Relative number of errors reported for void-unsafe open-source libraries with type-based and CAP-based local void safety rules.

analysis replaced type-based analysis for local variables in the compiler implementation in [15].

6.5 Related work

As we have seen in chapter 4, most null safety solutions rely on additional type marks to denote attachment status of a type. The approach is applied uniformly without any distinction to class attributes, arguments and local variables ([6, 17, 51, 62, 70]). While all the solutions are perfectly sound, usability and user experience are not considered a top priority for the language rules. This work inspects if the type-based void safety rules for local variables are reasonable for real life code or if they can be relaxed. It demonstrates that all required information in this case can be derived from code. This is similar to intraprocedural analysis performed by various static analyzers like the Checker Framework [72].

Also, other void safety proposals do no discuss any methodology to enable development of large systems where a complete set of classes is not readily available. My work addresses the issue by respecting exceptional behavior and explaining how it can be used when only a partial subset of concrete classes is available.

Formalizations of language type systems are not new. Gerwin Klein and Tobias Nipkow formalized a subset of Java in Isabelle/HOL ([33]). Later in [41] Andreas Lochbihler formalized Java memory model for concurrency. In either case the goal was to demonstrate absence of "method not found" errors in programs that are valid for a statically typed object-oriented language. Void safety rules formalized in this work cover a different aspect, absence of "object not found" errors that seems to be less demanding, because it is easier to deal with such types of errors. Nevertheless, it remains an important source of bugs in modern software, and formalizing safety rules in a machine-checkable way is an important step towards verified and safe software systems.

An algorithm to compute a set of attached variables might seem to be quite similar to definite assignment rules of [21] and formalized in [33]. However, it differs in several important aspects. Contemporary definite assignment and presented here transfer

functions do take into account context of branches with different outcome of preceding conditions, while formalization in [33, 41] does not. Moreover, a set of definitely assigned variables does not depend on initial set of variables. Such a set is useless because an uninitialized variable cannot be used as a source of a reattachment. This is different for void safety. Both attached and detachable variables can be used as a source or as a target of a reattachment. Therefore, unlike definite assignment, changes of a variable attachment state is two-way. Moreover, described here void safety rules rely on computation of greatest fixed points for loops. This is not needed for definite assignment checks. Furthermore, monotonicity of a transfer function becomes essential not only for soundness proofs, but also for showing that validity rules can be programmatically checked by a compiler.

6.6 Conclusion

In contrast to the previous solutions, this work proposes to refrain from using type declarations of local variables to check that they are not **Void** and relies solely on certified attachment patterns that allow for deriving attachment status from the code itself. It presents formally specified void safety rules using the Isabelle/HOL proof assistant and demonstrates superiority of the proposed validity rules compared to the type-based rules using benchmarks on real source code. Main contributions of this work are:

CAP-based safety rules for local variables Specification of void safety rules for local variables is done solely in terms of certified attachment patterns rather than a type system.

Machine-checkable formalization of void safety rules Void safety validity rules for local variables are formally specified in Isabelle/HOL that enables machine-checkable proofs of their soundness (see chapter 7). In particular the formalization includes:

Generalization of boolean operators A general representation of arbitrary boolean operators is proposed in a form of specially crafted conditional expressions that greatly simplifies formalization and makes it applicable to languages with different sets of the operators.

Introduction of "design mode" When designing large real-world programs, it is essential to enable development without a complete set of concrete classes. Unlike regular typing rules, void safety does not allow for leaving uninitialized variables of an attached type. The proposed mechanism allows for dealing with such gradual development without disabling validity rules during the design phase.

Addition of positive and negative scopes The rules for branching constructs take into account which branch is taken. A similar scheme can be used to adapt definite assignment rules described in [33] and later used in [41] to prove Java -like type system soundness to match current Java rules [21].

Specification of rules for loops. This work gives a detailed specification of transfer function equations and a validity predicate used for loops and proves why they can be used in the language specification preserving finiteness of the static analysis.

Quantitative comparison to a type-based solution Practical uselessness of attachment annotations for local variables is demonstrated on a real representative open source code base that shows practical advantages of using CAP-based rules instead of type-based ones.

7

Soundness: Mechanically-Checked Proofs

7.1 Overview

A complete proof that a void-safe program does not cause access on void target at run-time involves all three elements of the mechanism: a type system, restrictions for object initialization and certified attachment patterns. A soundness proof for the void-safe type system is similar to class-based type system soundness proofs and relies on the conformance principle that a variable of an attached type may be assigned only an expression of an attached type. This establishes an invariant that as soon as a variable of an attached type receives a value, it remains attached all the time. The proof can be carried out similar to regular type system soundness proofs, including those done in proof assistants like [33, 41].

The soundness proof for object initialization is similar to the one described in [69] with two major differences. Firstly, the *free* status of a current object does not last until the end of a creation procedure, but only up to the point when all attributes are set, with the reservation that the creation procedure is not called by another one with an incompletely initialized **Current**. Secondly, annotations are replaced with the requirement to avoid qualified feature calls in the context with incompletely initialized objects.

For initialization of **Current** two situations are possible. In the first case all attributes of the current class are set and there are no incompletely initialized objects in the current context. Because all reachable objects are fully initialized and all attributes of the current class are properly set, the current object is deeply initialized

and can be freely used before the creation procedure finishes. In the second case either some attributes of the current class are not properly set or the context has references to objects that are not completely initialized. Because qualified calls are disallowed in these conditions, the uninitialized attributes cannot be accessed and access on void target is impossible. Due to the requirement to set all attributes at the end of a creation procedure, all these objects will have all attributes set, and, taking into account that the only reachable objects are either previously fully initialized or are new with all attributes pointing to the old or new objects, i. e., also fully initialized, all objects become fully initialized in the context where all attributes of the current class are set and no callers passed an uninitialized **Current**.

In terms of [69] a reference to the current object is in the *free* state and all calls on it and all arguments of features, where it is passed, should be marked respectively. Given that qualified feature calls are forbidden in this case, we are left with unqualified feature calls and creation procedure calls. For unqualified calls, the effect of the special notation is achieved by preventing accesses on uninitialized attributes (according to the rules, all features involved in unqualified feature calls in a creation procedure are subject to the same checks as the ones applied to this creation procedure). For creation procedure calls, the requirement to have no direct or indirect qualified feature calls ensures the uninitialized objects do not escape into code not related to object creation. The rest of the proof is similar to the one described in [69].

The soundness proof for certified attachment patterns is based on the invariant that any local variable in the set of attached variables has an associated non-void value at run-time. The main dissimilarity from the type system and object initialization soundness proofs is that if a variable is attached before an expression, there is no guarantee it will be attached after the expression. Therefore, the corresponding equations of the transfer functions are much more complicated (e. g., compare figure 5.6 and figures 6.5 to 6.8). As explained in section 6.3.3, the implementation for certified attachment patterns uses the same code as the one for object initialization, augmented with removal of variables from the set of attached variables and with the new rules for scopes. Because there is no previous work with a mechanically-checked soundness proof for certified attachment patterns, in this chapter I focus on this part of the soundness proofs only. It covers several

important aspects of the model not reflected previously in one piece, namely:

- formalization in a proof assistant – none of the existing work on null safety is accompanied with a mechanically-checked proof, though there are such proofs for a type system ([33, 41]);
- "design mode" – exceptions are an essential part of the semantics and a methodology to work with incomplete sets of classes in a void-safe setting (section 6.2.4) and significantly affect proofs, in particular, when combined with scopes (see next), however they are not reflected in some models of null-safe programs ([42]);
- positive and negative scopes – different branches of conditional expressions are not reflected in the definite assignment analysis in [33, 41], a very simplified version of conditions is used in [43], the rules are formulated in terms of generated byte code rather than source code in [42, 67], and branching instructions are commonly ignored by authors of object initialization proposals ([3, 61, 64, 69]);
- non-monotone single step – unlike transfer functions for definite variable assignment in [33, 41] or object initialization in [69] where a set of variables that are initialized before an expression is a subset of the set after this expression, sets of attached variables before and after an expresson cannot be ordered, in particular, this leads to the requirement to find a fixed point for a loop that is not required in other analyses.

In order to cover all elements of void safety, a soundness proof has to involve a void-safe type system, object initialization rules and certified attachment patterns. Provided that the proofs for the first two are similar to previous work, this chapter is devoted to the soundness proofs for certified attachment patterns only.

The proofs have been checked with *Isabelle 2016*. Isabelle/HOL theories form a structure similar to an acyclic inheritance graph in an object-oriented language (see [74]). Figure 7.1 gives a birds-eye overview of theory dependencies. The complete theories code is available in appendix B.

The theory names in the picture are truncated for brevity. The names marked with an apostrophe (') correspond to the theories that take attachment status into account. As one can see, the

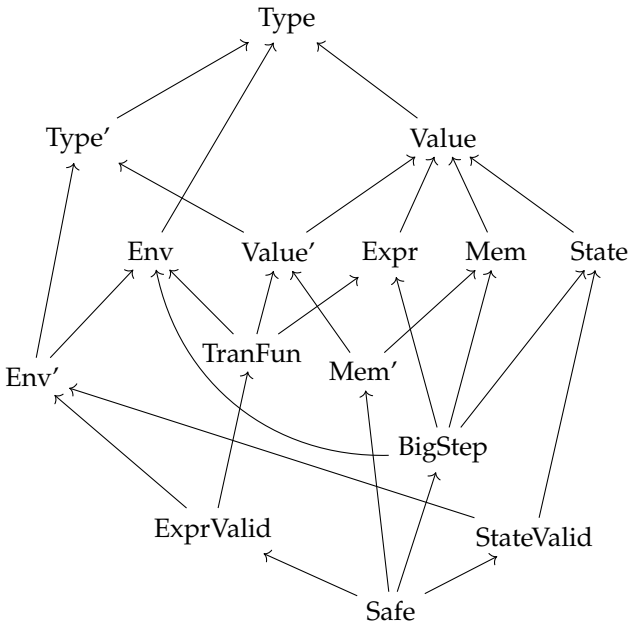

Figure 7.1. Simplified graph of theories.

big-step semantics theory does not depend on attachment properties. Indeed, except for program reflection and generic types, attachment properties do not change run-time behavior. They just guarantee absence of calls on void target. The theories structure confirms that attachment-aware theories are mostly subject for compile time checks and have no effect on execution apart from making it safe.

The formalization is done with a big-step semantics style that is known to be suitable for proving preservation properties ("nothing bad ever happens"), but has issues with proving progress properties ("something good always happens"). This limitation is addressed by considering two different semantics: void-unsafe and void-safe. Both demonstrated to be equivalent as soon as void safety rules are satisfied. A similar proof scheme can be applied to small-step semantics to prove progress property in that formalism if required.

7.2 State validity

A property that a local variable considered attached at compile time has an associated object at run-time is captured by the notion of a valid state. A state of a program is modeled by two functions: a function that maps local variable names to their values (a stack) and a function that maps memory addresses to object values (a heap). This work discusses only local variables, so the heap part can be arbitrary. Information about local variable types is available from an environment Γ.

Definition 7.1. *A local state l is valid with respect to an environment Γ iff for every local in Γ the state l has a value for this local:* B.20.1 p. 233

$$\Gamma \vdash l \surd_s{}^{\mathsf{E}} \equiv \forall n\ T.\ \Gamma\ n = \lfloor T \rfloor \longrightarrow (\exists v.\ l\ n = \lfloor v \rfloor)$$

Definition 7.2. *A local state l is valid with respect to an attachment set A iff for any local variable name in A there is a local variable of this name in l attached to an object provided that A is not $\top$:* B.20.1 p. 234

$$A \vdash l \surd_s{}^{\mathsf{A}} \equiv$$
$$A \neq \top \longrightarrow (\forall n.\ n \in^{\top} A \longrightarrow (\exists v.\ l\ n = \lfloor v \rfloor \wedge v \neq \mathit{Void}_v))$$

Definition 7.3 (Void-safe state)**.** *For an environment Γ and an attachment set A, a state (l, h) is void-safe iff it is valid with respect to Γ and A at the same time:*

$$\Gamma, A \vdash (l, h) \surd_s \equiv \Gamma \vdash l \surd_s{}^{\mathsf{E}} \wedge A \vdash l \surd_s{}^{\mathsf{A}}$$ B.20.1 p. 236

For an environment Γ, a state s is attachment-valid iff it is void-safe for Γ and an empty attachment set:

$$\Gamma \vdash s \surd_s \equiv \Gamma, \lceil \varnothing \rceil \vdash s \surd_s$$ B.20.3 p. 238

Most important properties of the state validity function are anti-monotonicity and state validity preservation when the corresponding attachment set changes:

Lemma 7.1 (Attachment state anti-monotonicity)**.** B.20.1 p. 235

$$B \leqslant A \wedge A \neq \top \wedge A \vdash l \surd_s{}^{\mathsf{A}} \Longrightarrow B \vdash l \surd_s{}^{\mathsf{A}}$$

Lemma 7.2. *Detaching, attaching and updating a local:* B.20.1 p. 235

$$A \vdash l \surd_s{}^{\mathsf{A}} \implies A \ominus \mathit{name} \vdash l(\mathit{name} \mapsto \mathit{value}) \surd_s{}^{\mathsf{A}}$$
$$\mathit{value} \neq \mathit{Void}_v \wedge A \vdash l \surd_s{}^{\mathsf{A}} \implies A \oplus \mathit{name} \vdash l(\mathit{name} \mapsto \mathit{value}) \surd_s{}^{\mathsf{A}}$$
$$\mathit{value} \neq \mathit{Void}_v \wedge A \vdash l \surd_s{}^{\mathsf{A}} \implies A \vdash l(\mathit{name} \mapsto \mathit{value}) \surd_s{}^{\mathsf{A}}$$

7.3 The semantics

The language semantics is defined in a big-step style as an Isabelle/HOL inductive predicate on transitions from an initial expression-state pair to a resulting one (figures 7.2 and 7.3). The rules are similar to those used in type system soundness proofs (e. g., [32, 33, 41]). The key differences are in the additional rules for object tests and in a modified rule for feature calls.

An object test evaluates to *True* if the expression evaluates to an object of an expected type ($\text{TEST}_{\texttt{True}}$). In this case the local storage is updated for the object test local with the computed object. The specification uses an abstract function *has_type* that is not instantiated. Therefore, the proofs do not depend on the actual run-time type check.

If either of the conditions is not met, e. g., the expression evaluates to *Void* or to an object of a non-conforming type, the state is not changed and the object test evaluates to *False* ($\text{TEST}_{\texttt{False}}$).

Note that there is only one (non-exceptional) rule for a feature call. This is the major difference from the traditional big-step semantics specifications. What if a target of a call will be *Void*? Would not it mean that execution may be stuck? The answer is given in section 7.4.2.

The exception propagation rules (figure 7.3) are similar to the rules used in type soundness proofs.

Conventionally the big-step semantics is shown to end up for a given expression in a final state, meaning an exception or a value.

B.6.2 p. 182

Definition 7.4 (Final expression)**.** *An expression is called final if it is an exception or a value:*

$$\mathit{Final}\ e = ((\exists v.\ e = \mathit{Value}\ v) \vee e = \mathit{Exception})$$

B.9.2 p. 186

Lemma 7.3 (Finality of big-step semantics)**.** *If there is a big-step transition for an expression e from state s to an expression e′ and state s′ then e′ is final:*

$$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow \mathit{Final}\ e'$$

B.9.1 p. 184

$$\frac{}{\Gamma \vdash \langle \mathit{Value}\ v, (l, m)\rangle \Rightarrow \langle \mathit{Value}\ v, (l, m)\rangle}\ \textsc{Value}$$

$$\frac{l\ n = \lfloor v \rfloor}{\Gamma \vdash \langle \mathit{Local}\ n, (l, m)\rangle \Rightarrow \langle \mathit{Value}\ v, (l, m)\rangle}\ \textsc{Local}$$

$$\frac{\Gamma \vdash \langle e_1, s\rangle \Rightarrow \langle \mathit{unit}, s'\rangle \wedge \Gamma \vdash \langle e_2, s'\rangle \Rightarrow \langle e_2{}', s''\rangle}{\Gamma \vdash \langle e_1\ ;;\ e_2, s\rangle \Rightarrow \langle e_2{}', s''\rangle}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, (l, m)\rangle}{\Gamma \vdash \langle n ::= e, s\rangle \Rightarrow \langle \mathit{unit}, (l(n \mapsto v), m)\rangle}\ \textsc{Assign}$$

$$\frac{\Gamma\ n = \lfloor T \rfloor \wedge \mathit{instance}\ m\ T = \lfloor (m', v) \rfloor}{\Gamma \vdash \langle \mathit{create}\ n, (l, m)\rangle \Rightarrow \langle \mathit{unit}, (l(n \mapsto v), m')\rangle}\ \textsc{Create}$$

$$\frac{\Gamma\ n = \lfloor T \rfloor \wedge \mathit{instance}\ m\ T = \mathit{None}}{\Gamma \vdash \langle \mathit{create}\ n, (l, m)\rangle \Rightarrow \langle \mathit{Exception}, (l, m)\rangle}\ \textsc{Create}_{\mathtt{fail}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_e\rangle \wedge v \neq \mathit{Void}_v \wedge \\ \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Value}\ vs, s'\rangle\end{array}}{\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow \langle \mathit{unit}, s'\rangle}\ \textsc{Call}$$

$$\frac{\Gamma \vdash \langle b, s\rangle \Rightarrow \langle \mathit{True}_c, s'\rangle \wedge \Gamma \vdash \langle e_1, s'\rangle \Rightarrow \langle e_1{}', s''\rangle}{\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end}, s\rangle \Rightarrow \langle e_1{}', s''\rangle}\ \textsc{If}_{\mathtt{True}}$$

$$\frac{\Gamma \vdash \langle b, s\rangle \Rightarrow \langle \mathit{False}_c, s'\rangle \wedge \Gamma \vdash \langle e_2, s'\rangle \Rightarrow \langle e_2{}', s''\rangle}{\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end}, s\rangle \Rightarrow \langle e_2{}', s''\rangle}\ \textsc{If}_{\mathtt{False}}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{True}_c, s'\rangle}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}, s\rangle \Rightarrow \langle \mathit{unit}, s'\rangle}\ \textsc{Loop}_{\mathtt{True}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{False}_c, s_e\rangle \wedge \Gamma \vdash \langle b, s_e\rangle \Rightarrow \langle \mathit{unit}, s_c\rangle \wedge \\ \Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}, s_c\rangle \Rightarrow \langle c', s'\rangle\end{array}}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}, s\rangle \Rightarrow \langle c', s'\rangle}\ \textsc{Loop}_{\mathtt{False}}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, (l, m)\rangle \wedge v \neq \mathit{Void}_v \wedge v\ \mathit{has\_type}\ T}{\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n, s\rangle \Rightarrow \langle \mathit{True}_c, (l(n \mapsto v), m)\rangle}\ \textsc{Test}_{\mathtt{True}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, (l, m)\rangle \wedge \\ \neg\ (v \neq \mathit{Void}_v \wedge v\ \mathit{has\_type}\ T)\end{array}}{\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n, s\rangle \Rightarrow \langle \mathit{False}_c, (l, m)\rangle}\ \textsc{Test}_{\mathtt{False}}$$

$$\frac{}{\Gamma \vdash \langle [], s\rangle\ [\Rightarrow]\ \langle [], s\rangle}\ \textsc{Arg}_{\mathtt{Nil}}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_e\rangle \wedge \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle es', s'\rangle}{\Gamma \vdash \langle e \cdot es, s\rangle\ [\Rightarrow]\ \langle \mathit{Value}\ v \cdot es', s'\rangle}\ \textsc{Arg}_{\mathtt{Cons}}$$

Figure 7.2. Big-step semantics: regular cases.

B.9.1
p. 184

$$\frac{}{\Gamma \vdash \langle \mathit{Exception}, s\rangle \Rightarrow \langle \mathit{Exception}, s\rangle}\ \textsc{Exception}$$

$$\frac{\Gamma \vdash \langle e_1, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle e_1 \mathbin{;;} e_2, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Seq}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle n ::= e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Assign}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle e\,.\,f\ (es), s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Call}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_e\rangle \wedge \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle \mathit{map\ Value\ vs} \mathbin{@} (\mathit{Exception} \cdot es'), s'\rangle}{\Gamma \vdash \langle e\,.\,f\ (es), s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Call}_{Arg\text{-}ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle e \cdot es, s\rangle\ [\Rightarrow]\ \langle \mathit{Exception} \cdot es, s'\rangle}\ \textsc{Arg}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{attached\ t\ e\ as\ n}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Test}_{ex}$$

$$\frac{\Gamma \vdash \langle b, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{if\ b\ then\ e_1\ else\ e_2\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{If}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{until\ e\ loop\ b\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Loop}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{False}_c, s_e\rangle \wedge \Gamma \vdash \langle b, s_e\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{until\ e\ loop\ b\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Loop}_{False\text{-}ex}$$

Figure 7.3. Big-step semantics: exception propagation.

7.4 Safety

7.4.1 *Preservation theorem*

Anti-monotonicity of an attachment state allows to prove that as soon as the state is valid in one of the branches of a conditional expression, it is valid for the expression as a whole. Intuitively there is more information in one branch of a conditional expression and therefore there are more attached variables, so if a state is valid for one branch it is valid for the whole expression with less attached variables.

Lemma 7.4. *If a local state l is valid in a context of either branch of a conditional expression, it is valid in the context of the whole expression:* B.21.1 p. 248

$$A \rhd + c \rhd e_1 \vdash l \surd_s{}^A \wedge A \rhd + c \rhd e_1 \neq \top \implies A \rhd \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} \vdash l \surd_s{}^A$$

$$A \rhd - c \rhd e_2 \vdash l \surd_s{}^A \wedge A \rhd - c \rhd e_2 \neq \top \implies A \rhd \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} \vdash l \surd_s{}^A$$

Proof. Follows from the definition of the transfer function and lemma 7.1. □

The big-step semantics preserves valid state for both exceptional (denoted by $\top$, see section 6.2.4) and non-exceptional attachment sets (denoted by $\lceil a \rceil$):

Lemma 7.5. $\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, \top \vdash s \surd_s \implies \Gamma, \top \vdash s' \surd_s$ B.21.1 p. 240

Lemma 7.6. $\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, \lceil a \rceil \vdash s \surd_s \implies \exists b.\ \Gamma, \lceil b \rceil \vdash s' \surd_s$ B.21.1 p. 241

On the other hand, if a final expression (definition 7.4) is not an exception, the attachment set for the initial expression remains non-exceptional. If there is a transition from an initial state for a void-safe expression to a final state that generates a value rather than an exception, then the code after the expression is reachable. Therefore, if void-safe analysis of the expression starts with an attachment set different from a top element, it produces a non-top attachment set:

Lemma 7.7 (Reachability preservation)**.** B.21.1 p. 242

$$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge A \vdash e : T \wedge e' \neq \mathit{Exception} \wedge A \neq \top \implies A \rhd e \neq \top$$

Proof. By structural induction on the big-step semantics predicate for all mutually recursive transfer functions. □

The main result of this section is an attachment preservation theorem telling that if an expression is void-safe and its evaluation starts in a void-safe state and completes, then it either results in an exception or in a value that is not *Void* if the expression type is attached. The following lemma states this formally.

B.21.2 p. 268 **Lemma 7.8** (Attachment preservation step)**.**

$$\begin{aligned} &\Gamma, A \vdash s \surd_s && \wedge \\ &A = \lceil a \rceil && \wedge \\ &\Gamma \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle && \wedge \\ &A \vdash e : T && \wedge \\ &e' \neq \mathit{Exception} && \Longrightarrow \exists T'.\ A \rhd e \vdash e' : T' \wedge T' \leqslant T \end{aligned}$$

Proof. The proof is done by structural induction on the big-step semantics predicate and uses lemmas 7.2 to 7.4. For every induction case it shows that a state remains valid using lemmas 7.5 and 7.6 and taking into account preservation of non-exceptional attachment set (lemma 7.7) and applying an inductive hypothesis to finish the proof. □

Replacing variables with initial state values, the lemma gives:

B.21.2 p. 272 **Theorem 7.1** (Attachment preservation)**.**

$$\begin{aligned} &\Gamma \vdash \langle e, \varnothing \rangle \Rightarrow \langle e', s' \rangle \ \wedge \vdash e : \mathit{Attached} \\ &\qquad \Longrightarrow\ e' = \mathit{Exception} \vee (\exists v.\ e' = \mathit{Value}\ v \wedge v \neq \mathit{Void}_v) \end{aligned}$$

7.4.2 *Equivalence of safe and unsafe semantics*

Ideally void safety should be a corollary of two theorems: preservation and progress. The third safety theorem – determinism – makes no sense for most concurrent environments. Unfortunately the progress theorem cannot be proved with classical big-step semantics because it deals only with final states (lemma 7.3) and cannot describe intermediate ones. To address this issue there are at least two options:

- Use "clocked" big-step semantics [65] or a similar abstraction that distinguishes between a stuck state and divergence ([2, 59, 66]).
- Use small-step semantics.

The first option is straightforward: the current rules can be adapted accordingly. The main drawback is missing support for concurrency that cannot be easily expressed with big-step semantics.

The second option is more attractive because it allows for proving the progress theorem directly and can enable concurrency in

the semantics. Unfortunately, it is not applicable to the current formalization because it does not provide type information. In the big-step rules for conditional expressions and loops the semantics is specified with an assumption that a branch or exit condition is evaluated to a boolean value. Intermediate proof steps involving the branch or exit condition in small-step semantics need to be accompanied with type information to make sure the resulting value is indeed of the boolean type.

Can the requirement to have type information in the semantics rules be avoided, so that only the part of interest is kept for consideration? Here is an idea. Let's assume that the type system is sound. Then both type preservation and progress theorems are true with respect to the associated small-step semantics. Assume that the original semantics is specified not taking void safety into account. Then consider semantics that expects void safety. If both, void-safety aware and void-safety unaware, semantics can be shown to be equivalent for programs that satisfy void safety rules, preservation and progress theorems can be derived for the void-safe semantics from their void-unsafe counterparts.

The approach can be demonstrated with big-step semantics as well. To this end two semantics definitions are considered. The void-safe version is the one described in section 7.3. The void-unsafe version differs from the safe one just by a single rule. The rule makes sure that if in a void-unsafe program a target of a call is **Void**, an exception is raised (figure 7.4). The exception here is the famous `NullPointerException`.

The void-safe version does not handle the case when the target of a call is *Void*. Therefore, it can become stuck if at some step it encounters such a target. From the programming language point of view this corresponds to undefined behavior at run-time. On the other hand, the void-unsafe version raises an exception in the same situation and proceeds. The following theorem connects both versions for void-safe programs and shows that void-safe semantics is never stuck because of null pointer dereferencing.

Lemma 7.9. *Void-unsafe semantics has the same effect as a void-safe one for a void-safe expression in a void-safe state:* $A \vdash e : T \wedge \Gamma, A \vdash s \surd_s \wedge A \neq \top \implies \Gamma \vdash \langle e, s\rangle \Rightarrow' \langle e', s'\rangle \implies \Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle$ B.22 p. 277

Proof. By structural induction on the unsafe version of the semantics. In the case $\textsc{Call}_{fail}^{unsafe}$ a call target's type is *Attached*

B.9.1 p. 184

Safe:
$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_e\rangle \ \wedge v \neq \mathit{Void}_\nu \ \wedge \quad \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Value}\ vs, s'\rangle}{\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow \langle \mathit{unit}, s'\rangle}\ \textsc{Call}$$

B.10.1 p. 188

Unsafe:
$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow' \langle \mathit{Value}\ v, s_e\rangle \ \wedge v \neq \mathit{Void}_\nu \ \wedge \quad \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]'\ \langle \mathit{map}\ \mathit{Value}\ vs, s'\rangle}{\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow' \langle \mathit{unit}, s'\rangle}\ \textsc{Call}^{unsafe}$$
$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow' \langle \mathit{Value}\ v, s'\rangle \wedge v = \mathit{Void}_\nu}{\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow' \langle \mathit{Exception}, s'\rangle}\ \textsc{Call}^{unsafe}_{fail}$$

Figure 7.4. Feature call rule in safe and unsafe big-step semantics.

because the call expression is void-safe. With the induction hypothesis we get that the target evaluates to a non-void value. This contradicts a premise of $\textsc{Call}^{unsafe}_{fail}$.

For $\textsc{Loop}_{ex}$ from the premise of void-safety of a loop we have that its exit condition e is void-safe in $A \rhd\!* (- \ e \ \rhd \ c)$. Using lemma 6.5 with monotonicity of the validity predicate (lemma 6.10) the expression e appears to be void-safe in A. With the induction hypothesis this proves the case.

Proofs for compound instructions such as conditional, sequence, etc. in addition rely on lemma 7.7. □

So, if an expression is void-safe, it gives exactly the same result regardless of the semantics behind. This effectively demonstrates absence of `NullPointerException` in null-safe programs.

B.22 p. 277

Theorem 7.2 (Semantics equivalence)**.** *Void-safe (⇒) and void-unsafe (⇒′) semantics of a void-safe program with an initial void-safe state are equivalent:* $\vdash e\ \surd_e \wedge \Gamma \vdash s_0\ \surd_s \implies \Gamma \vdash \langle e, s_0\rangle \Rightarrow' \langle v, s\rangle = \Gamma \vdash \langle e, s_0\rangle \Rightarrow \langle v, s\rangle$

Proof. Safe semantics trivially implies unsafe one because it has fewer rules. The reverse is proved using lemma 7.9. □

The complete theory code can be found in appendix B. All the scripts have been checked with *Isabelle 2016*.

7.5 Related work

Type system soundness of conventional object-oriented languages became a hot research area with a release of Java that claimed to be absolutely type-safe (cf. [29] that explicitly states undefined behavior of C++ in certain cases). Gerwin Klein and Tobias Nipkow in [33] presented a formal proof for a subset of Java in Isabelle/HOL using big-step semantics. Unfortunately big-step semantics is not good for reasoning about concurrent programs. Andreas Lochbihler in [41] updated the proof to use small-step semantics instead and formalized Java memory model. The current work focuses on void safety rules for Eiffel described in chapter 6. The concurrency model of Eiffel is quite different from Java. Even though Benjamin Morandi et al. in [52] formalized the concurrency semantics in Maude, its correctness is not formally proved. Therefore, this work uses big-step semantics to describe and to reason about void safety guarantees.

Leaving concurrency aside, big-step semantics does not distinguish between stuck and diverging states. Jeremy Siek in [65, 66] demonstrated how to deal with that, so the big-step semantics could be changed accordingly. Instead, this work shows that safe and unsafe versions of big step semantics become equivalent when void safety rules are satisfied.

Soundness proofs for null-safe analyses of object-oriented languages either consider branching expressions in a very limited form or do not consider them at all. Manuel Fahndrich and Songtao Xia in [18] specify a conditional expression in the form `ifnull x then y else z` that is quite similar to non-voidness tests. Unfortunately, they do not consider how to handle nesting of such expressions, ignore loops, do not include exceptions in the formal model and do not provide a complete soundness proof for their delayed types proposal.

Xin Qi and Andrew C. Myers describe in [61, 62] masked types as a type-based solution to the problem of object initialization. Their formal model omits branching expressions and exceptions altogether. The formal model used by Yoav Zibin et al. in [76] and the one used by Marco Servetto et al. [64], both to prove soundness of object initialization, suffer from the same issues. Branching expressions are also omitted by Alexander J. Summers and Peter Müller in [69, 70]. Therefore, these models cannot be used for cer-

tified attachment patterns. However, Alexander J. Summers and Peter Müller provide a detailed soundness proof for a very simple language that can be extended with missing constructs and modified to replace annotation-based typing rules with dataflow-analysis-based ones.

Chris Male et al. describe a Java bytecode verifier in [42]. The formal model misses exceptions, though it has a conditional instruction `ifceq` and a control transfer operator `goto`. Fausto Spoto proposes in [67] a way to deal with exceptions in Java bytecode, but does not include the `goto` instruction that is needed to model loops. Amogh Margoor and Raghavan Komondoor in [43] propose two techniques to improve precision of null-dereference verification, but their formal model does not include exceptions, loops and nested conditional expressions. Because in all three cases the analysis is done on bytecode, there is no immediate correspondence with the source code rules. In fact, in the second and the third analyses no language rules are employed at all and potential null pointer dereferencing is checked using abstract interpretation.

7.6 Conclusion

Certified attachment patterns are an essential part of void safety guarantees in modern OO languages. This work formalizes them in Isabelle/HOL and proves some safety properties with respect to big-step semantics. The main contributions of the work are:

Mechanical verification of the preservation theorem for void-safe programs The preservation theorem for void-safe programs is formally specified and mechanically checked in a proof assistant environment with the accompanying definitions and formal property proofs for a valid execution state.

Formal proof of conditional equivalence of safe and unsafe semantics Void-safe and void-unsafe big-step operational semantics are formally specified and proved to be equivalent for void-safe programs.

A

Code Migration

A.1 General information

Legend:

- N – non-void-safe code
- T – transitional void safety
- C – complete void safety

Table A.1. Void safety status of public libraries.

Library	6.2	6.3	6.4	6.5	6.6	6.7	6.8	7.0	7.1	7.2	7.3	13.11	14.05	15.01	15.08	15.12	16.05
api-wrapper			T	N	T	T	T	T	T	T	C	C	C	C	C	C	C
argument_parser		N	T	N	T	T	T	C	C	C	C	C	C	C	C	C	C
base	N	T	T	N	T	T	T	C	C	C	C	C	C	C	C	C	C
base_extension	N	N	T	N	T	T	T	C	C	C	C	C	C	C	C	C	C
base_original	N	N			N	N											
cocoa			T		T	T	T	T	T	T	T	C	C	C	C	C	C
com	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
curl	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
diff	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
docking	N	N	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T
editor	N	N	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T
edk						T	T	T	T	T	T	T	T	T	T	T	C
eiffel2java	N	N	T	N	T	T	T	T	T	T	C	C	C	C	C	C	C
encoding	N	N	T	N	T	T	T	T	T	T	C	C	C	C	C	C	C
event	N	N	N	N	T	T	T	T	T	T	T	T	C	C	C	C	C
gobo_extension	N	N	N	N	T	N	N	N	N	N	T	T	C	C	C	C	C
graph	N	N	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T
i18n	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
iphone						T	T	T	T	T	T	T	C	C	C	C	C
kmp_matcher				N	T	T	T	C	C	C	C	C	C	C	C	C	C
lex	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
libevent						T	T	T	T	T	T	T	C	C	C	C	C
logging							T	T	T	T	C	C	C	C	C	C	C
mel	N	N			N	N	N	N	N	N	N	N	N	N	N	N	N
memory_analyzer	N	N	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T
net	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
net_ipv6		N															
objective_c			T		T	T	T	T	T	T	T	C	C	C	C	C	C
parse	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
patterns	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
preferences	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
process	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
ribbon							T	T	T	T	T	T	C	C	C	C	C
store	N	N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
testing		N	T	N	T	T	T	T	T	T	T	T	C	C	C	C	C
testing_eweasel						T	T	T	T	T	T	T	C	C	C	C	C
thread	N	N	T	N	T	T	T	C	C	C	C	C	C	C	C	C	C
time	N	N	T	N	T	T	T	C	C	C	C	C	C	C	C	C	C
uri													C	C	C	C	C
uuid	N	N	N	N	T	T	T	C	C	C	C	C	C	C	C	C	C
vision	N	N			N	N											
vision2	N	N	T	N	T	T	T	T	T	T	C	C	C	C	C	C	C
vision2_extension		N	T	N	T	T	T	T	T	T	T	C	C	C	C	C	C
vision2_for_gtk12		N	T	N	N	N											
web	N	N	T	N	T	T	T	T	T	T	C	C	C	C	C	C	C
web_browser						T	T	T	T	T	C	C	C	C	C	C	C
wel	N	N	T	N	T	T	T	T	T	T	C	C	C	C	C	C	C
wizard	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
xml_parser						T	T	C	C	C	C	C	C	C	C	C	C
xml_tree						T	T	T	C	C	C	C	C	C	C	C	C
zmq												C	C	C	C	C	C

Table A.2. LOC in public libraries in different releases.

Library	6.2	6.3	6.4	6.5	6.6	6.7	6.8	7.0	7.1	7.2	7.3	13.11	14.05	15.01	15.08	15.12	16.05
api-wrapper	0	0	1386	1445	1445	1448	1448	1448	1445	1447	1447	1447	1447	1447	1447	1447	1447
argument_parser	0	5657	6505	6523	6605	6625	6625	6645	6566	6798	6798	6798	6798	6798	6797	6797	6797
base	89648	95311	91844	92302	96851	98396	99567	100113	101337	110379	111965	113921	122960	123037	120725	120731	121422
base_extension	1157	1157	1174	2667	2706	2713	2713	2766	2869	3026	3555	3555	3541	3538	3538	3538	3573
base_original	80905	80905	0	0	80905	80905	0	0	0	0	0	0	0	0	0	0	0
cocoa	0	0	12080	0	23049	23049	23049	23049	23049	23102	23102	23129	23169	23169	23169	23169	23169
com	14391	14391	14462	14462	14462	14463	14463	14463	14270	14318	14318	14318	14318	14318	14318	14318	14318
curl	1817	2107	2199	2214	2259	2259	2259	2993	3517	3517	3521	3529	3556	3556	3556	3556	3556
diff	795	795	1047	1047	1044	1059	1059	1059	1059	1064	1064	1064	1041	1041	1041	1041	1041
docking	57757	58378	60842	61282	64287	64493	64718	64679	64307	64592	64592	64584	64585	64585	64545	64545	64517
editor	20126	20588	21598	21672	22669	22489	22511	22586	22728	22734	22768	23243	23262	23262	23270	23269	23267
edk	0	0	0	0	0	3110	3110	3111	3113	3113	3113	3113	3113	3113	3113	3113	3171
eiffel2java	5233	5233	5259	5259	5259	5259	5259	5259	5259	5259	5260	5260	5260	5260	5260	5260	5260
encoding	1391	2567	2924	2924	2985	3353	3393	3433	3433	3490	3488	3514	3515	3516	3516	3516	3516
event	384	384	384	384	384	384	384	390	390	390	390	390	390	390	390	390	390
gobo_extension	84	296	299	299	299	299	299	299	299	285	290	290	290	290	290	290	290
graph	8976	8979	8979	8979	9354	9375	9476	9471	9309	9345	9345	9345	9345	9345	9345	9345	9316
i18n	11543	11564	12996	12996	13157	13163	13163	13170	13165	13489	13499	13499	13333	13334	13334	13334	13334
iphone	0	0	0	0	0	2720	2720	2722	2722	2722	2722	2722	2722	2722	2722	2722	2722
kmp_matcher	0	0	0	873	953	953	953	953	953	945	945	946	946	946	946	946	946
lex	5495	5495	5963	5963	5991	5992	5992	5992	5992	5992	5993	5993	5857	5857	5857	5857	5857
libevent	0	0	0	0	0	1047	1047	1054	1054	1054	1054	1054	1124	1124	1124	1124	1124
logging	0	0	0	0	0	0	1261	1307	1307	1307	1307	1307	1605	1605	1605	1605	1605
mel	49037	49037	0	0	49037	49037	49037	49037	0	0	0	0	0	0	0	0	0
memory_analyzer	6958	7828	7826	7826	8120	8120	8120	8120	8124	8126	8131	8131	8131	8131	8228	8228	8228
net	12769	12769	14891	14882	14949	14943	14951	14978	14982	14980	14980	14980	14905	14905	15455	15455	15455
net_ipv6	0	14634	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
objective_c	0	0	1255	0	6061	6061	6061	6047	6047	6049	6049	6071	6081	6081	6081	6081	6081
parse	1453	1453	1660	1660	1660	1660	1660	1660	1660	1660	1668	1668	1689	1689	1689	1689	1689
patterns	459	459	459	459	459	459	459	459	459	459	459	459	459	459	459	459	459
preferences	8756	8768	8905	8925	8961	9024	9038	9040	9011	10838	10838	10838	10870	10870	10918	10930	10930
process	8024	8459	8970	8969	9404	9409	9438	9448	9108	9232	9232	9232	8999	8999	8999	8999	8999
ribbon	0	0	0	0	0	0	5546	7062	7062	7799	7688	7688	7653	7650	7650	7650	7650
store	22806	22806	23712	23715	24666	24497	25470	26486	30033	30719	30697	30697	32646	32618	32612	32612	32612
testing	0	4129	6110	6168	6347	6110	6118	6119	6139	6253	6249	6249	6230	6237	6235	6235	6235
testing_eweasel	0	0	0	0	0	7815	7815	7815	7815	7800	7800	7800	7758	7758	7758	7758	7758
thread	1982	1987	2656	2656	3158	3117	3117	3122	3104	3104	3104	3104	3104	3104	3104	3104	3104
time	7146	7146	7244	7244	7250	7250	7250	7220	7255	7298	7768	7768	7768	7768	7768	7768	7768
uri	0	0	0	0	0	0	0	0	0	0	0	0	3191	3191	3206	3222	3222
uuid	394	394	521	521	519	519	519	519	519	525	525	525	525	525	525	525	525
vision	88178	88178	0	0	88178	88178	0	0	0	0	0	0	0	0	0	0	0
vision2	355384	355670	376537	376641	387511	387568	388177	378748	421957	422993	424111	277289	278142	278380	278690	278690	278582
vision2_extension	0	1303	1303	1303	1324	1324	1324	1324	1308	1527	1527	1527	1521	1521	1521	1521	1521
vision2_for_gtk12	0	15320	15320	15320	15320	15320	0	0	0	0	0	0	0	0	0	0	0
web	3210	3210	3243	3243	3243	3243	3243	3243	3243	3239	3235	3235	3235	3235	3235	3235	3235
web_browser	0	0	0	0	0	2400	2400	2375	2351	2371	2371	2287	2406	2406	2406	2406	2406
wel	90989	92015	93790	93795	94008	94007	94396	95875	96850	97105	97421	97429	97960	97960	97975	97975	97985
wizard	2767	2767	2767	2767	2762	2762	2762	2762	2766	2454	2454	2454	2454	2454	2454	2454	2454
xml_parser	0	0	0	0	0	5474	5480	5480	5553	8105	8105	8115	9357	9357	9357	9366	9366
xml_tree	0	0	0	0	0	0	3705	3991	4056	4068	4422	4422	4422	4422	4422	4460	4463
zmq	0	0	0	0	0	0	0	0	0	0	0	2222	2223	2223	2223	2223	2223

A.2 Migration from void-unsafe to transitional level of void safety

Table A.3. Migration of non-void-safe code to transitional void safety.

Library	LOC				Classes	
	Inserted	Deleted	Modified	Total	Modified	Total
api_wrapper	0	0	0	1447	12	12
argument_parser	21	1	18	6645	3	38
base	9859	9313	927	100113	100	398
base_extension	53	0	13	2766	5	22
cocoa	30	3	7	23129	3	123
curl	27	0	1	3556	1	18
diff	0	23	259	1041	2	8
edk	98	40	67	3171	10	35
eiffel2java	1	0	11	5260	2	22
encoding	0	2	15	3488	20	20
event	0	0	0	390	0	2
gobo_extension	0	0	0	290	0	4
i18n	67	233	498	13333	17	70
iphone	0	0	0	2722	0	23
kmp_matcher	0	0	0	953	0	3
lex	100	236	488	5857	13	20
libevent	70	0	0	1124	1	11
logging	0	0	0	1307	9	9
net	57	132	584	14905	23	80
objective_c	26	4	14	6071	5	44
parse	33	12	138	1689	6	9
preferences	99	67	229	10870	17	57
process	35	268	209	8999	12	38
ribbon	114	149	19	7653	9	71
store	3500	1551	2411	32646	95	184
testing	38	57	153	6230	12	35
testing_eweasel	10	52	140	7758	11	61
thread	6	1	4	3122	2	26
time	0	30	39	7220	4	36
uuid	0	0	0	519	0	3
vision2	1662	544	2663	424111	1358	2078
vision2_extension	0	0	0	1527	0	18
web	0	4	31	3235	1	24
web_browser	0	0	0	2371	7	7
wel	385	69	263	97421	103	582
xml_parser	3	3	4	5480	2	35
xml_tree	12	0	4	4068	2	28
Total	53867	10659	60525	788984	2912	3825

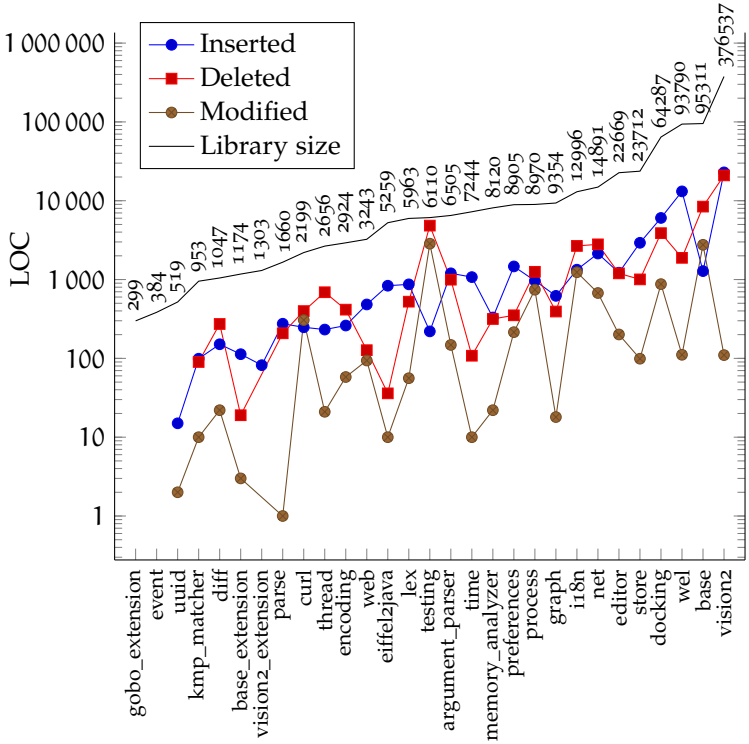

Figure A.1. Number of lines (LOC) changed in non-void-safe public libraries to bring them to transitional void safety level.

A.3 Migration from transitional to complete level of void safety

Table A.4. Migration from transitional to complete void safety.

Library	LOC				Classes	
	Inserted	Deleted	Modified	Total	Modified	Total
argument_parser	996	148	1200	6505	37	37
base	8422	2759	1280	95311	147	375
base_extension	19	3	113	1174	6	6
curl	398	306	248	2199	17	14
diff	273	22	151	1047	8	8
docking	3880	875	6048	64287	183	223
editor	1198	201	1222	22669	53	80
eiffel2java	36	10	835	5259	22	22
encoding	415	58	261	2924	17	17
event	0	0	0	384	0	2
gobo_extension	0	0	0	299	0	4
graph	393	18	621	9354	33	42
i18n	2670	1238	1334	12996	73	69
kmp_matcher	90	10	99	953	3	3
lex	524	56	869	5963	20	20
memory_analyzer	316	22	330	8120	24	35
net	2797	675	2137	14891	80	80
parse	208	1	275	1660	9	9
preferences	352	215	1467	8905	40	40
process	1254	743	966	8970	45	36
store	1005	99	2926	23712	128	128
testing	4833	2852	220	6110	47	31
thread	690	21	232	2656	24	24
time	108	10	1074	7244	36	36
uuid	0	2	15	519	2	3
vision2	20977	110	22895	376537	1254	1877
vision2_extension	0	0	82	1303	15	15
web	127	94	483	3243	24	24
wel	1886	111	13142	93790	565	565
Total	16306	12794	9209	822487	1867	4254

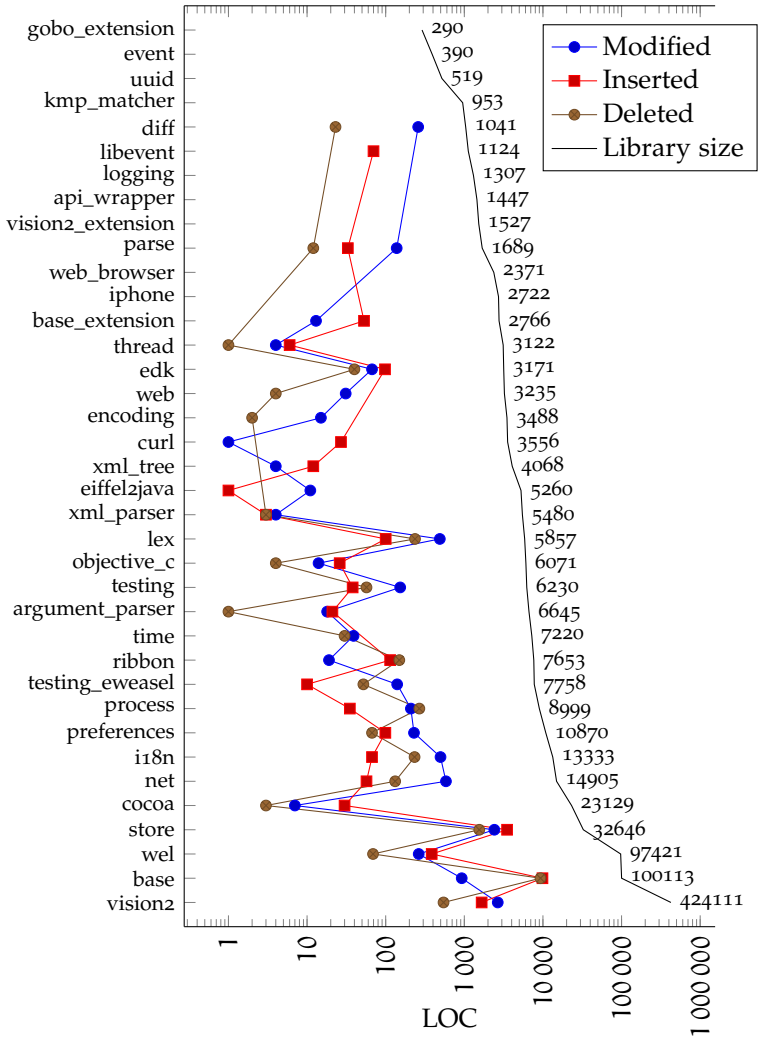

Figure A.2. Number of lines (LOC) changed in transitionally void-safe public libraries to make them completely void-safe.

B

Theories code

B.1 Common definitions

theory *Common* **imports** *Main*
~~/src/HOL/Library/LaTeXsugar
~~/src/HOL/Library/OptionalSugar
begin

no_notation *floor* (⌊_⌋)
no_notation *ceiling* (⌈_⌉)
notation *Some* (⌊_⌋)

end

B.2 Identifiers

theory *Name* **imports** *Common* **begin**

— Local variable name
type_synonym *vname* = *string*

— Feature name
type_synonym *fname* = *string*

— Class name
type_synonym *cname* = *string*

end

B.3 Types

theory *Type* **imports** *Name* **begin**

type_synonym *'a generic_list* = *'a list*

— A type is either a unit type (no value) or a class type (some value).
datatype *'a type* =

UnitType |
ClassType *'a cname* *'a type generic_list*

end

B.4 Type environment

theory *Environment* **imports** *Type* **begin**

type_synonym *'a environment* = *vname* ⇒ *'a type option*

end

B.5 Values

theory *Value* **imports** *Type* **begin**

datatype *'basic_value object_value* =
Basic *'basic_value* |
Boolean bool |
User

datatype *'basic_value reference_value* =
Void |
AttachedObject *'basic_value object_value*

datatype *'basic_value value* =
Unit |
Object *'basic_value object_value* |
Reference *'basic_value reference_value*

definition *is_value v* ≡ *v* ≠ *Unit*

primrec *is_expanded* :: *'basic_value value* ⇒ *bool* **where**
is_expanded Unit ⟷ *False* |
is_expanded (*Object* _) ⟷ *True* |
is_expanded (*Reference* _) ⟷ *False*

primrec *is_reference* :: *'basic_value value* ⇒ *bool* **where**
is_reference Unit ⟷ *False* |
is_reference (*Object* _) ⟷ *False* |
is_reference (*Reference* _) ⟷ *True*

lemma *is_reference v* ⟹ ¬ *is_expanded v*

by (*cases v*, *simp_all*)

lemma *is_expanded v* ⟹ ¬ *is_reference v*
by (*cases v*, *simp_all*)

lemma *value_is_expanded_or_reference*:
is_value v ⟹ *is_expanded v* ∨ *is_reference v*
by (*cases v*, *simp_all add*: *is_value_def*)

primrec *is_attached* :: *'basic_value value* ⇒ *bool* **where**
is_attached Unit = *True* |
is_attached (*Object* _) = *True* |
is_attached (*Reference x*) = (*case x of AttachedObject* _ ⇒ *True* | _ ⇒ *False*)

abbreviation *Void*$_v$ ≡ *Reference Void*

lemma *is_attached_if_not_void* [*iff*]: *is_attached x* ⟷ *x* ≠ *Void*$_v$
by (*cases x*, *insert reference_value.exhaust*, *force*+)

primrec *is_basic* :: *'basic_value value* ⇒ *bool* **where**
is_basic Unit = *False* |
is_basic (*Object x*) = (*case x of Boolean* _ ⇒ *True* | *Basic* _ ⇒ *True* | _ ⇒ *False*) |
is_basic (*Reference x*) = *False*

primrec *is_boolean* :: *'basic_value value* ⇒ *bool* **where**
is_boolean Unit = *False* |
is_boolean (*Object x*) = (*case x of Boolean* _ ⇒ *True* | _ ⇒ *False*) |
is_boolean (*Reference x*) = *False*

lemma *expanded_is_attached*: *is_expanded v* ⟹ *is_attached v*
by (*cases v*, *simp_all*)

lemma *basic_is_expanded*: *is_basic v* ⟹ *is_expanded v*
by (*cases v*, *simp_all*)

lemma *boolean_is_basic*: *is_boolean v* ⟹ *is_basic v*
proof (*cases v*)
case (*Object x*)
moreover assume *is_boolean v*
ultimately show *?thesis* **by** (*cases x*, *simp_all*)
qed *simp_all*

lemma *basic_is_attached*: *is_basic v* ⟹ *is_attached v*
by *auto*

lemma *boolean_is_expanded*: *is_boolean v* $\Longrightarrow$ *is_expanded v*
by (*simp add*: *boolean_is_basic basic_is_expanded*)

lemma *boolean_is_attached*: *is_boolean v* $\Longrightarrow$ *is_attached v*
by *auto*

end

B.6 Expression

theory *Expression* **imports** *Value* **begin**

B.6.1 Expressions

Abstract syntax.

datatype ('*b*, '*t*) *expression* =
Value '*b value* |
Local vname |
Sequence ('*b*, '*t*) *expression* ('*b*, '*t*) *expression* (_ ;; _ [80, 81] 80) |
Assignment vname ('*b*, '*t*) *expression* (_ ::= _ [1000, 81] 81) |
Creation vname (*create* _ [81] 81) |
Call ('*b*, '*t*) *expression fname* ('*b*, '*t*) *expression list*
(_ . _ '(_') [90, 99, 0] 90) |
If ('*b*, '*t*) *expression* ('*b*, '*t*) *expression* ('*b*, '*t*) *expression*
(*if* _ *then* _ *else* _ *end* [80, 80, 80] 81) |
Loop ('*b*, '*t*) *expression* ('*b*, '*t*) *expression* (*until* _ *loop* _ *end* [80, 81] 81) |
Test '*t option* ('*b*, '*t*) *expression vname* (*attached* _ _ *as* _ [80, 81, 81] 81) |
Exception

lemma *value_neq_exception*[*simp*, *intro*]: (*Value v*) $\neq$ *Exception*
by *simp*

abbreviation *unit* $\equiv$ *Value Unit*

lemma *inj_Value* [*simp*]: *inj Value*
by (*simp add*: *inj_on_def*)

B.6.2 Final computations

Is an expression final?

inductive *final* :: ('*b*, '*t*) *expression* $\Rightarrow$ *bool* **where**
final (*Value v*) |
final Exception

declare *final.cases* [*elim*]
declare *final.intros* [*simp*]

lemmas *finalE* [*consumes 1*, *case_names Value Exception*] = *final.cases*

lemma *final_iff*: *final e* = (∃ *v*. *e* = *Value v*) ∨ (*e* = *Exception*)
by *auto*

B.6.3 *Boolean expressions*

abbreviation $False_c \equiv Value\ (Object\ (Boolean\ False))$
abbreviation $True_c \equiv Value\ (Object\ (Boolean\ True))$

Does an expression definitely evaluate to False?

fun *is_false* **where**
is_false_true: *is_false* (c ;; $False_c$) ⟷ *True* |
is_false_false: *is_false* _ ⟷ *False*

Does an expression definitely evaluate to True?

fun *is_true* **where**
is_true_true: *is_true* (c ;; $True_c$) ⟷ *True* |
is_true_false: *is_true* _ ⟷ *False*

end

B.7 Object heap

theory *Memory* **imports** *Value* **begin**

locale *memory* =
fixes
instantiated :: *'memory* ⇒ *'a type* ⇒ (*'memory* × *'b value*) *option*
assumes
direct_instance: *instantiated m t* = ⌊(*m'*, *v*)⌋ ⟹ *v* ≠ *Unit* ∧ *is_attached v*
begin
end

definition
instantiated :: *'memory* ⇒ *'a type* ⇒ (*'memory* × *'b value*) *option*
where
instantiated m t = *Some* (*m*, *Reference* (*AttachedObject User*))
declare *instantiated_def* [*simp*]

interpretation *basic_memory*: *memory*

instantiated
by (*unfold_locales*, *auto*)

definition *instance* ≡ *instantiated* ::
'memory ⇒ *'a type* ⇒ (*'memory* × *'b value*) *option*

consts *is_instance_of* :: *'b value* ⇒ *'t* ⇒ *bool*

end

B.8 Memory state

theory *State* **imports** *Value* **begin**

type_synonym *'value_type local_state* =
vname ⇒ *'value_type value option*

type_synonym (*'value_type*, *'memory*) *state* =
'value_type local_state × *'memory*

end

B.9 Void-safe Big-step semantics

theory *BigStep* **imports**
Environment
Expression
Memory
State
begin

B.9.1 Big-step semantics rules

abbreviation *has_type_rep* (**infix** *has'_type 60*) **where**
has_type_rep v T ≡ (*T* = *None* ∨ *is_instance_of v T*)

Void-safe big-step semantics predicate: there is no rule for a call when its target is void.

inductive
big_step :: *'a environment* ⇒ (*'b*, *'t*) *expression* ⇒ (*'b*, *'m*) *state* ⇒
(*'b*, *'t*) *expression* ⇒ (*'b*, *'m*) *state* ⇒ *bool*
(_ ⊢ ⟨_, _⟩ ⇒ ⟨_, _⟩ [*80*, *0*, *0*] *81*)
and
big_steps :: *'a environment* ⇒ (*'b*, *'t*) *expression list* ⇒ (*'b*, *'m*) *state* ⇒

```
  ('b, 't) expression list ⇒ ('b, 'm) state ⇒ bool
  (_ ⊢ ⟨_, _⟩ [⇒] ⟨_, _⟩ [80, 0, 0] 81)
for
 Γ :: 'a environment
where
 Value: Γ ⊢ ⟨Value v, (l, m)⟩ ⇒ ⟨Value v, (l, m)⟩ |
 Local: l n = ⌊v⌋ ⟹ Γ ⊢ ⟨Local n, (l, m)⟩ ⇒ ⟨Value v, (l, m)⟩ |
 Seq: ⟦Γ ⊢ ⟨e1, s⟩ ⇒ ⟨unit, s'⟩; Γ ⊢ ⟨e2, s'⟩ ⇒ ⟨e2', s''⟩⟧ ⟹
  Γ ⊢ ⟨e1;; e2, s⟩ ⇒ ⟨e2', s''⟩ |
 Assign: ⟦Γ ⊢ ⟨e, s⟩ ⇒ ⟨Value v, (l, m)⟩⟧ ⟹
  Γ ⊢ ⟨n ::= e, s⟩ ⇒ ⟨unit, (l (n ↦ v), m)⟩ |
 Create: ⟦Γ n = ⌊T⌋; instance m T = ⌊(m', v)⌋⟧ ⟹
  Γ ⊢ ⟨create n, (l, m)⟩ ⇒ ⟨unit, (l (n ↦ v), m')⟩ |
 Create_fail: ⟦Γ n = ⌊T⌋; instance m T = (None::('m × 'b value) option)⟧ ⟹
  Γ ⊢ ⟨create n, (l, m)⟩ ⇒ ⟨Exception, (l, m)⟩ |
 Call: ⟦
   Γ ⊢ ⟨e, s⟩ ⇒ ⟨Value v, se⟩;
   v ≠ Void_v;
   Γ ⊢ ⟨es, se⟩ [⇒] ⟨map Value vs, s'⟩
  ⟧ ⟹ Γ ⊢ ⟨e . f (es), s⟩ ⇒ ⟨unit, s'⟩ |
 If_true: ⟦Γ ⊢ ⟨b, s⟩ ⇒ ⟨True_c, s'⟩; Γ ⊢ ⟨e1, s'⟩ ⇒ ⟨e1', s''⟩⟧ ⟹
  Γ ⊢ ⟨if b then e1 else e2 end, s⟩ ⇒ ⟨e1', s''⟩ |
 If_false: ⟦Γ ⊢ ⟨b, s⟩ ⇒ ⟨False_c, s'⟩; Γ ⊢ ⟨e2, s'⟩ ⇒ ⟨e2', s''⟩⟧ ⟹
  Γ ⊢ ⟨if b then e1 else e2 end, s⟩ ⇒ ⟨e2', s''⟩ |
 Loop_true: ⟦Γ ⊢ ⟨e, s⟩ ⇒ ⟨True_c, s'⟩⟧ ⟹
  Γ ⊢ ⟨until e loop b end, s⟩ ⇒ ⟨unit, s'⟩ |
 Loop_false: ⟦
     Γ ⊢ ⟨e, s⟩ ⇒ ⟨False_c, se⟩;
     Γ ⊢ ⟨b, se⟩ ⇒ ⟨unit, sc⟩;
     Γ ⊢ ⟨until e loop b end, sc⟩ ⇒ ⟨c', s'⟩
   ⟧ ⟹ Γ ⊢ ⟨until e loop b end, s⟩ ⇒ ⟨c', s'⟩ |
 Test_true: ⟦Γ ⊢ ⟨e, s⟩ ⇒ ⟨Value v, (l, m)⟩; v ≠ Void_v ∧ v has_type T⟧ ⟹
   Γ ⊢ ⟨attached T e as n, s⟩ ⇒ ⟨True_c, (l (n ↦ v), m)⟩ |
 Test_false: ⟦Γ ⊢ ⟨e, s⟩ ⇒ ⟨Value v, (l, m)⟩; ¬ (v ≠ Void_v ∧ v has_type T)⟧ ⟹
   Γ ⊢ ⟨attached T e as n, s⟩ ⇒ ⟨False_c, (l, m)⟩ |
 Nil: Γ ⊢ ⟨[], s⟩ [⇒] ⟨[], s⟩ |
 Cons: ⟦Γ ⊢ ⟨e, s⟩ ⇒ ⟨Value v, se⟩; Γ ⊢ ⟨es, se⟩ [⇒] ⟨es', s'⟩⟧ ⟹
  Γ ⊢ ⟨e # es, s⟩ [⇒] ⟨(Value v) # es', s'⟩ |
— Exception propagation
 Exception: Γ ⊢ ⟨Exception, s⟩ ⇒ ⟨Exception, s⟩ |
 SeqEx: ⟦Γ ⊢ ⟨e1, s⟩ ⇒ ⟨Exception, s'⟩⟧ ⟹ Γ ⊢ ⟨e1;; e2, s⟩ ⇒ ⟨Exception, s'⟩ |
 AssignEx: ⟦Γ ⊢ ⟨e, s⟩ ⇒ ⟨Exception, s'⟩⟧ ⟹
  Γ ⊢ ⟨n ::= e, s⟩ ⇒ ⟨Exception, s'⟩ |
 CallEx: ⟦Γ ⊢ ⟨e, s⟩ ⇒ ⟨Exception, s'⟩⟧ ⟹
  Γ ⊢ ⟨e . f (es), s⟩ ⇒ ⟨Exception, s'⟩ |
 CallArgEx: ⟦
```

$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_e\rangle;$
$\Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Value}\ vs\ @\ \mathit{Exception}\ \#\ es', s'\rangle$
$]] \Longrightarrow \Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle\ |$
IfEx: $[\![\Gamma \vdash \langle b, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle\ |$
LoopEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle\ |$
Loop$_{false}$*Ex*: $[\![$
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{False}_c, s_e\rangle;$
$\Gamma \vdash \langle b, s_e\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$
$]] \Longrightarrow \Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle\ |$
TestEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{attached}\ t\ e\ \mathit{as}\ n, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle\ |$
ConsEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle e\ \#\ es, s\rangle\ [\Rightarrow]\ \langle \mathit{Exception}\ \#\ es, s'\rangle$

lemmas *big_step_induct* =
big_step_big_steps.inducts[*split_format*(*complete*)]
declare *big_step_big_steps.intros*[*simp*,*intro*]
inductive_cases *ValueE*[*elim*]: $\Gamma \vdash \langle \mathit{Value}\ v, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *LocalE*[*elim*!]: $\Gamma \vdash \langle \mathit{Local}\ n, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *SeqE*[*elim*!]: $\Gamma \vdash \langle c_1;; c_2, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *AssignE*[*elim*!]: $\Gamma \vdash \langle v::=x, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *CreateE*[*elim*!]: $\Gamma \vdash \langle \mathit{create}\ v, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *CallE*[*elim*!]: $\Gamma \vdash \langle v\ .\ f\ (a), s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *IfE*[*elim*!]: $\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{end}, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *LoopE*[*elim*!]: $\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end}, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *TestE*[*elim*!]: $\Gamma \vdash \langle \mathit{attached}\ t\ e\ \mathit{as}\ x, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *ExceptionE*[*elim*!]: $\Gamma \vdash \langle \mathit{Exception}, s\rangle \Rightarrow \langle c', s'\rangle$
inductive_cases *ConsE*[*elim*]: $\Gamma \vdash \langle e\ \#\ es, s\rangle\ [\Rightarrow]\ \langle es', s'\rangle$

B.9.2 *Final state*

Is an expression final?

definition *Final* :: $('b, 't)$ *expression* $\Rightarrow$ *bool* **where**
$\mathit{Final}\ e \longleftrightarrow (\exists\ v.\ e = \mathit{Value}\ v) \vee e = \mathit{Exception}$

Are expressions final?

definition
Finals es $\longleftrightarrow$
$(\exists\ vs.\ es = \mathit{map}\ \mathit{Value}\ vs) \vee$
$(\exists\ vs\ es'\ .\ es = \mathit{map}\ \mathit{Value}\ vs\ @\ (\mathit{Exception}\ \#\ es'))$

If there is a transition according to the big-step predicate, the resulting expression is final.

lemma
fixes
e e' :: ('*b*, '*t*) *expression* **and** *es es'* :: ('*b*, '*t*) *expression list*
shows
big_step_final: $\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow$ *Final e'* **and**
big_step_finals: $\Gamma \vdash \langle es, s\rangle\ [\Rightarrow]\ \langle es', s'\rangle \Longrightarrow$ *Finals es'*
proof (*induction rule*: *big_step_big_steps.inducts*)
case *Cons* **then show** *?case* **using** *Finals_def append_Cons*
by (*metis* (*no_types*) *list.simps*(9))
next
case *ConsEx* **then show** *?case* **using** *Finals_def map_append* **by** *blast*
qed (*simp_all add*: *Final_def Finals_def*)

There is only an identity transition from a final expression.

lemma *no_progress*:
assumes *Final e*
shows $\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e, s\rangle$
using *assms*
by (*metis Value Exception Final_def prod.collapse*)

If final expression is not an exception, it is a value.

lemma *big_step_final_value*:
assumes
HS: $\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle$ **and**
HE: $e' \neq$ *Exception*
obtains *v* **where** $e' =$ *Value v*
proof −
from *HS* **have** *Final e'* **by** (*rule big_step_final*)
with *HE* **have** $\exists\ v.\ e' =$ *Value v* **by** (*simp add*: *Final_def*)
then show *?thesis* **using** *that* **by** *auto*
qed

end

B.10 Void-unsafe Big-step semantics

theory *BigStep_unsafe* **imports**
Environment
Expression
Memory
State
begin

B.10.1 *Big-step semantics rules*

abbreviation *has_type_rep* (**infix** *has'_type 60*) **where**
has_type_rep v T ≡ (*T* = *None* ∨ *is_instance_of v T*)

Void-unsafe big-step semantics predicate: there is a rule for a call when its target is void.

inductive
big_step :: *'a environment* ⇒ (*'b*, *'t*) *expression* ⇒ (*'b*, *'m*) *state* ⇒
(*'b*, *'t*) *expression* ⇒ (*'b*, *'m*) *state* ⇒ *bool*
(_ ⊢ ⟨_, _⟩ ⇒ ⟨_, _⟩ [80, *o*, *o*] 81)
and
big_steps :: *'a environment* ⇒ (*'b*, *'t*) *expression list* ⇒ (*'b*, *'m*) *state* ⇒
(*'b*, *'t*) *expression list* ⇒ (*'b*, *'m*) *state* ⇒ *bool*
(_ ⊢ ⟨_, _⟩ [⇒] ⟨_, _⟩ [80, *o*, *o*] 81)
for
Γ :: *'a environment*
where
Value: Γ ⊢ ⟨*Value v*, (*l*, *m*)⟩ ⇒ ⟨*Value v*, (*l*, *m*)⟩ |
Local: *l n* = ⌊*v*⌋ ⟹ Γ ⊢ ⟨*Local n*, (*l*, *m*)⟩ ⇒ ⟨*Value v*, (*l*, *m*)⟩ |
Seq: ⟦Γ ⊢ ⟨c_1, *s*⟩ ⇒ ⟨*unit*, *s'*⟩; Γ ⊢ ⟨c_2, *s'*⟩ ⇒ ⟨c_2', *s''*⟩⟧ ⟹
Γ ⊢ ⟨c_1;; c_2, *s*⟩ ⇒ ⟨c_2', *s''*⟩ |
Assign: ⟦Γ ⊢ ⟨*e*, *s*⟩ ⇒ ⟨*Value v*, (*l*, *m*)⟩⟧ ⟹
Γ ⊢ ⟨*n* ::= *e*, *s*⟩ ⇒ ⟨*unit*, (*l* (*n* ↦ *v*), *m*)⟩ |
Create: ⟦Γ *n* = ⌊*T*⌋; *instance m T* = ⌊(*m'*, *v*)⌋⟧ ⟹
Γ ⊢ ⟨*create n*, (*l*, *m*)⟩ ⇒ ⟨*unit*, (*l* (*n* ↦ *v*), *m'*)⟩ |
$Create_{fail}$: ⟦Γ *n* = ⌊*T*⌋; *instance m T* = (*None*::(*'m* × *'b value*) *option*)⟧ ⟹
Γ ⊢ ⟨*create n*, (*l*, *m*)⟩ ⇒ ⟨*Exception*, (*l*, *m*)⟩ |
Call: ⟦
Γ ⊢ ⟨*e*, *s*⟩ ⇒ ⟨*Value v*, s_e⟩;
v ≠ $Void_v$;
Γ ⊢ ⟨*es*, s_e⟩ [⇒] ⟨*map Value vs*, *s'*⟩
⟧ ⟹ Γ ⊢ ⟨*e* . *f* (*es*), *s*⟩ ⇒ ⟨*unit*, *s'*⟩ |
$Call_{fail}$: ⟦Γ ⊢ ⟨*e*, *s*⟩ ⇒ ⟨*Value v*, *s'*⟩; *v* = $Void_v$⟧ ⟹
Γ ⊢ ⟨*e* . *f* (*es*), *s*⟩ ⇒ ⟨*Exception*, *s'*⟩ |
If_{true}: ⟦Γ ⊢ ⟨*b*, *s*⟩ ⇒ ⟨$True_c$, *s'*⟩; Γ ⊢ ⟨c_1, *s'*⟩ ⇒ ⟨c_1', *s''*⟩⟧ ⟹
Γ ⊢ ⟨*if b then* c_1 *else* c_2 *end*, *s*⟩ ⇒ ⟨c_1', *s''*⟩ |
If_{false}: ⟦Γ ⊢ ⟨*b*, *s*⟩ ⇒ ⟨$False_c$, *s'*⟩; Γ ⊢ ⟨c_2, *s'*⟩ ⇒ ⟨c_2', *s''*⟩⟧ ⟹
Γ ⊢ ⟨*if b then* c_1 *else* c_2 *end*, *s*⟩ ⇒ ⟨c_2', *s''*⟩ |
$Loop_{true}$: ⟦Γ ⊢ ⟨*e*, *s*⟩ ⇒ ⟨$True_c$, *s'*⟩⟧ ⟹
Γ ⊢ ⟨*until e loop c end*, *s*⟩ ⇒ ⟨*unit*, *s'*⟩ |
$Loop_{false}$: ⟦
Γ ⊢ ⟨*e*, *s*⟩ ⇒ ⟨$False_c$, s_e⟩;
Γ ⊢ ⟨*c*, s_e⟩ ⇒ ⟨*unit*, s_c⟩;
Γ ⊢ ⟨*until e loop c end*, s_c⟩ ⇒ ⟨*c'*, *s'*⟩
⟧ ⟹ Γ ⊢ ⟨*until e loop c end*, *s*⟩ ⇒ ⟨*c'*, *s'*⟩ |

Test$_{true}$: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, (l, m)\rangle; v \neq \mathit{Void}_{\mathbf{v}} \wedge v\ \mathit{has\_type}\ T]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n, s\rangle \Rightarrow \langle \mathit{True}_{\mathbf{c}}, (l\ (n \mapsto v), m)\rangle$ |
Test$_{false}$: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, (l, m)\rangle; \neg\ (v \neq \mathit{Void}_{\mathbf{v}} \wedge v\ \mathit{has\_type}\ T)]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n, s\rangle \Rightarrow \langle \mathit{False}_{\mathbf{c}}, (l, m)\rangle$ |
Nil: $\Gamma \vdash \langle [], s\rangle\ [\Rightarrow]\ \langle [], s\rangle$ |
Cons: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_{\mathbf{e}}\rangle; \Gamma \vdash \langle es, s_{\mathbf{e}}\rangle\ [\Rightarrow]\ \langle es', s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle e\ \#\ es, s\rangle\ [\Rightarrow]\ \langle (\mathit{Value}\ v)\ \#\ es', s'\rangle$ |
— Exception propagation
Exception: $\Gamma \vdash \langle \mathit{Exception}, s\rangle \Rightarrow \langle \mathit{Exception}, s\rangle$ |
SeqEx: $[\![\Gamma \vdash \langle c_1, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle c_1;; c_2, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
AssignEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle n ::= e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
CallEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
CallArgEx: $[\![$
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_{\mathbf{e}}\rangle;$
$\Gamma \vdash \langle es, s_{\mathbf{e}}\rangle\ [\Rightarrow]\ \langle \mathit{map\ Value\ vs}\ @\ \mathit{Exception}\ \#\ es', s'\rangle$
$]\!] \Longrightarrow \Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
IfEx: $[\![\Gamma \vdash \langle b, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{if\ b\ then\ c1\ else\ c2\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
LoopEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{until\ e\ loop\ c\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
Loop$_{false}$*Ex*: $[\![$
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{False}_{\mathbf{c}}, s_{\mathbf{e}}\rangle;$
$\Gamma \vdash \langle c, s_{\mathbf{e}}\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$
$]\!] \Longrightarrow \Gamma \vdash \langle \mathit{until\ e\ loop\ c\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
TestEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle \mathit{attached}\ \_\ e\ \mathit{as}\ \_, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle$ |
ConsEx: $[\![\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle]\!] \Longrightarrow$
$\Gamma \vdash \langle e\ \#\ es, s\rangle\ [\Rightarrow]\ \langle \mathit{Exception}\ \#\ es, s'\rangle$

no_notation *big_step* ($\_ \vdash \langle \_, \_\rangle \Rightarrow \langle \_, \_\rangle$ [80, 0, 0] 81)
no_notation *big_steps* ($\_ \vdash \langle \_, \_\rangle\ [\Rightarrow]\ \langle \_, \_\rangle$ [80, 0, 0] 81)
abbreviation *big_step_rep* ($\_ \vdash \langle \_, \_\rangle \Rightarrow' \langle \_, \_\rangle$ [80, 0, 0] 81)
where *big_step_rep* ≡ *big_step*
abbreviation *big_steps_rep* ($\_ \vdash \langle \_, \_\rangle\ [\Rightarrow]'\ \langle \_, \_\rangle$ [80, 0, 0] 81)
where *big_steps_rep* ≡ *big_steps*

declare *big_step_big_steps.intros*[*simp*,*intro*]

end

B.11 Types with attachment status

theory *TypeAttachment* **imports** *Type* **begin**

B.11.1 Type abstraction describing attachment status

datatype *attachment_type* =
Attached | — Expanded or attached reference type
Detachable — Detachable reference type

B.11.1.1 Attachment types lattice

instantiation *attachment_type* :: *complete_lattice*
begin
definition *Inf X* ≡ (*if Attached* ∈ *X then Attached else Detachable*)
definition *Sup X* ≡ (*if Detachable* ∈ *X then Detachable else Attached*)
definition *inf A B* ≡ (*if A* = *Attached then Attached else B*)
definition *sup A B* ≡ (*if A* = *Detachable then Detachable else B*)
definition [*simp*]: *bot* ≡ *Attached*
definition [*simp*]: *top* ≡ *Detachable*
definition *less_eq A B* ≡ *A* = *B* ∨ (*A* = *Attached* ∧ *B* = *Detachable*)
definition *less A B* ≡ *A* = *Attached* ∧ *B* = *Detachable*
instance proof
fix *x y* :: *attachment_type*
show (*x* < *y*) = (*x* ⩽ *y* ∧ ¬ *y* ⩽ (*x*::*attachment_type*))
using *less_attachment_type_def less_eq_attachment_type_def* **by** *auto*
show *x* ⩽ *x*
by (*simp add*: *less_eq_attachment_type_def*)
show *inf x y* ⩽ *x*
by (*metis attachment_type.exhaust inf_attachment_type_def*
less_eq_attachment_type_def)
show *inf x y* ⩽ *y*
by (*metis attachment_type.exhaust inf_attachment_type_def*
less_eq_attachment_type_def)
show *x* ⩽ *sup x y*
by (*metis attachment_type.exhaust sup_attachment_type_def*
less_eq_attachment_type_def)
show *y* ⩽ *sup x y*
by (*metis attachment_type.exhaust sup_attachment_type_def*
less_eq_attachment_type_def)
show *Inf* {} = (*top* :: *attachment_type*)
by (*simp add*: *Inf_attachment_type_def*)
show *Sup* {} = (*bot* :: *attachment_type*)
by (*simp add*: *Sup_attachment_type_def*)
next

fix $x\ y\ z$ :: *attachment_type*
assume $x \leqslant y\ y \leqslant z$
then show $x \leqslant z$ **using** *less_eq_attachment_type_def* **by** *auto*
next
fix $x\ y$ :: *attachment_type*
assume $x \leqslant y\ y \leqslant x$
then show $x = y$ **using** *less_eq_attachment_type_def* **by** *auto*
next
fix $x\ y\ z$ :: *attachment_type*
assume $x \leqslant y\ x \leqslant z$
then show $x \leqslant inf\ y\ z$ **using** *inf_attachment_type_def* **by** *auto*
next
fix $x\ y\ z$ :: *attachment_type*
assume $y \leqslant x\ z \leqslant x$
then show $sup\ y\ z \leqslant x$ **using** *sup_attachment_type_def* **by** *auto*
next
fix x :: *attachment_type* **and** A
assume $x \in A$
then show $Inf\ A \leqslant x$
by (*metis Inf_attachment_type_def attachment_type.exhaust less_eq_attachment_type_def*)
next
fix x :: *attachment_type* **and** A
assume $x \in A$
then show $x \leqslant Sup\ A$
by (*metis Sup_attachment_type_def attachment_type.exhaust less_eq_attachment_type_def*)
next
fix z :: *attachment_type* **and** A
assume $\bigwedge x.\ x \in A \Longrightarrow z \leqslant x$
then show $z \leqslant Inf\ A$
using *Inf_attachment_type_def attachment_type.exhaust less_eq_attachment_type_def* **by** *auto*
next
fix z :: *attachment_type* **and** A
assume $\bigwedge x.\ x \in A \Longrightarrow x \leqslant z$
then show $Sup\ A \leqslant z$
using *Sup_attachment_type_def attachment_type.exhaust less_eq_attachment_type_def* **by** *auto*
qed
end

instantiation *attachment_type* :: *distrib_lattice*
begin
instance proof
fix $x\ y\ z$ :: *attachment_type*

```
  show sup x (inf y z) = inf (sup x y) (sup x z)
   using sup_attachment_type_def by auto
 qed
end

abbreviation (input) attachment_type_less_eq_rep (_ →ₐ _ [90, 90] 90)
 where A →ₐ B ≡ A ⩽ (B :: attachment_type)
```

B.11.1.2 *Conformance of attachment types*

```
lemma attachment_conforming_to_transitive [trans]:
 ⟦A →ₐ B; B →ₐ C⟧ ⟹ A →ₐ C
  by simp

lemma attachment_conforming_to_reflexive [simp]: A →ₐ A
 by simp

lemma attachment_conforming_to_antisymmetric:
 ⟦A →ₐ B; B →ₐ A⟧ ⟹ A = B
  by simp

lemma attachment_conforming_to_asymmetric: ¬ A →ₐ B ⟹ B →ₐ A
 using less_eq_attachment_type_def
 by (metis (full_types) attachment_type.exhaust)

lemma conforms_to_detachable [simp]: T →ₐ Detachable
 using attachment_conforming_to_asymmetric less_eq_attachment_type_def
 by auto

lemma conforms_to_attached [intro?]: T →ₐ Attached ⟹ T = Attached
 by (simp add: less_eq_attachment_type_def)

lemma attached_conforms_to: Attached →ₐ T
 using attachment_type.exhaust less_eq_attachment_type_def by auto
```

B.11.1.3 *Conformance of attachment type lists*

```
fun list_conformance :: attachment_type list ⇒ attachment_type list ⇒ bool
 ((_/ [⩽] _) [51, 51] 50)
where
 [] [⩽] [] ⟷ True
| A # As [⩽] B # Bs ⟷ A ⩽ B ∧ As [⩽] Bs
| _ [⩽] _ ⟷ False

lemma conformant_lists_zip1_length:
 As [⩽] Bs ⟹ length As = length (zip As Bs)
```

proof (*induction As arbitrary*: *Bs*, *simp*)
fix *A As Bs*
assume
A # As [⩽] *Bs*
then obtain *B' Bs'* **where** *B' # Bs' = Bs* **and** *A ⩽ B' ∧ As* [⩽] *Bs'*
by (*metis list.discI list.inject list_conformance.elims*(2))
moreover assume
⋀*Bs*. *As* [⩽] *Bs* ⟹ *length As = length* (*zip As Bs*)
ultimately show *length* (*A # As*) = *length* (*zip* (*A # As*) *Bs*) **by** *auto*
qed

lemma *conformant_lists_zip2_length*:
As [⩽] *Bs* ⟹ *length Bs = length* (*zip As Bs*)
proof (*induction Bs arbitrary*: *As*, *simp*)
fix *As B Bs*
assume
As [⩽] *B # Bs*
then obtain *A' As'* **where** *A' # As' = As* **and** *A' ⩽ B ∧ As'* [⩽] *Bs*
by (*metis list.discI list.inject list_conformance.elims*(2))
moreover assume
⋀*As*. *As* [⩽] *Bs* ⟹ *length Bs = length* (*zip As Bs*)
ultimately show *length* (*B # Bs*) = *length* (*zip As* (*B # Bs*)) **by** *auto*
qed

lemma *conformant_lists_length*: *As* [⩽] *Bs* ⟹ *length As = length Bs*
by (*simp only*: *conformant_lists_zip1_length conformant_lists_zip2_length*)

B.11.1.4 *Upper bound of attachment status*

abbreviation (*input*) *upper_bound A B* ≡ *sup A* (*B* :: *attachment_type*)

abbreviation (*input*) *lower_bound A B* ≡ *inf A* (*B* :: *attachment_type*)

lemma *upper_bound_left*: *A* →ₐ *upper_bound A B*
by *simp*

lemma *upper_bound_right*: *B* →ₐ *upper_bound A B*
by *simp*

lemma *upper_bound_definition*:
A →ₐ *upper_bound A B ∧ B* →ₐ *upper_bound A B*
by *simp*

lemma *upper_bound_commute*: *upper_bound A B = upper_bound B A*
by (*rule sup_commute*)

lemma *upper_bound_assoc*: *upper_bound* (*upper_bound A B*) *C* =
upper_bound A (*upper_bound B C*)
by (*rule sup_assoc*)

lemma *upper_bound_conf*:
A $\rightarrow_{\mathfrak{a}}$ *B* $\Longrightarrow$ *upper_bound A C* $\rightarrow_{\mathfrak{a}}$ *upper_bound B C*
using *sup.mono* **by** *blast*

lemma *upper_bound_bot_left*:
upper_bound A B $\rightarrow_{\mathfrak{a}}$ *Attached* $\Longrightarrow$ *A* = *Attached*
by (*simp add*: *conforms_to_attached*)

lemma *upper_bound_bot_right*:
upper_bound A B $\rightarrow_{\mathfrak{a}}$ *Attached* $\Longrightarrow$ *B* = *Attached*
by (*simp add*: *conforms_to_attached*)

lemma *upper_bound_bot_left'*:
upper_bound A B = *Attached* $\Longrightarrow$ *A* = *Attached*
by (*metis bot_attachment_type_def sup_eq_bot_iff*)

lemma *upper_bound_bot_right'*:
upper_bound A B = *Attached* $\Longrightarrow$ *B* = *Attached*
by (*metis bot_attachment_type_def sup_eq_bot_iff*)

lemma *upper_bound_absorb_right* [*simp*]: *upper_bound Attached A* = *A*
by (*simp add*: *sup_attachment_type_def*)

lemma *upper_bound_absorb_left* [*simp*]: *upper_bound A Attached* = *A*
by (*metis bot_attachment_type_def sup_bot.right_neutral*)

definition *attachment_type_Union* :: *attachment_type set* $\Rightarrow$ *attachment_type*
where
attachment_type_Union A =
(*if Attached* $\in$ *A then Attached else Detachable*)

definition *attachment_type_Inter* :: *attachment_type set* $\Rightarrow$ *attachment_type*
where
attachment_type_Inter A =
(*if* (*Detachable* $\in$ *A*) *then Detachable else Attached*)

B.11.1.5 *Attached type properties*

definition *is_attached_type* :: *attachment_type* $\Rightarrow$ *bool* **where**
is_attached_type t $\longleftrightarrow$ *t* = *Attached*

lemma *attached_is_attached* [*simp*]: *is_attached_type Attached*

by (*simp only*: *is_attached_type_def*)

lemma *is_attached_conforms_to* [*simp*]: *is_attached_type A* $\Longrightarrow$ *A* $\rightarrow_{\mathfrak{a}}$ *B*
by (*simp add*: *attached_conforms_to is_attached_type_def*)

lemma *is_attached_not_upper_bound* [*simp*]:
is_attached_type A $\Longrightarrow$ *upper_bound A B* = *B* **and**
is_attached_type A $\Longrightarrow$ *upper_bound B A* = *B*
by (*auto simp add*: *attachment_conforming_to_antisymmetric*)

lemma *is_attached_conformance*:
is_attached_type B $\longrightarrow$ *is_attached_type A* $\Longrightarrow$ *A* $\rightarrow_{\mathfrak{a}}$ *B*
using *attachment_conforming_to_asymmetric is_attached_type_def*
less_eq_attachment_type_def **by** *auto*

lemma *conformance_consistency*:
A $\rightarrow_{\mathfrak{a}}$ *B* $\Longrightarrow$ *is_attached_type B* $\longrightarrow$ *is_attached_type A*
using *less_eq_attachment_type_def* **by** *auto*

B.11.2 *Attachment status of types*

datatype *attachment_mark* =
No_attachment_mark |
Attached_mark |
Detachable_mark

type_synonym *attachable_type* = *attachment_mark type*

end

B.12 Values with attachment status

theory *ValueAttachment* **imports** *TypeAttachment Value* **begin**

B.12.1 *Attachment type of simple expressions*

definition *declared_attachment_type_of_constant* ::
'a value $\Rightarrow$ *attachment_type* ($\mathcal{T}_{\mathfrak{c}}$)
where
$\mathcal{T}_{\mathfrak{c}}$ *v* = (*case v of*
Unit $\Rightarrow$ *Attached* |
Object _ $\Rightarrow$ *Attached* |
Reference x $\Rightarrow$ (*case x of AttachedObject _* $\Rightarrow$ *Attached* | *_* $\Rightarrow$ *Detachable*))
declare *declared_attachment_type_of_constant_def* [*simp*]

lemma *expanded_attachment*: *is_expanded v* $\Longrightarrow$ $\mathcal{T}_c$ *v* = *Attached*
by (*cases v*, *simp_all*)

lemma *reference_attachment*:
is_reference v $\Longrightarrow$ $\mathcal{T}_c$ *v* = *Attached* $\vee$ $\mathcal{T}_c$ *v* = *Detachable*
proof (*cases v*)
case (*Reference x*) **thus** *?thesis* **by** (*cases x*, *simp_all*)
qed *simp_all*

definition *is_attached_value* :: *'a value* $\Rightarrow$ *bool* **where**
is_attached_value v = *is_attached_type* ($\mathcal{T}_c$ *v*)
declare *is_attached_value_def* [*simp*]

lemma *attached_value_is_attached_constant* [*iff*]:
is_attached v $\longleftrightarrow$ *is_attached_value v*
proof (*cases v*)
case (*Reference x*)
then show *?thesis* **by** (*cases x*) (*simp_all add*: *is_attached_type_def*)
qed *simp_all*

lemma *attached_value_is_not_void* [*iff*]: *is_attached_value v* $\longleftrightarrow$ *v* $\neq$ $Void_v$
using *is_attached_if_not_void* **by** *fastforce*

lemma *attached_typed_value_is_not_void* [*iff*]:
$\mathcal{T}_c$ *v* = *Attached* $\longleftrightarrow$ *v* $\neq$ $Void_v$
using *attached_value_is_not_void is_attached_type_def* **by** *auto*

lemma *detachable_typed_value_is_void* [*iff*]:
$\mathcal{T}_c$ *v* = *Detachable* $\longleftrightarrow$ *v* = $Void_v$
proof (*cases v*)
case (*Reference x*) **then show** *?thesis*
by (*cases x*) (*simp_all add*: *is_attached_type_def*)
qed *simp_all*

end

B.13 Attachment properties of object heap

theory *MemoryAttachment* **imports** *Memory ValueAttachment* **begin**

lemma *instance_is_attached*:
instance m t = $\lfloor$(*m'*, *v*)$\rfloor$ $\Longrightarrow$ *is_attached_value v*
proof −
assume *instance m t* = $\lfloor$(*m'*, *v*)$\rfloor$

then have
HA: *Value.is_attached v* **by** (*auto simp add*: *instance_def*)
then show *?thesis*
proof (*cases v*)
case (*Reference x*)
then show *?thesis* **by** (*cases x*, *insert Reference HA*, *simp_all*)
qed *simp_all*
qed

lemma *instance_is_not_void*: *instance m t* = ⌊(*m'*, *v*)⌋ ⟹ *v* ≠ *Void*$_v$
using *instance_is_attached* **using** *attached_value_is_not_void* **by** *simp*

lemma *instance_is_value*: *instance m t* = ⌊(*m'*, *v*)⌋ ⟹ *is_value v*
proof −
assume *instance m t* = ⌊(*m'*, *v*)⌋
moreover hence *is_attached_value v* **by** (*rule instance_is_attached*)
ultimately show *?thesis*
by (*simp add*: *basic_memory.direct_instance instance_def is_value_def*)
qed

end

B.14 Type environment with attachment marks

theory *EnvironmentAttachment* **imports** *TypeAttachment Environment*
begin

type_synonym *attachment_environment* = *attachment_mark environment*

end

B.15 Expression with attached types

theory *ExpressionAttachment* **imports** *Expression TypeAttachment* **begin**

type_synonym *'b attachable_expression* = (*'b*, *attachable_type*) *expression*

end

B.16 Set with absorbing top element

theory *TopSet* **imports**
~~/*src/HOL/Lattice/Orders*

```
 Common
begin

datatype 'a topset =
  Top |
  Set (the: 'a set) (⌈_⌉)

declare [[coercion λx::'a set. Set x]]

definition topset_subset :: 'a topset ⇒ 'a topset ⇒ bool
where
  topset_subset A B ≡
   case B of Top ⇒ True | ⌈b⌉ ⇒ (case A of Top ⇒ False | ⌈a⌉ ⇒ a ⊆ b)

instantiation topset :: (type) complete_lattice
begin

  definition less_eq_topset_def: less_eq_topset ≡ topset_subset
  definition less_topset_def:
   less n1 n2 ≡ (topset_subset n1 n2) ∧ ¬ (topset_subset n2 n1)

  definition topset_union :: 'a topset ⇒ 'a topset ⇒ 'a topset (infixl ⊔ 65)
   where
     A ⊔ B  ≡
      case A of Top ⇒ Top | Set a ⇒ (case B of Top ⇒ Top | Set b ⇒ Set (a ∪ b))

  definition topset_inter :: 'a topset ⇒ 'a topset ⇒ 'a topset (infixl ⊓ 70)
   where
     A ⊓ B  ≡
      case A of Top ⇒ B | Set a ⇒ (case B of Top ⇒ A | Set b ⇒ Set (a ∩ b))

  definition topset_Union :: 'a topset set ⇒ 'a topset where
   topset_Union A = (if Top ∈ A then Top else ⌈⋃ {a. ⌈a⌉ ∈ A}⌉)

  definition topset_Inter :: 'a topset set ⇒ 'a topset where
   topset_Inter A = (if (∃ a. ⌈a⌉ ∈ A) then ⌈⋂ {a. ⌈a⌉ ∈ A}⌉ else Top)

  definition [simp]: Inf = topset_Inter
  definition [simp]: Sup = topset_Union
  definition [simp]: bot = Set {}
  definition [iff]: top = Top
  definition [simp]: inf = topset_inter
  definition [simp]: sup = topset_union
  instance proof
   fix x y :: 'a topset
   show (x < y) = (x ⩽ y ∧ ¬ y ⩽ x)
```

```
  by (simp add: less_topset_def less_eq_topset_def)
next
 fix x y :: 'a topset
 show x ⩽ x
  by (cases x, simp_all add: topset_subset_def less_eq_topset_def)
next
 fix x y z :: 'a topset
 assume x ⩽ y y ⩽ z
 then show x ⩽ z
  apply (cases x)
  apply (cases y)
 apply simp
 apply (simp add: less_eq_topset_def topset_subset_def)
 apply (cases z)
 apply (simp add: less_eq_topset_def topset_subset_def)
 apply (cases y)
 apply (simp add: less_eq_topset_def topset_subset_def)
 by (metis less_eq_topset_def order.trans topset.simps(5) topset_subset_def)
next
 fix x y :: 'a topset
 assume x ⩽ y y ⩽ x
 then show x = y
   apply (cases x)
   apply (cases y)
   apply (simp_all add: topset_subset_def less_eq_topset_def)
   apply (cases y)
   apply simp_all
  done
next
 fix x y :: 'a topset
 show inf x y ⩽ x
   apply (cases x)
   apply (simp add: topset_subset_def less_eq_topset_def)
   apply (cases y)
   apply (simp_all add: less_eq_topset_def topset_inter_def
     topset_subset_def)
  done
next
 fix x y :: 'a topset
 show inf x y ⩽ y
   apply (cases y)
   apply (simp add: topset_subset_def less_eq_topset_def)
   apply (cases x)
   apply (simp_all add: topset_inter_def topset_subset_def
     less_eq_topset_def)
  done
```

```
next
  fix x y z :: 'a topset
  assume x ⩽ y x ⩽ z
  then show x ⩽ inf y z
    apply (cases y)
    apply (simp add: topset_inter_def)
    apply (cases z)
    apply (simp add: topset_inter_def)
    apply (cases x)
    apply (simp_all add: topset_inter_def topset_subset_def
      less_eq_topset_def)
  done
next
  fix x y :: 'a topset
  show x ⩽ sup x y
    by (simp add: less_eq_topset_def topset_union_def topset.case_eq_if
      topset_subset_def)
next
  fix x y :: 'a topset
  show y ⩽ sup x y
    apply (cases x)
    apply (simp add: topset_subset_def topset_union_def less_eq_topset_def)
    apply (cases y)
    apply (simp_all add: topset_subset_def topset_union_def
      less_eq_topset_def)
  done
next
  fix x y z:: 'a topset
  assume y ⩽ x z ⩽ x
  then show sup y z ⩽ x
    apply (cases y)
    apply (simp add: topset_union_def)
    apply (cases z)
    apply (simp add: topset_union_def)
    apply (cases x)
    apply (simp_all add: topset_subset_def topset_union_def
      less_eq_topset_def)
  done
next
  fix x:: 'a topset and A
  assume x ∈ A
  then show Inf A ⩽ x
  proof (cases x)
    case Top then show ?thesis by (simp add: topset_subset_def
                less_eq_topset_def)
  next
```

```
    case (Set s)
    moreover then have ⋂ {a. ⌈a⌉ ∈ A} ⊆ s using ‹x ∈ A› by blast
    ultimately show ?thesis using ‹x ∈ A› by
     (auto simp add: topset_subset_def topset_Inter_def less_eq_topset_def)
   qed
  next
   fix z:: 'a topset and A
   assume
    H: ⋀x. x ∈ A ⟹ z ⩽ x
   then show z ⩽ Inf A
   proof (cases z)
    case Top with H show ?thesis
     by (metis Inf_topset_def less_eq_topset_def topset.simps(4)
      topset.simps(5) topset_Inter_def topset_subset_def)
   next
    case (Set s)
    assume z = ⌈s⌉
    with H have ⋀x. ⌈x⌉ ∈ A ⟹ s ⩽ x
     using topset_subset_def less_eq_topset_def
  by (metis topset.simps(5))
    then have s ⩽ ⋂ {x. ⌈x⌉ ∈ A} by auto
    then have
     HH: ⌈s⌉ ⩽ ⌈⋂ {x. ⌈x⌉ ∈ A}⌉
      by (simp add: less_eq_topset_def topset_subset_def)
    show ?thesis
    proof (cases Inf A)
     case Top then show ?thesis
      by (simp add: less_eq_topset_def topset_subset_def)
    next
     case (Set a)
     have z = ⌈s⌉ by (simp add: ‹z = ⌈s⌉›)
     also have ... ⩽ ⌈⋂ {x. ⌈x⌉ ∈ A}⌉ using HH by simp
     also have ... = Inf A using Set
      by (metis Inf_topset_def topset.distinct(1) topset_Inter_def)
     finally show ?thesis by simp
    qed
   qed
  next
   fix x:: 'a topset and A
   assume x ∈ A
   then show x ⩽ Sup A
   proof (cases x)
    case Top then show ?thesis using ‹x ∈ A› topset_Union_def
     by (simp add: less_eq_topset_def topset_subset_def)
   next
    case (Set s)
```

```
    moreover then have s ⊆ ⋃ {a. ⌈a⌉ ∈ A} using ‹x ∈ A› by blast
    ultimately show ?thesis using ‹x ∈ A›
     by (simp add: topset_Union_def topset_subset_def less_eq_topset_def)
   qed
 next
   fix z:: 'a topset and A
   assume
    H: ⋀x. x ∈ A ⟹ x ⩽ z
   then show Sup A ⩽ z
   proof (cases z)
    case Top with H show ?thesis
     by (simp add: topset_subset_def less_eq_topset_def)
   next
    case (Set s)
    assume z = ⌈s⌉
    with H have ⋀x. ⌈x⌉ ∈ A ⟹ ⌈x⌉ ⩽ z by simp
    with Set have ⋀x. ⌈x⌉ ∈ A ⟹ x ⩽ s
     by (simp add: less_eq_topset_def topset_subset_def)
    then have ⋃ {x. ⌈x⌉ ∈ A} ⩽ s by auto
    then have
     HH: ⌈⋃ {x. ⌈x⌉ ∈ A}⌉ ⩽ ⌈s⌉
      by (simp add: topset_subset_def less_eq_topset_def)
    show ?thesis
    proof (cases Sup A)
     case Top then show ?thesis using H HH Set topset_Union_def by force
    next
     case (Set a)
     then have Sup A = ⌈⋃ {x. ⌈x⌉ ∈ A}⌉
      by (metis Sup_topset_def topset.distinct(1) topset_Union_def)
     also have ... ⩽ ⌈s⌉ using HH by simp
     also have ... = z by (simp add: ‹z = ⌈s⌉›)
     finally show ?thesis by simp
    qed
   qed
 next
   show Inf {} = (top:: 'a topset) by (simp add: topset_Inter_def)
 next
   show Sup {} = (bot:: 'a topset) by (simp add: topset_Union_def)
 qed

 declare
   Inf_topset_def
   Sup_topset_def
   bot_topset_def
   inf_topset_def
   sup_topset_def [simp del]
```

```
 notation Top (⊤)

end

instantiation topset :: (type) distrib_lattice
begin
 instance proof
  fix x y z:: 'a topset
  show sup x (inf y z) = inf (sup x y) (sup x z)
   apply (cases x)
   apply (metis inf_sup_absorb sup_top_left top_topset_def)
   apply (cases y)
   apply (metis inf_top_left sup_top_right top_topset_def)
   apply (cases z)
   apply (metis inf_top_right sup_top_right top_topset_def)
   apply (simp add: topset_inter_def topset_union_def Un_Int_distrib
     sup_topset_def)
  done
 qed
end

definition topset_member :: 'a ⇒ 'a topset ⇒ bool (infix ∈⊤ 50)
where
 x ∈⊤ A ≡ (case A of Top ⇒ True | ⌈a⌉ ⇒ x ∈ a)

lemma topset_member_top [simp]: c ∈⊤ ⊤
 by (simp add: topset_member_def)

lemma topset_member_set [iff]: c ∈⊤ ⌈a⌉ ⟷ c ∈ a
 by (simp add: topset_member_def)

definition topset_insert :: 'a ⇒ 'a topset ⇒ 'a topset where
 topset_insert a B = (case B of Top ⇒ Top | Set b ⇒ ⌈{x. x = a ∨ x ∈ b}⌉)

lemma topset_subsetI [intro!]: ⟦⋀x. x ∈⊤ A ⟹ x ∈⊤ B; A ≠ ⊤⟧ ⟹ A ⩽ B
proof −
 assume
  H1: ⋀x. x ∈⊤ A ⟹ x ∈⊤ B and
  H2: A ≠ ⊤
 then obtain a where
  H3: A = ⌈a⌉ using topset.exhaust by auto
 show ?thesis
 proof (cases B)
  case Top then show ?thesis using top_greatest [of A] by simp
 next
```

```
    case (Set b)
    with H1 H3 topset_member_def have ⋀x. x ∈ a ⟹ x ∈ b by fastforce
    then have a ⊆ b by auto
    with H3 Set show ?thesis
     by (simp add: less_eq_topset_def topset_subset_def)
  qed
qed

lemma topset_subsetD [elim, intro?]: ⟦A ⩽ B; c ∈⊤ A⟧ ⟹ c ∈⊤ B
proof (cases A)
  case Top
  assume
    Hle: A ⩽ B and
    Hin: c ∈⊤ A and
    HA: A = Top
  then have A = ⊤ by simp
  with Hle have B = ⊤ using dual_order.antisym by fastforce
  then show ?thesis using Hin HA by simp
next
  case (Set a)
  assume
    Hle: A ⩽ B and
    Hin: c ∈⊤ A and
    HA: A = ⌈a⌉
  show ?thesis
  proof (cases B)
    case Top
    have c ∈⊤ ⊤ by (rule topset_member_top)
    with Top show ?thesis by simp
  next
    case HB: (Set b)
    with Hin Hle HA show ?thesis
      by (simp add: less_eq_topset_def set_mp topset_member_def
        topset_subset_def)
  qed
qed

lemma rev_topset_subsetD [intro?]: ⟦c ∈⊤ A; A ⩽ B⟧ ⟹ c ∈⊤ B
  by (simp add: topset_subsetD)

lemma topset_subsetCE [elim]:
 A ⩽ B ⟹ (¬ c ∈⊤ A ⟹ P) ⟹ (c ∈⊤ B ⟹ P) ⟹ P
   — Classical elimination rule.
  by (meson topset_subsetD)

lemma topset_subset_eq: A ≠ ⊤ ⟹ A ⩽ B = (∀x. x ∈⊤ A ⟶ x ∈⊤ B)
```

by *blast*

lemma *contra_topset_subsetD*: $A \leqslant B \Longrightarrow \neg\ c \in^{\top} B \Longrightarrow \neg\ c \in^{\top} A$ **by** *blast*

lemma *topset_insert_top* [*simp*]: *topset_insert x* $\top = \top$
by (*simp add*: *topset_insert_def*)

lemma *topset_insert_set*: *topset_insert x* $\lceil a \rceil = \lceil$*insert x a*$\rceil$
by (*simp add*: *insert_compr topset_insert_def*)

lemma *topset_insertI1*: $a \in^{\top}$ *topset_insert a B*
by (*cases B*, *simp_all add*: *topset_insert_def topset_member_def*)

lemma *topset_insertI2*: $a \in^{\top} B \Longrightarrow a \in^{\top}$ *topset_insert b B*
by (*cases B*, *simp_all add*: *topset_insert_def topset_member_def*)

lemma *topset_subset_insertI*: $B \leqslant$ *topset_insert a B*
by (*cases B*) (*auto simp add*: *topset_insertI2*)

lemma *topset_subset_insertI2*: $A \leqslant B \Longrightarrow A \leqslant$ *topset_insert b B*
apply (*cases B*)
apply *simp*
by (*meson order.trans topset_subset_insertI*)

lemma *topset_subset_insert*:
$\neg\ x \in^{\top} A \Longrightarrow (A \leqslant$ *topset_insert x B*$) = (A \leqslant B)$
proof (*cases A*)
assume
H: $\neg\ x \in^{\top} A$
case *Top* **with** *H* **show** *?thesis* **by** (*simp add*: *topset_member_def*)
next
case (*Set a*)
assume
H: $\neg\ x \in^{\top} A$ **and**
HA: $A = \lceil a \rceil$
then have
HM: $x \notin a$ **by** (*simp add*: *topset_member_def*)
show *?thesis*
proof (*cases B*)
case *Top* **then show** *?thesis* **by** (*simp add*: *topset_member_def*)
next
case (*Set b*)
from *HM* **have** $(a \subseteq$ *insert x b*$) = (a \subseteq b)$ **by** (*simp add*: *subset_insert*)
with *HA Set* **show** *?thesis*
by (*simp add*: *less_eq_topset_def topset_insert_set topset_subset_def*)

qed
qed

lemma *topset_le_top* [*simp*]: $A \leqslant \top$
using *top_greatest* [*of A*] **by** *simp*

lemma *topset_top_le* [*simp*]: $\top \leqslant A \Longrightarrow A = \top$
by (*simp add*: *dual_order.antisym*)

lemma *topset_union_assoc*: $A \sqcup (B \sqcup C) = A \sqcup B \sqcup C$
by (*metis sup.assoc sup_topset_def*)

lemma *topset_inter_assoc*: $A \sqcap (B \sqcap C) = A \sqcap B \sqcap C$
by (*metis inf.assoc inf_topset_def*)

lemma *topset_union_commute*: $A \sqcup B = B \sqcup A$
using *sup.commute sup_topset_def* **by** *metis*

lemma *topset_inter_commute*: $A \sqcap B = B \sqcap A$
using *inf.commute inf_topset_def* **by** *metis*

lemma *topset_union_idem* [*simp*]: $A \sqcup A = A$
by (*metis sup.idem sup_topset_def*)

lemma *topset_inter_idem* [*simp*]: $A \sqcap A = A$
by (*metis inf.idem inf_topset_def*)

lemma *topset_union_bot_right* [*simp*]: $A \sqcup \lceil\{\}\rceil = A$
by (*simp add*: *topset.case_eq_if topset_union_def*)

lemma *topset_union_bot_left* [*simp*]: $\lceil\{\}\rceil \sqcup A = A$
by (*simp add*: *topset.case_eq_if topset_union_def*)

lemma *topset_inter_bot_right* [*simp*]: $A \sqcap \lceil\{\}\rceil = \lceil\{\}\rceil$
by (*metis bot_topset_def inf_bot_right inf_topset_def*)

lemma *topset_inter_bot_left* [*simp*]: $\lceil\{\}\rceil \sqcap A = \lceil\{\}\rceil$
by (*metis bot_topset_def inf_bot_left inf_topset_def*)

lemma
topset_union_top_right [*simp*]: $A \sqcup \top = \top$ **and**
topset_union_top_left [*simp*]: $\top \sqcup A = \top$
by (*cases A*) (*simp_all add*: *topset_union_def*)

lemma *topset_union_topD1*: $A \sqcup B \neq \top \Longrightarrow A \neq (\top :: 'a\ topset)$

by *auto*

lemma *topset_union_topD2*: $A \sqcup B \neq \top \Longrightarrow B \neq (\top :: 'a\ topset)$
by *auto*

lemma *topset_union_topI* [*intro*]:
$[\![A \neq \top; B \neq \top]\!] \Longrightarrow A \sqcup B \neq (\top :: 'a\ topset)$
by (*simp add*: *topset.case_eq_if topset_union_def*)

lemma
topset_inter_top_right [*simp*]: $A \sqcap \top = A$ **and**
topset_inter_top_left [*simp*]: $\top \sqcap A = A$
by (*cases A*) (*simp_all add*: *topset_inter_def*)

lemma *topset_inter_topD1*: $A \sqcap B = \top \Longrightarrow A = (\top :: 'a\ topset)$
apply (*cases A*)
apply *simp*
apply (*cases B*)
apply (*simp_all add*: *topset_inter_def*)
done

lemma *topset_inter_topD2*: $A \sqcap B = \top \Longrightarrow B = (\top :: 'a\ topset)$
apply (*cases B*)
apply *simp*
apply (*cases A*)
apply (*simp_all add*: *topset_inter_def*)
done

lemma *topset_inter_subset_iff*: $C \leqslant A \sqcap B = (C \leqslant A \wedge C \leqslant B)$
proof −
have $(C \leqslant inf\ A\ B) = (C \leqslant A \wedge C \leqslant B)$ **by** (*fact le_inf_iff*)
then show *?thesis* **by** *simp*
qed

definition *topset_add* :: $'a\ topset \Rightarrow 'a \Rightarrow 'a\ topset$ (**infixl** $\oplus$ 65)
where
$A \oplus b \equiv$ *case A of Top* $\Rightarrow$ *Top* | *Set a* $\Rightarrow \lceil insert\ b\ a \rceil$

definition *topset_rem* :: $'a\ topset \Rightarrow 'a \Rightarrow 'a\ topset$ (**infixl** $\ominus$ 65)
where
$A \ominus b \equiv$ *case A of Top* $\Rightarrow$ *Top* | *Set a* $\Rightarrow \lceil a - \{b\} \rceil$

lemma *topset_add_insert*: $A \oplus b = topset\_insert\ b\ A$
by (*cases A*) (*simp_all add*: *topset_add_def topset_insert_set*)

lemma *topset_add_union*: $A \oplus b = A \sqcup \lceil \{b\} \rceil$

by (*cases A*) (*simp_all add*: *topset_add_insert topset_insert_set*
topset_union_def)

lemma *topset_top_add* [*simp*]: $\top \oplus x = \top$
by (*simp add*: *topset_add_def*)

lemma *topset_top_rem* [*simp*]: $\top \ominus x = \top$
by (*simp add*: *topset_rem_def*)

lemma *topset_set_add*: $\lceil a \rceil \oplus b = \lceil \mathit{insert}\ b\ a \rceil$
by (*simp add*: *topset_add_def*)

lemma *topset_set_rem*: $\lceil a \rceil \ominus b = \lceil a - \{b\} \rceil$
by (*simp add*: *topset_rem_def*)

lemma *topset_add_subset*: $A \leqslant A \oplus b$
proof (*cases A*)
case *Top* **then show** *?thesis* **using** *topset_member_top* **by** *simp*
next
case (*Set a*)
have $a \subseteq \mathit{insert}\ b\ a$ **by** (*rule subset_insertI*)
then have $\lceil a \rceil \leqslant \mathit{topset\_insert}\ b \lceil a \rceil$ **using** *topset_subset_insertI* **by** *fastforce*
with *Set* **show** *?thesis* **using** *topset_set_add*
by (*simp add*: *topset_add_insert*)
qed

lemma *topset_rem_subset*: $A \ominus b \leqslant A$
by (*cases A*) (*simp_all add*: *Diff_subset topset_subset_def*
topset_set_rem less_eq_topset_def)

lemma *topset_add_absorb* [*iff*]: $A \oplus b = \top \longleftrightarrow A = \top$
by (*simp add*: *topset.case_eq_if topset_add_def*)

lemma *topset_rem_absorb* [*iff*]: $A \ominus b = \top \longleftrightarrow A = \top$
by (*simp add*: *topset.case_eq_if topset_rem_def*)

lemma *topset_add_right_commute* [*iff*]: $A \oplus x \oplus y = A \oplus y \oplus x$
by (*metis topset_add_union topset_union_assoc topset_union_commute*)

lemma *topset_rem_right_commute* [*iff*]: $A \ominus x \ominus y = A \ominus y \ominus x$
by (*cases A*) (*simp, metis Diff_insert Diff_insert2 topset_set_rem*)

lemma *topset_union_upper1*: $A \leqslant A \sqcup B$
by (*metis sup_ge1 sup_topset_def*)

lemma *topset_union_upper2*: $B \leqslant A \sqcup B$

by (*metis sup_ge2 sup_topset_def*)

lemma *topset_inter_lower1* [*simp*]: $A \sqcap B \leqslant A$
by (*metis inf_le1 inf_topset_def*)

lemma *topset_inter_lower2* [*simp*]: $A \sqcap B \leqslant B$
by (*metis inf_le2 inf_topset_def*)

lemma *topset_insert_absorb*: $a \in^{\top} A \Longrightarrow$ *topset_insert a A* $= A$
by (*cases A*) (*simp_all add*: *insert_absorb*
topset_insert_set topset_member_def)

lemma *topset_union_mono* [*simp*]:
assumes $A \leqslant B$ **and** $C \leqslant D$
shows $A \sqcup C \leqslant B \sqcup D$
using *assms*
by (*metis sup_mono sup_topset_def*)

lemma *topset_union_mono1* [*simp*]:
assumes $A \leqslant B$
shows $A \sqcup C \leqslant B \sqcup C$
proof −
have $C \leqslant C$ **by** *simp*
from *assms this* **show** *?thesis* **by** (*rule topset_union_mono*)
qed

lemma *topset_union_mono2* [*simp*]:
assumes $A \leqslant B$
shows $C \sqcup A \leqslant C \sqcup B$
proof −
have $C \leqslant C$ **by** *simp*
from *this assms* **show** *?thesis* **by** (*rule topset_union_mono*)
qed

lemma *topset_inter_mono* [*simp*]:
assumes $A \leqslant B$ **and** $C \leqslant D$
shows $A \sqcap C \leqslant B \sqcap D$
using *assms*
by (*metis inf_mono inf_topset_def*)

lemma *topset_inter_mono1* [*simp*]:
assumes $A \leqslant B$
shows $A \sqcap C \leqslant B \sqcap C$
proof −
have $C \leqslant C$ **by** *simp*
from *assms this* **show** *?thesis* **by** (*rule topset_inter_mono*)

qed

lemma *topset_inter_mono2* [*simp*]:
assumes $A \leqslant B$
shows $C \sqcap A \leqslant C \sqcap B$
proof −
have $C \leqslant C$ **by** *simp*
from *this assms* **show** *?thesis* **by** (*rule topset_inter_mono*)
qed

lemma *topset_inter_mono_arg1*: *mono* ($\lambda A.\ A \sqcap B$)
by (*simp add*: *monoI*)

lemma *topset_inter_mono_arg2*: *mono* ($\lambda B.\ A \sqcap B$)
by (*simp add*: *monoI*)

lemma *topset_add_mono*: $A \leqslant B \Longrightarrow A \oplus x \leqslant B \oplus x$
proof −
assume
HS: $A \leqslant B$
then have
$x \in^{\top} A \oplus x$
$x \in^{\top} B \oplus x$ **by** (*simp_all add*: *topset_add_insert topset_insertI1*)
with *HS* **have**
HS': $\forall\ y.\ y \in^{\top} A \oplus x \longrightarrow y \in^{\top} B \oplus x$
by (*auto simp add*: *topset.case_eq_if topset_add_def topset_member_def*)
show *?thesis*
proof (*cases A*)
case *Top*
then have $B = \top$ **using** *HS* **by** *simp*
then show *?thesis* **by** *simp*
next
case *Set*
then show *?thesis* **by** (*simp add*: *HS' topset_subsetI*)
qed
qed

lemma *topset_rem_mono*: $A \leqslant B \Longrightarrow A \ominus x \leqslant B \ominus x$
proof (*cases A*)
case *Top* **then show** $A \leqslant B \Longrightarrow A \ominus x \leqslant B \ominus x$
by (*simp add*: *topset.case_eq_if topset_rem_def*)
next
case (*Set a*)
assume
H: $A \leqslant B$ **and**
HA: $A = \lceil a \rceil$

show *?thesis*
proof (*cases B*)
case *Top* **then show** *?thesis* **using** *topset_member_top* **by** *simp*
next
case (*Set b*)
with *H HA* **have** $a \subseteq b$
by (*simp add*: *topset_subset_def less_eq_topset_def*)
then have $a - \{x\} \subseteq b - \{x\}$ **by** (*simp add*: *Diff_mono*)
with *HA Set* **show** *?thesis*
by (*simp add*: *topset_rem_def topset_subset_def less_eq_topset_def*)
qed
qed

lemma *topset_add_rem_mono*: $A \leqslant B \Longrightarrow A \ominus x \leqslant B \oplus y$
proof−
assume
H: $A \leqslant B$
have $A \ominus x \leqslant A$ **by** (*rule topset_rem_subset*)
also have $A \leqslant B$ **by** (*rule H*)
also have $B \leqslant B \oplus y$ **by** (*rule topset_add_subset*)
finally show *?thesis* **by** *simp*
qed

lemma *topset_add_rem_mono1*: $A \leqslant B \Longrightarrow A \ominus x \leqslant B \oplus x$
using *topset_add_rem_mono* **by** (*simp add*: *less_eq_topset_def*)

lemma *topset_inter_union_distrib*: $A \sqcap (B \sqcup C) = (A \sqcap B) \sqcup (A \sqcap C)$
using *inf_sup_distrib1* **by** (*metis inf_topset_def sup_topset_def*)

lemma *topset_inter_union_distrib2*: $(B \sqcup C) \sqcap A = (B \sqcap A) \sqcup (C \sqcap A)$
by (*simp add*: *topset_inter_commute topset_inter_union_distrib*)

lemma *topset_union_inter_distrib*: $A \sqcup (B \sqcap C) = (A \sqcup B) \sqcap (A \sqcup C)$
using *sup_inf_distrib1* **by** (*metis inf_topset_def sup_topset_def*)

lemma *topset_union_inter_distrib2*: $(B \sqcap C) \sqcup A = (B \sqcup A) \sqcap (C \sqcup A)$
by (*simp add*: *topset_union_commute topset_union_inter_distrib*)

lemma *topset_inter_add_distrib*: $(A \oplus c) \sqcap (B \oplus c) = (A \sqcap B) \oplus c$
by (*simp add*: *topset_add_union topset_union_inter_distrib2*)

lemma *topset_inter_rem_distrib*: $(A \ominus c) \sqcap (B \ominus c) = (A \sqcap B) \ominus c$
proof (*cases A*)
case *Top* **then show** *?thesis* **by** *simp*
next
case *HA*: (*Set a*)

```
  thus ?thesis
  proof (cases B)
    case Top
    then have B ⊖ c = ⊤ and A ⊓ B = A by simp_all
    then show ?thesis using topset_inter_def topset_rem_def
    by simp
  next
    case HB: (Set b)
    from HA have A ⊖ c = ⌈a − {c}⌉ using topset_set_rem by simp
    moreover from HB have B ⊖ c = ⌈b − {c}⌉ using topset_set_rem by simp
    ultimately have (A ⊖ c) ⊓ (B ⊖ c) = ⌈a − {c}⌉ ⊓ ⌈b − {c}⌉ by simp
    also have ... = ⌈(a − {c}) ∩ (b − {c})⌉ by (simp add: topset_inter_def)
    also have ... = ⌈(a ∩ b) − {c}⌉ by blast
    moreover from HA HB have A ⊓ B = ⌈a ∩ b⌉
     by (simp add: topset_inter_def)
    then have (A ⊓ B) ⊖ c = ⌈(a ∩ b) − {c}⌉ using topset_set_rem by simp
    ultimately show ?thesis by simp
  qed
qed

lemma topset_inter_rem: B ≠ ⊤ ⟹ A ⊓ (B ⊖ c) = (A ⊓ B) ⊖ c
  apply (cases A)
  apply simp
  apply (cases B)
  apply (simp_all add: Int_Diff topset_inter_def topset_set_rem)
done

lemma topset_add_rem: A ⊕ x ⊖ x = A ⊖ x
 by (simp add: topset_add_def topset_rem_def topset_union_def
    topset.case_eq_if)



lemma topset_gfp_inter_left: gfp (λ x. A ⊓ f x) ⩽ A
proof −
    have gfp (λ x. A ⊓ f x) ⩽ gfp (λ B. A) by (simp add: gfp_mono)
    then show ?thesis by (simp add: gfp_unfold monoI)
qed

lemma topset_lfp_inter_left: lfp (λ x. A ⊓ f x) ⩽ A
proof −
    have lfp (λ x. A ⊓ f x) ⩽ lfp (λ B. A) by (simp add: lfp_mono)
    then show ?thesis by (simp add: lfp_unfold monoI)
qed

end
```

B.17 Loop operator

theory *LoopOperator* **imports** *TopSet* **begin**

definition *loop_function f A* = (λ *X*. *A* ⊓ *f X*)
declare *loop_function_def* [*simp*]

lemma *loop_function_mono1*: *mono* (*loop_function f*)
by (*simp add*: *le_funI monoI*)

lemma *loop_function_mono2*: *mono f* ⟹ *mono* (*loop_function f A*)
by (*simp add*: *mono_def*)

definition *loop_operator f A* = *gfp* (*loop_function f A*)
declare *loop_operator_def* [*simp*]

lemma *loop_operator_mono*: *mono* (*loop_operator f*)
by (*rule monoI*) (*simp add*: *gfp_mono*)

lemma *loop_operator_mono'*:
A ⩽ *B* ⟹ *loop_operator f A* ⩽ *loop_operator f B*
using *loop_operator_mono* **by** (*rule monoD*) *simp*

lemma *loop_operator_unfold*:
mono f ⟹ *loop_operator f A* = *loop_function f A* (*loop_operator f A*)
proof −
let *?F* = *loop_function f A*
assume
mono f
then have *mono ?F* **using** *loop_function_mono2* **by** *simp*
then have *gfp ?F* = *?F* (*gfp ?F*) **by** (*rule gfp_unfold*)
then show *?thesis* **by** *simp*
qed

lemma *loop_operator_idem*:
assumes *mono f*
shows *loop_operator f* (*loop_operator f x*) = *loop_operator f x*
proof −
{ **fix** *tt* :: *'a topset*
obtain
tta :: (*'a topset* ⇒ *'a topset*) ⇒ *'a topset* **and**
ttb :: (*'a topset* ⇒ *'a topset*) ⇒ *'a topset* **where**
ff1: ∀ *f*. *tta f* ⩽ *ttb f* ∧ ¬ *f* (*tta f*) ⩽ *f* (*ttb f*) ∨ *mono f*
by (*metis* (*no_types*) *monoI*)
have *ff2*: ∀ *f*. ¬ *mono f* ∨
Sup {*t*. (*t* :: *'a topset*) ⩽ *f t*} ⩽ *f* (*Sup* {*t*. *t* ⩽ *f t*})

```
  by (metis gfp_def gfp_lemma2)
have ff3: ∀ t f. ¬ (t :: 'a topset) ⩽ f t ∨ t ⩽ Sup {t. t ⩽ f t}
  by (metis (full_types) gfp_def gfp_upperbound)
have ff4: ∀ t ta. (t :: 'a topset) ⊓ ta ⩽ ta
  by (metis order_refl topset_inter_subset_iff)
have ff5: ∀ t. gfp (λt. x ⊓ f t) ⊓ f t ⩽ Sup {t. t ⩽ x ⊓ f t}
  by (simp add: gfp_def)
have ff6: ∀ t. gfp (λt. x ⊓ f t) ⊓ f t ⩽ f t
  using ff4 by metis
have ff7: ∀ t ta. (t :: 'a topset) ⊓ ta = ta ⊓ t
  using ff4 by (simp add: eq_iff topset_inter_subset_iff)
have ff8: ∀ fa. x ⊓ f (tta fa) ⩽ x ⊓ f (ttb fa) ∨ mono fa
  using ff4 ff1 by (metis (no_types) assms dual_order.trans monoD
      topset_inter_lower1 topset_inter_subset_iff)
then have ff9: mono (λt. x ⊓ f t)
  using ff1 by metis
then have ff10: Sup {t. t ⩽ x ⊓ f t} ⩽ x ⊓ f (Sup {t. t ⩽ x ⊓ f t})
  using ff2 by meson
then have ff11: Sup {t. t ⩽ x ⊓ f t} ⩽ x
  by (meson topset_inter_subset_iff)
then have ff12: ∀ t. gfp (λt. x ⊓ f t) ⊓ f t ⩽ x
  using ff5 by (meson dual_order.trans)
then have ff13:
∀ t. gfp (λt. x ⊓ f t) ⊓ f (gfp (λt. x ⊓ f t) ⊓ f t) ⩽ f x
  using ff6 by (meson assms dual_order.trans monoD)
have ∀ t. gfp (λt. x ⊓ f t) ⊓ f t ⩽ f x
  using ff10 ff9 ff5
  by (meson dual_order.trans monoD topset_inter_subset_iff)
then have ff14:
∀ t. gfp (λt. x ⊓ f t) ⊓ f t ⩽ gfp (λt. x ⊓ f t) ⊓ f x
  using ff5 by (simp add: gfp_def topset_inter_subset_iff)
have Sup {t. t ⩽ x ⊓ f t} ⩽ f x
  using ff11 ff10 ff9
  by (meson dual_order.trans monoD topset_inter_subset_iff)
then have Sup {t. t ⩽ x ⊓ f t} ⩽ f x ⊓ Sup {t. t ⩽ x ⊓ f t}
  by (meson order_refl topset_inter_subset_iff)
then have ff15: Sup {t. t ⩽ x ⊓ f t} = gfp (λt. x ⊓ f t) ⊓ f x
  using ff7 by (metis (no_types) eq_iff gfp_def topset_inter_lower1)
then have ff16: ∀ t ta. ¬ gfp (λt. x ⊓ f t) ⊓ f t ⩽ ta ∨
gfp (λt. x ⊓ f t) ⊓ f t ⩽ ta ⊓ (gfp (λt. x ⊓ f t) ⊓ f x)
  by (simp add: gfp_def topset_inter_subset_iff)
have ff17: gfp (λt. x ⊓ f t) ⊓ f x ⩽
gfp (λt. x ⊓ f t) ⊓ f (gfp (λt. x ⊓ f t) ⊓ f x)
  using ff15 ff10 by (simp add: gfp_def topset_inter_subset_iff)
then have ff18: gfp (λt. x ⊓ f t) ⊓ f x ⩽
Sup {t. t ⩽ gfp (λt. x ⊓ f t) ⊓ f t}
```

```
    using ff3 by meson
  have ff19: gfp (λt. x ⊓ f t) ⊓ f x =
   gfp (λt. x ⊓ f t) ⊓ f (gfp (λt. x ⊓ f t) ⊓ f x)
    using ff17 ff13 ff5
     by (simp add: eq_iff gfp_def topset_inter_subset_iff)
  have ff20: ∀ t. gfp (λt. x ⊓ f t) ⊓ f t =
   x ⊓ f t ⊓ (gfp (λt. x ⊓ f t) ⊓ f x)
    using ff16 ff12 by (meson eq_iff topset_inter_subset_iff)
  have ∀ fa. mono fa ∨ gfp (λt. x ⊓ f t) ⊓ f (tta fa) ⩽
   x ⊓ f (ttb fa) ⊓ (gfp (λt. x ⊓ f t) ⊓ f x)
    using ff16 ff12 ff8 ff6
     by (meson dual_order.trans topset_inter_subset_iff)
  then have ff21: mono (λt. gfp (λt. x ⊓ f t) ⊓ f t)
   using ff20 ff1 by (metis (no_types))
  then have ff22: gfp (λt. x ⊓ f t) ⊓ f (gfp (λt. x ⊓ f t) ⊓ f x) ⩽
   gfp (λt. x ⊓ f t) ⊓ f (Sup {t. t ⩽ gfp (λt. x ⊓ f t) ⊓ f t})
    using ff18 by (meson monoD)
  have ff23: Sup {t. t ⩽ gfp (λt. x ⊓ f t) ⊓ f t} ⩽
   gfp (λt. x ⊓ f t) ⊓ f (Sup {t. t ⩽ gfp (λt. x ⊓ f t) ⊓ f t})
    using ff21 ff2 by meson
  have gfp (λt. x ⊓ f t) ⊓ f x =
   gfp (λt. x ⊓ f t) ⊓ f (Sup {t. t ⩽ gfp (λt. x ⊓ f t) ⊓ f t})
    using ff22 ff19 ff14 eq_iff by auto
  then have Sup {t. t ⩽ gfp (λt. x ⊓ f t) ⊓ f t} =
   gfp (λt. x ⊓ f t) ⊓ f x
    using ff23 ff18 by auto
  then have gfp (λt. x ⊓ f t) = gfp (λt. gfp (λt. x ⊓ f t) ⊓ f t) ∨
   x ⊓ f tt = gfp (λt. x ⊓ f t) ⊓ f tt
    using ff15 by (simp add: gfp_def) }
 then have gfp (λt. gfp (λt. x ⊓ f t) ⊓ f t) = gfp (λt. x ⊓ f t)
  by metis
 then show ?thesis by (simp only: loop_operator_def loop_function_def)
qed

lemma loop_operator_leo: mono f ⟹ loop_operator f A ⩽ A
 by (metis eq_iff loop_function_def loop_operator_unfold
   topset_inter_subset_iff)

lemma loop_operator_le1:
 mono f ⟹ loop_operator f A ⩽ loop_operator f (f A)
 by (simp only: loop_operator_def loop_function_def)
  (smt gfp_least gfp_upperbound mono_sup sup.absorb_iff2 sup.boundedE
   topset_inter_subset_iff)

end
```

B.18 Transfer function

theory *TransferFunction* **imports**
ValueAttachment LoopOperator Environment Expression
begin

B.18.1 Transfer function without scopes

fun
$\mathcal{A}'$ :: ('b, 't) *expression* ⇒ *vname topset* ⇒ *vname topset* **and**
$\mathcal{A}s'$ :: ('b, 't) *expression list* ⇒ *vname topset* ⇒ *vname topset* **and**
$\mathcal{AT}'$ :: ('b, 't) *expression* ⇒ *vname topset* ⇒ *bool*
where

— Access

$\mathcal{A}'$*_Value*: $\mathcal{A}'$ (*Value v*) $A = A$ |
$\mathcal{A}'$*_Local*: $\mathcal{A}'$ (*Local n*) $A = A$ |

— Reattachment

$\mathcal{A}'$*_Assign*: $\mathcal{A}'$ (*n* ::= *e*) A = (*if* $\mathcal{AT}'$ *e* A *then* $\mathcal{A}'$ *e* $A \oplus n$ *else* $\mathcal{A}'$ *e* $A \ominus n$) |
$\mathcal{A}'$*_Create*: $\mathcal{A}'$ (*create n*) $A = A \oplus n$ |
$\mathcal{A}'$*_Call*: $\mathcal{A}'$ (*e* . *f* (*a*)) A = $\mathcal{A}s'$ *a* ($\mathcal{A}'$ *e* A) |

— Compound instructions

$\mathcal{A}'$*_Seq*: $\mathcal{A}'$ (c_1;; c_2) $A = \mathcal{A}'$ c_2 ($\mathcal{A}'$ c_1 A) |
$\mathcal{A}'$*_If*: $\mathcal{A}'$ (*if b then* c_1 *else* c_2 *end*) $A = \mathcal{A}'$ c_1 ($\mathcal{A}'$ *b* A) ⊓ $\mathcal{A}'$ c_2 ($\mathcal{A}'$ *b* A) |
$\mathcal{A}'$*_Loop*:
$\mathcal{A}'$ (*until e loop c end*) $A = \mathcal{A}'$ *e* (*loop_operator* (λ *B*. $\mathcal{A}'$ *c* ($\mathcal{A}'$ *e* *B*)) A) |
$\mathcal{A}'$*_Exception*: $\mathcal{A}'$ *Exception* $A = \top$ |

— Boolean expressions

$\mathcal{A}'$*_Test*: $\mathcal{A}'$ (*attached t e as n*) $A = \mathcal{A}'$ *e* A |

— List of expressions

$\mathcal{A}s'$*_Nil*: $\mathcal{A}s'$ [] $A = A$ |
$\mathcal{A}s'$*_Cons*: $\mathcal{A}s'$ (*e* # *es*) A = $\mathcal{A}s'$ *es* ($\mathcal{A}'$ *e* A) |

— Is value attached?

$\mathcal{AT}'$*_Value*: $\mathcal{AT}'$ (*Value v*) $A \longleftrightarrow v \neq Void_{\mathcal{V}}$ |

$\mathcal{AT}'$_Local: $\mathcal{AT}'$ (*Local n*) $A \longleftrightarrow n \in^{\top} A$ |
$\mathcal{AT}'$_If: $\mathcal{AT}'$ (*if b then* c_1 *else* c_2 *end*) $A \longleftrightarrow$
$\mathcal{AT}'\, c_1\ (\mathcal{A}'\, b\ A) \wedge \mathcal{AT}'\, c_2\ (\mathcal{A}'\, b\ A)$ |

— Fallback

$\mathcal{AT}'$_Other: $\mathcal{AT}'$ _ $A \longleftrightarrow$ *True*

abbreviation (**output**) $\mathcal{A}'$_rep (**infixl** $\triangleright$ 72)
where $\mathcal{A}'$_rep $A\ c \equiv \mathcal{A}'\, c\ A$
abbreviation (**output**) $\mathcal{A}s'$_rep (**infixl** $\triangleright\triangleright$ 72)
where $\mathcal{A}s'$_rep $A\ es \equiv \mathcal{A}s'\, es\ A$
abbreviation (**output**) $\mathcal{AT}'$_rep (**infixl** $\hookrightarrow$ 71)
where $\mathcal{AT}'$_rep $A\ c \equiv \mathcal{AT}'\, c\ A$
abbreviation (**output**) *loop_computation_rep'* (_ $\triangleright*$ '(_ $\triangleright$ _') [73, 73, 73])
where *loop_computation_rep'* $A\ e\ c \equiv$ *loop_operator* ($\lambda X.\ \mathcal{A}'\, c\ (\mathcal{A}'\, e\ X)$) A

B.18.2 *Transfer function with scopes*

fun
$\mathcal{A}$ :: ('b, 't) *expression* $\Rightarrow$ *vname topset* $\Rightarrow$ *vname topset* **and**
$\mathcal{A}t$ :: ('b, 't) *expression* $\Rightarrow$ *vname topset* $\Rightarrow$ *vname topset* **and**
$\mathcal{A}f$ :: ('b, 't) *expression* $\Rightarrow$ *vname topset* $\Rightarrow$ *vname topset* **and**
$\mathcal{A}s$ :: ('b, 't) *expression list* $\Rightarrow$ *vname topset* $\Rightarrow$ *vname topset* **and**
$\mathcal{AT}$ :: ('b, 't) *expression* $\Rightarrow$ *vname topset* $\Rightarrow$ *bool*
where

— Access

$\mathcal{A}$_Value: $\mathcal{A}$ (*Value v*) $A = A$ |
$\mathcal{A}$_Local: $\mathcal{A}$ (*Local n*) $A = A$ |

— Reattachment

$\mathcal{A}$_Assign: $\mathcal{A}$ (n ::= e) $A = (\mathit{if}\ \mathcal{AT}\ e\ A\ \mathit{then}\ \mathcal{A}\ e\ A \oplus n\ \mathit{else}\ \mathcal{A}\ e\ A \ominus n)$ |
$\mathcal{A}$_Create: $\mathcal{A}$ (*create n*) $A = A \oplus n$ |
$\mathcal{A}$_Call: $\mathcal{A}$ (e . f (a)) $A = \mathcal{A}s\ a\ (\mathcal{A}\ e\ A)$ |

— Compound instructions

$\mathcal{A}$_Seq: $\mathcal{A}$ (e_1;; e_2) $A = \mathcal{A}\ e_2\ (\mathcal{A}\ e_1\ A)$ |
$\mathcal{A}$_If: $\mathcal{A}$ (*if c then* e_1 *else* e_2 *end*) $A = \mathcal{A}\ e_1\ (\mathcal{A}t\ c\ A) \sqcap \mathcal{A}\ e_2\ (\mathcal{A}f\ c\ A)$ |
$\mathcal{A}t$_If: $\mathcal{A}t$ (*if c then* e_1 *else* e_2 *end*) $A =$
(*if is_false* e_1 *then* $\mathcal{A}t\ e_2\ (\mathcal{A}f\ c\ A)$ *else*
(*if is_false* e_2 *then* $\mathcal{A}t\ e_1\ (\mathcal{A}t\ c\ A)$ *else*

$\mathcal{A}$ (*if c then* e_1 *else* e_2 *end*) A)) |
$\mathcal{A}f\_If$: $\mathcal{A}f$ (*if c then* e_1 *else* e_2 *end*) A =
(*if is_true* e_1 *then* $\mathcal{A}f$ e_2 ($\mathcal{A}f$ *c* A) *else*
(*if is_true* e_2 *then* $\mathcal{A}f$ e_1 ($\mathcal{A}t$ *c* A) *else*
$\mathcal{A}$ (*if c then* e_1 *else* e_2 *end*) A)) |
$\mathcal{A}\_Loop$:
$\mathcal{A}$ (*until e loop b end*) A = $\mathcal{A}t$ *e* (*loop_operator* (λ *B*. $\mathcal{A}$ *b* ($\mathcal{A}f$ *e B*)) A) |
$\mathcal{A}\_Exception$: $\mathcal{A}$ *Exception* A = ⊤ |

— Boolean expressions

$\mathcal{A}\_Test$: $\mathcal{A}$ (*attached t e as n*) A = $\mathcal{A}$ *e* A |
$\mathcal{A}t\_Test2$: $\mathcal{A}t$ (*attached T* (*Local n'*) *as n*) A = A ⊕ *n'* ⊕ *n* |
$\mathcal{A}t\_Test1$: $\mathcal{A}t$ (*attached T e as n*) A = $\mathcal{A}$ *e* A ⊕ *n* |

— List of expressions

$\mathcal{A}s\_Nil$: $\mathcal{A}s$ [] A = A |
$\mathcal{A}s\_Cons$: $\mathcal{A}s$ (*e* # *es*) A = $\mathcal{A}s$ *es* ($\mathcal{A}$ *e* A) |

— Is value attached?

$\mathcal{AT}\_Value$: $\mathcal{AT}$ (*Value v*) A ⟷ *v* ≠ *Void*$_v$ |
$\mathcal{AT}\_Local$: $\mathcal{AT}$ (*Local n*) A ⟷ *n* ∈$^\top$ A |
$\mathcal{AT}\_If$: $\mathcal{AT}$ (*if c then* e_1 *else* e_2 *end*) A ⟷
$\mathcal{AT}$ e_1 ($\mathcal{A}t$ *c* A) ∧ $\mathcal{AT}$ e_2 ($\mathcal{A}f$ *c* A) |

— Fallback

$\mathcal{A}t\_Other$: $\mathcal{A}t$ *e* A = (*if is_false e then* ⊤ *else* $\mathcal{A}$ *e* A) |
$\mathcal{A}f\_Other$: $\mathcal{A}f$ *e* A = (*if is_true e then* ⊤ *else* $\mathcal{A}$ *e* A) |
$\mathcal{AT}\_Other$: $\mathcal{AT}$ _ A ⟷ *True*

lemmas $\mathcal{A}\_induct$ = $\mathcal{A}\_\mathcal{A}t\_\mathcal{A}f\_\mathcal{A}s\_\mathcal{AT}$.*induct*[*split_format*(*complete*)]

abbreviation $\mathcal{A}\_rep$ (**infixl** ▷ 72) **where** A ▷ *c* ≡ $\mathcal{A}$ *c* A
abbreviation $\mathcal{A}\_true\_rep$ (**infixl** ▷+ 72) **where** A ▷+ *b* ≡ $\mathcal{A}t$ *b* A
abbreviation $\mathcal{A}\_false\_rep$ (**infixl** ▷− 72) **where** A ▷− *b* ≡ $\mathcal{A}f$ *b* A
abbreviation $\mathcal{A}s\_rep$ (**infixl** ▷▷ 72) **where** $\mathcal{A}s\_rep$ A *es* ≡ $\mathcal{A}s$ *es* A
abbreviation $\mathcal{AT}\_rep$ (**infixl** ↪ 71) **where** A ↪ *c* ≡ $\mathcal{AT}$ *c* A

definition *loop_step_def* [*simp*]: *loop_step e c A* = A ▷− *e* ▷ *c*
abbreviation (*input*) *loop_computation*
where *loop_computation e c A* ≡ *loop_operator* (λ*X*. *X* ▷− *e* ▷ *c*) A
abbreviation *loop_computation_rep* (_ ▷∗ *'*(− _ ▷ _*'*) [73, 73, 73])

where $A \rhd* (- e \rhd c) \equiv \mathit{loop\_operator}\ (\lambda X.\ X \rhd- e \rhd c)\ A$

lemma *transfer_fold*: $A \rhd\rhd \mathit{es} = \mathit{fold}\ (\lambda\ e\ X.\ X \rhd e)\ \mathit{es}\ A$
by (*induction es arbitrary*: A) *simp_all*

lemma *transfer_unfold*: $A \rhd e \rhd\rhd \mathit{es} = A \rhd\rhd (e \# \mathit{es})$
by *simp*

lemma *unreachable_if_false*: *is_false* $e \Longrightarrow A \rhd+ e = \top$
by (*cases e*) *simp_all*

lemma *unreachable_if_true*: *is_true* $e \Longrightarrow A \rhd- e = \top$
by (*cases e*) *simp_all*

lemma
fixes e :: $('b, 't)$ *expression* **and** es :: $('b, 't)$ *expression list*
shows
$\mathcal{A}\_mono'$: $\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd e \leqslant Y \rhd e$ **and**
$\mathcal{A}t\_mono'$: $\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd+ e \leqslant Y \rhd+ e$ **and**
$\mathcal{A}f\_mono'$: $\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd- e \leqslant Y \rhd- e$ **and**
$\mathcal{A}s\_mono'$: $\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd\rhd \mathit{es} \leqslant Y \rhd\rhd \mathit{es}$ **and**
$\mathcal{A}\mathcal{T}\_mono'$: $\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \hookrightarrow e \longrightarrow Y \hookrightarrow e$
proof (*induction rule*: $\mathcal{A}\_\mathcal{A}t\_\mathcal{A}f\_\mathcal{A}s\_\mathcal{A}\mathcal{T}.induct$)
fix v :: $'b$ *value* **and** $A\ X\ Y$ :: *vname topset*
assume
H: $X \leqslant Y$
from H **show** $X \rhd (\mathit{Value}\ v) \leqslant Y \rhd \mathit{Value}\ v$ **by** *simp*
from H **show** $X \rhd+ (\mathit{Value}\ v) \leqslant Y \rhd+ \mathit{Value}\ v$ **by** *simp*
from H **show** $X \rhd- (\mathit{Value}\ v) \leqslant Y \rhd- \mathit{Value}\ v$ **by** *simp*
from H **show** $X \hookrightarrow (\mathit{Value}\ v) \longrightarrow Y \hookrightarrow \mathit{Value}\ v$ **by** *auto*
fix n
from H **show** $X \rhd (\mathit{Local}\ n) \leqslant Y \rhd \mathit{Local}\ n$ **by** *simp*
from H **show** $X \rhd+ (\mathit{Local}\ n) \leqslant Y \rhd+ \mathit{Local}\ n$ **by** *simp*
from H **show** $X \rhd- (\mathit{Local}\ n) \leqslant Y \rhd- \mathit{Local}\ n$ **by** *simp*
from H **show** $X \hookrightarrow (\mathit{Local}\ n) \longrightarrow Y \hookrightarrow \mathit{Local}\ n$
by (*auto simp add*: *topset_subsetD*)
from H **show** $X \rhd \mathit{create}\ n \leqslant Y \rhd \mathit{create}\ n$ **by** (*simp add*: *topset_add_mono*)
then show $X \rhd+ \mathit{create}\ n \leqslant Y \rhd+ \mathit{create}\ n$ **by** *simp*
then show $X \rhd- \mathit{create}\ n \leqslant Y \rhd- \mathit{create}\ n$ **by** *simp*
then show $X \hookrightarrow \mathit{create}\ n \longrightarrow Y \hookrightarrow \mathit{create}\ n$ **by** *simp*
then show $X \rhd \mathit{Exception} \leqslant Y \rhd \mathit{Exception}$ **using** *topset_member_top*
by *simp*
then show $X \rhd+ \mathit{Exception} \leqslant Y \rhd+ \mathit{Exception}$ **using** *topset_member_top*
by *simp*
then show $X \rhd- \mathit{Exception} \leqslant Y \rhd- \mathit{Exception}$ **using** *topset_member_top*
by *simp*

```
  then show X ↪ Exception ⟶ Y ↪ Exception by simp
  fix c1 c2 :: ('b, 't) expression
  from H show X ↪ c1 ;; c2 ⟶ Y ↪ c1 ;; c2 by simp
  assume
    ⋀X Y. X ⩽ Y ⟹ X ▷ c1 ⩽ Y ▷ c1
    ⋀X Y. X ⩽ Y ⟹ X ▷ c2 ⩽ Y ▷ c2
  with H show X ▷ c1 ;; c2 ⩽ Y ▷ c1 ;; c2 by simp
next
  fix c1 c2 :: ('b, 't) expression and A X Y :: vname topset
  assume
    ⋀X Y. ⟦¬ is_false (c1 ;; c2); X ⩽ Y⟧ ⟹ X ▷ c1 ;; c2 ⩽ Y ▷ c1 ;; c2
    X ⩽ Y
  then show X ▷+ c1 ;; c2 ⩽ Y ▷+ c1 ;; c2 by simp
next
  fix c1 c2 :: ('b, 't) expression and A X Y :: vname topset
  assume
    ⋀X Y. ⟦¬ is_true (c1 ;; c2); X ⩽ Y⟧ ⟹ X ▷ c1 ;; c2 ⩽ Y ▷ c1 ;; c2
    X ⩽ Y
  then show X ▷− c1 ;; c2 ⩽ Y ▷− c1 ;; c2 by simp
next
  fix n and e :: ('b, 't) expression and A X Y :: vname topset
  assume
    H: X ⩽ Y
  then show X ↪ n ::= e ⟶ Y ↪ n ::= e by simp
  assume
    ⋀X Y. X ⩽ Y ⟹ X ↪ e ⟶ Y ↪ e
    ⋀X Y. ⟦A ↪ e; X ⩽ Y⟧ ⟹ X ▷ e ⩽ Y ▷ e
    ⋀X Y. ⟦¬ A ↪ e; X ⩽ Y⟧ ⟹ X ▷ e ⩽ Y ▷ e
  with H show X ▷ n ::= e ⩽ Y ▷ n ::= e
   using topset_add_rem_mono1 topset_add_mono topset_rem_mono
    by (metis (no_types) A_Assign)

next
  fix
   e :: ('b, 't) expression and f and
   a :: ('b, 't) expression list and A X Y :: vname topset
  assume
    H: X ⩽ Y
  then show X ↪ e . f (a) ⟶ Y ↪ e . f (a) by simp
  assume
    ⋀X Y. X ⩽ Y ⟹ X ▷ e ⩽ Y ▷ e
    ⋀X Y. X ⩽ Y ⟹ X ▷▷ a ⩽ Y ▷▷ a
  with H show X ▷ e . f (a) ⩽ Y ▷ e . f (a) by simp
next
  fix e :: ('b, 't) expression and n a and A X Y :: vname topset
  assume
```

$\bigwedge X\ Y.\ [\![\neg\ is\_false\ (e\ .\ n\ (a));\ X \leqslant Y]\!] \Longrightarrow X \rhd e\ .\ n\ (a) \leqslant Y \rhd e\ .\ n\ (a)$
$X \leqslant Y$
then show $X \rhd+ e\ .\ n\ (a) \leqslant Y \rhd+ e\ .\ n\ (a)$ **by** *simp*
next
fix $e :: ('b,\ 't)$ *expression* **and** $n\ a$ **and** $A\ X\ Y ::$ *vname topset*
assume
$\bigwedge X\ Y.\ [\![\neg\ is\_true\ (e\ .\ n\ (a));\ X \leqslant Y]\!] \Longrightarrow X \rhd e\ .\ n\ (a) \leqslant Y \rhd e\ .\ n\ (a)$
$X \leqslant Y$
then show $X \rhd- e\ .\ n\ (a) \leqslant Y \rhd- e\ .\ n\ (a)$ **by** *simp*
next
fix $b\ c_1\ c_2 :: ('b,\ 't)$ *expression* **and** $A\ X\ Y ::$ *vname topset*
assume
$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd+ b \leqslant Y \rhd+ b$
$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd c_1 \leqslant Y \rhd c_1$
$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd- b \leqslant Y \rhd- b$
$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd c_2 \leqslant Y \rhd c_2$
$X \leqslant Y$
then show $X \rhd$ *if b then* c_1 *else* c_2 *end* $\leqslant Y \rhd$ *if b then* c_1 *else* c_2 *end*
by *simp*
next
fix $b\ e_1\ e_2 :: ('b,\ 't)$ *expression* **and** $A\ X\ Y ::$ *vname topset*
assume
$\bigwedge X\ Y.\ [\![is\_false\ e_1;\ X \leqslant Y]\!] \Longrightarrow X \rhd- b \leqslant Y \rhd- b$
$\bigwedge X\ Y.\ [\![is\_false\ e_1;\ X \leqslant Y]\!] \Longrightarrow X \rhd+ e_2 \leqslant Y \rhd+ e_2$
$\bigwedge X\ Y.\ [\![\neg\ is\_false\ e_1;\ is\_false\ e_2;\ X \leqslant Y]\!] \Longrightarrow X \rhd+ b \leqslant Y \rhd+ b$
$\bigwedge X\ Y.\ [\![\neg\ is\_false\ e_1;\ is\_false\ e_2;\ X \leqslant Y]\!] \Longrightarrow X \rhd+ e_1 \leqslant Y \rhd+ e_1$
$\bigwedge X\ Y.\ [\![\neg\ is\_false\ e_1;\ \neg\ is\_false\ e_2;\ X \leqslant Y]\!] \Longrightarrow$
$X \rhd$ *if b then* e_1 *else* e_2 *end* $\leqslant Y \rhd$ *if b then* e_1 *else* e_2 *end*
$X \leqslant Y$
then show $X \rhd+$ *if b then* e_1 *else* e_2 *end* $\leqslant Y \rhd+$ *if b then* e_1 *else* e_2 *end*
by *simp*
next
fix $b\ e_1\ e_2 :: ('b,\ 't)$ *expression* **and** $A\ X\ Y ::$ *vname topset*
assume
$\bigwedge X\ Y.\ [\![is\_true\ e_1;\ X \leqslant Y]\!] \Longrightarrow X \rhd- b \leqslant Y \rhd- b$
$\bigwedge X\ Y.\ [\![is\_true\ e_1;\ X \leqslant Y]\!] \Longrightarrow X \rhd- e_2 \leqslant Y \rhd- e_2$
$\bigwedge X\ Y.\ [\![\neg\ is\_true\ e_1;\ is\_true\ e_2;\ X \leqslant Y]\!] \Longrightarrow X \rhd+ b \leqslant Y \rhd+ b$
$\bigwedge X\ Y.\ [\![\neg\ is\_true\ e_1;\ is\_true\ e_2;\ X \leqslant Y]\!] \Longrightarrow X \rhd- e_1 \leqslant Y \rhd- e_1$
$\bigwedge X\ Y.\ [\![\neg\ is\_true\ e_1;\ \neg\ is\_true\ e_2;\ X \leqslant Y]\!] \Longrightarrow$
$X \rhd$ *if b then* e_1 *else* e_2 *end* $\leqslant Y \rhd$ *if b then* e_1 *else* e_2 *end*
$X \leqslant Y$
then show $X \rhd-$ *if b then* e_1 *else* e_2 *end* $\leqslant Y \rhd-$ *if b then* e_1 *else* e_2 *end*
by *simp*
next
fix $b\ c_1\ c_2 :: ('b,\ 't)$ *expression* **and** $A\ X\ Y ::$ *vname topset*
assume

$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd+ b \leqslant Y \rhd+ b$
$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \hookrightarrow c_1 \longrightarrow Y \hookrightarrow c_1$
$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \rhd- b \leqslant Y \rhd- b$
$\bigwedge X\ Y.\ X \leqslant Y \Longrightarrow X \hookrightarrow c_2 \longrightarrow Y \hookrightarrow c_2$
$X \leqslant Y$
then show $X \hookrightarrow$ *if b then* c_1 *else* c_2 *end* $\longrightarrow Y \hookrightarrow$ *if b then* c_1 *else* c_2 *end*
by (*meson* $\mathcal{AT}$*_If*)
next
fix *e b* :: ('*b*, '*t*) *expression* **and** *A X Y* :: *vname topset*
assume
H: $X \leqslant Y$
then show $X \hookrightarrow$ *until e loop b end* $\longrightarrow Y \hookrightarrow$ *until e loop b end* **by** *simp*
assume
lHi: $\bigwedge x\ X\ Y.\ [\![X \leqslant Y]\!] \Longrightarrow X \rhd- e \leqslant Y \rhd- e$ **and**
lHb: $\bigwedge x\ X\ Y.\ [\![X \leqslant Y]\!] \Longrightarrow X \rhd b \leqslant Y \rhd b$ **and**
lHe: $\bigwedge X\ Y.\ [\![X \leqslant Y]\!] \Longrightarrow X \rhd+ e \leqslant Y \rhd+ e$
then have *mono* $(\lambda x.\ x \rhd- e \rhd b)$ **if** $\neg$ *is_false e* **using** *that*
by (*simp add*: *monoI*)
with *H* **have** $X \rhd* (- e \rhd b) \leqslant Y \rhd* (- e \rhd b)$ **using**
loop_operator_mono'
by *simp*
then have $X \rhd* (- e \rhd b) \rhd+ e \leqslant Y \rhd* (- e \rhd b) \rhd+ e$ **using** *lHe*
by *simp*
then show $X \rhd$ *until e loop b end* $\leqslant Y \rhd$ *until e loop b end* **by** *simp*
next
fix *e b* :: ('*b*, '*t*) *expression* **and** *A X Y* :: *vname topset*
assume
$\bigwedge X\ Y.\ [\![\neg$ *is_false* (*until e loop b end*); $X \leqslant Y]\!] \Longrightarrow$
$X \rhd$ *until e loop b end* $\leqslant Y \rhd$ *until e loop b end*
$X \leqslant Y$
then show $X \rhd+$ *until e loop b end* $\leqslant Y \rhd+$ *until e loop b end* **by** *simp*
next
fix *e b* :: ('*b*, '*t*) *expression* **and** *A X Y* :: *vname topset*
assume
$\bigwedge X\ Y.\ [\![\neg$ *is_true* (*until e loop b end*); $X \leqslant Y]\!] \Longrightarrow$
$X \rhd$ *until e loop b end* $\leqslant Y \rhd$ *until e loop b end*
$X \leqslant Y$
then show $X \rhd-$ *until e loop b end* $\leqslant Y \rhd-$ *until e loop b end* **by** *simp*
next
fix *t* **and** *e* :: ('*b*, '*t*) *expression* **and** *n n'* **and** *A X Y* :: *vname topset*
show $X \hookrightarrow$ *attached t e as n* $\longrightarrow Y \hookrightarrow$ *attached t e as n* **by** *simp*
assume
$X \leqslant Y$
then show $X \rhd+$ *attached t* (*Local n'*) *as n* $\leqslant$
$Y \rhd+$ *attached t* (*Local n'*) *as n*
by (*simp add*: *topset_add_mono*)

```
  assume
    ⋀X Y. X ⩽ Y ⟹ X ▷ e ⩽ Y ▷ e
    X ⩽ Y
  then show X ▷ attached t e as n ⩽ Y ▷ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n
   by (cases e) (simp_all add: topset_add_mono)
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
  then show X ▷+ attached t e as n ⩽ Y ▷+ attached t e as n by simp
next
  fix A X Y :: vname topset
  assume
    H: X ⩽ Y
  then show X ▷▷ [] ⩽ Y ▷▷ [] by simp
  fix e :: ('b, 't) expression and es :: ('b, 't) expression list
  assume
    ⋀X Y. X ⩽ Y ⟹ X ▷ e ⩽ Y ▷ e
    ⋀X Y. X ⩽ Y ⟹ X ▷▷ es ⩽ Y ▷▷ es
  with H show X ▷▷ (e # es) ⩽ Y ▷▷ (e # es) by simp
qed simp_all

lemma 𝒜_mono: mono (λ X. X ▷ e) by (simp add: monoI 𝒜_mono')
lemma 𝒜t_mono: mono (λ X. X ▷+ e) by (simp add: monoI 𝒜t_mono')
lemma 𝒜f_mono: mono (λ X. X ▷− e) by (simp add: monoI 𝒜f_mono')
lemma 𝒜s_mono: mono (λ X. X ▷▷ e) by (simp add: monoI 𝒜s_mono')
lemma 𝒜𝒯_mono: mono (λ X. X ↪ e) by (simp add: monoI 𝒜𝒯_mono')

lemma loop_computation_leo:
  fixes e b :: ('b, 't) expression
  shows loop_computation e b A ⩽ A
proof −
  have ⋀X. A ⊓ (X ▷− e ▷ b) ⩽ A by simp
  then show ?thesis
    by (metis gfp_least loop_function_def loop_operator_def
      topset_inter_subset_iff)
qed

lemma loop_step_mono: mono (loop_step e b)
  by (simp add: 𝒜_mono' 𝒜f_mono' monoI)

lemma loop_computation_mono: mono (loop_computation e b)
```

```
  by (metis loop_operator_mono' monoI)

lemma loop_computation_le1:
  fixes e b :: ('b, 't) expression
  shows loop_computation e b A ⩽ A ▷− e ▷ b
proof −
  let ?f = (λX. X ▷− e ▷ b)
  have
    lHm: mono ?f by (simp add: A_mono' Af_mono' monoI)
  then have loop_operator ?f A = loop_function ?f A (loop_operator ?f A)
    by (rule loop_operator_unfold)
  also from lHm have ... ⩽ loop_function ?f A A
    using loop_function_mono2 loop_operator_leo by (metis monoD)
  finally show ?thesis by (simp add: topset_inter_subset_iff)
qed

lemma loop_application1:
 A ▷ until e loop b end ⩽ A ▷− e ▷ b ▷ until e loop b end
proof −
  let ?f = (λX. X ▷− e ▷ b)
  have mono ?f by (simp add: A_mono' Af_mono' monoI)
  then have loop_operator ?f A ⩽ loop_operator ?f (?f A)
   by (rule loop_operator_le1)
  then show ?thesis by (simp add: A_mono' At_mono')
qed

lemma
  A_to_all [simp]: A e ⊤ = ⊤ and
  At_to_all [simp]: At e ⊤ = ⊤ and
  Af_to_all [simp]: Af e ⊤ = ⊤
proof (induction e)
case (Loop b c)
  then show A (until b loop c end) ⊤ = ⊤
   by (simp_all add: topset_Union_def gfp_def)
  then show At (until b loop c end) ⊤ = ⊤
   by (simp_all add: topset_Union_def gfp_def)
  then show Af (until b loop c end) ⊤ = ⊤
   by (simp_all add: topset_Union_def gfp_def)
case (Test t e n)
  then show
    lHt: A (attached t e as n) ⊤ = ⊤ by (cases e) simp_all
  then show At (attached t e as n) ⊤ = ⊤ by (cases e) simp_all
  from lHt show Af (attached t e as n) ⊤ = ⊤ by simp
case (Call e f a)
  then show ⊤ ▷ e . f (a) = ⊤ by (induction a, simp_all)
  then show ⊤ ▷+ e . f (a) = ⊤ by simp
```

```
  then show ⊤ ▷− e . f (a) = ⊤ by simp
qed simp_all

lemma reachability:
  assumes
    HR: B ▷ e ≠ ⊤ and
    HS: A ⩽ B
  shows
    A ▷ e ≠ ⊤
proof
  assume
    lH: A ▷ e = ⊤
  from HS have A ▷ e ⩽ B ▷ e by (rule A_mono')
  with lH have ⊤ ⩽ B ▷ e by simp
  then have B ▷ e = ⊤ using topset_top_le by simp
  with HR show False by simp
qed

lemma
  scope_true_ge: A ▷ e ⩽ A ▷+ e and
  scope_false_ge: A ▷ e ⩽ A ▷− e
proof (induction e arbitrary: A)
  fix A
  case (Test t e n)
  then show A ▷ (attached t e as n) ⩽ A ▷+ (attached t e as n)
    by (cases e) (simp_all add: topset_add_insert topset_subset_insertI2)
  then show A ▷ (attached t e as n) ⩽ A ▷− (attached t e as n) by simp
next
  case (If b c1 c2)
  fix A
  show A ▷ (if b then c1 else c2 end) ⩽ A ▷+ (if b then c1 else c2 end)
   using If.IH by (metis A_If At_If eq_iff order.trans
        topset_inter_lower1 topset_inter_lower2)
  show A ▷ (if b then c1 else c2 end) ⩽ A ▷− (if b then c1 else c2 end)
   using If.IH by (metis A_If Af_If eq_iff order.trans
        topset_inter_lower1 topset_inter_lower2)
qed simp_all

lemma
  reachability_with_scope_true: A ▷+ e ≠ ⊤ ⟶ A ▷ e ≠ ⊤ and
  reachability_with_scope_false: A ▷− e ≠ ⊤ ⟶ A ▷ e ≠ ⊤
  by (metis scope_true_ge topset_top_le) (metis scope_false_ge topset_top_le)

lemma attachment_loop_condition: A ▷ until e loop b end ⩽ A ▷+ e
proof −
  let ?F = λ B. B ▷− e ▷ b
```

```
 have mono ?F
  by (simp add: A_mono' Af_mono' monoD monoI topset_inter_mono_arg2)
 then have
  loop_computation e b A ⩽ loop_function ?F A (loop_computation e b A)
   using loop_operator_unfold by (metis order_refl)
 then have loop_computation e b A ⩽ A
  by (simp add: topset_inter_subset_iff)
 then have ?thesis by (simp add: A_mono' At_mono')
 moreover have A ▷+ e = ⊤ if is_false e
  using unreachable_if_false that by blast
 then have A ▷ until e loop b end = A ▷+ e if is_false e
  using unreachable_if_false that by simp
 ultimately show ?thesis by simp
qed

end
```

B.19 Expression void safety

```
theory ExpressionValidity imports
 EnvironmentAttachment
 ExpressionAttachment
 TransferFunction
begin
```

B.19.1 *Attachment validity rules and type checks*

A predicate that tells if an expression is valid with respect to attachment rules and what is the expected type of the expression.

```
inductive
 AT :: vname topset ⇒ 'b attachable_expression ⇒ attachment_type ⇒ bool
  (_ ⊢ _ : _ [60, 0, 60] 60) and
 ATs :: vname topset ⇒ 'b attachable_expression list ⇒
  attachment_type list ⇒ bool
  (_ ⊢ _ [:] _ [60, 0, 60] 60)
where
 AT_ValueDet: v = Void_v ⟹ A ⊢ Value v: Detachable |
 AT_ValueAtt: v ≠ Void_v ⟹ A ⊢ Value v: Attached |
 AT_LocalAtt: n ∈⊤ A ⟹ A ⊢ Local n: Attached |
 AT_LocalDet: ¬ n ∈⊤ A ⟹ A ⊢ Local n: Detachable |
 AT_Seq: ⟦A ⊢ e1: Attached; A e1 A ⊢ e2: Attached⟧ ⟹
  A ⊢ (e1;; e2): Attached |
 AT_Assign: A ⊢ e: T ⟹ A ⊢ (n ::= e): Attached |
 AT_Create: A ⊢ (create n): Attached |
```

AT_Call: ⟦*A* ⊢ *e*: *Attached*; *A e A* ⊢ *a* [:] *Ts*⟧ ⟹
A ⊢ (*e* . *f* (*a*)): *Attached* |
AT_If: ⟦
A ⊢ *b*: *Attached*;
A ▷+ *b* ⊢ e_1: T_1;
A ▷− *b* ⊢ e_2: T_2
⟧ ⟹ *A* ⊢ *if b then* e_1 *else* e_2 *end*: *upper_bound* T_1 T_2 |
AT_Loop: ⟦
loop_computation e b A ⊢ *e*: *Attached*;
loop_computation e b A ▷− *e* ⊢ *b*: *Attached*
⟧ ⟹ *A* ⊢ *until e loop b end*: *Attached* |
AT_Test: *A* ⊢ *e*: *T* ⟹ *A* ⊢ (*attached t e as n*): *Attached* |
AT_Exception: *A* ⊢ *Exception*: *Attached* |
ATs_Nil: *A* ⊢ [] [:] [] |
ATs_Cons: ⟦*A* ⊢ *e*: *T*; *A e A* ⊢ *es* [:] *Ts*⟧ ⟹ *A* ⊢ *e* # *es* [:] *T* # *Ts*

declare *AT_ATs.intros*[*intro*!]

inductive_simps *ATs_iffs* [*iff*]:
A ⊢ [] [:] *Ts*
A ⊢ *e*#*es* [:] *T*#*Ts*
A ⊢ *e*#*es* [:] *Ts*

lemmas *at_induct* = *AT_ATs.induct*[*split_format*(*complete*)]
inductive_cases *ValueE*[*elim*!]: *A* ⊢ *Value v*: *T*
inductive_cases *LocalE*[*elim*!]: *A* ⊢ *Local n*: *T*
inductive_cases *SeqE*[*elim*!]: *A* ⊢ *c1* ;; *c2*: *T*
inductive_cases *AssignE*[*elim*!]: *A* ⊢ *n* ::= *e*: *T*
inductive_cases *CreateE*[*elim*!]: *A* ⊢ *create n*: *T*
inductive_cases *CallE*[*elim*!]: *A* ⊢ *n* . *f* (*a*): *T*
inductive_cases *IfE*[*elim*!]: *A* ⊢ *if b then c1 else c2 end*: *T*
inductive_cases *LoopE*[*elim*!]: *A* ⊢ *until e loop c end*: *T*
inductive_cases *TestE*[*elim*!]: *A* ⊢ *attached t e as x*: *T*
inductive_cases *ExceptionE*[*elim*!]: *A* ⊢ *Exception*: *T*

B.19.2 *Type checks properties*

If an expression is void-safe with two types, these types are the same.

lemma *attachment_unique*:
fixes
e :: *'b attachable_expression* **and** *es* :: *'b attachable_expression list*
shows
attachment_unique_e: ⟦*A* ⊢ *e* : *T*; *A* ⊢ *e* : *T'*⟧ ⟹ *T* = *T'* **and**
attachment_unique_es: ⟦*A* ⊢ *es* [:] *Ts*; *A* ⊢ *es* [:] *Ts'*⟧ ⟹ *Ts* = *Ts'*
by (*induction arbitrary*: *T'* **and** *Ts'* *rule*: *AT_ATs.inducts*)

((*blast?*, *fastforce*)+)

If an expression has an attached type, it is attached in the same context.

lemma *checked_attached_is_attached*:
$A \vdash e$: *Attached* $\Longrightarrow A \hookrightarrow e$
proof (*induction e arbitrary*: A)
case (*If b c1 c2*)
then have
$A \vdash b$: *Attached* **by** *blast*
from *If.prems*
obtain T_1 T_2 **where**
$A \rhd+ b \vdash c1 : T_1$ $A \rhd- b \vdash c2 : T_2$ *upper_bound* T_1 T_2 = *Attached*
by *force*
moreover then have T_1 = *Attached* **and** T_2 = *Attached*
using *upper_bound_bot_left'* *upper_bound_bot_right'* **by** *blast*+
ultimately have
$A \rhd+ b \hookrightarrow c1$ **and**
$A \rhd- b \hookrightarrow c2$ **using** *AT_mono'* *If.IH* *topset_union_upper1* **by** *blast*+
then show *?case* **by** *simp*
qed *auto*

lemma *list_attachment_length*:
$A \vdash es$ [:] $Ts \Longrightarrow$ *length es* = *length Ts*
by (*induction es arbitrary*: A Ts) (*insert ATs.cases*, *fastforce*, *auto*)

lemma *list_attachment_validity*:
$A \vdash es1$ @ $es2$ [:] $Ts \Longrightarrow A \rhd\rhd es1 \vdash es2$ [:] *drop* (*length es1*) Ts
proof (*induction es1 arbitrary*: A Ts)
case *Nil* **then show** *?case* **by** *simp*
next
fix *es* :: *'a attachable_expression list*
case (*Cons e es A Ts*)
then obtain *es'* **where** *es'* = *es* @ *es2* **by** *simp*
moreover with *Cons* **have** $A \vdash [e]$ @ *es'* [:] Ts **by** *simp*
ultimately have $A \rhd\rhd [e] \vdash es$ @ $es2$ [:] *drop 1* Ts **by** *auto*
then show *?case*
by (*metis Cons.IH One_nat_def TransferFunction.As_Nil drop_drop*
list.size(4) *transfer_unfold*)
qed

lemma *list_attachment_validity_head*:
$A \vdash e$ # es [:] $Ts \Longrightarrow A \vdash e$: *hd Ts*
by (*induction Ts*) *auto*

lemma *list_attachment_validity_tail*:
$A \vdash e$ # es [:] $Ts \Longrightarrow A \rhd e \vdash es$ [:] *tl Ts*

by *auto*

The void safety predicate is monotone.

lemma
fixes
e :: *'b attachable_expression* **and**
es :: *'b attachable_expression list*
shows
AT_mono: $A \vdash e : T_A \Longrightarrow (\bigwedge B.\ A \leqslant B \Longrightarrow \exists\ T_B.\ B \vdash e : T_B \wedge T_B \leqslant T_A)$
and
ATs_mono: $A \vdash es\ [:]\ Tsa \Longrightarrow (\bigwedge B.\ A \leqslant B \Longrightarrow \exists\ Tsb.\ B \vdash es\ [:]\ Tsb)$
proof (*induction arbitrary*: *rule*:*AT_ATs.inducts*)
case *AT_LocalAtt* **then show** *?case* **using** *attached_conforms_to* **by** *blast*
next
case *AT_LocalDet* **then show** *?case* **using** *conforms_to_detachable* **by** *blast*
next
case (*AT_Seq A c1 c2*)
with *A_mono'* **have** $A \rhd c1 \leqslant B \rhd c1$ **by** *metis*
with *AT_Seq* **show** *?case* **using** *conforms_to_attached* **by** *blast*
next
case *AT_Assign* **then show** *?case* **using** *attached_conforms_to* **by** *blast*
next
case (*AT_Call A e es Ts*)
with *A_mono'* **have** $A \rhd e \leqslant B \rhd e$ **by** *metis*
with *AT_Call* **show** *?case* **using** *conforms_to_attached*
by (*metis AT_ATs.AT_Call*)
next
case (*AT_If A c e1 T1 e2 T2*)
then have
lHc: $B \vdash c : Attached$ **using** *conforms_to_attached* **by** *auto*
from *AT_If* **obtain** *T1' T2'* **where**
lHe1: $A \rhd+ c \vdash e1 : T1'$ **and**
lHT1: $T1' \to_a T1$ **and**
lHe2: $A \rhd- c \vdash e2 : T2'$ **and**
lHT2: $T2' \to_a T2$ **using** *is_attached_conformance* **by** *blast*+
from *AT_If* **have**
$A \rhd+ c \leqslant B \rhd+ c$
$A \rhd- c \leqslant B \rhd- c$
by (*simp_all add*: *At_mono' Af_mono'*)
with *lHe1 lHe2 AT_If* **have**
$\exists Tb.\ B \rhd+ c \vdash e1 : Tb \wedge Tb \to_a T1'$
$\exists Tb.\ B \rhd- c \vdash e2 : Tb \wedge Tb \to_a T2'$
using *attachment_unique_e* **by** *force*+
then obtain *Tb1 Tb2* **where**
lHBe1:$B \rhd+ c \vdash e1 : Tb1$ **and**
lHTb1: $Tb1 \to_a T1'$ **and**

```
 lHBe2: B ▷− c ⊢ e2 : Tb2 and
 lHTb2: Tb2 →a T2' by blast
with lHc have
 lHB: B ⊢ if c then e1 else e2 end : upper_bound Tb1 Tb2 by auto
from lHT1 lHTb1 lHT2 lHTb2 have
 Tb1 →a T1 and
 Tb2 →a T2
  using attachment_conforming_to_transitive by blast+
then have upper_bound Tb1 Tb2 →a upper_bound T1 T2 using sup.mono
 by blast
then show ?case using lHB by meson
next
case (AT_Loop e b A)
then have loop_computation e b B ⊢ e : Attached
 using conforms_to_attached loop_operator_mono' by metis
moreover from AT_Loop have loop_computation e b B ▷− e ⊢ b : Attached
 using Af_mono' conforms_to_attached loop_operator_mono' by metis
ultimately show ?case using attached_conforms_to by blast
next
case AT_Test then show ?case using attached_conforms_to by blast
next
case (ATs_Cons A e T es Ts)
 with A_mono' have A ▷ e ⩽ B ▷ e by metis
 with ATs_Cons show ?case by auto
qed auto
```

If the source expression of an assignment instruction is of an attached type, the set of attached variables after the expression includes the target variable.

```
lemma attached_assignment_set:
 assumes
  HA: A ⊢ (n ::= e): Attached and
  HE: A ⊢ e: Attached
 shows A (n ::= e) A = (A e A) ⊕ n
using HE HA
proof (induction e arbitrary: Γ A)
 case (If b c1 c2)
 moreover then have
  lHA1: A ▷+ b ⊢ c1 : Attached and
  lHA2: A ▷− b ⊢ c2 : Attached
    apply (metis IfE upper_bound_bot_left')
     by (metis IfE calculation(4) upper_bound_bot_right')
 moreover with If have
  A ▷+ b ⊢ n ::= c1 : Attached and
  A ▷− b ⊢ n ::= c2 : Attached
   using AT_Assign AssignE IfE by blast+
```

ultimately have $\mathcal{A}\ (n ::= c1)\ (A \rhd+ b) = (\mathcal{A}\ c1\ (A \rhd+ b)) \oplus n$ **and**
$\mathcal{A}\ (n ::= c2)\ (A \rhd- b) = (\mathcal{A}\ c2\ (A \rhd- b)) \oplus n$ **by** *simp_all*
from *lHA1* **have**
lHA1': $A \rhd+ b \hookrightarrow c1$ **by** (*rule checked_attached_is_attached*)
from *lHA2* **have**
lHA2': $A \rhd- b \hookrightarrow c2$ **by** (*rule checked_attached_is_attached*)
have $A \rhd n ::= (\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{end}) =$
(*if* $(A \hookrightarrow (\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{end}))$ *then*
$(A \rhd (\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{end})) \oplus n$ *else*
$(A \rhd (\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{end})) \ominus n$) **by** *simp*
also have ... $= (\mathit{if}\ ((A \rhd+ b \hookrightarrow c1) \wedge (A \rhd- b \hookrightarrow c2))\ \mathit{then}$
$((A \rhd+ b \rhd c1) \sqcap (A \rhd- b \rhd c2)) \oplus n$ *else*
$((A \rhd+ b \rhd c1) \sqcap (A \rhd- b \rhd c2)) \ominus n$) **by** *simp*
also with *lHA1' lHA2'* **have** ... $= ((A \rhd+ b \rhd c1) \sqcap (A \rhd- b \rhd c2)) \oplus n$
by *simp*
finally show *?case* **by** *simp*
qed *auto*

If a void-safe expression has a detachable type, it is not attached.

lemma *checked_detachable_is_detachable'*:
assumes
HA: $A \vdash e : T$ **and**
HT: $T = \mathit{Detachable}$
shows
$\neg\ A \hookrightarrow e$
using *assms*
proof (*induction e arbitrary*: $A\ T$)
case (*Value v*)
then show *?case* **by** *auto*
next
case (*If c* e_1 e_2)
moreover then obtain $T_1\ T_2$ **where**
$A \vdash c : \mathit{Attached}$
$A \rhd+ c \vdash e_1 : T_1$
$A \rhd- c \vdash e_2 : T_2$
$T = \mathit{upper\_bound}\ T_1\ T_2$
using *IfE* **by** *auto*
moreover with *If.prems* **have** $T_1 = \mathit{Detachable} \vee T_2 = \mathit{Detachable}$
by (*metis sup_attachment_type_def*)
ultimately have $\neg\ A \rhd+ c \hookrightarrow e_1 \vee \neg\ A \rhd- c \hookrightarrow e_2$ **by** *force*
then show *?case* **using** $\mathcal{AT}$*_If* **by** *simp*
qed *auto*

lemma *checked_detachable_is_detachable*: $A \vdash e : \mathit{Detachable} \Longrightarrow \neg\ A \hookrightarrow e$
by (*simp add*: *checked_detachable_is_detachable'*)

If the source expression of an assignment instruction is of a detachable type, the set of attached variables after the expression does not include the target variable.

lemma *detachable_assignment_set*:
 assumes
  HA: $A \vdash (n ::= e)$: *Attached* **and**
  HE: $A \vdash e$: *Detachable*
 shows $\mathcal{A}\ (n ::= e)\ A = (\mathcal{A}\ e\ A) \ominus n$
using *HE HA*
proof (*induction e arbitrary*: $\Gamma\ A$)
 case (*Value v*)
 then show *?case* **by** *auto*
next
 case (*If b c1 c2*)
 then show *?case* **using** *checked_detachable_is_detachable*
  using $\mathcal{A}$*_Assign* **by** *metis*
next
 case (*Loop e c*)
 then show *?case* **by** *auto*
qed *auto*

If a loop expression is void-safe, it is void-safe in the context of its complete evaluation.

lemma *AT_loop*: $A \vdash$ *until e loop c end* : $T \Longrightarrow$
 loop_computation e c A $\vdash$ *until e loop c end* : T
proof −
 assume
  lH: $A \vdash$ *until e loop c end* : T
 then have
  lHe: *loop_computation e c A* $\vdash e$: *Attached* **and**
  lHc: *loop_computation e c A* $\rhd\!-\ e \vdash c$: *Attached* **and**
  lHt: T = *Attached* **by** *auto*
 have *mono* ($\lambda X.\ X \rhd\!-\ e \rhd c$) **by** (*simp add*: $\mathcal{A}$*_mono′ Af_mono′ monoI*)
 then have
  loop_computation e c (*loop_computation e c A*) = *loop_computation e c A*
   using *loop_operator_idem* **by** *auto*
 with *lHe lHc* **have**
  loop_computation e c (*loop_computation e c A*) $\vdash e$: *Attached* **and**
  loop_computation e c (*loop_computation e c A*) $\rhd\!-\ e \vdash c$: *Attached*
   by *simp_all*
 with *lHt* **show** *?thesis* **by** (*simp add*: *AT_Loop*)
qed

If a loop expression is void-safe, it is void-safe in the context of its single-iteration evaluation.

lemma *AT_loop_step*: $A \vdash$ *until e loop c end* : $T \Longrightarrow$
$A \rhd\!\!- e \rhd c \vdash$ *until e loop c end* : T
by (*metis AT_loop AT_mono LoopE loop_computation_le1*)

A predicate that tells if an expression is void-safe with a given attachment type.

definition *attachment_type_valid_expression* ($\vdash$ _ : _ [60, 60] 60) **where**
attachment_type_valid_expression e T = *ExpressionValidity.AT* $\varnothing$ *e T*

A predicate that tells if an expression is void-safe.

definition *attachment_valid_expression* ($\vdash$ _ $\surd_e$) **where**
attachment_valid_expression e =
($\exists$ *T*. *attachment_type_valid_expression e T*)

end

B.20 Memory state validity

theory *StateValidity* **imports**
EnvironmentAttachment
State
TopSet
Value
begin

type_synonym
'local_type local_state = *'local_type State.local_state*
type_synonym
(*'local_type*, *'memory*) *state* = (*'local_type*, *'memory*) *State.state*

B.20.1 Run-time attachment status

Function *valid_local_values* tells if a given state of local variables is valid.

definition *valid_local_values*::
'local_type local_state ⇒ *bool* — Notation: $\vdash$ *local_state*
where
$\vdash$ *loc* $\longleftrightarrow$ ($\forall$ *name value*. *loc name* = $\lfloor$*value*$\rfloor$ $\longrightarrow$ *value* ≠ *Unit*)

definition *local_has_value*:: *'local_type local_state* ⇒ *vname* ⇒ *bool* **where**
local_has_value l name $\longleftrightarrow$ ($\exists$ *v*. *l name* = $\lfloor$*v*$\rfloor$)

definition *valid_local_state*::
attachment_environment ⇒ *'local_type local_state* ⇒ *bool*
where

```
 valid_local_state Γ l ⟷
    (∀ name T. Γ name = ⌊T⌋ ⟶ local_has_value l name)

lemma local_has_value': ⟦valid_local_state Γ l; Γ n = ⌊T⌋⟧ ⟹ ∃ v. l n = ⌊v⌋
  by (simp add: valid_local_state_def local_has_value_def)

lemma local_state_upd:
  ⟦valid_local_state Γ l⟧ ⟹
    valid_local_state Γ (l (name ↦ value))
using assms
  by (simp add: valid_local_state_def local_has_value_def)

definition valid_local_attachment_state::
  vname topset ⇒ 'local_type local_state ⇒ bool
where
 valid_local_attachment_state A l ⟷
    A ≠ ⊤ ⟶ (∀ name. name ∈⊤ A ⟶
    (∃ value. l name = ⌊value⌋ ∧ value ≠ Void_v))

lemma local_attachment_state_top [simp]: valid_local_attachment_state ⊤ l
  by (simp add: valid_local_attachment_state_def)

lemma local_attachment_state_bottom [simp]:
 valid_local_attachment_state ⌈∅⌉ l
   by (simp add: valid_local_attachment_state_def)

lemma local_attachment_state_union:
  ⟦valid_local_attachment_state A l; valid_local_attachment_state B l⟧ ⟹
    valid_local_attachment_state (A ⊔ B) l
proof (cases A, simp, cases B, simp)
  fix a b
  assume
    HA: A = ⌈a⌉ and
    HB: B = ⌈b⌉ and
    VA: valid_local_attachment_state A l and
    VB: valid_local_attachment_state B l
  then have
    HAB: A ⊔ B = ⌈a ∪ b⌉ using topset_union_def
     by (metis topset.simps(5))
  {
    fix name
    assume
      ∗: name ∈⊤ A ⊔ B
    with HAB have name ∈ a ∪ b by simp
    moreover from ∗ HA VA have
     ∃ value. l name = ⌊value⌋ ∧ value ≠ Void_v if name ∈⊤ A
```

```
      using that valid_local_attachment_state_def by blast
    moreover from ∗ HB VB have
     ∃ value. l name = ⌊value⌋ ∧ value ≠ Void_ν if name ∈⊤ B
      using that valid_local_attachment_state_def by blast
    ultimately have ∃ value. l name = ⌊value⌋ ∧ value ≠ Void_ν
     using HA HB by blast
  }
  then have ∀ name. name ∈⊤ A ⊔ B ⟶
    (∃ value. l name = ⌊value⌋ ∧ value ≠ Void_ν) by simp
  then show valid_local_attachment_state (A ⊔ B) l
   using valid_local_attachment_state_def by blast
qed

lemma local_attachment_state_exc:
  assumes
    valid_local_attachment_state A l and
    ¬ name ∈⊤ A
  shows valid_local_attachment_state A (l (name ↦ value))
using assms valid_local_attachment_state_def
 by (metis (mono_tags) fun_upd_other)

lemma local_attachment_state_anti_mono:
  assumes
    B ⩽ A and
    A ≠ ⊤
  shows
   valid_local_attachment_state A l ⟹ valid_local_attachment_state B l
    using assms
     by (simp add: topset_subsetD valid_local_attachment_state_def)

lemma local_attachment_state_sub:
  valid_local_attachment_state A l ⟹
  valid_local_attachment_state (A ⊖ name) (l (name ↦ value))
  apply (cases A)
  apply simp
  apply (simp add: topset_member_def valid_local_attachment_state_def
     topset_rem_def)
done

lemma local_attachment_state_add:
  assumes
    HA: value ≠ Void_ν and
    HV: valid_local_attachment_state A l
  shows valid_local_attachment_state (A ⊕ name) (l (name ↦ value))
proof (cases A)
  case Top then show ?thesis
```

```
  by (simp add: valid_local_attachment_state_def)
next
 case (Set a)
 with HV have ∀ n. n ∈⊤ A ⟶ (∃ v. l n = ⌊v⌋ ∧ v ≠ Void_v)
  using valid_local_attachment_state_def by auto
 with Set HA have ∀ n. n ∈ (a ∪ {name}) ⟶
 (∃ v. (l (name ↦ value)) n = ⌊v⌋ ∧ v ≠ Void_v)
  by (simp add: topset_member_def)
 with Set have ∀ n. n ∈⊤ (A ⊕ name) ⟶
 (∃ v. (l (name ↦ value)) n = ⌊v⌋ ∧ v ≠ Void_v)
  by (simp add: topset_add_def topset_member_def)
 with Set show ?thesis using valid_local_attachment_state_def by blast
qed

lemma local_attachment_state_upd:
 assumes
  HA: value ≠ Void_v and
  HV: valid_local_attachment_state A l
 shows valid_local_attachment_state A (l (name ↦ value))
using assms local_attachment_state_add local_attachment_state_exc
 topset_add_insert topset_insert_absorb by metis
```

Function *valid_state* tells if a given run-time state conforms to a specified attachment status of variables.

```
primrec valid_state::
 attachment_environment ⇒ vname topset ⇒ ('l, 'm) state ⇒ bool
  (_, _ ⊢ _ √s [60, 60, 60] 60)
   — Notation: environment, attached_vars ⊢ state √s
where
Γ, A ⊢ (loc, mem) √s ⟷
  valid_local_state Γ loc ∧
  valid_local_attachment_state A loc

lemma valid_stateI [intro]:
⟦valid_local_state Γ loc; valid_local_attachment_state A loc⟧ ⟹
Γ, A ⊢ (loc, mem) √s
  by simp

lemma valid_stateD1 [dest?]: Γ, A ⊢ (loc, mem) √s ⟹
 valid_local_state Γ loc  by simp

lemma valid_stateD2 [dest?]: Γ, A ⊢ (loc, mem) √s ⟹
 valid_local_attachment_state A loc by simp

lemma valid_state_anti_mono:
```

assumes
$B \leqslant A$ **and**
$A \neq \top$
shows $\Gamma, A \vdash s \surd_s \Longrightarrow \Gamma, B \vdash s \surd_s$
proof −
obtain l m **where**
$*$: $s = (l, m)$ **by** *fastforce*
moreover assume
$\Gamma, A \vdash s \surd_s$
ultimately have
valid_local_state Γ l **and**
valid_local_attachment_state A l
by *auto*
then have
valid_local_state Γ l **and**
valid_local_attachment_state B l
using *assms local_attachment_state_anti_mono* **by** *auto*
with $*$ **show** *?thesis* **by** *simp*
qed

lemma *valid_state_union*:
$[\![\Gamma, A \vdash (loc, mem) \surd_s; \Gamma, B \vdash (loc, mem) \surd_s]\!] \Longrightarrow \Gamma, A \sqcup B \vdash (loc, mem) \surd_s$
by (*simp add*: *local_attachment_state_union*)

B.20.2 *Run-time state decomposition.*

lemma *local_has_value*: $[\![\Gamma, A \vdash s \surd_s; \Gamma\ n = \lfloor T \rfloor]\!] \Longrightarrow \exists\ v.\ fst\ s\ n = \lfloor v \rfloor$
proof −
assume $\Gamma, A \vdash s \surd_s$ **and** $\Gamma\ n = \lfloor T \rfloor$
moreover obtain l m **where** $l = fst\ s$ **and** $m = snd\ s$ **by** *simp*
moreover have $[\![\Gamma, A \vdash (l, m) \surd_s; \Gamma\ n = \lfloor T \rfloor]\!] \Longrightarrow \exists\ v.\ l\ n = \lfloor v \rfloor$
by (*auto simp add*: *local_has_value'*)
ultimately show *?thesis* **by** *simp*
qed

lemma *local_is_attached*:
$[\![\Gamma, A \vdash s \surd_s; A \neq \top; n \in^\top A]\!] \Longrightarrow \exists\ v.\ fst\ s\ n = \lfloor v \rfloor \wedge v \neq Void_\nu$
proof −
assume $\Gamma, A \vdash s \surd_s$ **and** $A \neq \top$ **and** $n \in^\top A$
moreover obtain l m **where** $l = fst\ s$ **and** $m = snd\ s$ **by** *simp*
moreover have
$[\![\Gamma, A \vdash (l, m) \surd_s; A \neq \top; n \in^\top A]\!] \Longrightarrow \exists\ v.\ l\ n = \lfloor v \rfloor \wedge v \neq Void_\nu$
by (*auto simp add*: *valid_local_attachment_state_def*)
ultimately show *?thesis* **by** *simp*
qed

B.20.3 *Run-time state updates.*

lemma *attachment_set_sub*:
assumes $\Gamma, A \vdash (l, m)\ \surd_s$
shows $\Gamma, A \ominus name \vdash (l\ (name \mapsto value), m)\ \surd_s$
using *assms* **by** (*simp add*: *local_state_upd local_attachment_state_sub*)

lemma *attachment_set_add*:
assumes
$value \neq Void_v$ **and**
$\Gamma, A \vdash (l, m)\ \surd_s$
shows $\Gamma, A \oplus name \vdash (l\ (name \mapsto value), m)\ \surd_s$
using *assms local_attachment_state_add local_state_upd* **by** *auto*

lemma *attachment_set_int1*:
assumes
HV: $\Gamma, A1 \vdash (l1, m1)\ \surd_s$ **and**
HD1: $A1 \neq \top$ **and**
HD2: $A2 \neq \top$
shows
$\Gamma, A1 \sqcap A2 \vdash (l1, m1)\ \surd_s$
proof −
from *HD1 HD2* **have** $A1 \sqcap A2 \neq \top$ **using** *topset_inter_topD1* **by** *auto*
moreover have $A1 \sqcap A2 \leqslant A1$ **by** (*rule topset_inter_lower1*)
with *HV HD1 HD2* **show** *?thesis* **using** *valid_state_anti_mono* **by** *blast*
qed

lemma *attachment_set_int2*:
assumes
HV: $\Gamma, A2 \vdash (l2, m2)\ \surd_s$ **and**
HD2: $A2 \neq \top$ **and**
HD1: $A1 \neq \top$
shows
$\Gamma, A1 \sqcap A2 \vdash (l2, m2)\ \surd_s$
proof −
from *assms* **have** $\Gamma, A2 \sqcap A1 \vdash (l2, m2)\ \surd_s$ **by** (*rule attachment_set_int1*)
then show *?thesis* **by** (*simp add*: *topset_inter_commute*)
qed

definition *attachment_valid_state* $(\_ \vdash \_\ \surd_s)$ **where**
$attachment\_valid\_state\ \Gamma\ s = \Gamma, \varnothing \vdash s\ \surd_s$

end

B.21 Attachment correctness

theory *BigStepSafety* **imports**
BigStep MemoryAttachment ExpressionValidity StateValidity
begin

no_notation *floor* (⌊_⌋)
no_notation *Set.member* ((_/ : _) [51, 51] 50)

B.21.1 Preservation of valid run-time state

A void-safe assignment instruction preserves a void-safe state.

lemma *valid_state_preservation_by_value_assignment*:
assumes
HS: $\Gamma \vdash \langle n ::= \mathit{Value}\ v, (l, m)\rangle \Rightarrow \langle c', (l', m')\rangle$ **and**
VS: $\Gamma, A \vdash (l, m)\ \surd_s$
shows $\Gamma, \mathcal{A}\ (n ::= \mathit{Value}\ v)\ A \vdash (l', m')\ \surd_s$
using *assms*
proof (*cases A*)
case (*Set a*)
show *?thesis*
proof (*cases* $v \neq \mathit{Void}_v$)
case *True*
moreover with *Set* **have** $\mathcal{A}\ (n ::= \mathit{Value}\ v)\ A = A \oplus n$ **by** *simp*
moreover from *VS* **have** $\Gamma, A \vdash (l, m')\ \surd_s$ **by** *auto*
with *True* **have** $\Gamma, A \oplus n \vdash (l\ (n \mapsto v), m')\ \surd_s$ **using** *attachment_set_add*
by *fastforce*
moreover from *HS* **have** $*: l' = l\ (n \mapsto v)$ **by** *auto*
ultimately show *?thesis* **by** *auto*
next
case *False*
moreover from *False Set* **have** $\mathcal{A}\ (n ::= \mathit{Value}\ v)\ A = A \ominus n$ **by** *simp*
moreover from *VS* **have** $\Gamma, A \vdash (l, m')\ \surd_s$ **by** *auto*
then have $\Gamma, A \ominus n \vdash (l\ (n \mapsto v), m')\ \surd_s$ **by** (*rule attachment_set_sub*)
moreover from *HS* **have** $*: l' = l\ (n \mapsto v)$ **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
next
case *Top*
with *assms A_to_all* **show** *?thesis* **using** *local_state_upd* **by** *fastforce*
qed

A void-safe creation instruction preserves a void-safe state.

lemma *valid_state_preservation_by_creation*:
assumes

```
  HS: Γ ⊢ ⟨create n, (l, m)⟩ ⇒ ⟨c', (l', m')⟩ and
  HT: A ⊢ create n : T and
  VS: Γ, A ⊢ (l, m) √s and
  HE: c' ≠ Exception
 shows Γ, A (create n) A ⊢ (l', m') √s
proof −
 from HS HE obtain Tn v where
  Γ n = ⌊Tn⌋ and
  instance m Tn = Some (m', v) and
  ∗: l' = l (n ↦ v) by auto
 hence v ≠ Void_v by (auto simp add: instance_def)
 moreover from VS have Γ, A ⊢ (l, m') √s by auto
 ultimately have Γ, A ⊕ n ⊢ (l (n ↦ v), m') √s
  by (rule attachment_set_add)
 with ∗ show ?thesis by auto
qed
```

If there is a transition from a local-valid state, the resulting state is local-valid.

```
lemma valid_local_state_preservation:
 fixes
  e :: ('b, 't) expression and es :: ('b, 't) expression list
 assumes
  valid_local_state Γ (fst s)
 shows
  valid_local_state_preservation_e:
   Γ ⊢ ⟨e, s⟩ ⇒ ⟨e', s'⟩ ⟹ valid_local_state Γ (fst s') and
  valid_local_state_preservation_es:
   Γ ⊢ ⟨es, s⟩ [⇒] ⟨es', s'⟩ ⟹ valid_local_state Γ (fst s')
using assms
 by (induction rule: big_step_big_steps.inducts)
    (simp_all add: instance_is_value local_state_upd option.case_eq_if)

lemma valid_local_state_preservation':
 fixes
  e :: ('b, 't) expression and es :: ('b, 't) expression list
 assumes
  valid_local_state Γ l
 shows
  valid_local_state_preservation_e':
   Γ ⊢ ⟨e, (l, m)⟩ ⇒ ⟨e', (l', m')⟩ ⟹ valid_local_state Γ l' and
  valid_local_state_preservation_es':
   Γ ⊢ ⟨es, (l, m)⟩ [⇒] ⟨es', (l', m')⟩ ⟹ valid_local_state Γ l'
using assms valid_local_state_preservation by auto
```

If there is a transition from a valid state for unreachable code, the resulting state is valid for unreachable code.

lemma *indefinedness_preservation_e*:
assumes
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle$
$\Gamma, \top \vdash s \surd_s$
shows
$\Gamma, \top \vdash s' \surd_s$
using *assms local_attachment_state_top valid_local_state_preservation_e valid_stateD1 valid_stateI* **by** (*metis prod.collapse*)

lemma *indefinedness_preservation_es*:
assumes
$\Gamma \vdash \langle es, s\rangle$ [⇒] $\langle es', s'\rangle$
$\Gamma, \top \vdash s \surd_s$
shows
$\Gamma, \top \vdash s' \surd_s$
using *assms local_attachment_state_top valid_local_state_preservation_es valid_stateD1 valid_stateI* **by** (*metis prod.collapse*)

If there is a transition from a valid state for reachable code, there is some reachable code for which the new state is valid.

lemma
fixes
e :: ('*b*, '*t*) *expression* **and** *es* :: ('*b*, '*t*) *expression list*
shows
definedness_preservation_e:
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow \Gamma, \lceil a\rceil \vdash s \surd_s \Longrightarrow \exists\ b.\ \Gamma, \lceil b\rceil \vdash s' \surd_s$ **and**
definedness_preservation_es:
$\Gamma \vdash \langle es, s\rangle$ [⇒] $\langle es', s'\rangle \Longrightarrow \Gamma, \lceil a\rceil \vdash s \surd_s \Longrightarrow \exists\ b.\ \Gamma, \lceil b\rceil \vdash s' \surd_s$
using *assms*
proof (*induction arbitrary*: *a* **and** *a rule*: *big_step_big_steps.inducts*)
case (*Assign e s v l m n a*)
then obtain *b*::*vname set* **where** $\Gamma, \lceil b\rceil \vdash (l, m) \surd_s$ **by** *blast*
then have $\Gamma, \lceil b\rceil \ominus n \vdash (l(n \mapsto v), m) \surd_s$ **by** (*rule attachment_set_sub*)
thus *?case*
by (*metis topset.exhaust topset_inter_rem topset_inter_rem_distrib topset_inter_top_right*)
next
case (*Create n T m m' v l a*)
moreover with *Create* **obtain** *Tn* **and** *v* :: '*b value*
where $\Gamma\ n = \lfloor Tn\rfloor$ *instance m Tn* $= \lfloor(m', v)\rfloor$ **by** *simp*
moreover hence *instance m Tn* $= \lfloor(m', v)\rfloor$ **by** *simp*
hence *is_attached_value v* **by** (*rule instance_is_attached*)
ultimately show *?case* **using** *local_state_upd valid_local_attachment_state_def attached_value_is_not_void*
by *fastforce*
next

case ($Loop_{false}$ *e c*) **then show** *?case* **by** *meson*
next
case ($Test_{true}$ *e*) **then show** *?case*
by (*metis attachment_set_sub topset.collapse topset_rem_absorb*)
qed *auto*

If there is a transition from an expression known be True, the resulting expression is either a value True or an exception.

lemma *result_true*:
$[\![is\_true\ e; \Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle]\!] \Longrightarrow e' = True_c \vee e' = Exception$
using *is_true.elims*(2) **by** *blast*

If there is a transition from an expression known be False, the resulting expression is either a value False or an exception.

lemma *result_false*:
$[\![is\_false\ e; \Gamma \vdash \langle e, s\rangle \Rightarrow \langle c, s'\rangle]\!] \Longrightarrow c = False_c \vee c = Exception$
using *is_false.elims*(2) **by** *blast*

If there is a transition from a void-safe expression to an expression that is not an exception, the static analysis treats code after that expression as reachable if the code before the expression is reachable.

lemma
fixes
e :: *'b attachable_expression* **and** *es* :: *'b attachable_expression list*
assumes
$A \neq \top$
shows
reachability_preservation:
$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' \neq Exception \Longrightarrow$
$A \rhd e \neq \top$ **and**
reachability_preservation_scope_true:
$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = True_c \Longrightarrow$
$A \rhd+ e \neq \top$ **and**
reachability_preservation_scope_false:
$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = False_c \Longrightarrow$
$A \rhd- e \neq \top$ **and**
reachability_preservation_es:
$\Gamma \vdash \langle es, (l, m)\rangle\ [\Rightarrow]\ \langle es', (l', m')\rangle \Longrightarrow$
$A \vdash es\ [:]\ Ts \Longrightarrow$

```
   Exception ∉ set es' ⟹
   A ▷▷ es ≠ ⊤ and
  Γ ⊢ ⟨es, (l, m)⟩ [⇒] ⟨es', (l', m')⟩ ⟹ True and
  Γ ⊢ ⟨es, (l, m)⟩ [⇒] ⟨es', (l', m')⟩ ⟹ True
using assms
proof (induction arbitrary: A T and A Ts rule: big_step_induct)
  fix A :: vname topset and T
  case (Value v l m)
  assume A ≠ ⊤
  then show A ▷ (Value v) ≠ ⊤ by simp
  then show A ▷+ (Value v) ≠ ⊤ by simp
  then show A ▷− (Value v) ≠ ⊤ by simp
next
  fix A :: vname topset and T
  case (Local l n v m)
  assume A ≠ ⊤
  then show A ▷ (Local n) ≠ ⊤ by simp
  then show A ▷+ (Local n) ≠ ⊤ by simp
  then show A ▷− (Local n) ≠ ⊤ by simp
next
  fix A :: vname topset and T
  case (Seq c1 l m l1 m1 c2 c2' l' m')
  moreover assume
    c2' ≠ Exception
    A ⊢ c1 ;; c2 : T
    A ≠ ⊤
  ultimately show A ▷ c1 ;; c2 ≠ ⊤ by auto
next
  fix A :: vname topset and T
  case (Seq c1 l m l1 m1 c2 c2' l' m')
  moreover assume
    c2' = True_c
    A ⊢ c1 ;; c2 : T
    A ≠ ⊤
  ultimately show A ▷+ c1 ;; c2 ≠ ⊤
   using result_false by (cases is_false (c1 ;; c2)) fastforce+
next
  fix A :: vname topset and T
  case (Seq c1 l m l1 m1 c2 c2' l' m')
  moreover assume
    c2' = False_c
    A ⊢ c1 ;; c2 : T
    A ≠ ⊤
  ultimately show A ▷− c1 ;; c2 ≠ ⊤
   using result_true by (cases is_true (c1 ;; c2)) fastforce+
next
```

```
  fix A :: vname topset and T n
  case (Assign e l m v l' m')
  moreover assume
    A ⊢ n ::= e : T
    A ≠ ⊤
  ultimately show A ▷ n ::= e ≠ ⊤ by auto
  then show A ▷+ n ::= e ≠ ⊤ by simp
  then show A ▷− n ::= e ≠ ⊤ by simp
next
  fix A :: vname topset and T
  case (Create n Tn m m' v)
  assume
    A ≠ ⊤
  then show A ▷ create n ≠ ⊤ by simp
  then show A ▷+ create n ≠ ⊤ by simp
  then show A ▷− create n ≠ ⊤ by simp
next
  fix A :: vname topset and T
  case (Create_fail n Tn m)
  assume
    A ≠ ⊤
  then show A ▷ create n ≠ ⊤ by simp
  then show A ▷+ create n ≠ ⊤ by simp
  then show A ▷− create n ≠ ⊤ by simp
next
  fix A :: vname topset and T f
  case (Call e l m v le me es vs l' m')
  moreover assume
    lHc: A ⊢ e . f (es) : T and
    lHD: A ≠ ⊤
  moreover with Call.IH have A ▷ e ≠ ⊤ by auto
  moreover have Exception ∉ set (map Value vs) by auto
  moreover from lHc obtain Ts where A ▷ e ⊢ es [:] Ts by blast
  ultimately have A ▷ e ▷▷ es ≠ ⊤ by blast
  then show A ▷ e . f (es) ≠ ⊤ by auto
  then show A ▷+ e . f (es) ≠ ⊤ by simp
  then show A ▷− e . f (es) ≠ ⊤ by simp
next
  fix A :: vname topset and T c2
  case (If_true b l m lb mb c1 c1' l1 m1)
  moreover assume
    lHc1: c1' ≠ Exception and
    lHc: A ⊢ if b then c1 else c2 end : T and
    lHA: A ≠ ⊤
  ultimately have A ▷+ b ≠ ⊤ by auto
  moreover from lHc obtain Tc where A ▷+ b ⊢ c1 : Tc by auto
```

moreover from *lHc1* **have** $A \rhd+\ b \rhd c1 \neq \top$
using If_{true}.$IH(4)$ *calculation* **by** *blast*
ultimately show $A \rhd$ *if b then c1 else c2 end* $\neq \top$
using *topset_inter_topD1* **by** *auto*
next
fix A :: *vname topset* **and** T *c2*
case (If_{true} *b l m lb mb c1 c1' l1 m1*)
moreover assume
lHc1: $c1' = True_c$ **and**
lHc: $A \vdash$ *if b then c1 else c2 end* : T **and**
lHA: $A \neq \top$
ultimately have $A \rhd+\ b \neq \top$ **by** *auto*
moreover from *lHc* **obtain** *Tc* **where** $A \rhd+\ b \vdash c1 : Tc$ **by** *auto*
ultimately show $A \rhd+$ *if b then c1 else c2 end* $\neq \top$
using *lHc1* If_{true} *topset_inter_topD1 A_If value_neq_exception result_false At_If*
by (*metis expression.inject*(1) *object_value.inject*(2) *value.inject*(1))
next
fix A :: *vname topset* **and** T *c2*
case (If_{true} *b l m lb mb c1 c1' l1 m1*)
moreover assume
lHc1: $c1' = False_c$ **and**
lHc: $A \vdash$ *if b then c1 else c2 end* : T **and**
lHA: $A \neq \top$
ultimately have $A \rhd+\ b \neq \top$ **by** *auto*
moreover from *lHc* **obtain** *Tc* **where** $A \rhd+\ b \vdash c1 : Tc$ **by** *auto*
ultimately show $A \rhd-$ *if b then c1 else c2 end* $\neq \top$
using *lHc1* If_{true} *topset_inter_topD1 A_If value_neq_exception unreachable_if_true Af_If* **by** *metis*
next
fix A :: *vname topset* **and** T *c1*
case (If_{false} *b l m lb mb c2 c2' l2 m2*)
moreover assume
lHc1: $c2' \neq Exception$ **and**
lHc: $A \vdash$ *if b then c1 else c2 end* : T **and**
lHA: $A \neq \top$
ultimately have $A \rhd-\ b \neq \top$ **by** *auto*
moreover from *lHc* **obtain** *Tc* **where** $A \rhd-\ b \vdash c2 : Tc$ **by** *auto*
moreover from *lHc1* **have** $A \rhd-\ b \rhd c2 \neq \top$
using If_{false}.$IH(4)$ *calculation* **by** *blast*
ultimately show $A \rhd$ *if b then c1 else c2 end* $\neq \top$
using *topset_inter_topD2* **by** *auto*
next
fix A :: *vname topset* **and** T *c1*
case (If_{false} *b l m lb mb c2 c2' l2 m2*)
moreover assume

```
    lHc2: c2' = True_c and
    lHc: A ⊢ if b then c1 else c2 end : T and
    lHA: A ≠ ⊤
  ultimately have A ▷− b ≠ ⊤ by auto
  moreover from lHc obtain Tc where A ▷− b ⊢ c2 : Tc by auto
  ultimately show A ▷+ if b then c1 else c2 end ≠ ⊤
   using lHc2 If_false topset_inter_topD2 A_If value_neq_exception
    unreachable_if_false At_If by metis
next
  fix A :: vname topset and T c1
  case (If_false b l m lb mb c2 c2' l2 m2)
  moreover assume
    lHc2: c2' = False_c and
    lHc: A ⊢ if b then c1 else c2 end : T and
    lHA: A ≠ ⊤
  ultimately have A ▷− b ≠ ⊤ by auto
  moreover from lHc obtain Tc where A ▷− b ⊢ c2 : Tc by auto
  ultimately show A ▷− if b then c1 else c2 end ≠ ⊤
   using lHc2 If_false topset_inter_topD2 A_If value_neq_exception
    unreachable_if_true Af_If by metis
next
  fix A :: vname topset and T c
  case (Loop_true e l m)
  moreover assume
    lHc: A ⊢ until e loop c end : T and
    lHA: A ≠ ⊤
  then have gfp (λB. A ⊓ (B ▷− e ▷ c)) ⊢ e : Attached by auto
  moreover have gfp (λB. A ⊓ (B ▷− e ▷ c)) ⩽ A
   by (meson gfp_least topset_inter_subset_iff)
  ultimately have A ⊢ e : Attached using AT_mono by blast
  moreover have A ▷ until e loop c end ⩽ A ▷+ e
   by (rule attachment_loop_condition)
  moreover from lHA have A ▷+ e ≠ ⊤
   using Loop_true calculation by blast
  ultimately show A ▷ until e loop c end ≠ ⊤by auto
  then show A ▷+ until e loop c end ≠ ⊤ by simp
  then show A ▷− until e loop c end ≠ ⊤ by simp
next
  fix A :: vname topset and T
  case (Loop_false e l m le me c lc mc c' l' m')
  moreover assume
    lHc': c' ≠ Exception and
    lHc: A ⊢ until e loop c end : T and
    lHA: A ≠ ⊤
  ultimately show A ▷ until e loop c end ≠ ⊤ by simp
next
```

fix A :: *vname topset* **and** T
case ($Loop_{false}$ *e l m le me c lc mc c' l' m'*)
moreover assume
lHc': $c' = True_c$ **and**
lHc: $A \vdash$ *until e loop c end* : T **and**
lHA: $A \neq \top$
ultimately show $A \rhd+$ *until e loop c end* $\neq \top$ **by** *simp*
next
fix A :: *vname topset* **and** T
case ($Loop_{false}$ *e l m le me c lc mc c' l' m'*)
moreover assume
lHc': $c' = False_c$ **and**
lHc: $A \vdash$ *until e loop c end* : T **and**
lHA: $A \neq \top$
ultimately show $A \rhd-$ *until e loop c end* $\neq \top$ **by** *simp*
next
fix A :: *vname topset* **and** T *n*
case ($Test_{true}$ *e l m v l' m' t*)
moreover assume
lHv: $v \neq Void_v \wedge v$ *has_type t* **and**
lHc: $A \vdash$ *attached t e as n* : T **and**
lHA: $A \neq \top$
moreover then obtain *Te* **where** $A \vdash e$: *Te* **by** *blast*
ultimately have
lHe: $A \rhd e \neq \top$ **using** *value_neq_exception* **by** *blast*
from *lHe* **show** $A \rhd$ *attached t e as n* $\neq \top$ **by** *auto*
from *lHe* **show** $A \rhd+$ *attached t e as n* $\neq \top$
by (*cases e*) (*simp_all add*: *topset_union_topI*)
from *lHe* **show** $A \rhd-$ *attached t e as n* $\neq \top$
by (*cases e*) (*simp_all add*: *topset_union_topI*)
next
fix A :: *vname topset* **and** T *n*
case ($Test_{false}$ *e l m v l' m' t*)
moreover assume
lHv: $\neg$ ($v \neq Void_v \wedge v$ *has_type t*) **and**
lHc: $A \vdash$ *attached t e as n* : T **and**
lHA: $A \neq \top$
moreover then obtain *Te* **where** $A \vdash e$: *Te* **by** *blast*
ultimately have
lHe: $A \rhd e \neq \top$ **using** *value_neq_exception* **by** *blast*
from *lHe* **show** $A \rhd$ *attached t e as n* $\neq \top$ **by** *auto*
from *lHe* **show** $A \rhd+$ *attached t e as n* $\neq \top$
by (*cases e*) (*simp_all add*: *topset_union_topI*)
from *lHe* **show** $A \rhd-$ *attached t e as n* $\neq \top$
by (*cases e*) (*simp_all add*: *topset_union_topI*)
qed *auto*

lemma
reachability_preservation':
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' \neq Exception \Longrightarrow$
$A \neq \top \Longrightarrow$
$A \rhd e \neq \top$ **and**
reachability_preservation_scope_true':
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = True_c \Longrightarrow$
$A \neq \top \Longrightarrow$
$A \rhd+ e \neq \top$ **and**
reachability_preservation_scope_false':
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = False_c \Longrightarrow$
$A \neq \top \Longrightarrow$
$A \rhd- e \neq \top$ **and**
reachability_preservation_es':
$\Gamma \vdash \langle es, s\rangle\ [\Rightarrow]\ \langle es', s'\rangle \Longrightarrow$
$A \vdash es\ [:]\ Ts \Longrightarrow$
$Exception \notin set\ es' \Longrightarrow$
$A \neq \top \Longrightarrow$
$A \rhd\rhd es \neq \top$
using *reachability_preservation* **apply** (*metis prod.collapse*)
using *reachability_preservation_scope_true* **apply** (*metis prod.collapse*)
using *reachability_preservation_scope_false* **apply** (*metis prod.collapse*)
using *reachability_preservation_es* **apply** (*metis prod.collapse*)
done

If there is a transition from a void-safe expression in reachable code and the code after the expression is unreachable, the resulting expression is an exception.

lemma *exception_detection*:
assumes
HS: $\Gamma \vdash \langle c, (l, m)\rangle \Rightarrow \langle c', (l', m')\rangle$ **and**
HT: $A \vdash c : T$ **and**
HD: $A \neq \top$ **and**
HE: $A \rhd c = \top$
shows
$c' = Exception$
using *reachability_preservation assms* **by** *fastforce*

lemma *local_attachment_state_if1*:
assumes

HV: *valid_local_attachment_state* $(A \rhd+\ c \rhd e_1)\ l$ **and**
HD: $A \rhd+\ c \rhd e_1 \neq \top$
shows *valid_local_attachment_state* $(A \rhd \textit{if } c \textit{ then } e_1 \textit{ else } e_2 \textit{ end})\ l$
using *assms topset_inter_lower1*
by (*metis A_If local_attachment_state_anti_mono*)

lemma *local_attachment_state_if2*:
assumes
HV: *valid_local_attachment_state* $(A \rhd-\ c \rhd e_2)\ l$ **and**
HD: $A \rhd-\ c \rhd e_2 \neq \top$
shows *valid_local_attachment_state* $(A \rhd \textit{if } c \textit{ then } e_1 \textit{ else } e_2 \textit{ end})\ l$
using *assms topset_inter_lower2* **by** (*metis A_If local_attachment_state_anti_mono*)

If a transition from a void-safe expression of an attached type in a void-safe state yields a value, this value is attached.

lemma *valid_local_attachment_state_preservation''*:
fixes
e :: *'b attachable_expression* **and** *es* :: *'b attachable_expression list*
assumes
HV: *valid_local_attachment_state A l* **and**
HD: $A \neq \top$
shows

$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = \textit{Value } w \Longrightarrow$
$T = \textit{Attached} \Longrightarrow$
$w \neq \textit{Void}_{\nu}$ **and**

$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = \textit{Value } w \Longrightarrow$
valid_local_attachment_state $(A \rhd e)\ l'$ **and**

$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = \textit{True}_{\mathrm{c}} \Longrightarrow$
valid_local_attachment_state $(A \rhd+\ e)\ l'$ **and**

$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = \textit{False}_{\mathrm{c}} \Longrightarrow$
valid_local_attachment_state $(A \rhd-\ e)\ l'$ **and**

$\Gamma \vdash \langle es, (l, m)\rangle\ [\Rightarrow]\ \langle es', (l', m')\rangle \Longrightarrow$

```
    A ⊢ es [:] Ts ⟹
    es' = map Value ws ⟹
    Detachable ∉ set Ts ⟹
    Void_ν ∉ set ws and

    Γ ⊢ ⟨es, (l, m)⟩ [⇒] ⟨es', (l', m')⟩ ⟹
    A ⊢ es [:] Ts ⟹
    es' = map Value ws ⟹
    valid_local_attachment_state (A ▷▷ es) l' and
   Γ ⊢ ⟨es, (l, m)⟩ [⇒] ⟨es', (l', m')⟩ ⟹ True and
   Γ ⊢ ⟨es, (l, m)⟩ [⇒] ⟨es', (l', m')⟩ ⟹ True
using assms
proof (induction arbitrary: A T w and A Ts ws rule: big_step_induct)
 fix A T w
 case (Value v l m)
 assume
  lHt: A ⊢ Value v : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: (Value v) = (Value w)
 from lHv show valid_local_attachment_state (A ▷ Value v) l by simp
 then show valid_local_attachment_state (A ▷+ Value v) l by simp
 then show valid_local_attachment_state (A ▷− Value v) l by simp
 assume
  T = Attached
 with lHw lHt show w ≠ Void_ν by auto
next
 fix A T w
 case (Local l n v m)
 assume
  lHv: l n = ⌊v⌋ and
  lHT: A ⊢ Local n : T and
  lHV: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: (Value v) = (Value w)
 then show valid_local_attachment_state (A ▷ Local n) l by simp
 then show valid_local_attachment_state (A ▷+ Local n) l by simp
 then show valid_local_attachment_state (A ▷− Local n) l by simp
 assume
  T = Attached
 with lHA lHw lHT lHv lHV show w ≠ Void_ν
  using valid_local_attachment_state_def ExpressionValidity.LocalE
   by (metis attachment_type.distinct(1) expression.inject(1)
     option.inject)
next
 fix A T w
```

```
 case (Seq c1 l m l1 m1 c2 c2' l' m')
 assume
  lHT: A ⊢ c1 ;; c2 : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: c2' = Value w
 from lHT lHv lHA lHw show
 lHS: valid_local_attachment_state (A ▷ c1 ;; c2) l'
   using Seq.IH(2) Seq.IH(6) Seq.hyps(1) value_neq_exception
    exception_detection by fastforce
 from lHS show valid_local_attachment_state (A ▷+ c1 ;; c2) l' by simp
 from lHS show valid_local_attachment_state (A ▷− c1 ;; c2) l' by simp
 assume
  T = Attached
 with Seq.hyps Seq.IH lHT lHv lHA lHw show w ≠ Void_v
   using exception_detection by fastforce
next
 fix A T and w :: 'b value and  n
 case (Assign e l m v le me)
 assume
  lHT: A ⊢ n ::= e : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: unit = Value w
 from lHT lHv lHA show
 valid_local_attachment_state (A ▷ n ::= e) (le(n ↦ v))
   using attached_assignment_set detachable_assignment_set
    local_attachment_state_add local_attachment_state_sub
    by (metis (full_types) Assign.IH(1) Assign.IH(2)
     ExpressionValidity.AssignE attachment_type.exhaust)
 then show valid_local_attachment_state (A ▷+ n ::= e) (le(n ↦ v))
  by simp
 then show valid_local_attachment_state (A ▷− n ::= e) (le(n ↦ v))
  by simp
 assume
  T = Attached
 from lHw show w ≠ Void_v by simp
next
 fix A T and w :: 'b value and l :: 'b local_state
 case (Create n Tn m m' v)
 assume
  lHT: A ⊢ create n : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: unit = Value w
 from lHv Create.hyps show
```

```
  valid_local_attachment_state (A ▷ create n) (l(n ↦ v))
   using local_attachment_state_add A_Create instance_is_not_void
    by metis
 then show valid_local_attachment_state (A ▷+ create n) (l(n ↦ v))
  by simp
 then show valid_local_attachment_state (A ▷− create n) (l(n ↦ v))
  by simp
 assume
  T = Attached
 from lHw show w ≠ Void_v by simp
next
 fix A T and w :: 'b value and l :: 'b local_state
 case (Create_fail n Tn m)
 assume
  lHT: A ⊢ create n : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: Exception = Value w
 then show w ≠ Void_v by simp
 from lHv lHw show  valid_local_attachment_state (A ▷ create n) l
  by simp
 then show valid_local_attachment_state (A ▷+ create n) l by simp
 then show valid_local_attachment_state (A ▷− create n) l by simp
next
 fix A T and w :: 'b value and  f
 case (Call e l m v le me es vs l' m')
 assume
  lHT: A ⊢ e . f (es) : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: unit = Value w
 then show w ≠ Void_v by blast
 from lHT lHv lHA Call.IH show
  valid_local_attachment_state (A ▷ e . f (es)) l'
   using A_Call A_to_all local_attachment_state_top
    by (metis ExpressionValidity.CallE)
 then show valid_local_attachment_state (A ▷+ e . f (es)) l'
  by simp
 then show valid_local_attachment_state (A ▷− e . f (es)) l'
  by simp
next
 fix A T w c2
 case H: (If_true b l m lb mb c1 c1' l1 m1)
 assume
  lHT: A ⊢ if b then c1 else c2 end : T and
  lHv: valid_local_attachment_state A l and
```

```
    lHA: A ≠ ⊤ and
    lHw: c1' = Value w
  from lHT lHv lHA lHw H.hyps H.IH show
   valid_local_attachment_state (A ▷ if b then c1 else c2 end) l1
     using value_neq_exception exception_detection
       local_attachment_state_if1 reachability_preservation_scope_true
       by (metis ExpressionValidity.IfE)
 assume
  T = Attached
 with H lHT lHv lHA lHw show w ≠ Void_v
  using reachability_preservation_scope_true upper_bound_bot_left'
  by (metis ExpressionValidity.IfE)
next
  fix A T w c2
  case (If_true b l m lb mb c1 c1' l1 m1)
  moreover assume
    lH1: c1' = True_c and
    lHT: A ⊢ if b then c1 else c2 end : T and
    lHv: valid_local_attachment_state A l and
    lHA: A ≠ ⊤ and
    lHw: c1' = Value w
  then have valid_local_attachment_state (A ▷ if b then c1 else c2 end) l1
    using value_neq_exception exception_detection
      local_attachment_state_if1 reachability_preservation_scope_true
      by (metis If_true.IH(3) If_true.IH(6)
        If_true.hyps(1) If_true.hyps(2) ExpressionValidity.IfE)
  moreover from lH1 have ¬ is_false c1
   using If_true.hyps result_false by blast
  ultimately show
   valid_local_attachment_state (A ▷+ if b then c1 else c2 end) l1
     using If_true.IH lH1 lHT lHv by fastforce
next
  fix A T w c2
  case (If_true b l m lb mb c1 c1' l1 m1)
  moreover assume
    lH1: c1' = False_c and
    lHT: A ⊢ if b then c1 else c2 end : T and
    lHv: valid_local_attachment_state A l and
    lHA: A ≠ ⊤ and
    lHw: c1' = Value w
  then have valid_local_attachment_state (A ▷ if b then c1 else c2 end) l1
    using value_neq_exception exception_detection
      local_attachment_state_if1 reachability_preservation_scope_true
      by (metis If_true.IH(3) If_true.IH(6)
        If_true.hyps(1) If_true.hyps(2) ExpressionValidity.IfE)
  moreover from lH1 have ¬ is_true c1
```

```
  using If_true.hyps result_true by blast
 ultimately show
  valid_local_attachment_state (A ▷− if b then c1 else c2 end) l1
   using If_true.IH lH1 lHT lHv by fastforce
next
 fix A T w c1
 case H: (If_false b l m lb mb c2 c2' l2 m2)
 assume
  lHT: A ⊢ if b then c1 else c2 end : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: c2' = Value w
 from lHT lHv lHA lHw H show
  valid_local_attachment_state (A ▷ if b then c1 else c2 end) l2
   using value_neq_exception exception_detection
    local_attachment_state_if2 reachability_preservation_scope_false
    by (metis ExpressionValidity.IfE)
assume
 T = Attached
with lHT lHv lHA lHw H show w ≠ Void_v
  using reachability_preservation_scope_false upper_bound_bot_right'
  by (metis ExpressionValidity.IfE)
next
 fix A T w c1
 case (If_false b l m lb mb c2 c2' l2 m2)
 moreover assume
  lH2: c2' = True_c and
  lHT: A ⊢ if b then c1 else c2 end : T and
  lHv: valid_local_attachment_state A l and
  lHA: A ≠ ⊤ and
  lHw: c2' = Value w
 from lHT lHv lHA lHw have
  valid_local_attachment_state (A ▷ if b then c1 else c2 end) l2
   using value_neq_exception exception_detection
    local_attachment_state_if2 reachability_preservation_scope_false
    by (metis If_false.IH(4) If_false.IH(6)
     If_false.hyps(1) If_false.hyps(2) ExpressionValidity.IfE)
 moreover from lH2 have ¬ is_false c2
  using If_false.hyps result_false by blast
 ultimately show
  valid_local_attachment_state (A ▷+ if b then c1 else c2 end) l2
   using If_false.IH lH2 lHT lHv by fastforce
next
 fix A T w c1
 case (If_false b l m lb mb c2 c2' l2 m2)
 moreover assume
```

lH2: $c2' = False_c$ **and**
lHT: $A \vdash$ *if b then c1 else c2 end* : T **and**
lHv: *valid_local_attachment_state A l* **and**
lHA: $A \neq \top$ **and**
lHw: $c2' = Value\ w$
from *lHT lHv lHA lHw* **have**
valid_local_attachment_state $(A \rhd$ *if b then c1 else c2 end*$)$ *l2*
using *value_neq_exception exception_detection*
local_attachment_state_if2 reachability_preservation_scope_false
by (*metis* If_{false}*.IH*(4) If_{false}*.IH*(6)
If_{false}*.hyps*(1) If_{false}*.hyps*(2) *ExpressionValidity.IfE*)
moreover from *lH2* **have** $\neg$ *is_true c2*
using If_{false}*.hyps result_true* **by** *blast*
ultimately show
valid_local_attachment_state $(A \rhd -$ *if b then c1 else c2 end*$)$ *l2*
using If_{false}*.IH lH2 lHT lHv* **by** *fastforce*
next
fix $A\ T$ **and** w :: *'b value* **and** *c*
case $(Loop_{true}\ e\ l\ m\ le\ me)$
assume
lHT: $A \vdash$ *until e loop c end* : T **and**
lHv: *valid_local_attachment_state A l* **and**
lHA: $A \neq \top$ **and**
lHw: $unit = Value\ w$
then show $w \neq Void_v$ **by** *simp*
from *lHT* **have** *gfp* $(\lambda\ B.\ A \sqcap B \rhd - e \rhd c) \vdash e$: *Attached* **by** *blast*
moreover have *gfp* $(\lambda\ B.\ A \sqcap B \rhd - e \rhd c) \leqslant A$
by (*simp add*: *topset_gfp_inter_left*)
ultimately have $\exists\ Tb.\ A \vdash e$: $Tb \wedge Tb \rightarrow_a$ *Attached*
using *AT_mono* **by** *metis*
then have
lHe: $A \vdash e$: *Attached* **using** *conforms_to_attached* **by** *blast*
with $Loop_{true}$*.IH lHA lHv* **have**
valid_local_attachment_state $(A \rhd + e)$ *le* **by** *simp*
moreover from *lHA* $Loop_{true}$*.hyps lHe* **have** $A \rhd + e \neq \top$
by (*simp add*: *reachability_preservation_scope_true*)
moreover have $A \rhd$ *until e loop c end* $\leqslant A \rhd + e$
by (*rule attachment_loop_condition*)
ultimately show
valid_local_attachment_state $(A \rhd$ *until e loop c end*$)$ *le*
by (*simp add*: *local_attachment_state_anti_mono*)
then show
valid_local_attachment_state $(A \rhd +$ *until e loop c end*$)$ *le* **by** *simp*
then show
valid_local_attachment_state $(A \rhd -$ *until e loop c end*$)$ *le* **by** *simp*
next

fix $A\ T\ w$
case $(Loop_{false}\ e\ l\ m\ le\ me\ c\ lc\ mc\ c'\ l'\ m')$
assume
lHT: $A \vdash$ *until e loop c end* : T **and**
lHv: *valid_local_attachment_state* $A\ l$ **and**
lHA: $A \neq \top$ **and**
lHw: $c' =$ *Value* w
obtain $sc\ s'\ f$ **where**
$sc = (lc, mc)$ **and** $s' = (l', m')$ **and**
lHf: $f =$ *until e loop c end*
by *simp*
with $Loop_{false}$*.hyps* **have**
lHB: $\Gamma \vdash \langle f, sc\rangle \Rightarrow \langle c', s'\rangle$ **by** *simp*
from lHT **have** *gfp* $(\lambda\ B.\ A \sqcap B \rhd\!-\ e \rhd c) \vdash e$: *Attached* **by** *auto*
moreover have *gfp* $(\lambda\ B.\ A \sqcap B \rhd\!-\ e \rhd c) \leqslant A$
by (*simp add*: *topset_gfp_inter_left*)
ultimately have $\exists\ Tb.\ A \vdash e$: $Tb \wedge Tb \rightarrow_{\alpha}$ *Attached*
using *AT_mono* **by** *metis*
then have
$lHTe$: $A \vdash e$: *Attached* **using** *conforms_to_attached* **by** *blast*
with $Loop_{false}$*.IH* $lHv\ lHA$ **have**
$lHVe$: *valid_local_attachment_state* $(A \rhd\!-\ e)\ le$ **by** *simp*
moreover from $lHA\ Loop_{false}$*.hyps* $lHTe$ **have**
$lHDe$: $A \rhd\!-\ e \neq \top$ **by** (*simp add*: *reachability_preservation_scope_false*)
from lHT **have** $(gfp\ (\lambda B.\ A \sqcap (B \rhd\!-\ e \rhd c)) \rhd\!-\ e) \vdash c$: *Attached* **by** *auto*
moreover have $(gfp\ (\lambda B.\ A \sqcap (B \rhd\!-\ e \rhd c)) \rhd\!-\ e) \leqslant A \rhd\!-\ e$
by (*simp add*: *topset_gfp_inter_left Af_mono'*)
ultimately have $\exists\ Tb.\ (A \rhd\!-\ e) \vdash c$: $Tb \wedge Tb \rightarrow_{\alpha}$ *Attached*
using *AT_mono* **by** *metis*
then have
$lHTc$: $(A \rhd\!-\ e) \vdash c$: *Attached* **using** *conforms_to_attached* **by** *blast*
with $Loop_{false}$*.IH* $lHDe\ lHVe$ **have**
valid_local_attachment_state $(A \rhd\!-\ e \rhd c)\ lc$ **by** *simp*
moreover from $Loop_{false}$*.hyps* $lHDe\ lHTc$ **have**
$lHDc$: $A \rhd\!-\ e \rhd c \neq \top$ **by** (*simp add*: *reachability_preservation*)
moreover from lHT **have** $T =$ *Attached* **by** *auto*
with lHT **have**
$lHTc$: $A \rhd\!-\ e \rhd c \vdash$ *until e loop c end*: *Attached*
using *AT_loop_step* **by** *metis*
ultimately have
$lHVc$: *valid_local_attachment_state* $(A \rhd\!-\ e \rhd c \rhd$ *until e loop c end*$)\ l'$
using $Loop_{false}$*.IH* lHw **by** *simp*
from lHw **have** $c' \neq$ *Exception* **by** *simp*
from $Loop_{false}$*.hyps* $lHTc\ lHDc$ *this* **have**
$lHDl$: $A \rhd\!-\ e \rhd c \rhd$ *until e loop c end* $\neq \top$

```
    using reachability_preservation(1) by metis
  have A ▷ until e loop c end ⩽ A ▷− e ▷ c ▷ until e loop c end
   by (rule loop_application1)
  from this lHDl lHVc show
   valid_local_attachment_state (A ▷ until e loop c end) l'
    using local_attachment_state_anti_mono by blast
  then show
   valid_local_attachment_state (A ▷+ until e loop c end) l' by simp
  then show
   valid_local_attachment_state (A ▷− until e loop c end) l' by simp
  assume
   T = Attached
  moreover have
   ⟦Γ ⊢ ⟨f, sc⟩ ⇒ ⟨c', s'⟩; f = until e loop c end; c' = Value w⟧ ⟹
    w = Unit
   Γ ⊢ ⟨es, sc⟩ [⇒] ⟨es', s'⟩ ⟹ True
    by (induction rule: big_step_big_steps.inducts) auto
  moreover note lHf lHw lHB
  ultimately have w = Unit by simp
  then show w ≠ Void_v by simp
next
  fix A T and w :: 'b value and n
  case (Test_true e l m v le me t)
  assume
    lHa: v ≠ Void_v ∧ v has_type t and
    lHT: A ⊢ attached t e as n : T and
    lHv: valid_local_attachment_state A l and
    lHA: A ≠ ⊤ and
    lHw: True_c = Value w
  then show w ≠ Void_v by auto
  from lHA lHT lHa lHv show
    lHVc: valid_local_attachment_state (A ▷ attached t e as n) (le(n ↦ v))
      using Test_true.IH local_attachment_state_upd by fastforce
  then have
    lHVe: valid_local_attachment_state (A ▷ e) le
     using Test_true.IH lHA lHT lHv by auto
  moreover from lHa have
    lHa': is_attached v by simp
  ultimately have
    lHVn: valid_local_attachment_state (A ▷ e ⊕ n) (le(n ↦ v))
      using local_attachment_state_add by auto
  show valid_local_attachment_state (A ▷+ attached t e as n) (le(n ↦ v))
  proof (cases ∃ x. e = Local x)
    case True
    then obtain x where e = Local x by blast
    then have A ▷+ attached t e as n = A ▷ e ⊕ x ⊕ n by simp
```

```
    then have A ▷+ attached t e as n = A ▷ e ⊕ n ⊕ x by simp
    moreover have le x = ⌊v⌋
     using Test_true.hyps(1) ‹e = Local x› by blast
    then have (le (n ↦ v)) x = ⌊v⌋by simp
    moreover from lHa' have v ≠ Void_v by simp
    moreover note lHVn
    ultimately show ?thesis using local_attachment_state_add
     by (metis fun_upd_triv)
   next
    case False
    then have A ▷+ attached t e as n = A ▷ e ⊕ n
     by (cases e) simp_all
    then show ?thesis using lHVn by simp
   qed
   from lHVc show
    valid_local_attachment_state (A ▷− attached t e as n) (le(n ↦ v))
    by simp
  next
   fix A T and w :: 'b value and n
   case (Test_false e l m v le me t)
   assume
    lHa: ¬ (v ≠ Void_v ∧ v has_type t) and
    lHT: A ⊢ attached t e as n : T and
    lHv: valid_local_attachment_state A l and
    lHA: A ≠ ⊤ and
    lHw: False_c = Value w
   then show w ≠ Void_v by auto
   from lHA lHT lHv show
    lHVc: valid_local_attachment_state (A ▷ attached t e as n) le
     using Test_false.IH(2) by auto
   then have valid_local_attachment_state (A ▷ e) le by simp
   from lHVc show
    valid_local_attachment_state (A ▷− attached t e as n) le by simp
   assume
    False_c = True_c
   then show
    valid_local_attachment_state (A ▷+ attached t e as n) le by simp
  next
   fix A Ts and ws :: 'b value list
   case (Nil l m)
   assume
    lHV: valid_local_attachment_state A l and
    lHA: A ≠ ⊤
   show True by simp
   show True by simp
   assume
```

lHT: $A \vdash [] [:] Ts$ **and**
lHw: $[] = map\ Value\ ws$
from *lHV* **show** *valid_local_attachment_state* $(A \rhd\rhd [])\ l$ **by** *simp*
assume
Detachable $\notin$ *set Ts*
from *lHw* **show** $Void_{\mathcal{V}} \notin set\ ws$ **by** *simp*
next
fix *A Ts ws*
case (*Cons e l m v le me es es' l' m'*)
assume
lHV: *valid_local_attachment_state A l* **and**
lHA: $A \neq \top$
show *True* **by** *simp*
show *True* **by** *simp*
assume
lHT: $A \vdash e \# es [:] Ts$ **and**
lHw: $(Value\ v) \# es' = map\ Value\ ws$
from *lHT* **obtain** $T'\ Ts'$ **where**
lHeT: $A \vdash e : T'$ **and**
lHesT: $A \rhd e \vdash es [:] Ts'$ **and**
lHTc: $Ts = T' \# Ts'$
using *list_attachment_validity_tail* **by** *blast*
moreover from *lHw* **have**
lHesv: $es' = map\ Value\ (tl\ ws)$ **by** *auto*
moreover from *Cons.IH lHA lHV lHeT* **have**
lHeV: *valid_local_attachment_state* $(A \rhd e)$ *le* **by** *auto*
moreover from *Cons.hyps lHA lHeT* **have**
lHAe: $A \rhd e \neq \top$ **using** *reachability_preservation* **by** *fastforce*
moreover note *Cons.IH lHesT*
ultimately show
valid_local_attachment_state $(A \rhd\rhd (e \# es))\ l'$ **by** *auto*
from *lHw* **have**
$v = hd\ ws$ **by** *auto*
moreover assume
lHTsD: *Detachable* $\notin$ *set Ts*
with *lHT lHTc* **have**
lHeD: $T' = Attached$ **using** *attachment_type.exhaust* **by** *auto*
moreover note *Cons.IH lHA lHV lHeT*
ultimately have
lHwh: $hd\ ws \neq Void_{\mathcal{V}}$ **by** *blast*
from *lHTsD lHTc* **have**
lHesD: *Detachable* $\notin$ *set Ts'* **by** *simp*
with *Cons.IH lHesT lHesv lHeV lHAe* **have** $Void_{\mathcal{V}} \notin set\ (tl\ ws)$ **by** *simp*
with *lHwh lHw* **show** $Void_{\mathcal{V}} \notin set\ ws$
using *hd_Cons_tl Nil_is_map_conv* **by** *fastforce*
qed *auto*

A transition from a void-safe expression with a void-safe state that does not lead to an exception preserves the void-safe status of the state.

lemma *valid_local_attachment_state_preservation*:
 assumes
  HS: $\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle$ **and**
  HT: $A \vdash e : T$ **and**
   HV: *valid_local_attachment_state A l* **and**
   HE: $e' \neq$ *Exception*
 shows
  valid_local_attachment_state $(\mathcal{A}\ e\ A)\ l'$
proof (*cases A*)
 case *Top* **thus** *?thesis* **by** (*simp add*: *valid_local_attachment_state_def*)
next
 case (*Set a*)
 from *HS* **have** *Final e'* **by** (*simp add*: *big_step_final*)
 then obtain *v* **where** $e' = (Value\ v) \vee e' =$ *Exception*
  by (*auto simp add*: *Final_def*)
 hence $e' =$ *Value v* **by** (*simp add*: *HE*)
 with *HS HT HV Set* **show** *?thesis*
  using *valid_local_attachment_state_preservation''*(2) **by** *fastforce*
qed

If there is a transition from a void-safe expression of an attached type in reachable code with a void-safe state to a value, this value is not Void.

lemma *local_attachment_type_preservation*:
 assumes
  HS: $\Gamma \vdash \langle c, (l, m)\rangle \Rightarrow \langle Value\ v, (l', m')\rangle$ **and**
  HT: $A \vdash c$: *Attached* **and**
   HV: *valid_local_attachment_state A l* **and**
   HD: $A \neq \top$
 shows $v \neq Void_v$
using *assms valid_local_attachment_state_preservation''*(1) **by** *fastforce*

If there is a transition from a void-safe expression with a valid state to a value, the resulting state is valid with respect to the context after the expression.

lemma
 fixes
  e :: *'b attachable_expression* **and** *es* :: *'b attachable_expression list*
 assumes
  $\Gamma, A \vdash (l, m)\ \surd_s$
 shows
  valid_state_preservation:
  $\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
  $A \vdash e : T \Longrightarrow$

$e' = \mathit{Value}\ w \Longrightarrow$
$\Gamma, (A \rhd e) \vdash (l', m') \surd_s$ **and**
valid_state_preservation_scope_true:
$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = \mathit{True}_c \Longrightarrow$
$\Gamma, (A \rhd+ e) \vdash (l', m') \surd_s$ **and**
valid_state_preservation_scope_false:
$\Gamma \vdash \langle e, (l, m)\rangle \Rightarrow \langle e', (l', m')\rangle \Longrightarrow$
$A \vdash e : T \Longrightarrow$
$e' = \mathit{False}_c \Longrightarrow$
$\Gamma, (A \rhd- e) \vdash (l', m') \surd_s$ **and**
valid_state_preservation_es:
$\Gamma \vdash \langle es, (l, m)\rangle\ [\Rightarrow]\ \langle es', (l', m')\rangle \Longrightarrow$
$A \vdash es\ [:]\ Ts \Longrightarrow$
$es' = \mathit{map}\ \mathit{Value}\ ws \Longrightarrow$
$\Gamma, A \rhd\rhd es \vdash (l', m') \surd_s$ **and**

$\Gamma \vdash \langle es, (l, m)\rangle\ [\Rightarrow]\ \langle es', (l', m')\rangle \Longrightarrow \mathit{True}$ **and**

$\Gamma \vdash \langle es, (l, m)\rangle\ [\Rightarrow]\ \langle es', (l', m')\rangle \Longrightarrow \mathit{True}$
using *assms*
proof (*induction arbitrary*: *A T w* **and** *A Ts ws rule*: *big_step_induct*)
fix *A T w*
case (*Value v l m*)
assume
$A \vdash \mathit{Value}\ v : T$
$\Gamma, A \vdash (l, m) \surd_s$
$(\mathit{Value}\ v) = (\mathit{Value}\ w)$
then show $\Gamma, A \rhd \mathit{Value}\ v \vdash (l, m) \surd_s$ **by** *simp*
then show $\Gamma, A \rhd+ \mathit{Value}\ v \vdash (l, m) \surd_s$ **by** *simp*
then show $\Gamma, A \rhd- \mathit{Value}\ v \vdash (l, m) \surd_s$ **by** *simp*
next
fix *A T w*
case (*Local l n v m*)
assume
$l\ n = \lfloor v \rfloor$
$A \vdash \mathit{Local}\ n : T$
$\Gamma, A \vdash (l, m) \surd_s$
$(\mathit{Value}\ v) = (\mathit{Value}\ w)$
then show $\Gamma, A \rhd \mathit{Local}\ n \vdash (l, m) \surd_s$ **by** *simp*
then show $\Gamma, A \rhd+ \mathit{Local}\ n \vdash (l, m) \surd_s$ **by** *simp*
then show $\Gamma, A \rhd- \mathit{Local}\ n \vdash (l, m) \surd_s$ **by** *simp*
next
fix *A T w*
case (*Seq c1 l m l1 m1 c2 c2' l' m'*)

```
  assume
   A ⊢ c1 ;; c2 : T
   Γ, A ⊢ (l, m) √s
   c2 ′ = Value w
  then show
   S: Γ, A ▷ c1 ;; c2 ⊢ (l ′, m ′) √s
    using TransferFunction.A_Seq Seq.IH by auto
  from S show Γ, A ▷+ c1 ;; c2 ⊢ (l ′, m ′) √s by simp
  from S show Γ, A ▷− c1 ;; c2 ⊢ (l ′, m ′) √s by simp
next
  fix A T and w :: 'b value and n
  case (Assign e l m v le me)
  moreover assume
   A ⊢ n ::= e : T
   Γ, A ⊢ (l, m) √s
   unit = Value w
  ultimately show Γ, A ▷ n ::= e ⊢ (le(n ↦ v), me) √s
   using A_to_all
    attached_assignment_set detachable_assignment_set
    attachment_set_add attachment_set_sub
    valid_stateD1 valid_stateD2 valid_stateI
    local_state_upd valid_local_attachment_state_def
    local_attachment_type_preservation
   by (metis (full_types) ExpressionValidity.AssignE
    attachment_type.exhaust)
  then show Γ, A ▷+ n ::= e ⊢ (le(n ↦ v), me) √s by simp
  then show Γ, A ▷− n ::= e ⊢ (le(n ↦ v), me) √s by simp
next
  fix A T and w :: 'b value and l :: 'b local_state
  case (Create n Tn m m ′ v)
  assume
   A ⊢ create n : T
   Γ, A ⊢ (l, m) √s
   unit = Value w
  then show Γ, A ▷ create n ⊢ (l(n ↦ v), m ′) √s
   using value_neq_exception valid_state_preservation_by_creation
   by (metis Create.hyps(1) Create.hyps(2) big_step_big_steps.Create)
  then show Γ, A ▷+ create n ⊢ (l(n ↦ v), m ′) √s by simp
  then show Γ, A ▷− create n ⊢ (l(n ↦ v), m ′) √s by simp
next
  fix A T and w :: 'b value and l :: 'b local_state
  case (Create_fail n Tn m)
  assume
   A ⊢ create n : T
   Γ, A ⊢ (l, m) √s
   Exception = Value w
```

```
  then show Γ, A ▷ create n ⊢ (l, m) √s by simp
  then show Γ, A ▷+ create n ⊢ (l, m) √s by simp
  then show Γ, A ▷− create n ⊢ (l, m) √s by simp
next
  fix A T and w :: 'b value and f
  case (Call e l m v le me es vs l' m')
  assume
    A ⊢ e . f (es) : T
    Γ, A ⊢ (l, m) √s
    unit = Value w
  then show Γ, A ▷ e . f (es) ⊢ (l', m') √s using Call.IH by auto
  then show Γ, A ▷+ e . f (es) ⊢ (l', m') √s by simp
  then show Γ, A ▷− e . f (es) ⊢ (l', m') √s by simp
next
  fix A T w c2
  case (If_true b l m lb mb c1 c1' l' m')
  assume
    lHT: A ⊢ if b then c1 else c2 end : T and
    lHV: Γ, A ⊢ (l, m) √s and
    lHw: c1' = Value w
  then show
    lHVc: Γ, A ▷ if b then c1 else c2 end ⊢ (l', m') √s
      using A_to_all indefinedness_preservation_e
       valid_local_attachment_state_preservation''(2)
        valid_local_state_preservation_e'
        valid_stateD1 valid_stateD2 valid_stateI
      by (metis If_true.hyps(1) If_true.hyps(2)
        big_step_big_steps.If_true)
  assume
    c1' = True_c
  moreover with If_true.IH lHT lHV have
   Γ, A ▷+ b ▷+ c1 ⊢ (l', m') √s by auto
  ultimately show Γ, A ▷+ if b then c1 else c2 end ⊢ (l', m') √s
    using If_true.hyps(2) lHVc result_false by fastforce
next
  fix A T w c2
  case (If_true b l m lb mb c1 c1' l' m')
  assume
    lHT: A ⊢ if b then c1 else c2 end : T and
    lHV: Γ, A ⊢ (l, m) √s and
    lHw: c1' = Value w
  then have
    lHVc: Γ, A ▷ if b then c1 else c2 end ⊢ (l', m') √s
      using A_to_all indefinedness_preservation_e
       valid_local_attachment_state_preservation''(2)
        valid_local_state_preservation_e'
```

```
      valid_stateD1 valid_stateD2 valid_stateI
    by (metis If_true.hyps(1) If_true.hyps(2)
     big_step_big_steps.If_true)
  assume
   c1' = False_c
  with lHT lHV lHVc show Γ, A ▷− if b then c1 else c2 end ⊢ (l', m') √_s
   using A_to_all Af_to_all
    valid_local_attachment_state_preservation''(4)
    valid_stateD1 valid_stateD2 valid_stateI
   by (metis If_true.hyps(1) If_true.hyps(2)
    big_step_big_steps.If_true)
next
 fix A T w c1
 case (If_false b l m lb mb c2 c2' l' m')
 assume
  lHT: A ⊢ if b then c1 else c2 end : T and
  lHV: Γ, A ⊢ (l, m) √_s and
  lHw: c2' = Value w
 then show
  lHVc: Γ, A ▷ if b then c1 else c2 end ⊢ (l', m') √_s
   using A_to_all indefinedness_preservation_e
    valid_local_attachment_state_preservation''(2)
    valid_local_state_preservation_e'
    valid_stateD1 valid_stateD2 valid_stateI
   by (metis If_false.hyps(1) If_false.hyps(2)
    big_step_big_steps.If_false)
 assume
   c2' = True_c
 with lHT lHV lHVc show Γ, A ▷+ if b then c1 else c2 end ⊢ (l', m') √_s
  using A_to_all At_to_all
   valid_local_attachment_state_preservation''(3)
   valid_stateD1 valid_stateD2 valid_stateI
  by (metis If_false.hyps(1) If_false.hyps(2)
   big_step_big_steps.If_false)
next
 fix A T w c1
 case (If_false b l m lb mb c2 c2' l' m')
 assume
  lHT: A ⊢ if b then c1 else c2 end : T and
  lHV: Γ, A ⊢ (l, m) √_s and
  lHw: c2' = Value w
 then have
  lHVc: Γ, A ▷ if b then c1 else c2 end ⊢ (l', m') √_s
   using A_to_all indefinedness_preservation_e
    valid_local_attachment_state_preservation''(2)
    valid_local_state_preservation_e'
```

```
      valid_stateD1 valid_stateD2 valid_stateI
    by (metis If_false.hyps(1) If_false.hyps(2)
      big_step_big_steps.If_false)
  assume
    c2' = False_c
  with lHT lHV lHVc show Γ, A ▷− if b then c1 else c2 end ⊢ (l', m') √_s
    using A_to_all Af_to_all
      valid_local_attachment_state_preservation''(4)
      valid_stateD1 valid_stateD2 valid_stateI
    by (metis If_false.hyps(1) If_false.hyps(2)
      big_step_big_steps.If_false)
next
  fix A T and w :: 'b value and c
  case (Loop_true e l m le me)
  assume
    lHT: A ⊢ until e loop c end : T and
    lHV: Γ, A ⊢ (l, m) √_s and
    lHw: unit = Value w
  then have
    lHTe: loop_computation e c A ⊢ e: Attached by auto
  moreover have
    lHle: loop_computation e c A ⩽ A by (rule loop_computation_leo)
  ultimately have A ⊢ e: Attached
   using AT_mono conforms_to_attached by blast
  with lHTe lHV lHle show Γ, A ▷ until e loop c end ⊢ (le, me) √_s
    using A_to_all
      valid_state_anti_mono Loop_true.IH Loop_true.hyps
      attachment_loop_condition reachability_preservation_scope_true
      by metis
  then show Γ, A ▷+ until e loop c end ⊢ (le, me) √_s by simp
  then show Γ, A ▷− until e loop c end ⊢ (le, me) √_s by simp
next
  fix A T w
  case (Loop_false e l m le me c lc mc c' l' m')
  assume
    lHT: A ⊢ until e loop c end : T and
    lHV: Γ, A ⊢ (l, m) √_s and
    lHw: c' = Value w
  then show Γ, A ▷ until e loop c end ⊢ (l', m') √_s
    using indefinedness_preservation_e
      valid_local_attachment_state_preservation''(2)
      valid_local_state_preservation_e'
      valid_stateD1 valid_stateD2 valid_stateI
    by (metis Loop_false.IH(7) Loop_false.hyps(1)
      Loop_false.hyps(2) Loop_false.hyps(3)
      big_step_big_steps.Loop_false)
```

```
  then show Γ, A ▷+ until e loop c end ⊢ (l', m') √s by simp
  then show Γ, A ▷− until e loop c end ⊢ (l', m') √s by simp
next
  fix A T and w :: 'b value and n
  case (Test_true e l m v le me t)
  assume
    lHT: A ⊢ attached t e as n : T and
    lHV: Γ, A ⊢ (l, m) √s and
    lHw: True_c = Value w
  then show
    lHVc: Γ, A ▷ attached t e as n ⊢ (le(n ↦ v), me) √s
    using value_neq_exception local_attachment_state_upd local_state_upd
      valid_local_attachment_state_preservation
      valid_local_state_preservation_e'
      valid_stateD1 valid_stateD2 valid_stateI
    by (metis (no_types) Test_true.hyps(1) Test_true.hyps(2)
      ExpressionValidity.TestE TransferFunction.A_Test)
  with lHT lHV show Γ, A ▷+ attached t e as n ⊢ (le(n ↦ v), me) √s
    using A_to_all
      reachability_with_scope_true
      valid_local_attachment_state_preservation''(3)
      valid_stateD1 valid_stateD2 valid_stateI
    by (metis Test_true.hyps(1) Test_true.hyps(2)
      big_step_big_steps.Test_true)
  from lHVc show
  Γ, A ▷− attached t e as n ⊢ (le(n ↦ v), me) √s by simp
next
  fix A T and w :: 'b value and n
  case (Test_false e l m v le me t)
  assume
    lHT: A ⊢ attached t e as n : T and
    lHV: Γ, A ⊢ (l, m) √s and
    lHw: False_c = Value w
  with Test_false.hyps(1) have
    valid_local_state Γ le
     by (meson valid_local_state_preservation_e' valid_stateD1)
  then show
    lHVc: Γ, A ▷ attached t e as n ⊢ (le, me) √s
      using Test_false.IH(1) lHT lHV by auto
  from lHVc show Γ, A ▷− attached t e as n ⊢ (le, me) √s by simp
  assume
    False_c = True_c
  then show Γ, A ▷+ attached t e as n ⊢ (le, me) √s by simp
next
 fix A Ts ws
 case (Cons e l m v le me es es' l' m')
```

assume
$A \vdash e \# es\ [:]\ Ts$
$(Value\ v) \# es' = map\ Value\ ws$
$\Gamma, A \vdash (l, m)\ \surd_s$
with *Cons.IH* **show** $\Gamma, A \rhd\rhd (e \# es) \vdash (l', m')\ \surd_s$ **by** *auto*
next
fix *A Ts ws*
case (*ConsEx e l m l' m' es*)
assume
$A \vdash e \# es\ [:]\ Ts$
$Exception \# es = map\ Value\ ws$
$\Gamma, A \vdash (l, m)\ \surd_s$
then show $\Gamma, A \rhd\rhd (e \# es) \vdash (l', m')\ \surd_s$ **by** *blast*
qed *simp_all*

lemma
assumes
$\Gamma \vdash \langle c, s\rangle \Rightarrow \langle c', s'\rangle$
$A \vdash c : T$
$\Gamma, A \vdash s\ \surd_s$
shows
valid_state_preservation': $c' \neq Exception \Longrightarrow \Gamma, A \rhd c \vdash s'\ \surd_s$ **and**
valid_state_preservation_scope_true':
$c' = True_c \Longrightarrow \Gamma, A \rhd+ c \vdash s'\ \surd_s$ **and**
valid_state_preservation_scope_false':
$c' = False_c \Longrightarrow \Gamma, A \rhd- c \vdash s'\ \surd_s$
proof −
obtain *l m l' m'* **where**
lHs: $(l, m) = s$ **and**
lHs': $(l', m') = s'$ **using** *prod.collapse* **by** *blast*
from *valid_local_attachment_state_preservation*
valid_local_state_preservation'
show $c' \neq Exception \Longrightarrow \Gamma, A \rhd c \vdash s'\ \surd_s$
using *assms lHs lHs'* **by** *fastforce*
from *valid_state_preservation_scope_true*
show $c' = True_c \Longrightarrow \Gamma, A \rhd+ c \vdash s'\ \surd_s$
using *assms lHs lHs'* **by** *fastforce*
from *valid_state_preservation_scope_false*
show $c' = False_c \Longrightarrow \Gamma, A \rhd- c \vdash s'\ \surd_s$
using *assms lHs lHs'* **by** *fastforce*
qed

B.21.2 *Preservation of attachment property*

lemma *value_attachment*: $A \vdash Value\ v : \mathcal{T}_c\ v$

by (*cases v = Void*$_v$) (*fastforce*, *metis AT_ValueAtt attached_typed_value_is_not_void*)

lemma *values_attachment*: *A* ⊢ *map Value vs* [:] *map* $\mathcal{T}_c$ *vs*
using *value_attachment*
by (*induction vs arbitrary*: *A*) (*simp*, *fastforce*)

If there is a transition from a void-safe expression with valid state in reachable code to an expression different from an exception, the type of the latter conforms to the type of the original expression in the context after it.

theorem
fixes
e :: *'b attachable_expression* **and** *es* :: *'b attachable_expression list*
assumes
Γ, *A* ⊢ *s* $\surd_s$ **and**
A = ⌈*a*⌉
shows
attachment_preservation_step: Γ ⊢ ⟨*e*, *s*⟩ ⇒ ⟨*e'*, *s'*⟩ ⟹
A ⊢ *e* : *T* ⟹
e' ≠ *Exception* ⟹
∃ *T'*. *A* ▷ *e* ⊢ *e'* : *T'* ∧ *T'* ≤ *T* **and**
attachment_preservation_step_s: Γ ⊢ ⟨*es*, *s*⟩ [⇒] ⟨*es'*, *s'*⟩ ⟹
A ⊢ *es* [:] *Ts* ⟹
Exception ∉ *set es'* ⟹
∃ *Ts'*. *A* ⊢ *es'* [:] *Ts'*
using *assms*
proof (*induction arbitrary*: *A a T* **and** *A a Ts rule*:
big_step_big_steps.inducts)
case (*Value v l m A a*)
then show *?case* **by** *blast*
next
case (*Local l n v m A a*)
then show *?case*
proof (*cases T*)
case *Attached*
with *Local.hyps Local.prems* **show** *?thesis*
using *valid_local_attachment_state_def topset.discI* **by** *fastforce*
next
case *Detachable*
with *Local* **show** *?thesis* **by** *auto*
qed
next
case (*Seq* c_1 *s s'* c_2 c_2*' s'' A a*)
then have
lHS1: Γ ⊢ ⟨c_1, *s*⟩ ⇒ ⟨*unit*, *s'*⟩ **and**

```
    lHV: Γ, A ⊢ s √s
     by simp_all
   from Seq.prems have
    lHT1: A ⊢ c1: Attached and
    lHT2: A ▷ c1 ⊢ c2: Attached and
    lHT: T = Attached
     by auto
   from lHT1 lHS1 lHV have
    lHV1: Γ, A c1 A ⊢ s' √s using valid_state_preservation' by blast
   from lHS1 lHT1 ‹A = ⌈a⌉› have A ▷ c1 ≠ ⊤
    using reachability_preservation
     by (metis value_neq_exception prod.collapse topset.distinct(1))
   with Seq.prems lHT2 lHT lHV1 have
    ∃T'. A ▷ c1 ▷ c2 ⊢ c2': T' ∧ T' →a Attached
     using Seq.IH by (metis topset.collapse)
   then show ∃T'. (A ▷ c1 ;; c2) ⊢ c2': T' ∧ T' →a T
    by (simp add: lHT)
 next
   case Assign then show ?case by auto
 next
   case Create then show ?case by auto
 next
   case Call then show ?case by blast
 next
   case H: (If_true b s s' c1 c1' s'' c2 A a T)
   then have
    lHbT: A ⊢ b: Attached by blast
   with H.IH H.prems have
    Γ, (A ▷ b) ⊢ s' √s using value_neq_exception
     by (simp add: valid_state_preservation')
   from H.IH H.prems lHbT have
    lHS: Γ, (A ▷+ b) ⊢ s' √s using valid_state_preservation_scope_true
     by (metis prod.collapse)
   then have A ▷ b ⊢ True_c : Attached by auto
   from lHbT H.IH H.prems have
    A ▷ b ≠ ⊤ using reachability_preservation
     by (metis value_neq_exception prod.collapse topset.distinct(1))
   with lHbT H.IH H.prems have
    lHR: A ▷+ b ≠ ⊤ using reachability_preservation_scope_true
     by (metis prod.collapse topset.discI)
   from H obtain T1 where
    lHcT: A ▷+ b ⊢ c1 : T1 and
    lHcTT: T1 →a T using upper_bound_left by auto
   with lHR H.IH H.prems have A ▷+ b ▷ c1 ≠ ⊤
    using reachability_preservation by (metis prod.collapse)
   then obtain Tc where A ▷+ b ▷ c1 = ⌈Tc⌉
```

```
  using topset.exhaust by metis
 with H.IH lHS lHcT H.prems obtain T' where
  lIH1: A ▷+ b ▷ c1 ⊢ c1' : T' ∧ T' →a T1
    by (metis lHR topset.exhaust)
 from H.prems H.IH obtain v where
  c1' = Value v using Final_def big_step_final by metis
 with lIH1 have
  ∃T'. A ▷ if b then c1 else c2 end ⊢ c1' : T' ∧ T' →a T1 by auto
 with lHcTT show ?case
  using attachment_conforming_to_transitive by blast
next
 case H: (If_false b s s' c2 c2' s'' c1 A a T)
 then have
  lHbT: A ⊢ b : Attached by blast
 with H.IH H.prems have
  Γ, (A ▷ b) ⊢ s' √s
   using value_neq_exception by (simp add: valid_state_preservation')
 from H.IH H.prems lHbT have
  lHS: Γ, (A ▷− b) ⊢ s' √s using valid_state_preservation_scope_false
    by (metis prod.collapse)
 have A ▷ b ⊢ True_c : Attached by auto
 from lHbT H.IH H.prems have
  A ▷ b ≠ ⊤ using reachability_preservation
    by (metis value_neq_exception prod.collapse topset.distinct(1))
 with H.IH H.prems lHbT have
  lHR: A ▷− b ≠ ⊤ using reachability_preservation_scope_false
    by (metis old.prod.exhaust topset.discI)
 from H obtain T2 where
  lHcT: A ▷− b ⊢ c2 : T2 and
  lHcTT: T2 →a T using upper_bound_right by auto
 with lHR H.IH H.prems have A ▷− b ▷ c2 ≠ ⊤
  using reachability_preservation by (metis prod.collapse)
 then obtain Tc where A ▷− b ▷ c2 = ⌈Tc⌉
  using topset.exhaust by metis
 with H.IH lHS lHcT H.prems obtain T' where
  lIH1: A ▷− b ▷ c2 ⊢ c2' : T' ∧ T' →a T2
    by (metis lHR topset.exhaust)
 moreover from H.prems H.IH obtain v where
  c2' = Value v using Final_def big_step_final by metis
 with lIH1 have
  ∃T'. A ▷ if b then c1 else c2 end ⊢ c2' : T' ∧ T' →a T2 by auto
 with lHcTT show ?case
  using attachment_conforming_to_transitive by blast
next
 case (Loop_true e s s' c A a) then show ?case by auto
next
```

case ($Loop_{false}$ e s s_e c s_c c' s' A a T)
moreover then have
lHT: $T = Attached$ **by** *auto*
with $Loop_{false}$.*prems* **have** $A \vdash$ *until e loop c end*: *Attached* **by** *simp*
with $Loop_{false}$.*prems* **have**
lHTeo: *loop_computation e c A* $\vdash e$: *Attached* **and**
lHTco: *loop_computation e c A* $\rhd\!-\, e \vdash c$: *Attached*
by *auto*
then have
lHTe: $A \vdash e$: *Attached*
using *AT_mono conforms_to_attached loop_computation_leo* **by** *metis*
ultimately have
lHVe: Γ, $(A \rhd\!-\, e) \vdash s_e \surd_s$ **using** *valid_state_preservation_scope_false'*
by *metis*
from $Loop_{false}$.*prems* **have** $A \neq \top$ **by** *simp*
with $Loop_{false}$.*IH* $Loop_{false}$.*prems lHTe* **have** $A \rhd\!-\, e \neq \top$
using *reachability_preservation_scope_false prod.collapse* **by** *metis*
then obtain *ae* **where**
lHAe: $A \rhd\!-\, e = \lceil ae \rceil$ **using** *topset.exhaust* **by** *auto*
have *loop_computation e c A* $\leqslant A$ **using** *loop_computation_leo* **by** *metis*
then have *loop_computation e c A* $\rhd\!-\, e \leqslant A \rhd\!-\, e$
by (*simp add*: *Af_mono'*)
with *lHTco* **have**
lHTc: $A \rhd\!-\, e \vdash c$: *Attached*
using *AT_mono conforms_to_attached* **by** *blast*
have
lHEu: $unit \neq Exception$ **by** *simp*
from $Loop_{false}$.*IH lHTc lHVe lHEu* **have**
lHVc: Γ, $(A \rhd\!-\, e \rhd c) \vdash s_c \surd_s$
using *valid_state_preservation'* **by** *metis*
obtain l_e m_e l_c m_c **where** $s_e = (l_e, m_e)$ **and** $s_c = (l_c, m_c)$ **by** *fastforce*
with $Loop_{false}$.*IH* **have** $\Gamma \vdash \langle c, (l_e, m_e)\rangle \Rightarrow \langle unit, (l_c, m_c)\rangle$ **by** *simp*
with *lHAe lHTc lHEu* **have** $A \rhd\!-\, e \rhd c \neq \top$
by (*simp add*: *reachability_preservation*)
then obtain *ac* **where**
lHAc: $A \rhd\!-\, e \rhd c = \lceil ac \rceil$ **using** *topset.exhaust* **by** *auto*
with $Loop_{false}$.*IH* $Loop_{false}$.*prems AT_loop_step lHVc* **have**
lHTl: $\exists T'.\ A \rhd\!-\, e \rhd c \rhd$ *until e loop c end* $\vdash c' : T' \wedge T' \rightarrow_a T$
by *metis*
with *lHT* **have** $A \rhd\!-\, e \rhd c \rhd$ *until e loop c end* $\vdash c'$: *Attached*
using *conforms_to_attached* **by** *blast*
with $Loop_{false}$.*IH* $Loop_{false}$.*prems* **obtain** v **where**
$c' = Value\ v$ **using** *big_step_final_value* **by** *metis*
with *lHTl* **show** *?case* **by** *blast*
next

```
 case Test_true then show ?case by auto
next
 case Test_false then show ?case by auto
next
 case Nil then show ?case by simp
next
 case (Cons e s v se es es' s')
 from Cons.prems obtain Ts' where
  A ▷ e ⊢ es [:] Ts' using list_attachment_validity_tail by blast
 moreover from Cons have Γ, A ▷ e ⊢ se √s
  using value_neq_exception valid_state_preservation'
   by (metis ATs_iffs(3))
 moreover from Cons obtain b where A ▷ e = ⌈b⌉
  using value_neq_exception reachability_preservation
   by (metis ATs_iffs(3) prod.collapse topset.exhaust_sel)
 moreover note Cons
 ultimately have ∃a. A ▷ e ⊢ es' [:] a by auto
 then obtain Ts'' where
  lHTs: A ▷ e ⊢ es' [:] Ts'' by auto
 from Cons.prems have Exception ∉ set es' by simp
 moreover from Cons.IH have Finals es' using big_step_finals by metis
 ultimately have ∃ vs. map Value vs = es' using Finals_def
  by (metis in_set_conv_decomp)
 then obtain vs where es' = map Value vs by auto
 with lHTs have A ⊢ es' [:] Ts''
  using attachment_unique_es values_attachment by blast
 moreover obtain Te where A ▷ e ⊢ Value v : Te by blast
 ultimately have A ⊢ (Value v) # es' [:] Te # Ts'' by auto
 then show ?case by blast
qed simp_all
```

If there is a transition from a void-safe expression of an attached type with an empty state, the resulting expression is either an exception or a non-Void value.

```
theorem attachment_preservation:
 Γ ⊢ ⟨e, (empty, empty)⟩ ⇒ ⟨e', s'⟩ ⟹
 ⊢ e : Attached ⟹
 e' = Exception ∨ (∃ v. e' = Value v ∧ v ≠ Void_v)
using big_step_final_value local_attachment_state_bottom
 attachment_type_valid_expression_def
 local_attachment_type_preservation by (metis surj_pair topset.discI)

end
```

B.22 Conditional equivalence of void-safe and void-unsafe semantics

theory *BigStepEquivalence* **imports** *BigStepSafety BigStep_unsafe* **begin**

If there is a transition according to the void-safe big-step semantics, there is the same transition according to the void-unsafe one.

lemma
fixes
e :: ('b, 't) *expression* **and** *es* :: ('b, 't) *expression list*
shows
big_step_safe_implies_unsafe:
$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow \Gamma \vdash \langle e, s\rangle \Rightarrow' \langle e', s'\rangle$
and
big_steps_safe_implies_unsafe:
$\Gamma \vdash \langle es, s\rangle\ [\Rightarrow]\ \langle es', s'\rangle \Longrightarrow \Gamma \vdash \langle es, s\rangle\ [\Rightarrow]'\ \langle es', s'\rangle$
by (*induction rule*: *BigStep.big_step_big_steps.inducts*) *simp_all*

If there is a transition according to the void-unsafe big-step semantics for a void-safe expression starting from a void-safe state, there is the same transition according to the void-safe semantics.

lemma
fixes
e :: 'b *attachable_expression* **and** *es* :: 'b *attachable_expression list*
assumes
$\Gamma, A \vdash s\ \surd_s$ **and**
$A \neq \top$
shows
big_step_unsafe_implies_safe_step:
$\Gamma \vdash \langle e, s\rangle \Rightarrow' \langle e', s'\rangle \Longrightarrow A \vdash e : T \Longrightarrow \Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle$ **and**
big_step_unsafe_implies_safe_steps:
$\Gamma \vdash \langle es, s\rangle\ [\Rightarrow]'\ \langle es', s'\rangle \Longrightarrow A \vdash es\ [:]\ Ts \Longrightarrow \Gamma \vdash \langle es, s\rangle\ [\Rightarrow]\ \langle es', s'\rangle$
using *assms*
proof (*induction arbitrary*: *A T* **and** *A Ts*
rule: *BigStep_unsafe.big_step_big_steps.inducts*)
case *Value* **then show** *?case* **by** *simp*
next
case *Local* **then show** *?case* **by** *simp*
next
case (*Seq c1 s s' c2 c2' s''*)
then have $\Gamma \vdash \langle c1, s\rangle \Rightarrow \langle unit, s'\rangle$ **by** *auto*
moreover from *Seq* **have** $A \rhd c1 \neq \top$
using *value_neq_exception reachability_preservation*
by (*metis ExpressionValidity.SeqE prod.collapse*)
moreover with *Seq* **have** $\Gamma \vdash \langle c2, s'\rangle \Rightarrow \langle c2', s''\rangle$

```
    using value_neq_exception valid_state_preservation' by blast
  ultimately show ?case by simp
next
  case Assign then show ?case by auto
next
  case Create then show ?case by simp
next
  case Create_fail then show ?case by simp
next
 case (Call e s v se es vs s' n)
 then obtain Ts where
  lHeT: A ⊢ e : Attached and
  lHesTs: A ▷ e ⊢ es [:] Ts by blast
  with Call.IH Call.prems have
   lHeS: Γ ⊢ ⟨e, s⟩ ⇒ ⟨Value v, se⟩ by simp
  moreover with lHeT Call.prems have Γ, A ▷ e ⊢ se √s
  using value_neq_exception valid_state_preservation' by blast
 moreover from lHeS lHeT Call.prems have A ▷ e ≠ ⊤
  using value_neq_exception reachability_preservation
   by (metis prod.collapse)
 moreover note Call.IH lHesTs
 ultimately have Γ ⊢ ⟨es, se⟩ [⇒] ⟨map Value vs, s'⟩ by blast
  with lHeS Call.hyps show ?case by auto
next
  case (Call_fail e s v s' f es)
  from Call_fail.prems have A ⊢ e: Attached by auto
  with Call_fail.prems Call_fail.IH have v ≠ Void_v
    using StateValidity.valid_stateD2 local_attachment_type_preservation
    by (metis prod.collapse)
  with Call_fail.hyps show ?case by simp
next
  case (If_true b s s' c1 c1' s'' c2)
  then have
    lHb: Γ ⊢ ⟨b, s⟩ ⇒ ⟨True_c, s'⟩ by auto
  from If_true.prems have A ⊢ b : Attached by blast
  with If_true have A ▷+ b ≠ ⊤
   using reachability_preservation_scope_true by (metis prod.collapse)
  with lHb If_true.prems If_true.IH have Γ ⊢ ⟨c1, s'⟩ ⇒ ⟨c1', s''⟩
    using valid_state_preservation_scope_true' by blast
  with lHb show ?case by simp
next
  case (If_false b s s' c2 c2' s'' c1)
  then have
    lHb: Γ ⊢ ⟨b, s⟩ ⇒ ⟨False_c, s'⟩ by auto
  from If_false.prems have A ⊢ b : Attached by blast
  with If_false have A ▷− b ≠ ⊤ using reachability_preservation_scope_false
```

```
    by (metis prod.collapse)
  with lHb If_false.prems If_false.IH have Γ ⊢ ⟨c2, s'⟩ ⇒ ⟨c2', s''⟩
    using valid_state_preservation_scope_false' by blast
  with lHb show ?case by simp
next
  case (Loop_true e s s' c)
  then have TransferFunction.loop_computation e c A ⊢ e: Attached by auto
  moreover have
    lHle: TransferFunction.loop_computation e c A ⩽ A
     by (rule TransferFunction.loop_computation_leo)
  ultimately have A ⊢ e: Attached
   using ExpressionValidity.AT_mono conforms_to_attached by blast
  with Loop_true.prems Loop_true.IH have
    lHSe: Γ ⊢ ⟨e, s⟩ ⇒ ⟨True_c, s'⟩ by simp
  then show ?case by simp
next
  case (Loop_false e s se c sc c' s')
  then have
    lHLe: TransferFunction.loop_computation e c A ⊢ e: Attached by auto
  moreover have
    lHle: TransferFunction.loop_computation e c A ⩽ A
     by (rule TransferFunction.loop_computation_leo)
  ultimately have
    lHTe: A ⊢ e: Attached
     using ExpressionValidity.AT_mono conforms_to_attached by blast
  with Loop_false.prems Loop_false.IH have
    lHSe: Γ ⊢ ⟨e, s⟩ ⇒ ⟨False_c, se⟩ by simp
  from lHTe Loop_false have
    lHAe: A ▷— e ≠ ⊤ using reachability_preservation_scope_false
     by (metis prod.collapse)
  from lHSe lHTe Loop_false.prems have
    lHVe: Γ, A ▷— e ⊢ se √_s
     by (simp add: valid_state_preservation_scope_false')
  from Loop_false.prems have
    lHLc: TransferFunction.loop_computation e c A ▷— e ⊢ c: Attached by auto
  with lHle have
    lHTc: A ▷— e ⊢ c : Attached
     using ExpressionValidity.AT_mono conforms_to_attached Af_mono'
      by metis
  with lHAe lHVe Loop_false.IH have
    lHSc: Γ ⊢ ⟨c, se⟩ ⇒ ⟨unit, sc⟩ by simp
  with lHTc lHAe have
    lHAc: A ▷— e ▷ c ≠ ⊤
     using reachability_preservation value_neq_exception
      by (metis prod.collapse)
  from lHSc lHTc lHVe Loop_false.prems have
```

```
   lHVc: Γ, A ▷— e ▷ c ⊢ sc √s by (simp add: valid_state_preservation')
  with lHLe lHLc have A ▷— e ▷ c ⊢ until e loop c end : Attached
   by (simp add: AT_ATs.AT_Loop ExpressionValidity.AT_loop_step)
  with Loop_false.IH lHAc lHSc lHSe lHVc show ?case by simp
next
  case Test_true then show ?case by auto
next
  case Test_false then show ?case by auto
next
 case Nil then show ?case by simp
next
 case Cons then show ?case
  using value_neq_exception exception_detection valid_state_preservation'
   by (metis ATs_iffs(3) BigStep.big_step_big_steps.Cons prod.collapse)
next
  case Exception then show ?case by simp
next
  case SeqEx then show ?case by auto
next
  case AssignEx then show ?case by auto
next
  case CallEx then show ?case by auto
next
  case (CallArgEx e s v se es vs es' s' n)
  then have
   lHeT: A ⊢ e . n (es) : Attached by auto
  then obtain Ts where A ▷ e ⊢ es [:] Ts by blast
  from CallArgEx.IH CallArgEx.prems have
   lHse: Γ, A ▷ e ⊢ se √s using valid_state_preservation' by blast
 from CallArgEx have Γ ⊢ ⟨e, s⟩ ⇒ ⟨Value v, se⟩ by blast
  with lHeT lHse CallArgEx.IH CallArgEx.prems show ?case
   using value_neq_exception exception_detection
    BigStep.big_step_big_steps.CallArgEx ExpressionValidity.CallE
    by (metis surj_pair)
next
  case IfEx then show ?case by auto
next
  case (LoopEx e s s' c)
  then have TransferFunction.loop_computation e c A ⊢ e: Attached by auto
  moreover have
   lHle: TransferFunction.loop_computation e c A ⩽ A
    by (rule TransferFunction.loop_computation_leo)
  ultimately have
   A ⊢ e: Attached using ExpressionValidity.AT_mono by blast
  with LoopEx.IH LoopEx.prems show ?case by simp
next
```

```
 case (Loop_false Ex e s se c s')
 then have TransferFunction.loop_computation e c A ⊢ e: Attached by auto
 moreover have
  lHle: TransferFunction.loop_computation e c A ⩽ A
   by (rule TransferFunction.loop_computation_leo)
 ultimately have
  lHTe: A ⊢ e: Attached
   using ExpressionValidity.AT_mono conforms_to_attached by blast
 with Loop_false Ex.prems Loop_false Ex.IH have
  lHSe: Γ ⊢ ⟨e, s⟩ ⇒ ⟨False_c, se⟩ by simp
 from lHTe Loop_false Ex have
  lHAe: A ▷− e ≠ ⊤ using reachability_preservation_scope_false
    by (metis prod.collapse)
 from lHSe lHTe Loop_false Ex.prems have
  lHVe: Γ, A ▷− e ⊢ se √s
    by (simp add: valid_state_preservation_scope_false')
 from Loop_false Ex.prems have
  TransferFunction.loop_computation e c A ▷− e ⊢ c: Attached
    by auto
 with lHle have A ▷− e ⊢ c : Attached
   using ExpressionValidity.AT_mono conforms_to_attached Af_mono' by
metis
 with lHAe lHVe Loop_false Ex.IH have
  Γ ⊢ ⟨c, se⟩ ⇒ ⟨Exception, s'⟩ by simp
 with lHSe show ?case by simp
next
 case TestEx then show ?case by auto
next
 case ConsEx then show ?case by blast
qed

lemma big_step_unsafe_implies_safe:
 assumes
  Γ ⊢ ⟨p, s0⟩ ⇒' ⟨v, s⟩
  ⊢ p √e
  Γ ⊢ s0 √s
 shows
   Γ ⊢ ⟨p, s0⟩ ⇒ ⟨v, s⟩
using assms big_step_unsafe_implies_safe_step using topset.discI
 by (metis attachment_valid_expression_def attachment_valid_state_def
   attachment_type_valid_expression_def)
```

There is a transition for a void-safe expression from a void-safe state according to the void-unsafe semantics if and only if there is the same transition according to the void-safe one.

```
lemma big_step_unsafe_safe_eq:
```

assumes
$\vdash e \surd_e$
$\Gamma \vdash s_0 \surd_s$
shows
$\Gamma \vdash \langle e, s_0 \rangle \Rightarrow' \langle v, s \rangle \longleftrightarrow \Gamma \vdash \langle e, s_0 \rangle \Rightarrow \langle v, s \rangle$
using *big_step_unsafe_implies_safe big_step_safe_implies_unsafe*
using *assms* **by** *blast*

end

B.23 Class declaration

theory *Class* **imports** *Name* **begin**

B.23.1 *Declarations*

datatype *class_mark* = *NoMark* | *Deferred* | *Expanded*

type_synonym *'t attribute_signature* = *'t*
type_synonym *'t attribute_declaration* = *fname* × *'t*
type_synonym *'t argument_declaration* = *vname* × *'t*

type_synonym *'t routine_signature* = *'t argument_declaration list* × *'t*
— formal arguments, result type

type_synonym *'m routine_body* = *'m option*
— possible body

type_synonym (*'t*, *'m*) *routine_declaration* =
fname × *'t routine_signature* × *'m routine_body*

type_synonym (*'t*, *'m*) *class_body* =
class_mark × *cname list* × *'t attribute_declaration list* ×
(*'t*, *'m*) *routine_declaration list* × *fname list*
— class body = deferred mark, parent classes, attributes, routines, creation procedure names

type_synonym (*'t*, *'m*) *class_declaration* = *cname* × (*'t*, *'m*) *class_body*
— class declaration = class name, class body

B.23.2 *Routines*

A feature name of a given routine declaration.

definition

feature_name_from_routine:: ('t, 'm) *routine_declaration* ⇒ *fname*
where
feature_name_from_routine r = *fst r*

A routine declaration (if any) with the specified feature name in the given class body.

fun
routine_of_class ::
('t, 'm) *class_body* ⇒ *fname* ⇀ ('t, 'm) *routine_declaration*
where
routine_of_class (_, _, _, [], _) *f* = *None*
| *routine_of_class* (*d*, *p*, *a*, *r*#*rs*, *c*) *f* =
(*if feature_name_from_routine r* = *f then Some r*
else routine_of_class (*d*, *p*, *a*, *rs*, *c*) *f*)

A routine body of a given routine declaration.

definition *routine_body_from_routine_declaration* ::
('t, 'm) *routine_declaration* ⇒ 'm *routine_body*
where
routine_body_from_routine_declaration d ≡ *snd* (*snd d*)

end

B.24 System

theory *System* **imports** *Class* **begin**

datatype ('t, 'm) *system* = *System* ('t, 'm) *class_declaration list*

B.24.1 *Class properties*

Retrieve a class body (if any) by its name from a system.

primrec *class* :: ('t, 'm) *system* ⇒ *cname* ⇀ ('t, 'm) *class_body*
where
class (*System s*) = *map_of s*

B.24.1.1 *Routines*

Retrieve a routine body by the class and feature names from a system.

fun *routine_body* :: ('t, 'm) *system* ⇒ *cname* ⇒ *fname* ⇒ 'm *routine_body*
where
routine_body S c f = (*case class S c of*
None ⇒ *None* |

Some b ⇒ (case routine_of_class b f of
None ⇒ None |
Some d ⇒ routine_body_from_routine_declaration d))

end

B.25 Validity of creation procedures

theory *CreationValidity* **imports** *State System* **begin**

Abstract syntax.

datatype *('b, 't) expression =*
Value 'b value |
Current |
Local vname |
Attribute fname |
Sequence ('b, 't) expression ('b, 't) expression (_ ;; _ [80, 81] 80) |
LocalAssignment vname ('b, 't) expression (_ :=L _ [1000, 81] 81) |
AttributeAssignment fname ('b, 't) expression (_ :=A _ [1000, 81] 81) |
Creation cname fname ('b, 't) expression list
(create {_} . _ '(_') [81, 82, 0] 81) |
Call ('b, 't) expression fname ('b, 't) expression list
(_ . _ '(_') [90, 99, 0] 90) |
If ('b, 't) expression ('b, 't) expression ('b, 't) expression
(if _ then _ else _ end [80, 80, 80] 81) |
Loop ('b, 't) expression ('b, 't) expression (until _ loop _ end [80, 81] 81) |
Test 't option ('b, 't) expression vname (attached _ _ as _ [80, 81, 81] 81) |
Exception

B.25.1 *Unattached attributes*

A function that computes a set of unattached attributes for a given expression and a set of unattached attributes before it.

fun
𝒰 :: ('b, 't) expression ⇒ fname set ⇒ fname set **and**
𝒰s :: ('b, 't) expression list ⇒ fname set ⇒ fname set
where
U_Seq: *𝒰 (e₁;; e₂) V = 𝒰 e₂ (𝒰 e₁ V) |*
U_AssignLocal: *𝒰 (n :=L e) V = 𝒰 e V |*
U_AssignAttribute: *𝒰 (n :=A e) V = (𝒰 e V) − {n} |*
U_Create: *𝒰 (create {t} . n (es)) V = 𝒰s es V |*
U_QCall: *𝒰 (e . n (es)) V = 𝒰s es (𝒰 e V) |*
U_If: *𝒰 (if c then e₁ else e₂ end) V = 𝒰 e₁ (𝒰 c V) ∪ 𝒰 e₂ (𝒰 c V) |*
U_Loop: *𝒰 (until c loop b end) V = 𝒰 c V |*

U_Test: $\mathcal{U}$ (*attached t e as n*) $V = \mathcal{U}\ e\ V$ |
U_Exception: $\mathcal{U}$ *Exception* $V = \{\}$ |
U_Other: $\mathcal{U}$ _ $V = V$ |
U_Nil: $\mathcal{U}s$ [] $V = V$ |
U_Cons: $\mathcal{U}s$ (*e* # *es*) $V = \mathcal{U}s\ es\ (\mathcal{U}\ e\ V)$

abbreviation $\mathcal{U}$*_rep* **where** $\mathcal{U}$*_rep A e* $\equiv \mathcal{U}\ e\ A$
notation (**output**) $\mathcal{U}$*_rep* (**infixl** $\gg$ 72)
abbreviation $\mathcal{U}s$*_rep* (**infixl** $\gg$ 72) **where** $A \gg es \equiv \mathcal{U}s\ es\ A$

The function that computes a set of unattached attributes is monotone.

lemma *unattached_mono*:
fixes
e:: ('*b*, '*t*) *expression* **and** *es*:: ('*b*, '*t*) *expression list*
shows
mono ($\mathcal{U}\ e$) **and** *mono* ($\mathcal{U}s\ es$)
by (*induction rule*: $\mathcal{U}$_$\mathcal{U}s$.*induct*)
(*auto simp add*: *mono_def sup.coboundedI1 sup.coboundedI2*)

lemma *unattached_mono'*:
fixes
e:: ('*b*, '*t*) *expression* **and** *es*:: ('*b*, '*t*) *expression list*
assumes
$A \subseteq B$
shows
$\mathcal{U}\ e\ A \subseteq \mathcal{U}\ e\ B$ **and** $\mathcal{U}s\ es\ A \subseteq \mathcal{U}s\ es\ B$
by (*simp_all add*: *assms monoD unattached_mono*)

A set of unattached attributes after an expression is not greater than before it.

lemma *unattached_decrease*:
fixes
e:: ('*b*, '*t*) *expression* **and** *es*:: ('*b*, '*t*) *expression list*
shows
$\mathcal{U}\ e\ A = B \Longrightarrow B \subseteq A$ **and** $\mathcal{U}s\ es\ A = B \Longrightarrow B \subseteq A$
by (*induction arbitrary*: *B* **and** *B rule*: $\mathcal{U}$_$\mathcal{U}s$.*induct*) *fastforce*+

If a set of unattached attributes is not empty after an expression, it is not empty before it.

lemma *unattached_non_empty*:
fixes
e:: ('*b*, '*t*) *expression* **and** *es*:: ('*b*, '*t*) *expression list*
shows
$\mathcal{U}\ e\ V \neq \{\} \Longrightarrow V \neq \{\}$ **and** $\mathcal{U}s\ es\ V \neq \{\} \Longrightarrow V \neq \{\}$
using *unattached_decrease* **by** *fastforce*+

B.25.2 *Validity rule: simple attribute access safety*

type_synonym $('t, 'b)$ *expression_system* $= ('t, ('b, 't)$ *expression*$)$ *system*

A predicate that tells if an expression is valid with respect to a set of unattached attributes (simple strong version).

inductive
$cV :: fname\ set \Rightarrow ('b, 't)\ expression \Rightarrow bool\ (\_ \vdash \_\ \surd_c\ '\ [60, 60]\ 60)$ **and**
$cVs :: fname\ set \Rightarrow ('b, 't)\ expression\ list \Rightarrow bool\ (\_ \vdash \_\ [\surd_c]\ '\ [60, 60]\ 60)$
where
cV_Value: $V \vdash Value\ v\ \surd_c\ '\ |$
$cV\_Current$: $V = \{\} \Longrightarrow V \vdash Current\ \surd_c\ '\ |$
cV_Local: $V \vdash Local\ n\ \surd_c\ '\ |$
$cV\_Attribute$: $\neg\ n \in V \Longrightarrow V \vdash Attribute\ n\ \surd_c\ '\ |$
cV_Seq: $[\![V \vdash e_1\ \surd_c\ ';\ \mathcal{U}\ e_1\ V \vdash e_2\ \surd_c\ ']\!] \Longrightarrow V \vdash e_1\ ;;\ e_2\ \surd_c\ '\ |$
$cV\_AssignLocal$: $V \vdash e\ \surd_c\ ' \Longrightarrow V \vdash n :=_L e\ \surd_c\ '\ |$
$cV\_AssignAttribute$: $V \vdash e\ \surd_c\ ' \Longrightarrow V \vdash n :=_A e\ \surd_c\ '\ |$
cV_Create: $V \vdash es\ [\surd_c]\ ' \Longrightarrow V \vdash create\ \{t\}\ .\ n\ (es)\ \surd_c\ '\ |$
cV_QCall: $[\![V \vdash e\ \surd_c\ ';\ \mathcal{U}\ e\ V \vdash es\ [\surd_c]\ ']\!] \Longrightarrow V \vdash e\ .\ n\ (es)\ \surd_c\ '\ |$
cV_If: $[\![V \vdash c\ \surd_c\ ';\ \mathcal{U}\ c\ V \vdash e_1\ \surd_c\ ';\ \mathcal{U}\ c\ V \vdash e_2\ \surd_c\ ']\!] \Longrightarrow$
$V \vdash if\ c\ then\ e_1\ else\ e_2\ end\ \surd_c\ '\ |$
cV_Loop: $[\![V \vdash e\ \surd_c\ ';\ \mathcal{U}\ e\ V \vdash b\ \surd_c\ ']\!] \Longrightarrow V \vdash until\ e\ loop\ b\ end\ \surd_c\ '\ |$
cV_Test: $V \vdash e\ \surd_c\ ' \Longrightarrow V \vdash attached\ t\ e\ as\ n\ \surd_c\ '\ |$
$cV\_Exception$: $V \vdash Exception\ \surd_c\ '\ |$
cV_Nil: $V \vdash []\ [\surd_c]\ '\ |$
cV_Cons: $[\![V \vdash e\ \surd_c\ ';\ \mathcal{U}\ e\ V \vdash es\ [\surd_c]\ ']\!] \Longrightarrow V \vdash e\ \#\ es\ [\surd_c]\ '$

declare $cV\_cVs.intros$[*intro*!]

inductive_simps cVs_iffs [*iff*]:
$V \vdash []\ [\surd_c]\ '$
$V \vdash e\ \#\ es\ [\surd_c]\ '$

inductive_cases cV_ValueE[*elim*!]: $V \vdash Value\ v\ \surd_c\ '$
inductive_cases $cV\_CurrentE$[*elim*!]: $V \vdash Current\ \surd_c\ '$
inductive_cases cV_LocalE[*elim*!]: $V \vdash Local\ n\ \surd_c\ '$
inductive_cases $cV\_AttributeE$[*elim*!]: $V \vdash Attribute\ n\ \surd_c\ '$
inductive_cases cV_SeqE[*elim*!]: $V \vdash e_1\ ;;\ e_2\ \surd_c\ '$
inductive_cases $cV\_AssignLocalE$[*elim*!]: $V \vdash n :=_L e\ \surd_c\ '$
inductive_cases $cV\_AssignAttributeE$[*elim*!]: $V \vdash n :=_A e\ \surd_c\ '$
inductive_cases $cV\_CreateE$[*elim*!]: $V \vdash create\ \{t\}\ .\ n\ (es)\ \surd_c\ '$
inductive_cases cV_QCallE[*elim*!]: $V \vdash e\ .\ n\ (es)\ \surd_c\ '$
inductive_cases cV_IfE[*elim*!]: $V \vdash if\ c\ then\ e_1\ else\ e_2\ end\ \surd_c\ '$
inductive_cases cV_LoopE[*elim*!]: $V \vdash until\ e\ loop\ b\ end\ \surd_c\ '$
inductive_cases cV_TestE[*elim*!]: $V \vdash attached\ t\ e\ as\ n\ \surd_c\ '$
inductive_cases $cV\_ExceptionE$[*elim*!]: $V \vdash Exception\ \surd_c\ '$

B.25.3 *Access to current*

A function that tells if an expression accesses a current object.

fun
$\mathcal{C}$ :: ('b, 't) *expression* ⇒ *bool* (*has*'*_current*) **and**
$\mathcal{C}s$ :: ('b, 't) *expression list* ⇒ *bool* (*has*'*_current*)
where
C_Current: $\mathcal{C}$ *Current* ⟷ *True* |
C_Seq: $\mathcal{C}$ (e_1;; e_2) ⟷ $\mathcal{C}$ e_1 ∨ $\mathcal{C}$ e_2 |
C_AssignLocal: $\mathcal{C}$ (n :=$_L$ e) ⟷ $\mathcal{C}$ e |
C_AssignAttr: $\mathcal{C}$ (n :=$_A$ e) ⟷ $\mathcal{C}$ e |
C_Create: $\mathcal{C}$ (*create* {t} . n (es)) ⟷ $\mathcal{C}s$ es |
C_QCall: $\mathcal{C}$ (e . n (es)) ⟷ $\mathcal{C}$ e ∨ $\mathcal{C}s$ es |
C_If: $\mathcal{C}$ (*if* c *then* e_1 *else* e_2 *end*) ⟷ $\mathcal{C}$ c ∨ $\mathcal{C}$ e_1 ∨ $\mathcal{C}$ e_2|
C_Loop: $\mathcal{C}$ (*until* e *loop* b *end*) ⟷ $\mathcal{C}$ e ∨ $\mathcal{C}$ b|
C_Test: $\mathcal{C}$ (*attached* t e *as* n) ⟷ $\mathcal{C}$ e|
C_Other: $\mathcal{C}$ _ ⟷ *False* |
C_ArgNil: $\mathcal{C}s$ [] ⟷ *False* |
C_ArgCons: $\mathcal{C}s$ (e # es) ⟷ $\mathcal{C}$ e ∨ $\mathcal{C}s$ es

B.25.4 *Presence of qualified calls*

A set of creation procedures referenced by a given expression.

fun
creation_set :: ('b, 't) *expression* ⇒ (*cname* × *fname*) *set* ($\mathcal{S}$) **and**
creation_set_s :: ('b, 't) *expression list* ⇒ (*cname* × *fname*) *set* ($\mathcal{S}s$)
where
S_Seq: *creation_set* (e_1;; e_2) = *creation_set* e_1 ∪ *creation_set* e_2 |
S_AssignLocal: *creation_set* (n :=$_L$ e) = *creation_set* e |
S_AssignAttr: *creation_set* (n :=$_A$ e) = *creation_set* e |
S_Create: *creation_set* (*create* {c} . n (es)) = {(c, n)} ∪ *creation_set_s* es |
S_QCall: *creation_set* (e . n (es)) = *creation_set* e ∪ *creation_set_s* es |
S_If: *creation_set* (*if* c *then* e_1 *else* e_2 *end*) =
creation_set c ∪ *creation_set* e_1 ∪ *creation_set* e_2 |
S_Loop: *creation_set* (*until* e *loop* b *end*) = *creation_set* e ∪ *creation_set* b |
S_Test: *creation_set* (*attached* t e *as* n) = *creation_set* e |
S_Other: *creation_set* _ = {} |
S_Nil: *creation_set_s* [] = {} |
S_Cons: *creation_set_s* (e # es) = *creation_set* e ∪ *creation_set_s* es

A set of creation procedures reachable from a routine specified by class and feature names.

fun

creation_reachable$_1$ :: ('*t*, '*b*) *expression_system* ⇒ *cname* × *fname* ⇒ (*cname* × *fname*) *set*
where
creation_reachable$_1$ *S* (*c*, *f*) = (*case routine_body S c f of*
None ⇒ {} |
Some b ⇒ *creation_set b*)

fun
creation_reachable ::
('*t*, '*b*) *expression_system* ⇒ *cname* × *fname* ⇒ (*cname* × *fname*) *set*
where
creation_reachable S (*c*, *f*) =
lfp (λ *x*. {(*c*, *f*)} ∪ *x* ∪ (⋃ *y* ∈ *x*. *creation_reachable*$_1$ *S y*))

A predicate that tells if a creation procedure is reachable from a given routine.

inductive
creation_reachable_1' ::
('*t*, '*b*) *expression_system* ⇒ *cname* × *fname* ⇒ *cname* × *fname* ⇒ *bool*
for *S* :: ('*t*, '*b*) *expression_system*
where
⟦*routine_body S c f* = *Some b*; *g* ∈ *creation_set b*⟧ ⟹
creation_reachable_1' S (*c*, *f*) *g*

lemma *creation_reachable_def* [*iff*]:
(*c*, *f*) ∈ *creation_reachable*$_1$ *S* (c_0, f_0) =
creation_reachable_1' S (c_0, f_0) (*c*, *f*)
by (*cases routine_body S* c_0 f_0) (*simp_all add*: *creation_reachable_1'.simps*)

fun
creation_reachable' ::
('*t*, '*b*) *expression_system* ⇒ *cname* × *fname* ⇒ *cname* × *fname* ⇒ *bool*
where
creation_reachable' S (*c*, *f*) = (*creation_reachable_1' S*)** (*c*, *f*)

Ω tells if there are immediate qualified calls in an expression.

fun
Ω :: ('*b*, '*t*) *expression* ⇒ *bool* **and**
Ω*s* :: ('*b*, '*t*) *expression list* ⇒ *bool*
where
Q_Seq: Ω (*e*$_1$;; *e*$_2$) ⟷ Ω *e*$_1$ ∨ Ω *e*$_2$ |
Q_AssignLocal: Ω (*n* :=$_L$ *e*) ⟷ Ω *e* |
Q_AssignAttribute: Ω (*n* :=$_A$ *e*) ⟷ Ω *e* |
Q_Create: Ω (*create* {*t*} . *n* (*es*)) ⟷ Ω*s es* |
Q_QCall: Ω (*e* . *n* (*es*)) ⟷ *True* |

Q_If: $\mathcal{Q}$ (*if c then* e_1 *else* e_2 *end*) $\longleftrightarrow$ $\mathcal{Q}$ *c* $\vee$ $\mathcal{Q}$ e_1 $\vee$ $\mathcal{Q}$ e_2 |
Q_Loop: $\mathcal{Q}$ (*until e loop b end*) $\longleftrightarrow$ $\mathcal{Q}$ *e* $\vee$ $\mathcal{Q}$ *b* |
Q_Test: $\mathcal{Q}$ (*attached t e as n*) $\longleftrightarrow$ $\mathcal{Q}$ *e* |
Q_Other: $\mathcal{Q}$ _ $\longleftrightarrow$ *False* |
Q_Nil: $\mathcal{Q}s$ [] $\longleftrightarrow$ *False* |
Q_Cons: $\mathcal{Q}s$ (*e # es*) $\longleftrightarrow$ $\mathcal{Q}$ *e* $\vee$ $\mathcal{Q}s$ *es*

Are there qualified feature calls in a given routine?

primrec *has_immediate_qualified_in_routine* ::
(*'t*, *'b*) *expression_system* $\Rightarrow$ *cname* $\times$ *fname* $\Rightarrow$ *bool*
where
has_immediate_qualified_in_routine S (*c*, *f*) $\longleftrightarrow$ (*case routine_body S c f of*
None $\Rightarrow$ *False* | (*Some b*) $\Rightarrow$ $\mathcal{Q}$ *b*)

Are there qualified feature calls in any routine reachable from a given creation procedure?

definition *has_qualified* ::
(*'t*, *'b*) *expression_system* $\Rightarrow$ *cname* $\times$ *fname* $\Rightarrow$ *bool*
where
has_qualified S c $\longleftrightarrow$
($\exists$ *x* $\in$ *creation_reachable S c*. *has_immediate_qualified_in_routine S x*)

The following function tells if Current is not used when not all attributes are set.

fun
$\mathcal{V}$ :: (*'b*, *'t*) *expression* $\Rightarrow$ *fname set* $\Rightarrow$ *bool* (*safe*) **and**
$\mathcal{V}s$ :: (*'b*, *'t*) *expression list* $\Rightarrow$ *fname set* $\Rightarrow$ *bool* (*safe*)
where
V_Current: $\mathcal{V}$ *Current V* $\longleftrightarrow$ *V* = {} |
V_Seq: $\mathcal{V}$ (e_1 ;; e_2) *V* $\longleftrightarrow$ $\mathcal{V}$ e_1 *V* $\wedge$ $\mathcal{V}$ e_2 ($\mathcal{U}$ e_1 *V*) $\vee$ $\mathcal{U}$ (e_1;;e_2) *V* = {} |
V_AssignLocal: $\mathcal{V}$ (*n* :=$_L$ *e*) *V* $\longleftrightarrow$ $\mathcal{V}$ *e V* |
V_AssignAttribute: $\mathcal{V}$ (*n* :=$_A$ *e*) *V* $\longleftrightarrow$ $\mathcal{V}$ *e V* $\vee$ $\mathcal{U}$ (*n* :=$_A$ *e*) *V* = {} |
V_Create: $\mathcal{V}$ (*create* {*t*} . *n* (*es*)) *V* $\longleftrightarrow$ $\mathcal{V}s$ *es V* |
V_QCall: $\mathcal{V}$ (*e* . *n* (*es*)) *V* $\longleftrightarrow$ $\mathcal{V}s$ (*e # es*) *V* |
V_If: $\mathcal{V}$ (*if c then* e_1 *else* e_2 *end*) *V* $\longleftrightarrow$
($\mathcal{V}$ *c V* $\wedge$ $\mathcal{V}$ e_1 ($\mathcal{U}$ *c V*) $\vee$ $\mathcal{U}s$ [*c*, e_1] *V* = {}) $\wedge$
($\mathcal{V}$ *c V* $\wedge$ $\mathcal{V}$ e_2 ($\mathcal{U}$ *c V*) $\vee$ $\mathcal{U}s$ [*c*, e_2] *V* = {}) |
V_Loop: $\mathcal{V}$ (*until e loop b end*) *V* $\longleftrightarrow$
($\mathcal{V}$ *e V* $\wedge$ $\mathcal{V}$ *b* ($\mathcal{U}$ *e V*)) $\vee$ $\mathcal{U}s$ [*e*, *b*] *V* = {} |
V_Test: $\mathcal{V}$ (*attached t e as n*) *V* $\longleftrightarrow$ $\mathcal{V}$ *e V* |
V_Other: $\mathcal{V}$ _ *V* $\longleftrightarrow$ *True* |
V_Nil: $\mathcal{V}s$ [] *V* $\longleftrightarrow$ *True* |
V_Cons: $\mathcal{V}s$ (*e # es*) *V* $\longleftrightarrow$ $\mathcal{V}$ *e V* $\wedge$ $\mathcal{V}s$ *es* ($\mathcal{U}$ *e V*) $\vee$ $\mathcal{U}s$ (*e # es*) *V* = {}

lemma *empty_implies_safe*:

fixes
e :: ('*b*, '*t*) *expression* **and** *es* :: ('*b*, '*t*) *expression list*
shows
$\mathcal{U}\ e\ V = \{\} \Longrightarrow \mathcal{V}\ e\ V$ **and**
$\mathcal{U}s\ es\ V = \{\} \Longrightarrow \mathcal{V}s\ es\ V$
using *unattached_non_empty* **by** (*induction rule*: $\mathcal{U}$_$\mathcal{U}s$.*induct*) *auto*

lemma *no_current_implies_safe*:
fixes
e :: ('*b*, '*t*) *expression* **and** *es* :: ('*b*, '*t*) *expression list*
shows
$\neg\ \mathcal{C}\ e \Longrightarrow \mathcal{V}\ e\ V$ **and**
$\neg\ \mathcal{C}s\ es \Longrightarrow \mathcal{V}s\ es\ V$
by (*induction arbitrary*: *V* **and** *V rule*: $\mathcal{C}$_$\mathcal{C}s$.*induct*) *simp_all*

lemma *cV_implies_safe*:
fixes
e :: ('*b*, '*t*) *expression* **and** *es* :: ('*b*, '*t*) *expression list*
shows
$V \vdash e\ \surd_c\ ' \Longrightarrow \mathcal{V}\ e\ V$ **and**
$V \vdash es\ [\surd_c]\ ' \Longrightarrow \mathcal{V}s\ es\ V$
by (*induction rule*: *cV_cVs*.*inducts*) *simp_all*

lemma *safe_mono'*:
fixes
e:: ('*b*, '*t*) *expression* **and** *es*:: ('*b*, '*t*) *expression list*
assumes
$A \leqslant B$
shows
$\mathcal{V}\ e\ B \Longrightarrow \mathcal{V}\ e\ A$ **and** $\mathcal{V}s\ es\ B \Longrightarrow \mathcal{V}s\ es\ A$
using *assms*
apply (*induction arbitrary*: *A* **and** *A rule*: $\mathcal{V}$_$\mathcal{V}s$.*induct*)
apply *auto*[1]
using *unattached_mono'*(1)
apply (*metis CreationValidity.V_Seq bot.extremum_uniqueI*)
using *CreationValidity.V_AssignLocal* **apply** *blast*
apply (*metis CreationValidity.V_AssignAttribute*
bot.extremum_uniqueI unattached_mono'(1))
using *CreationValidity.V_Create* **apply** *blast*
using *CreationValidity.V_QCall* **apply** *blast*
apply (*smt CreationValidity.V_If bot.extremum_uniqueI*
unattached_mono'(1) *unattached_mono'*(2))
apply (*metis CreationValidity.V_Loop bot.extremum_uniqueI*
unattached_mono'(1) *unattached_mono'*(2))
using *CreationValidity.V_Test* **apply** *blast*
apply *simp*

apply *simp*
apply *simp*
apply *simp*
apply *simp*
by (*metis CreationValidity.V_Cons bot.extremum_uniqueI*
unattached_mono'(1) *unattached_mono'*(2))

lemma *safe_mono*:
fixes
e:: ('*b*, '*t*) *expression* **and** *es*:: ('*b*, '*t*) *expression list*
shows
antimono (*V e*) **and** *antimono* (*Vs es*)
by (*simp_all add*: *antimonoI safe_mono'*)

B.25.5 *Validity rule: circular references*

A predicate that tells if an expression is valid with respect to a set of unattached attributes (advanced weaker version).

inductive
CV :: ('*t*, '*b*) *expression_system* ⇒ *fname set* ⇒ ('*b*, '*t*) *expression* ⇒ *bool*
(_, _ ⊢ _ $\surd_c$ [60, 54, 60] 60) **and**
CVs ::
('*t*, '*b*) *expression_system* ⇒ *fname set* ⇒ ('*b*, '*t*) *expression list* ⇒ *bool*
(_, _ ⊢ _ [$\surd_c$] [60, 54, 60] 60)
for *S*
where
CV_Value: *S*, *V* ⊢ *Value v* $\surd_c$ |
CV_Current: *S*, *V* ⊢ *Current* $\surd_c$ |
CV_Local: *S*, *V* ⊢ *Local n* $\surd_c$ |
CV_Attribute: ¬ *n* ∈ *V* ⟹ *S*, *V* ⊢ *Attribute n* $\surd_c$ |
CV_Seq: ⟦*S*, *V* ⊢ e_1 $\surd_c$; *S*, *U* e_1 *V* ⊢ e_2 $\surd_c$⟧ ⟹ *S*, *V* ⊢ e_1 ;; e_2 $\surd_c$ |
CV_AssignLocal: *S*, *V* ⊢ *e* $\surd_c$ ⟹ *S*, *V* ⊢ *n* :=$_L$ *e* $\surd_c$ |
CV_AssignAttribute: *S*, *V* ⊢ *e* $\surd_c$ ⟹ *S*, *V* ⊢ *n* :=$_A$ *e* $\surd_c$ |
CV_Create: ⟦*S*, *V* ⊢ *es* [$\surd_c$]; *Vs es V* ∨ ¬ *has_qualified S* (*c*, *n*)⟧ ⟹
S, *V* ⊢ *create* {*c*} . *n* (*es*) $\surd_c$ |
CV_QCall: ⟦*S*, *V* ⊢ *e* $\surd_c$; *S*, *U e V* ⊢ *es* [$\surd_c$]; *Vs* (*e* # *es*) *V*⟧ ⟹
S, *V* ⊢ *e* . *n* (*es*) $\surd_c$ |
CV_If: ⟦*S*, *V* ⊢ *c* $\surd_c$; *S*, *U c V* ⊢ e_1 $\surd_c$; *S*, *U c V* ⊢ e_2 $\surd_c$⟧ ⟹
S, *V* ⊢ *if c then* e_1 *else* e_2 *end* $\surd_c$ |
CV_Loop: ⟦*S*, *V* ⊢ *e* $\surd_c$; *S*, *U e V* ⊢ *b* $\surd_c$⟧ ⟹ *S*, *V* ⊢ *until e loop b end* $\surd_c$ |
CV_Test: *S*, *V* ⊢ *e* $\surd_c$ ⟹ *S*, *V* ⊢ *attached t e as n* $\surd_c$ |
CV_Exception: *S*, *V* ⊢ *Exception* $\surd_c$ |
CV_Nil: *S*, *V* ⊢ [] [$\surd_c$] |
CV_Cons: ⟦*S*, *V* ⊢ *e* $\surd_c$; *S*, *U e V* ⊢ *es* [$\surd_c$]⟧ ⟹ *S*, *V* ⊢ *e* # *es* [$\surd_c$]

declare *CV_CVs.intros*[*intro*!]

inductive_simps *CVs_iffs* [*iff*]:
$S, V \vdash [] \ [\surd_c]$
$S, V \vdash e \# es \ [\surd_c]$

inductive_cases *CV_ValueE*[*elim*!]: $S, V \vdash Value\ v\ \surd_c$
inductive_cases *CV_CurrentE*[*elim*!]: $S, V \vdash Current\ \surd_c$
inductive_cases *CV_LocalE*[*elim*!]: $S, V \vdash Local\ n\ \surd_c$
inductive_cases *CV_AttributeE*[*elim*!]: $S, V \vdash Attribute\ n\ \surd_c$
inductive_cases *CV_SeqE*[*elim*!]: $S, V \vdash e_1\ ;;\ e_2\ \surd_c$
inductive_cases *CV_AssignLocalE*[*elim*!]: $S, V \vdash n :=_L e\ \surd_c$
inductive_cases *CV_AssignAttributeE*[*elim*!]: $S, V \vdash n :=_A e\ \surd_c$
inductive_cases *CV_CreateE*[*elim*!]: $S, V \vdash create\ \{t\}\ .\ n\ (es)\ \surd_c$
inductive_cases *CV_QCallE*[*elim*!]: $S, V \vdash e\ .\ n\ (es)\ \surd_c$
inductive_cases *CV_IfE*[*elim*!]: $S, V \vdash if\ c\ then\ e_1\ else\ e_2\ end\ \surd_c$
inductive_cases *CV_LoopE*[*elim*!]: $S, V \vdash until\ e\ loop\ b\ end\ \surd_c$
inductive_cases *CV_TestE*[*elim*!]: $S, V \vdash attached\ t\ e\ as\ n\ \surd_c$
inductive_cases *CV_ExceptionE*[*elim*!]: $S, V \vdash Exception\ \surd_c$

The stronger validity implies the weaker one.

lemma *cV_implies_CV*:
fixes
$e :: ('b, 't)$ *expression* **and** $es :: ('b, 't)$ *expression list*
shows
$V \vdash e\ \surd_c' \Longrightarrow S, V \vdash e\ \surd_c$ **and**
$V \vdash es\ [\surd_c]' \Longrightarrow S, V \vdash es\ [\surd_c]$
by (*induction e* **and** *es rule*: *cV_cVs.inducts*)
(*auto simp add*: *cV_implies_safe*)

The initialization validity predicate is monotone.

lemma *CV_mono*:
assumes
$A \leqslant B$
shows
$S, B \vdash e\ \surd_c \Longrightarrow S, A \vdash e\ \surd_c$ **and**
$S, B \vdash es\ [\surd_c] \Longrightarrow S, A \vdash es\ [\surd_c]$
using *assms*
proof (*induction arbitrary*: *A* **and** *A rule*: *CV_CVs.inducts*)
case *CV_QCall*
with *unattached_mono'*(1) *safe_mono'*(2) **show** *?case*
using *CV_CVs.CV_QCall* **by** *metis*
next
case *CV_Create*
with *safe_mono'*(2) **show** *?case* **using** *CV_CVs.CV_Create* **by** *blast*
qed (*fastforce simp add*: *unattached_mono'*)+

end

B.26 Formal generic conformance

theory *GenericConformance* **imports** *Main* **begin**

datatype *type* =
Expanded |
AttachedReference |
DetachableReference

fun *conforms* :: *type* ⇒ *type* ⇒ *bool* (**infixl** ⩽ 60) **where**
conforms Expanded _ = True
| *conforms AttachedReference AttachedReference = True*
| *conforms AttachedReference DetachableReference = True*
| *conforms DetachableReference DetachableReference = True*
| *conforms _ _ = False*

datatype *mark* =
EmptyMark |
Attached |
Detachable

fun *unmark* :: *type* ⇒ *mark* ⇒ *type* ('(_, _') 70) **where**
unmark Expanded _ = Expanded
| *unmark AttachedReference Detachable = DetachableReference*
| *unmark AttachedReference _ = AttachedReference*
| *unmark DetachableReference Attached = AttachedReference*
| *unmark DetachableReference _ = DetachableReference*

fun *is_attached* **where**
is_attached DetachableReference = False
| *is_attached _ = True*

lemma (⋀ *T*. *is_attached* (*unmark T m*)) ⟹ *m = Attached*
by (*metis is_attached.elims*(2) *mark.exhaust type.distinct*(4)
type.distinct(5) *unmark.simps*(2) *unmark.simps*(6))

lemma *generic_is_attached*:
(∀ *A*. *conforms A C* ⟶ *is_attached* (*unmark A m*)) ⟷
C = Expanded ∨
C = AttachedReference ∧ *m* ≠ *Detachable* ∨
m = Attached
using *conforms.simps*

apply (*cases C*)
apply (*fastforce elim*: *conforms.elims*)
apply (*cases m*)
apply (*fastforce elim*: *conforms.elims*)
apply (*fastforce elim*: *conforms.elims*)
apply *fastforce*
apply (*cases m*)
apply *fastforce*
apply (*fastforce elim*: *conforms.elims*)
by *fastforce*

no_notation *less_eq* ((_/ ⩽ _) [51, 51] 50)

lemma *generic_is_attached_n*:
Cs ≠ [] ⟹
(∀ *A*. (∀ *C*. *C* ∈ *set Cs* ⟶ *A* ⩽ *C*) ⟶ *is_attached* (*unmark A m*)) ⟷
(∃ *C*. *C* ∈ *set Cs* ∧ *C* = *Expanded*) ∨
(∃ *C*. *C* ∈ *set Cs* ∧ *C* = *AttachedReference* ∧ *m* ≠ *Detachable*) ∨
m = *Attached*
apply (*cases* (∃ *C*. *C* ∈ *set Cs* ∧
(*C* = *Expanded* ∨
C = *AttachedReference* ∧ *m* ≠ *Detachable* ∨
m = *Attached*)))
using *generic_is_attached* **apply** *auto*[1]
by (*smt conforms.elims*(3) *is_attached.simps*(1) *last_in_set*
type.distinct(5) *unmark.elims unmark.simps*(2))

fun *self_initializing* **where**
self_initializing AttachedReference = *False*
| *self_initializing* _ = *True*

lemma *generic_self_initializing*:
(∀ *A*. *conforms A C* ⟶ *self_initializing* (*unmark A m*)) ⟷
m = *Detachable* ∨ *C* = *Expanded*
using *conforms.simps* **by** (*cases C*, *fastforce elim*: *conforms.elims*)
(*cases m*, *fastforce*+, (*fastforce elim*: *conforms.elims*))+

lemma *generic_self_initializing_n*:
Cs ≠ [] ⟹ (∀ *A*. (∀ *C*. *C* ∈ *set Cs* ⟶ *conforms A C*) ⟶
self_initializing (*unmark A m*)) ⟷
(∃ *C*. *C* ∈ *set Cs* ∧ *C* = *Expanded*) ∨ *m* = *Detachable*
apply (*cases m* = *Detachable*)
using *generic_self_initializing last_in_set* **apply** *blast*
by (*smt conforms.elims*(3) *conforms.simps*(5) *conforms.simps*(6)
self_initializing.elims(2) *self_initializing.elims*(3)
type.distinct(1) *type.distinct*(5) *unmark.elims*)

fun *generic_conforms* :: *mark* ⇒ *mark* ⇒ *bool* (**infixl** $\leqslant_G$ *40*) **where**
generic_conforms _ Detachable = *True*
| *generic_conforms Attached _* = *True*
| *generic_conforms EmptyMark EmptyMark* = *True*
| *generic_conforms _ _* = *False*

lemma ⟦*generic_conforms Sm Tm*; *is_attached* (*unmark T Tm*)⟧ ⟹
is_attached (*unmark T Sm*)
by (*cases T*, *auto*, *cases Tm*, *auto*, *cases Sm*, *auto*, *cases Sm*, *auto*,
cases Tm, *auto*, *cases Sm*, *auto*)

lemma ⟦*generic_conforms Sm Tm*; ¬ *self_initializing* (*unmark T Tm*)⟧ ⟹
¬ *self_initializing* (*unmark T Sm*)
by (*cases T*, *auto*, *cases Tm*, *auto*, *cases Sm*, *auto*,
cases Sm, *auto*, *cases Tm*, *auto*, *cases Sm*, *auto*)

lemma *generic_conforms_is_correct*:
generic_conforms m_S m_T ⟹ *conforms* (*unmark T* m_S) (*unmark T* m_T)
by (*cases T*, *simp*, (*cases* m_T, (*cases* m_S, *simp*, *simp*, *simp*)+)+)

no_notation *conforms* (**infixl** ⩽ *60*)
notation *less_eq* ((_/ ⩽ _) [*51*, *51*] *50*)

lemma *generic_conforms_is_maximal*:
(⋀ *T* m_T m_S. ⟦
f m_S m_T;
is_attached (*unmark T* m_T);
¬ *self_initializing* (*unmark T* m_T)
⟧ ⟹
is_attached (*unmark T* m_S) ∧ ¬ *self_initializing* (*unmark T* m_S)) ⟹
f m_S m_T ⩽ *generic_conforms* m_S m_T
by (*metis generic_conforms.elims*(*3*) *is_attached.simps*(*1*)
is_attached.simps(*3*) *le_boolI unmark.simps*(*2*) *unmark.simps*(*3*)
unmark.simps(*5*) *unmark.simps*(*6*) *unmark.simps*(*7*)
self_initializing.simps(*1*))

notation *conforms* (**infixl** ⩽ *60*)

end

Index

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[2] Nada Amin and Tiark Rompf. "Type Soundness Proofs with Definitional Interpreters." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 666–679. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009866.

[3] Davide Ancona and Elena Zucca. "Corecursive Featherweight Java." In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. FTfJP '12. Beijing, China: ACM, 2012, pp. 3–10. ISBN: 978-1-4503-1272-1. DOI: 10.1145/2318202.2318205.

[4] Apple Inc. *Programming with Objective-C*. Sept. 17, 2014. URL: https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/ (visited on 2016-05-15).

[5] Karine Arnout and Bertran Meyer. "Finding Implicit Contracts in .NET Components." In: *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5–8, 2002, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 285–318. ISBN: 978-3-540-39656-7. DOI: 10.1007/978-3-540-39656-7_12.

[6] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. "Specification and Verification: The Spec# Experience." In: *Commun. ACM* 54.6 (June 2011). Ed. by Moshe Y. Vardi, pp. 81–91. ISSN: 0001-0782. DOI: 10.1145/1953122.1953145.

[7] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Fifth Edition of a Recommendation. http://www.w3.org/TR/2008/REC-xml-20081126/. W3C, Nov. 2008.

[8] *Common Vulnerabilities and Exposures*. 2017. URL: http://cve.mitre.org/ (visited on 2017-04-27).

[9] *Common Weakness Enumeration*. 2017. URL: https://cwe.mitre.org/ (visited on 2017-01-15).

[10] *Computer emergency response teams*. 2017. URL: http://www.cert.org/ (visited on 2017-01-15).

[11] *Creating a new void-safe project*. Community portal for Eiffel, 2016. URL: https://www.eiffel.org/doc/eiffel/Creating%20a%20new%20void-safe%20project (visited on 2016-12-30).

[12] Ecma International. *ECMA-367: Eiffel analysis, design and programming language*. 2nd. (Alexander V. Kogtenkov is a member of the committee.) Geneva, Switzerland: Ecma International, June 2006. URL: http://www.ecma-international.org/publications/standards/Ecma-367.htm.

[13] Ecma International. *ECMA-262: ECMAScript Language Specification*. 6.0. Geneva, Switzerland: Ecma International, June 2015.

[14] *Eiffel compatibility options*. Community portal for Eiffel, 2016. URL: https://www.eiffel.org/doc/eiffelstudio/Eiffel%20compatibility%20options (visited on 2016-12-30).

[15] Eiffel Software. *EiffelStudio 16.05 Releases*. July 2016. URL: https://dev.eiffel.com/EiffelStudio_16.05_Releases (visited on 2016-09-15).

[16] Eiffel Software. *EiffelStudio 16.11 Releases*. Nov. 2016. URL: https://dev.eiffel.com/EiffelStudio_16.11_Releases (visited on 2016-11-15).

[17] Manuel Fähndrich and K. Rustan M. Leino. "Declaring and Checking Non-null Types in an Object-oriented Language." In: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*. OOPSLA '03. Anaheim, California, USA: ACM, 2003, pp. 302–312. ISBN: 1-58113-712-5. DOI: 10.1145/949305.949332.

[18] Manuel Fähndrich and Songtao Xia. "Establishing Object Invariants with Delayed Types." In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 337–350. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297052.

[19] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended Static Checking for Java." In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI '02. Berlin, Germany: ACM, 2002, pp. 234–245. ISBN: 1-58113-463-0. DOI: 10.1145/512529.512558.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[21] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 9780133900699.

[22] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Maintenance Release. Oracle America, Inc. and/or its affiliates., Mar. 2015.

[23] PHP Documentation Group. *PHP Manual*. Ed. by Peter Cowburn. May 14, 2016. URL: http://php.net/manual/ (visited on 2016-05-15).

[24] Tony Hoare. "Null references: The billion dollar mistake." In: *Presentation at QCon London* (2009).

[25] *Projects – Isabelle Community wiki*. Apr. 2016. URL: https://isabelle.in.tum.de/community/Projects (visited on 2016-09-15).

[26] ISO. *ISO/IEC 23270:2006(E): Information technology — Programming languages — C#*. 2nd. Geneva, Switzerland: International Organization for Standardization, Sept. 1, 2006.

[27] ISO. *ISO/IEC 25436:2006(E): Information technology — Eiffel: Analysis, Design and Programming Language*. 1st. International standard ISO/IEC 25436. (Alexander V. Kogtenkov is a member of the committee.) Geneva, Switzerland: ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission), Dec. 1, 2006.

[28] ISO. *ISO/IEC 30170:2012(E): Information technology — Programming languages — Ruby*. 1st. Geneva, Switzerland: International Organization for Standardization, Apr. 15, 2012.

[29] ISO. *ISO/IEC 14882:2014(E): Information technology — Programming languages — C++*. 4th. Geneva, Switzerland: International Organization for Standardization, Dec. 15, 2014.

[30] JetBrains. *Kotlin Language Documentation*. Jan. 31, 2017. URL: https://kotlinlang.org/docs/kotlin-docs.pdf (visited on 2017-01-31).

[31] JetBrains. *Kotlin Language Specification*. Jan. 31, 2017. URL: https://jetbrains.github.io/kotlin-spec/kotlin-spec.pdf (visited on 2017-01-31).

[32] Gerwin Klein. "Verified Java Bytecode Verification." PhD thesis. Institut für Informatik, Technische Universität München, 2003. URL: http://www4.in.tum.de/~kleing/diss/.

[33] Gerwin Klein and Tobias Nipkow. "A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler." In: *ACM Trans. Program. Lang. Syst.* 28.4 (July 2006), pp. 619–695. ISSN: 0164-0925. DOI: 10.1145/1146809.1146811.

[34] Alexander Kogtenkov. *Void-safety: tag info*. Stackoverflow, July 15, 2016. URL: http://stackoverflow.com/tags/void-safety/info (visited on 2016-12-30).

[35] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. "Alias and Change Calculi, Applied to Frame Inference." In: *CoRR* abs/1307.3189 (2013). URL: http://arxiv.org/abs/1307.3189.

[36] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. "Alias calculus, change calculus and frame inference." In: *Science of Computer Programming* 97, Part 1 (2015). Special Issue on New Ideas and Emerging Results in Understanding Software, pp. 163–172. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.11.006.

[37] A.V. Kogtenkov. "Mechanically Proved Practical Local Null Safety." In: *Proceedings of the Institute for System Programming of the RAS* 28.5 (Dec. 2016), pp. 27–54. ISSN: 2079-8156 (Print), 2220-6426 (Online). DOI: 10.15514/ISPRAS-2016-28(5)-2.

[38] J.-L. Lassez, V.L. Nguyen, and E.A. Sonenberg. "Fixed point theorems and semantics: a folk tale." In: *Information Processing Letters* 14.3 (1982), pp. 112–116. ISSN: 0020-0190. DOI: 10.1016/0020-0190(82)90065-5.

[39] K. Rustan M. Leino. "Data Groups: Specifying the Modification of Extended State." In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. Vancouver, British Columbia, Canada: ACM, 1998, pp. 144–153. ISBN: 1-58113-005-8. DOI: 10.1145/286936.286953.

[40] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Maintenance Release. Oracle America, Inc. and/or its affiliates., Feb. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 2016-12-28).

[41] Andreas Lochbihler. "A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler." PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012. DOI: 10.5445/KSP/1000028867.

[42] Chris Male, David J. Pearce, Alex Potanin, and Constantine Dymnikov. "Formalisation and implementation of an algorithm for bytecode verification of @NonNull types." In: *Sci-*

ence of Computer Programming 76.7 (2011), pp. 587–608. ISSN: 0167-6423. DOI: 10.1016/j.scico.2010.10.004.

[43] Amogh Margoor and Raghavan Komondoor. "Two techniques to improve the precision of a demand-driven null-dereference verification approach." In: *Science of Computer Programming* 98, Part 4 (2015), pp. 645–679. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.09.006.

[44] *Mediator pattern*. 2016. URL: https://en.wikipedia.org/wiki/Mediator_pattern (visited on 2016-12-23).

[45] Erik Meijer and Wolfram Schulte. "Unifying Tables, Objects and Documents." In: *Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL 2003)*. Aug. 2003. URL: https://www.microsoft.com/en-us/research/publication/unifying-tables-objects-and-documents/.

[46] Bertrand Meyer. *Object-oriented software construction*. 2nd ed. Prentice Hall, 1997.

[47] Bertrand Meyer. "Attached Types and Their Application to Three Open Problems of Object-Oriented Programming." In: *ECOOP 2005 – Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings*. Ed. by Andrew P. Black. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–32. ISBN: 978-3-540-31725-8. DOI: 10.1007/11531142_1.

[48] Bertrand Meyer. "The Dependent Delegate Dilemma." In: *Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems Marktoberdorf, Germany 3–15 August 2004*. Ed. by Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare. Dordrecht: Springer Netherlands, 2005, pp. 105–118. ISBN: 978-1-4020-3532-6. DOI: 10.1007/1-4020-3532-2_4.

[49] Bertrand Meyer. *Targeted expressions: safe object creation with void safety*. July 30, 2012. URL: http://se.ethz.ch/~meyer/publications/online/targeted.pdf (visited on 2017-05-08).

[50] Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. "Void Safety: Putting an End To the Plague of Null Dereferencing." In: *Dr.Dobbs Journal* online (Sept. 1, 2009).

URL: http://drdobbs.com/architecture-and-design/219500827.

[51] Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. "Avoid a Void: The Eradication of Null Dereferencing." In: *Reflections on the Work of C.A.R. Hoare*. Ed. by A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood. History of Computing. Springer London, 2010, pp. 189–211. ISBN: 978-1-84882-912-1. DOI: 10.1007/978-1-84882-912-1_9.

[52] Benjamin Morandi, Mischael Schill, Sebastian Nanz, and Bertrand Meyer. "Prototyping a Concurrency Model." In: *Proceedings of the 2013 13th International Conference on Application of Concurrency to System Design*. ACSD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 170–179. ISBN: 978-0-7695-5035-0. DOI: 10.1109/ACSD.2013.21.

[53] Robert Morgan. *Building an Optimizing Compiler*. Newton, MA, USA: Digital Press, 1998. ISBN: 1-55558-179-X.

[54] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[55] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3540654100.

[56] Piotr Nienaltowski. "Practical framework for contract-based concurrent object-oriented programming." Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 17061. PhD thesis. Swiss Federal Institute Of Technology Zürich, 2007. DOI: 10.3929/ethz-a-005363875.

[57] David von Oheimb. "Analyzing Java in Isabelle/HOL: *Formalization, Type Safety and Hoare Logic*." http://ddvo.net/diss/. PhD thesis. Technische Universität München, 2001.

[58] *Open Web Application Security Project*. 2017. URL: https://www.owasp.org/ (visited on 2017-01-15).

[59] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. "Functional Big-Step Semantics." In: *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin

Heidelberg, 2016, pp. 589–615. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_23.

[60] *PHP Bug Tracking System*. 2016. URL: https://bugs.php.net/ (visited on 2016-05-25).

[61] Xin Qi and Andrew C. Myers. *Masked Types*. Tech. rep. Oct. 28, 2008. URL: http://hdl.handle.net/1813/11563 (visited on 2016-12-25).

[62] Xin Qi and Andrew C. Myers. "Masked Types for Sound Object Initialization." In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: ACM, 2009, pp. 53–65. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480890.

[63] Guido van Rossum and the Python development team. *The Python Language Reference*. Release 3.5.1. Python Software Foundation, May 15, 2016. URL: https://docs.python.org/3/download.html (visited on 2016-05-15).

[64] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. "The Billion-Dollar Fix." In: *ECOOP 2013 – Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 205–229. ISBN: 978-3-642-39038-8. DOI: 10.1007/978-3-642-39038-8_9.

[65] Jeremy Siek. *Big-step, diverging or stuck?* July 2012. URL: http://siek.blogspot.ch/2012/07/big-step-diverging-or-stuck.html (visited on 2016-09-15).

[66] Jeremy Siek. *Type Safety in Three Easy Lemmas*. May 2013. URL: http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html (visited on 2016-09-15).

[67] Fausto Spoto. "Precise null-pointer analysis." English. In: *Software & Systems Modeling* 10.2 (2011), pp. 219–252. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0132-5.

[68] Raymie Stata. *ESCJ 2: Improving the safety of Java*. Dec. 2, 1995. URL: http://kindsoftware.com/products/opensource/ESCJava2/ESCTools/docs/design-notes/escj02.html (visited on 2017-04-27).

[69] Alexander J. Summers and Peter Müller. *Freedom before commitment. simple flexible initialisation for non-full types*. Tech. rep. 716. Zurich, Switzerland: ETH Zurich, Department of Computer Science, 2010. DOI: 10.3929/ethz-a-006904372.

[70] Alexander J. Summers and Peter Müller. "Freedom Before Commitment: A Lightweight Type System for Object Initialisation." In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. 548115. Portland, Oregon, USA: ACM, 2011, pp. 1013–1032. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048142.

[71] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific J. Math.* 5.2 (1955), pp. 285–309. URL: http://projecteuclid.org/euclid.pjm/1103044538.

[72] *The Checker Framework 2.1.10*. Apr. 3, 2017. URL: https://checkerframework.org/ (visited on 2017-05-08).

[73] *Void-safe changes to Eiffel libraries*. Community portal for Eiffel. URL: https://www.eiffel.org/doc/eiffel/Void-safe%20changes%20to%20Eiffel%20libraries (visited on 2016-12-30).

[74] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. Feb. 2016. URL: http://isabelle.in.tum.de/dist/Isabelle2016/doc/isar-ref.pdf (visited on 2016-09-20).

[75] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. "Effective Dynamic Detection of Alias Analysis Errors." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 279–289. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491439.

[76] Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. "Object Initialization in X10." In: *ECOOP 2012 – Object-Oriented Programming: 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Ed. by James Noble. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 207–231. ISBN: 978-3-642-31057-7. DOI: 10.1007/978-3-642-31057-7_10.