Mechanically Proved Practical Local Null Safety

A.V. Kogtenkov <kwaxer@mail.ru>
ETH Zürich
Universitätstrasse 19, 8092 Zürich, Switzerland

Abstract. Null pointer dereferencing is a well-known bug in object-oriented programs. It can be avoided by adding special validity rules to a language in which programs are written. Are the rules sufficient to ensure absence of such exceptions? This work focuses on null safety for intra-procedural context where no additional type annotations are needed and formalizes the rules in Isabelle/HOL proof assistant. It then proves null-safety preservation theorem for big-step semantics in a computer-checkable way. Finally, it demonstrates that with such rules null-safe and null-unsafe semantics are equivalent.

Keywords: null safety; void safety; static analysis; Eiffel; formal methods; big-step operational semantics; preservation theorem; operational semantics equivalence.

DOI: 10.15514/ISPRAS-2016-28(5)-2

For citation: Kogtenkov A.V. Mechanically Proved Practical Local Null Safety. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016. pp. 27-54. DOI: 10.15514/ISPRAS-2016-28(5)-2

1. Introduction

In his talk at a conference in 2009 Tony Hoare called his invention of the null reference in 1965 a "billion-dollar mistake" ([8]). 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 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.

Provided that most popular languages do not prevent null-pointer dereferencing at compile time, it remains one of the day-to-day issue discovered in open source and private software. As of May 2016 a public database of cybersecurity vulnerabilities known as Common Vulnerabilities and Exposures (CVE$^{®}$) [3] operated by *MITRE* and funded by Computer Emergency Readiness Team (*CERT*) has 727 entries

mentioning null pointer dereference bugs explicitly. The distribution by years is shown in figure 1.

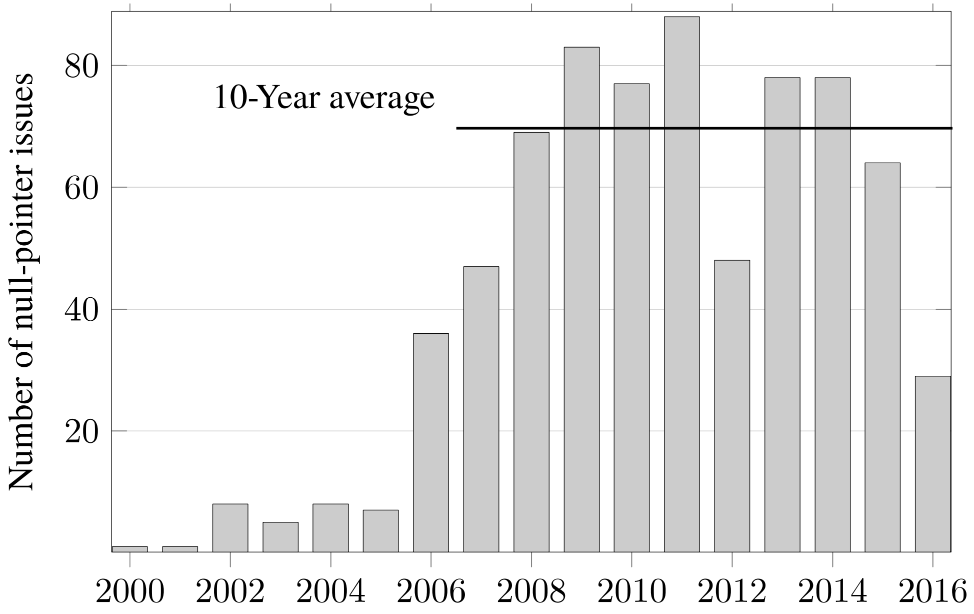

Fig. 1. Null pointer issues (such as null pointer dereferencing) in Common Vulnerabilities and Exposures database

A solution to this problem was proposed in [15] as an extension of the type system of Eiffel with a set of so called certified attachment patterns (CAP). Similar approach was proposed for Spec#, but was not adopted for inclusion into C# and the Common Runtime Language because of difficulties caused by required changes in the underlying platform and unsoundness of the prototype implementation ([2]). Indeed, it was discovered that a simple extension of the type system with "non-null" types does not allow for safe initialization of objects ([22]) and the void-safety property can be compromised. (In Eiffel null references are known as void references, hence the name "void safety". In this paper *void* and *null* are used interchangeably.) The same paper proposed a fix by introducing special annotations [Free] and [Unclassified] to source code and some new rules that should be checked by a compiler. Because the cases when the annotation is required are rare, an attempt to use a light-weight solution that does not require any additional annotation is implemented in [5]. Even though both approaches were shown to be usable, neither is accompanied with a formal machine-checkable soundness proof of the proposed type systems combined with additional restrictions placed on source code.

Current work formalizes the part related to certified attachment patterns, relaxes void safety rules for local intra-procedural context and proves that the rules ensure void safety with a generic proof assistant Isabelle/HOL.

2. Overview

Existing proposals ([2, 15, 22]) that address void safety issue 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., [4]) meaning that types of variables are specified explicitly. This type information is then used to check reattachment validity rules. For example, an assignment $x := y$ would be valid only when a type of y conforms to a 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, the locals can be changed only in a current feature. As a result it should be possible to get rid of type annotations altogether. It turns out that CAPs are absolutely sufficient for local variables and attachment annotations can be safely discarded.

The CAPs for local variables can also be applied to function's **Result**. For example, one can replace the code on the left with the code on the right:

```
foo: X
    do
        if attached something as r then
            Result := r
        else
            Result := something_else_attached
        end
    end
```

```
foo: X
    do
        Result := something
        if not attached Result then
            Result := something_else_attached
        end
    end
```

This allows not only for less code in new classes, but also for keeping original code unchanged if it follows this pattern.

Certified attachment patterns in [4] treat every boolean connective as a single use case: 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 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. As a result, expressions of any complexity or nesting can be supported.

Even though simple branches and loop conditions were taken care by the original CAPs, the rules did not cover loop bodies. Simple analyses like definite assignment required by Java [7] can be done in one pass because on every iteration a variable can only become assigned, not the reverse. This does not work 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 on figure 2.

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

Fig. 2. 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.

A set of CAPs specified in [4] do ensure void safety. However, they cannot be used in practice for any large scale application without provision for rules to escape void safety checks. It is just physically 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 and there are no effective descendants of this class yet, the program will not compile. The solution adopted in [6] is to rely on exceptions, including forced checks of assertions. This triggers so called "design mode" when compiler ignores attachment status in type checks. Modeling the mode in the formalization essentially affects proofs but makes it possible to show soundness of real-life analysis.

To my knowledge this is the first attempt to formalize void safety rules and program semantics with attachment properties in a proof assistant environment. Moreover, this is the first time a formal void safety model is mechanically checked.

Presented formalization is done with big-step semantics style that is known to be suitable for proving preservation property, but to have issues with proving progress property. To address this, two different semantics are considered: void-unsafe and void-safe. It is demonstrated that both are 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.

In order to remain sound the formalization relies on attachment properties of expressions. Therefore, extending the approach to an inter-procedural context requires non-local void safety guarantees that can be achieved with a void-safe type system. This is done in Eiffel by augmenting its type system with **attached** and **detachable** marks added to type declarations and by specifying conformance and initialization rules that ensure an expression of an attached type always yields an object. Abstracting away language dissimilarities, the proposed rules can be used in other languages including mainstream ones as soon as they strengthen their type systems to be not only type-safe, but also null-safe.

All presented void safety rules are also implemented by me in the compiler [6] and are in production.

3. Formalization

Isabelle/HOL was successfully used in different projects starting from algebraic topology to verification of an operating system micro-kernel ([9]). It is build 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 ([11, 12, 13]). 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), 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.

3.1 Translation of source language

Source language syntax is modeled in Isabelle/HOL with appropriate constructors of a datatype *expression* (figure 3). 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.

$expression$ =

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 expression
until $expression$ loop $expression$ end |	– Loop
attached $type$ $expression$ as $name$ |	– Object test
Exception	– Exception

Fig. 3. Datatype expression

Voidness tests are source language expressions that check if a particular expression evaluates to *Void* at run-time or not:

expression /= Void

However, there is a more powerful construct that can be used instead of voidness tests: object tests. 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. When the type part is absent, the object test 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 still reflected in the formal semantics of the object test expression (see section 5.1).

3.2 Practical considerations

3.2.1 Design mode and unreachable code

As mentioned in section 2, 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 [6] 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.

[12] proposes to model definite assignment property in presence of exceptions in Jinja with a 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. This approach perfectly works to model exceptional cases and unreachable code during attachment analysis too. But instead of using somewhat ad hoc rules to handle *set option*, a new type *topset* is introduced. It is obtained from a regular *set* type by adding a new top element. The operations are defined as shown in figure 4.

$$\begin{aligned} X \sqsubseteq \top &= True \\ \top \sqsubseteq \lceil A \rceil &= False \\ \lceil A \rceil \sqsubseteq \lceil B \rceil &= A \subseteq B \end{aligned} \qquad \begin{aligned} X \sqcup \top &= \top \\ \top \sqcup X &= \top \\ \lceil A \rceil \sqcup \lceil B \rceil &= \lceil A \cup B \rceil \end{aligned}$$

$$\begin{aligned} x \in^{\top} \top &= True \\ x \in^{\top} \lceil A \rceil &= x \in A \end{aligned} \qquad \begin{aligned} X \sqcap \top &= X \\ \top \sqcap X &= X \\ \lceil A \rceil \sqcap \lceil B \rceil &= \lceil A \cap B \rceil \end{aligned}$$

Fig. 4. Operations on topset.

The type *topset* is proved to be a complete lattice and a distributive lattice. These properties are essential in proofs involving fixed point of a transfer function (section 4.1).

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. But neither insertion nor removal changes a top element $\top$:

$$\begin{aligned} A \oplus x &= A \sqcup \lceil \{x\} \rceil \\ A \ominus x &= A \sqcap \lceil \overline{\{x\}} \rceil \end{aligned} \qquad \begin{aligned} \top \oplus x &= \top \\ \top \ominus x &= \top \end{aligned}$$

3.2.2 Operator transformation

A source code snippet, where a variable is considered attached because of a previous test to *Void* or when it is an object test local (see [4]), is called an attachment scope of this variable. The following kinds of scopes exist:

1. **Control flow scope –** an attachment scope based on language constructs that change execution flow.
2. **Operator scope** – an attachment scope based on semistrict boolean 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 5.

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**

Fig. 5. Scope combinations (code fragments where variable x is considered attached are marked with ...).

The language standard [4] 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 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 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 3) or addition of 2 new constructors to this datatype and introduction of new datatypes for operators with specific operator kinds. In both cases all induction-based proofs would have to be performed for 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. Every boolean expression can be translated into a conditional expression with nested boolean constants and optional sequences. The conversions of the boolean operators mentioned earlier and some others added for completeness are listed in figure 6. Following the terminology used in [4] they are called unfolded forms of boolean operators.

Operator	Original expression	Translation	
Negation	**not** *e*	**if** *e* **then False else True end**	(1)
Conjunction	*e1* **and then** *e2*	**if** *e1* **then** *e2* **else False end**	(2)
	e1 **and** *e2*	**if** *e1* **then** *e2* **else** *e2*; **False end**	(3)
Disjunction	*e1* **or else** *e2*	**if** *e1* **then True else** *e2* **end**	(4)
	e1 **or** *e2*	**if** *e1* **then** *e2*; **True else** *e2* **end**	(5)
Implication	*e1* **implies** *e2*	**if** *e1* **then** *e2* **else True end**	(6)
	not *e1* **or** *e2*	**if** *e1* **then** *e2* **else** *e2*; **True end**	(7)
Converse nonimplication	**not** *e1* **and then** *e2*	**if** *e1* **then False else** *e2* **end**	(8)
	not *e1* **and** *e2*	**if** *e1* **then** *e2*; **False else** *e2* **end**	(9)

Fig. 6. 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:

$$is\_false\ (c\ ;;\ False_c) = True \qquad is\_true\ (c\ ;;\ True_c) = True$$
$$is\_false\ \_ = False \qquad is\_true\ \_ = False$$

The cases from figure 6, when instead of an expression followed by a constant there is just a single constant **False** or **True**, can be represented by the 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

$$is\_true\ (if\ b\ then\ e_1\ else\ e_2\ end) = (if\ (is\_true\ b)\ then\ is\_true\ e_1)$$
$$\vee\ (if\ (is\_false\ b)\ then\ is\_true\ e_2) \vee (is\_true\ e_1 \wedge is\_true\ e_2)$$

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 ([17, 18]).

Read-only scopes. General rules that define scopes are intermingled with the rules of an attachment status transfer function if (unlike [4]) the scopes are seen as means to determine potential attachment status of an arbitrary variable, not just a read-only one. For the sake of simplicity, consider the scopes of read-only variables first because they may be defined without bringing general attachment rules into play.

Scopes are defined for two cases: when an associated expression evaluates to **True** and when it evaluates to **False**:

Definition 3.1 (Scope function)**.** *A function that computes a set of read-only variables that are considered attached for an expression that evaluates to a particular boolean value is called a* ***scope function****.*

A scope function for an expression e that evaluates to True (False) is called ***positive (negative)*** *and is denoted* $+[e]$ *(*$-[e]$*).*

The rules to compute scope function are shown in figure 7. Only expressions that can produce non-empty sets of attached variables are listed. In all other cases the associated sets are empty. The notation $\lceil\{\ldots\}\rceil$ is used for sets adjusted to handle design mode (section 3.2.1).

$$+[\mathit{attached\ t\ Local\ n'\ as\ n}] = \lceil\{n', n\}\rceil \qquad (10)$$

$$+[\mathit{attached\ t\ e\ as\ n}] = \lceil\{n\}\rceil \qquad (11)$$

$$\text{if } e \text{ is not a variable}$$

$$+[\_] = \lceil\varnothing\rceil \qquad (12)$$

$$-[\_] = \lceil\varnothing\rceil \qquad (13)$$

$$+[\mathit{if\ b\ then\ e_1\ else\ e_2\ end}] = \begin{cases} -[b] \sqcup +[e_2] & \text{if } \mathit{is\_false}\ e_1 \\ +[b] \sqcup +[e_1] & \text{if } \mathit{is\_false}\ e_2 \\ \lceil\varnothing\rceil & \text{otherwise} \end{cases} \qquad (14)$$

$$-[\mathit{if\ b\ then\ e_1\ else\ e_2\ end}] = \begin{cases} -[b] \sqcup -[e_2] & \text{if } \mathit{is\_true}\ e_1 \\ +[b] \sqcup -[e_1] & \text{if } \mathit{is\_true}\ e_2 \\ \lceil\varnothing\rceil & \text{otherwise} \end{cases} \qquad (15)$$

Fig. 7. Scope rules for read-only variables

The rules for object tests follow the explanations ealier: if an object test evaluates to $True_c$ (positive scope function), the corresponding object test variable is attached. Moreover, if the object test expression is a variable it is also known to be attached.

Two other cases cover general conditional expressions. 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 object test locals involved in its sub-expressions. The reason is that information whether an object test succeeded, be it a conditional expression b or one of the branch expressions e_1 or e_2, is lost in that case.

Consider one of the cases when a branch expression meets a condition to produce a known constant value, for example, $\mathit{is\_false}\ e_1$. Because this is the rule for positive scope function $+[\mathit{if\ b\ then\ e_1\ else\ e_2\ end}]$, the computed sets correspond to the case when the conditional expression evaluates to $True_c$. From the condition $\mathit{is\_false}\ e_1$ we know that b could not have been evaluated to $True_c$. Also, we know that the only case to get $True_c$ for the whole expression is to get $True_c$ for e_2. Therefore, a set of attached variables in that case is a union of the negative scope function for b and a positive scope function for e_2.

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

$$\begin{array}{lll} & +[\textbf{not not attached } x] & \\ = & +[\textbf{if not attached } x \textbf{ then False else True end}] & \text{by (6)} \\ = & -[\textbf{not attached } x] \sqcup +[\textbf{True}] & \text{by (14)} \\ = & -[\textbf{not attached } x] & \text{by (12)} \\ = & -[\textbf{if attached } x \textbf{ then False else True end}] & \text{by (6)} \\ = & +[\textbf{attached } x] \sqcup -[\textbf{False}] & \text{by (15)} \\ = & +[\textbf{attached } x] & \text{by (13)} \end{array}$$

What if for a given conditional expression both $\mathit{is\_false}\ e_1$ and $\mathit{is\_false}\ e_2$ would be true? Would the positive scope function yield a consistent result? For sub-expressions the function gives $+[e_1] = \lceil\varnothing\rceil$ and $+[e_2] = \lceil\varnothing\rceil$. So the result for the whole conditional is $+[b]$ and $-[b]$ at the same time that looks weird. The puzzle is solved by noticing that in this case the whole conditional expression evaluates to $False_c$ and does not fit the assumption that it produces $True_c$ (see definition 3.1).

4. Attachment properties

4.1 Transfer function

Given a set of attached variables A, a transfer function $A \triangleright e$ computes a set of attached variables for a given expression e. It is defined inductively as 5 mutually recursive functions:

$\cdot \triangleright \cdot$	the transfer function itself (figure 8)
$\cdot \triangleright\!+ \cdot$ $\cdot \triangleright\!- \cdot$	computes a set of attached variables with an assumption that the expression evaluates to true/false (positive/negative scope) (figure 10)
$\cdot \triangleright\triangleright \cdot$	computes a set of attached variables for a given list of expressions (used to model arguments in feature calls) (figure 11)
$\cdot \hookrightarrow \cdot$	tells if a given expression is attached (figure 9)

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

Fig. 8. Transfer function

Let's have a look at the most interesting cases. 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 initial set otherwise.

An attachment status of an expression is $True$ if it is a value other than $Void$, a local in the set of attached variables, or, a conditional expression with both branches attached (figure 9). Note that an attachment status of a conditional branch takes into account whether it is positive or negative.

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

Fig. 9. Attachment status function

A similar formula is used for the transfer function on a conditional expression: one branch is evaluated with an assumption that a condition is true (the part $A \triangleright\!+ b$) and the other one – when it is false (the part $A \triangleright\!- b$).

A special value $\top$ is used for an exception to indicate that all variables can be safely considered as attached.

Positive and negative transfer functions (figure 10) differ from a regular transfer function only in two cases: for object tests and for conditional expressions. They mimic the scope function discussed in section 3.2.2.

$$
\begin{aligned}
&A \triangleright\!+ \mathit{attached}\ T\ \mathit{Local}\ n'\ \mathit{as}\ n && = A \sqcup \lceil\{n', n\}\rceil\\
&A \triangleright\!+ \mathit{attached}\ T\ e\ \mathit{as}\ n && = A \triangleright e \sqcup \lceil\{n\}\rceil \quad \text{if } e \text{ is not a variable}\\
&A \triangleright\!+ \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} && = \begin{cases} A \triangleright\!- b \triangleright\!+ e_2 & \text{if } \mathit{is\_false}\ e_1\\ A \triangleright\!+ b \triangleright\!+ e_1 & \text{if } \mathit{is\_false}\ e_2\\ A \triangleright \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} & \text{otherwise}\end{cases}\\
&A \triangleright\!- \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} && = \begin{cases} A \triangleright\!- b \triangleright\!- e_2 & \text{if } \mathit{is\_true}\ e_1\\ A \triangleright\!+ b \triangleright\!- e_1 & \text{if } \mathit{is\_true}\ e_2\\ A \triangleright \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} & \text{otherwise}\end{cases}\\
&A \triangleright\!+ e && = A \triangleright e\\
&A \triangleright\!- e && = A \triangleright e
\end{aligned}
$$

Fig. 10. Transfer functions for positive and negative scopes

For a loop the transfer function is specified using a loop operator. A loop body is evaluated in a negative branch of an exit condition and the effect of the loop as a whole is evaluated in a positive branch of the same condition. The loop operator is defined as a greatest fixed point for $\lambda X. X \triangleright\!- e \triangleright b$ where e is an exit condition and b is a loop body:

$$A \triangleright\!* (- e \triangleright b) \equiv \mathit{gfp}\ (\lambda X.\ A \sqcap X \triangleright\!- e \triangleright b)$$

The strange form of a loop function reflects what is implemented in the compiler. It reuses the same set of classes and functions for both conditional expressions and for loops. The transfer function for loops is then implemented by iterating over a loop until it stabilizes. A theorem from *HOL-Library* states that in this case the result is equal to the greatest fixed point. The theorem depends on monotonicity of the function. Firstly, observe that the loop function is monotone on both arguments, but we need only monotonicity on the last one:

Lemma 4.1 (Loop function monotonicity)**.** $\mathit{mono}\,f \implies \mathit{mono}\ (\lambda X.\ A \sqcap f\,X)$

Then, instead of proving lemmas with a specific loop function, a generalized version can be used: $\mathit{loop\_operator}\,f\,A \equiv \mathit{gfp}\ (\lambda x.\ A \sqcap f\,x)$. The loop operator is monotone and idempotent on both arguments:

Lemma 4.2 (Loop operator monotonicity)**.** $\mathit{mono}\ (\mathit{loop\_operator}\,f)$

Proof. From monotonicity of greatest fixed point. □

Lemma 4.3 (Loop operator unfolding**).**

$$mono\, f \Longrightarrow loop\_operator\, f\, A = loop\_function\, f\, A\ (loop\_operator\, f\, A)$$

Lemma 4.4 (Loop operator idempotence**).**

$$mono\, f \Longrightarrow loop\_operator\, f\ (loop\_operator\, f\, x) = loop\_operator\, f\, x$$

As one would expect, an application of a loop operator produces a smaller set of attached variables:

Lemma 4.5. $mono\, f \Longrightarrow loop\_operator\, f\, A \le A$

$$loop\_operator\, f\, A \le loop\_operator\, f\ (f\, A)$$

To conclude this section let's look at the rules for an expression list modeling arguments. Arguments of a call are subject to chained processing even though it might seem unnecessary. It turns out that an attachment status of an object test local could be affected because of the rules for "design mode" (section 3.2.1). The transfer function for every argument is evaluated in the context of a previous one (figure 11) or in the context of a target (for the first argument). The same effect can be achieved by using Isabelle/HOL function fold.

$$A \rhd\!\rhd [\,] \;=\; A \qquad\qquad A \rhd\!\rhd (e \cdot es) \;=\; A \rhd e \rhd\!\rhd es$$

Fig. 11. Transfer functions for argument lists

Lemma 4.6. A $\rhd\!\rhd$ es = fold (λ e X. X $\rhd$ e) es A

According to [12] 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.

Here is an essential property of the transfer function that will be used later: it is monotonic. Intuitively this means that the more attached variables are known before an expression, the more there are after the expression:

Lemma 4.7 (Transfer function monotonicity**).** $mono\ (\lambda X.\ X \rhd c)$

Proof. By structural induction on all 5 mutually recursive function definitions. □

4.2 Expression validity

Validity rules are specified in Isabelle/HOL using an inductive predicate

$$\Gamma, A \vdash e : T$$

where Γ is an environment, A – a set of attached variables, e – an expression being checked, T – either $Attached$ or $Detachable$ – an attachment status of the expression e. $Attached$ means the expression produces a value that is not $Void$, $Detached$ means this value may be $Void$. If the predicate is true, the expression satisfies void safety rules in the given environment and attachment set and its type is T. The rules to compute predicate are shown in figure 12.

$$\frac{v \ne Void_v}{\Gamma, A \vdash Value\ v : Attached}\ \textsc{Value}_{att} \qquad \frac{v = Void_v}{\Gamma, A \vdash Value\ v : Detachable}\ \textsc{Value}_{det}$$

$$\frac{n \in^{\top} A}{\Gamma, A \vdash Local\ n : Attached}\ \textsc{Local}_{att} \qquad \frac{\neg\, n \in^{\top} A}{\Gamma, A \vdash Local\ n : Detachable}\ \textsc{Local}_{det}$$

$$\frac{}{\Gamma, A \vdash Exception : Attached}\ \textsc{Exception}$$

$$\frac{\Gamma, A \vdash e_1 : Attached \wedge \Gamma, A \rhd e_1 \vdash e_2 : Attached}{\Gamma, A \vdash e_1\ ;;\ e_2 : Attached}\ \textsc{Seq}$$

$$\frac{\Gamma, A \vdash e : T}{\Gamma, A \vdash n ::= e : Attached}\ \textsc{Assign} \qquad \frac{}{\Gamma, A \vdash create\ n : Attached}\ \textsc{Create}$$

$$\frac{\Gamma, A \vdash e : Attached \wedge \Gamma, A \rhd e \vdash a\ [:]\ Ts}{\Gamma, A \vdash e\ .\ f\ (a) : Attached}\ \textsc{Call}$$

$$\frac{\Gamma, A \vdash e : T}{\Gamma, A \vdash attached\ t\ e\ as\ n : Attached}\ \textsc{Test}$$

$$\frac{\Gamma, A \vdash b : Attached \wedge \Gamma, A \rhd\!+ b \vdash e_1 : T_1 \wedge \Gamma, A \rhd\!- b \vdash e_2 : T_2}{\Gamma, A \vdash if\ b\ then\ e_1\ else\ e_2\ end : upper\_bound\ T_1\ T_2}\ \textsc{If}$$

$$\frac{\Gamma, A \rhd\!* (-\ e \rhd b) \vdash e : Attached \wedge \Gamma, A \rhd\!* (-\ e \rhd b) \rhd\!- e \vdash b : Attached}{\Gamma, A \vdash until\ e\ loop\ b\ end : Attached}\ \textsc{Loop}$$

$$\frac{}{\Gamma, A \vdash [\,]\ [:]\ [\,]}\ \textsc{Arg}_{Nil} \qquad \frac{\Gamma, A \vdash e : T \wedge \Gamma, A \rhd e \vdash es\ [:]\ Ts}{\Gamma, A \vdash e \cdot es\ [:]\ T \cdot Ts}\ \textsc{Arg}_{Cons}$$

Fig. 12. Void safety rules

There are two rules for local variables: if a local variable name is in the set of attached variables, 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 ($\textsc{If}$). The upper bound is $Detachable$ if any of the operands is $Detachable$, and $Attached$ otherwise.

The validity predicate is properly defined, i.e. it cannot be true for attached and detachable types at the same time:

Lemma 4.8 (Attachment type uniqueness**).** *A valid expression has one attachment type:* $\Gamma, A \vdash e : T \wedge \Gamma, A \vdash e : T' \Longrightarrow T = T'$

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 4.9 (Attachment type monotonicity**).**

$$A \le B \wedge \Gamma, A \vdash e : T_A \Longrightarrow \exists T_B.\ \Gamma, B \vdash e : T_B \wedge T_B \to_a T_A$$

Proof. The proof is done by structural induction on the predicate definition. It relies on monotonicity of transfer function (lemma 4.7) for compound expressions such as sequences and calls. For a conditional expression, types of both branches can be obtained thanks to lemma 4.8 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 4.2). □

If a loop is valid in a given context, it is valid in a context obtained by a single or multiple application of the loop exit condition and loop body:

Lemma 4.10. *A loop remains valid after applying its transfer function to a set of attached variables one or any number of times:*

$$\Gamma, A \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T \Longrightarrow \Gamma, A \rhd\!-\ e \rhd c \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T$$

$$\Gamma, A \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T \Longrightarrow \Gamma, A \rhd\!*\ (-\ e \rhd c) \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T$$

Proof. Follows from monotonicity of transfer function and expression validity predicate (lemmas 4.7 and 4.9), idempotence of loop operator (lemma 4.4) and lemma 4.5. □

A notion of void-safe expressions is defined using the expression validity predicate with or without an associated context:

Definition 4.1 (Void-safe expression)**.** *An expression e is void-safe with type T in an environemnt Γ iff there is type that satisfies expression validity predicate with an empty set of attached variables:*

$$\Gamma \vdash e : T \equiv \Gamma, \lceil\varnothing\rceil \vdash e : T$$

An expression e is void-safe in an environemnt Γ iff there is type T with which e is void-safe in an environment Γ:

$$\Gamma \vdash e \surd_e \equiv \exists T.\ \Gamma \vdash e : T$$

In the context of null safety, if a local variable is considered attached at compile time, it should have an associated object at run-time. This property is captured by the notion of a valid state.

4.3 State validity

A state of a program is modeled by two functions: a function that maps local variable names to their value (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 denoted in earlier formulas as Γ.

Definition 4.2. *A local state l is valid w.r.t. an environment Γ iff for every local in Γ the state l has a value for this local:*

$$\Gamma \vdash l \surd_s{}^E \equiv \forall\, \mathit{name}\ T.\ \Gamma\ \mathit{name} = \lfloor T \rfloor \longrightarrow (\exists\, v.\ l\ \mathit{name} = \lfloor v \rfloor)$$

Definition 4.3. *A local state l is valid w.r.t. an attachment set A iff for every local in A the state l has an attached value for this local provided that A is not $\top$:*

$$A \vdash l \surd_s{}^A \equiv A \neq \top \longrightarrow (\forall\, n.\ n \in^\top A \longrightarrow (\exists\, v.\ l\ n = \lfloor v \rfloor \wedge v \neq \mathit{Void}_v))$$

Definition 4.4 (Void-safe state)**.** *For an environment Γ with an attachment set A, a state (l, h) is void-safe iff for any local variable name in A there is a local variable of this name in l attached to an object:*

$$\Gamma, A \vdash (l, h)\ \surd_s \equiv \Gamma \vdash l \surd_s{}^E \wedge A \vdash l \surd_s{}^A$$

For an environment Γ, a state s is attachment-valid iff it is void-safe for Γ with an empty attachment set:

$$\Gamma \vdash s \surd_s \equiv \Gamma, \lceil\varnothing\rceil \vdash s \surd_s$$

Most important properties of the state validity function are anti-monotonicity and what could be said about state validity if the corresponding attachment set changes:

Lemma 4.11 (Attachment state anti-monotonicity)**.**

$$B \le A \wedge A \neq \top \wedge A \vdash l \surd_s{}^A \Longrightarrow B \vdash l \surd_s{}^A$$

Lemma 4.12. *Detaching, attaching and updating a local:*

$$\begin{aligned} A \vdash l \surd_s{}^A &\Longrightarrow A \ominus \mathit{name} \vdash l(\mathit{name} \mapsto \mathit{value}) \surd_s{}^A \\ \mathit{value} \neq \mathit{Void}_v \wedge A \vdash l \surd_s{}^A &\Longrightarrow A \oplus \mathit{name} \vdash l(\mathit{name} \mapsto \mathit{value}) \surd_s{}^A \\ \mathit{value} \neq \mathit{Void}_v \wedge A \vdash l \surd_s{}^A &\Longrightarrow A \vdash l(\mathit{name} \mapsto \mathit{value}) \surd_s{}^A \end{aligned}$$

5. Local null safety

5.1 Big-step semantics

The big-step semantics is defined in Isabelle/HOL as an inductive predicate on transitions from an initial expression-state pair to a resulting one (figures 13 and 14). The rules are similar to those used in type system soundness proofs (e.g., [11, 12, 13]). The key differences are in the additional rules for object tests and in a modified rule for feature calls.

$$\frac{}{\Gamma \vdash \langle \mathit{Value}\ v,\ (l,\ m)\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle}\ \textsc{Value} \qquad \frac{l\ n = \lfloor v \rfloor}{\Gamma \vdash \langle \mathit{Local}\ n,\ (l,\ m)\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle}\ \textsc{Local}$$

$$\frac{\Gamma \vdash \langle e_1,\ s\rangle \Rightarrow \langle \mathit{unit},\ s'\rangle \wedge \Gamma \vdash \langle e_2,\ s'\rangle \Rightarrow \langle e_2',\ s''\rangle}{\Gamma \vdash \langle e_1\ ;;\ e_2,\ s\rangle \Rightarrow \langle e_2',\ s''\rangle}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle}{\Gamma \vdash \langle n ::= e,\ s\rangle \Rightarrow \langle \mathit{unit},\ (l(n \mapsto v),\ m)\rangle}\ \textsc{Assign}$$

$$\frac{\Gamma\ n = \lfloor T \rfloor \wedge \mathit{instance}\ m\ T = \lfloor (m',\ v) \rfloor}{\Gamma \vdash \langle \mathit{create}\ n,\ (l,\ m)\rangle \Rightarrow \langle \mathit{unit},\ (l(n \mapsto v),\ m')\rangle}\ \textsc{Create}$$

$$\frac{\Gamma\ n = \lfloor T \rfloor \wedge \mathit{instance}\ m\ T = \mathit{None}}{\Gamma \vdash \langle \mathit{create}\ n,\ (l,\ m)\rangle \Rightarrow \langle \mathit{Exception},\ (l,\ m)\rangle}\ \textsc{Create}_{fail}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ s_e\rangle \wedge v \neq \mathit{Void}_v \wedge \Gamma \vdash \langle es,\ s_e\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Value}\ vs,\ s'\rangle}{\Gamma \vdash \langle e\ .\ f\ (es),\ s\rangle \Rightarrow \langle \mathit{unit},\ s'\rangle}\ \textsc{Call}$$

$$\frac{\Gamma \vdash \langle b,\ s\rangle \Rightarrow \langle \mathit{True}_c,\ s'\rangle \wedge \Gamma \vdash \langle e_1,\ s'\rangle \Rightarrow \langle e_1',\ s''\rangle}{\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end},\ s\rangle \Rightarrow \langle e_1',\ s''\rangle}\ \textsc{If}_{True}$$

$$\frac{\Gamma \vdash \langle b,\ s\rangle \Rightarrow \langle \mathit{False}_c,\ s'\rangle \wedge \Gamma \vdash \langle e_2,\ s'\rangle \Rightarrow \langle e_2',\ s''\rangle}{\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end},\ s\rangle \Rightarrow \langle e_2',\ s''\rangle}\ \textsc{If}_{False}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{True}_c,\ s'\rangle}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s\rangle \Rightarrow \langle \mathit{unit},\ s'\rangle}\ \textsc{Loop}_{True}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{False}_c,\ s_e\rangle\ \wedge \Gamma \vdash \langle b,\ s_e\rangle \Rightarrow \langle \mathit{unit},\ s_c\rangle \wedge \\ \Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s_c\rangle \Rightarrow \langle c',\ s'\rangle\end{array}}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s\rangle \Rightarrow \langle c',\ s'\rangle}\ \textsc{Loop}_{False}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle \wedge v \neq \mathit{Void}_v \wedge v\ \mathit{has\_type}\ T}{\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n,\ s\rangle \Rightarrow \langle \mathit{True}_c,\ (l(n \mapsto v),\ m)\rangle}\ \textsc{Test}_{True}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle \wedge \neg\ (v \neq \mathit{Void}_v \wedge v\ \mathit{has\_type}\ T)}{\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n,\ s\rangle \Rightarrow \langle \mathit{False}_c,\ (l,\ m)\rangle}\ \textsc{Test}_{False}$$

$$\frac{}{\Gamma \vdash \langle [],\ s\rangle\ [\Rightarrow]\ \langle [],\ s\rangle}\ \textsc{Arg}_{Nil} \qquad \frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ s_e\rangle \wedge \Gamma \vdash \langle es,\ s_e\rangle\ [\Rightarrow]\ \langle es',\ s'\rangle}{\Gamma \vdash \langle e \cdot es,\ s\rangle\ [\Rightarrow]\ \langle \mathit{Value}\ v \cdot es',\ s'\rangle}\ \textsc{Arg}_{Cons}$$

Fig. 13. Big-step semantics: regular cases

An object test evaluates to $True$ if its object test expression is attached and is of an expected type ($\textsc{Test}_{True}$). In that case the local storage is updated for an object test local to have a computed value. The specification uses an abstract function has_type that is not instantiated. Therefore, all the proofs do not depend on the actual run-time type check.

If any of the conditions is not met, e.g. a value is void or its type is not expected, the state is not changed and the object test evaluates to $False$ ($\textsc{Test}_{False}$).

Note that there is only one (non-exceptional) rule for a feature call. This is the major difference from traditional big-step semantics specifications. What if a target of a call will be $Void$? Would not it mean that excution may be stuck? The answer is given in section 5.3.

Exception propagation rules (figure 14) are not different from the rules used in type soundness proofs.

$$\frac{}{\Gamma \vdash \langle \mathit{Exception},\ s\rangle \Rightarrow \langle \mathit{Exception},\ s\rangle}\ \textsc{Exception} \qquad \frac{\Gamma \vdash \langle e_1,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle e_1\ ;;\ e_2,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{Seq}_{ex}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle n ::= e,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{Assign}_{ex}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle e\ .\ f\ (es),\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{Call}_{ex}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ s_e\rangle \wedge \Gamma \vdash \langle es,\ s_e\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Value}\ vs\ @\ (\mathit{Exception} \cdot es'),\ s'\rangle}{\Gamma \vdash \langle e\ .\ f\ (es),\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{Call}_{Arg-ex}$$

$$\frac{\Gamma \vdash \langle b,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end},\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{If}_{ex}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle \mathit{attached}\ t\ e\ \mathit{as}\ n,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{Test}_{ex} \qquad \frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle e \cdot es,\ s\rangle\ [\Rightarrow]\ \langle \mathit{Exception} \cdot es,\ s'\rangle}\ \textsc{Arg}_{ex}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{Loop}_{ex}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{False}_c,\ s_e\rangle \wedge \Gamma \vdash \langle b,\ s_e\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s\rangle \Rightarrow \langle \mathit{Exception},\ s'\rangle}\ \textsc{Loop}_{False\text{-}ex}$$

Fig. 14. Big-step semantics: exception propagation

Conventionally the big-step semantics is shown to end up for a given expression in a final state, meaning an exception or a value.

Definition 5.1 (Final expression)**.** *An expression is called final if it is an exception or a value:*

$$Final\ e \equiv e = Exception \vee \exists v.\ e = Value\ v$$

Lemma 5.1 (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:*

$$\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle e',\ s'\rangle \Longrightarrow Final\ e'$$

5.2 Preservation theorem

Anti-monotonicity of attachment state allows to prove that as soon as a 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 5.2. *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:*

$$A \rhd\!+ c \rhd e_1 \vdash l \sqrt{}_s{}^A \wedge A \rhd\!+ c \rhd e_1 \neq \top \implies A \rhd \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} \vdash l \sqrt{}_s{}^A$$
$$A \rhd\!- c \rhd e_2 \vdash l \sqrt{}_s{}^A \wedge A \rhd\!- c \rhd e_2 \neq \top \implies A \rhd \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} \vdash l \sqrt{}_s{}^A$$

Proof. Follows from the definition of transfer function and lemma 4.11.□

The big-step semantics preserves valid state for both exceptional (denoted by $\top$, see section 3.2.1) and non-exceptional attachment sets (denoted by $\lceil a \rceil$):

Lemma 5.3. $\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle e',\ s'\rangle \wedge \Gamma,\ \top \vdash s \sqrt{}_s \Longrightarrow \Gamma,\ \top \vdash s' \sqrt{}_s$

Lemma 5.4. $\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle e',\ s'\rangle \wedge \Gamma,\ \lceil a \rceil \vdash s \sqrt{}_s \Longrightarrow \exists b.\ \Gamma,\ \lceil b \rceil \vdash s' \sqrt{}_s$

From the other hand, if a final expression (definition 5.1) is not an exception, an attachment set for the initial expression remains non-exceptional:

Lemma 5.5.

$$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, A \vdash e : T \wedge e' \neq \mathit{Exception} \wedge A \neq \top \Longrightarrow A \rhd e \neq \top$$

Proof. By structural induction on 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.

Lemma 5.6 (Attachment preservation step)**.**

$$\Gamma, A \vdash s \sqrt{}_s \wedge A = \lceil a \rceil \wedge \Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, A \vdash e : T \wedge e' \neq \mathit{Exception} \Longrightarrow$$
$$\exists\, T'.\ \Gamma, A \rhd e \vdash e' : T' \wedge T' \rightarrow_a T$$

Proof. The proof is done by structural induction on big-step semantics predicate and uses lemmas 4.12, 5.1 and 5.2. For every induction case is shows that a state remains valid using lemmas 5.3 and 5.4 and taking into account preservation of non-exceptional attachment set (lemma 5.5) and applying an inductive hypothesis to finish the proof. □

Replacing variables with initial state values, the lemma gives:

Theorem 5.1 (Attachment preservation)**.**

$$\Gamma \vdash \langle e, \varnothing\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma \vdash e : \mathit{Attached} \Longrightarrow$$
$$e' = \mathit{Exception} \vee (\exists\, v.\ e' = \mathit{Value}\ v \wedge v \neq \mathit{Void}_v)$$

5.3 Equivalence of safe and unsafe semantics

Ideally void safety should be a corollary of two theorems: preservation and progress. The third one – determinism – cannot be proved in most concurrent environments, so it is not considered here. Unfortunately a progress theorem cannot be proved with classical big-step semantics because it deals only with final states (lemma 5.1). Therefore, it is impossible to describe intermediate states. At least two options exist:

- Use clocked big-step semantics [20] or similar abstraction that distinguishes between stuck state and divergence (e.g., [1, 19, 21]).
- Use small-step semantics.

The first option is straightforward: the current rules can be adapted to distinguish between stuck state and divergence. 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 a progress theorem directly. 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 branch or exit conditions are evaluated to a boolean value, i.e. no initial or intermediate type information is required to state the rule. For small-step semantics this is not the case: for intermediate

steps to be sound we need to know that these intermediate steps preserve the property that the expression type is boolean.

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 w.r.t. the associated small-step semantics. Assume that the original semantics is specified not taking void safety into account. Then consider a 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 5.1. 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 15). The exception here is the famous NullPointerException.

Safe:
$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_e\rangle \wedge v \neq \mathit{Void}_v \wedge \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle \mathit{map\ Value\ vs}, s'\rangle}{\Gamma \vdash \langle e \mathbin{.} f\ (es), s\rangle \Rightarrow \langle \mathit{unit}, s'\rangle}\ \textsc{Call}$$

Unsafe:
$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow' \langle \mathit{Value}\ v, s_e\rangle \wedge v \neq \mathit{Void}_v \wedge \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]'\ \langle \mathit{map\ Value\ vs}, s'\rangle}{\Gamma \vdash \langle e \mathbin{.} f\ (es), s\rangle \Rightarrow' \langle \mathit{unit}, s'\rangle}\ \textsc{Call}^{unsafe}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow' \langle \mathit{Value}\ v, s'\rangle \wedge v = \mathit{Void}_v}{\Gamma \vdash \langle e \mathbin{.} f\ (es), s\rangle \Rightarrow' \langle \mathit{Exception}, s'\rangle}\ \textsc{Call}^{unsafe}_{fail}$$

Fig. 15. Feature call rule in safe and unsafe big-step semantics

It turns out that if an expression is null-safe, it gives exactly the same result regardless of the semantics behind. This effectively demonstrates absence of NullPointerException in null-safe programs.

Theorem 5.2 (Semantics equivalence)**.** *Void-safe ($\Rightarrow$) and void-unsafe ($\Rightarrow'$) semantics of a void-safe program with an initial void-safe state are equivalent:*

$$\Gamma \vdash e \sqrt{}_e \wedge \Gamma \vdash s_0 \sqrt{}_s \Longrightarrow \Gamma \vdash \langle e, s_0\rangle \Rightarrow' \langle v, s\rangle = \Gamma \vdash \langle e, s_0\rangle \Rightarrow \langle v, s\rangle$$

5.4 Practical results

The core part of the local code analysis described in the paper is implemented in 19 Eiffel classes of about 2.5KLOC in total. Instead of immutable attachment sets it uses mutable ones, optimized with bitwise operations. Branching instructions, such as loops, and conditional instructions and expressions, share the same code base that explains a form of the loop transfer function (section 4.1). All code is open source and is available at https://dev.eiffel.com/Source_Code.

First void safety checks were added to Eiffel in *EiffelStudio 6.3* at the time when public libraries almost reached a million lines of code. Migration of public libraries took several releases and is still an ongoing work for few remaining libraries that are not completely void-safe.

The current work suggests relaxed rules for local variable declarations and **Result** where no special type annotations are required. All attachment information is derived by static code analysis. This analysis was implemented in [6]. In contrast to previous releases, no changes to public libraries were required. This confirms theoretical soundness of the approach in practice. The rules also remove unneeded annotation burden from programmers and allow for simpler code.

All theories code is proved with *Isabelle 2016* and is available at https://bitbucket.org/kwaxer/void_safety/ (tag 1.2.2).

6. Related work

Three key ingredients of void safety – a type system that allows for specifying attachment properties, certified attachment patterns that allow for expressions of otherwise detachable type to be used when attached values are expected, and validity rules for safe reattachment and initialization – were already used by [4] in 2006. Research efforts were then mostly focused on making sure the typing rules do ensure soundness: [15] explains why the rules work, [2] reports issues with object initialization in a naïve implementation, [22] proposes a solution that requires introduction of new concepts into the type system to represent partially initialized objects with explicit additional annotations, [14] discusses how additional annotations can be avoided if strict modularity is not required. None of the papers went into the details of usability and user experience, inspecting if the rules are reasonable for real-life cases. Nor did the papers discuss how to perform the development of large systems where a complete set of classes is not readily available. This work addresses the first issue by proposing a set of rules for local variables and demonstrates that all the required information for them can be derived from the code. The second issue is solved by changing validity rules to respect exceptional behavior at run-time.

Type system soundness of conventional object-oriented languages became a hot research area with release of Java that claimed to be absolutely type-safe (cf. [10] that explicitly states undefined behavior in certain cases). [12] 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. [13] updated the proof to use small-step semantics instead and formalized Java memory model. The current work focuses on void safety rules introduced in Eiffel. Its concurrency model is quite different from Java. Even though [16] formalized the semantics in Maude, its correctness is not formally proved. So, the work uses big-step semantics to describe and to reason about void safety guarantees.

An algorithm to compute a set of attached variables might seem to be quite similar to definite assignment rules of [7] and formalized by [12]. 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 [12, 13] 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 an attached or detachable variable can be used as a source or as a target of a reattachment. Finally, described here void safety rules rely on computation of greatest fixed points for loops. This is not needed for definite assignment computation. More similarity with type soundness proofs can be seen in monotonicity of a transfer function, though for void safety this property is also essential for showing that validity rules can be programmatically checked by a compiler.

Leaving concurrency aside, big-step semantics does not distinguish between stuck and diverging states. [20, 21] demonstrate 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 met.

7. 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 w.r.t. big-step semantics. The novelty of the work is in:

- Generalization of attachment rules for boolean operators. (In particular similar scheme can be used to adapt definite assignment rules described in [12] and later used in [13] to prove Java-like type system soundness to match current Java rules [7].)
- Introduction of "design mode" to void safety rules required to analyze real-world programs.
- Specification of void safety rules for loops.
- Demonstration of theoretical and practical uselessness of attachment annotations for local variables.
- Mechanically verified preservation theorem for void-safe programs and conditional equivalence of void-safe and void-unsafe semantics that can be applied to show complete void safety (both for local contexts only).

The work covers only one part of three key elements of a void-safe language. It needs to be extended to deal with the other two, namely:

- Type system – required to show safety with regular feature calls.
- Initialization – required to show safety for object creation when some objects are in an intermediate half-initialized state.

References

[1]. Nada Amin and Tiark Romp. "Type Soundness Proofs with Definitional Interpreters". In: *OOPSLA*. 2016. url: https://www.cs.purdue.edu/homes/ rompf/papers/amin-draft2016a.pdf. Submitted.
[2]. Mike Barnett et al. "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.
[3]. *Common Vulnerabilities and Exposures*. http://cve.mitre.org/. 2016. (Visited on 2016-05-25).
[4]. Ecma International. *ECMA-367: Eiffel analysis, design and programming language*. 2nd. Geneva, Switzerland: Ecma International, June 2006. url: http://www.ecma-international.org/publications/standards/Ecma367.htm.
[5]. Eiffel Software. *EiffelStudio 7.3 Releases*. https://dev.eiffel.com/ EiffelStudio_7.3_Releases. July 2013. (Visited on 2016-05-14).
[6]. Eiffel Software. *EiffelStudio 16.05 Releases*. https://dev.eiffel.com/ EiffelStudio_16.05_Releases. July 2016. (Visited on 2016-09-15).
[7]. James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. isbn: 9780133900699.
[8]. Tony Hoare. "Null references: The billion dollar mistake". In: *Presentation at QCon London* (2009).
[9]. *Projects – Isabelle Community wiki*. https://isabelle.in.tum.de/ community/Projects. Apr. 2016. (Visited on 2016-09-15).
[10]. ISO. *ISO/IEC 14882:2014(E): Information technology — Programming languages — C++*. 4th. Geneva, Switzerland: International Organization for Standardization, Dec. 15, 2014.
[11]. Gerwin Klein. "Verified Java Bytecode Verification". PhD thesis. Institut für Informatik, Technische Universität München, 2003. url: http://www4.in. tum.de/~kleing/diss/.
[12]. 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.
[13]. 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. url: http://digbib.ubka.unikarlsruhe.de/volltexte/1000028867.
[14]. Bertrand Meyer. *Targeted expressions: safe object creation with void safety*. http://se.ethz.ch/~meyer/publications/online/targeted.pdf. July 2012. (Visited on 2016-09-24).
[15]. 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.
[16]. Benjamin Morandi et al. "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. url: http://dx.doi.org/10.1109/ACSD.2013.21.

[17]. Robert Morgan. *Building an Optimizing Compiler*. Newton, MA, USA: Digital Press, 1998. isbn: 1-55558-179-X.
[18]. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. isbn: 1-55860320-4.
[19]. Scott Owens et al. "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: 10.1007/978-3-662-494981_23. url: http://dx.doi.org/10.1007/978-3-662-49498-1_23.
[20]. Jeremy Siek. *Big-step, diverging or stuck?* http://siek.blogspot.ch/2012/07/big-step-diverging-or-stuck.html. July 2012. (Visited on 2016-09-15).
[21]. Jeremy Siek. *Type Safety in Three Easy Lemmas*. http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html. May 2013. (Visited on 2016-09-15).
[22]. 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*. Ed. by . OOPSLA '11. Portland, Oregon, USA: ACM,2011,pp.1013–1032.isbn:978-1-4503-0940-0.doi:10.1145/2048066.2048142.

Автоматическое доказательство безопасности локальных пустых указателей

А.В. Когтенков <kwaxer@mail.ru>
ETH Zürich
8092 Цюрих, Швейцария, Universitätstrasse 19

Аннотация. Разыменование пустого указателя – это хорошо известная ошибка, встречающаяся в объектно-ориентированных программах. Ее можно избежать путем добавления к языку, на котором пишется программа, специальных правил приложимости. Достаточно ли этих правил для гарантии отсутствия таких исключительных ситуаций? Данная статья посвящена безопасности пустых указателей во внутрипроцедурном контексте, в котором не требуются какие-либо дополнительные аннотации. Правила формализуются в системе автоматического доказательства теорем Isabelle/HOL. Затем доказывается теорема о сохранении безопасности пустых указателей в крупношаговой семантике. Наконец, демонстрируется, что при наличии таких правил семантики с безопасностью пустых указателей и без нее эквивалентны.

Ключевые слова: безопасность пустых указателей; статический анализ; Eiffel; формальные методы; крупношаговая операционная семантика; теорема о сохранении; эквивалентность операционных семантик

DOI: 10.15514/ISPRAS-2016-28(5)-2

Для цитирования: Когтенков А.В. Автоматическое доказательство безопасности локальных пустых указателей. Труды ИСП РАН, том 28, вып. 5, 2016, стр.. 27-54 (на английском). DOI: 10.15514/ISPRAS-2016-28(5)-2

Список литературы

[1]. Nada Amin and Tiark Romp. "Type Soundness Proofs with Definitional Interpreters". In: *OOPSLA*. 2016. url: https://www.cs.purdue.edu/homes/ rompf/papers/amin-draft2016a.pdf. Submitted.

[2]. Mike Barnett et al. "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.

[3]. *Common Vulnerabilities and Exposures*. http://cve.mitre.org/. 2016. (Visited on 2016-05-25).

[4]. Ecma International. *ECMA-367: Eiffel analysis, design and programming language*. 2nd. Geneva, Switzerland: Ecma International, June 2006. url: http://www.ecma-international.org/publications/standards/Ecma367.htm.

[5]. Eiffel Software. *EiffelStudio 7.3 Releases*. https://dev.eiffel.com/ EiffelStudio_7.3_Releases. July 2013. (Visited on 2016-05-14).

[6]. Eiffel Software. *EiffelStudio 16.05 Releases*. https://dev.eiffel.com/ EiffelStudio_16.05_Releases. July 2016. (Visited on 2016-09-15).

[7]. James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. isbn: 9780133900699.

[8]. Tony Hoare. "Null references: The billion dollar mistake". In: *Presentation at QCon London* (2009).

[9]. *Projects – Isabelle Community wiki*. https://isabelle.in.tum.de/ community/Projects. Apr. 2016. (Visited on 2016-09-15).

[10]. ISO. *ISO/IEC 14882:2014(E): Information technology — Programming languages — C++*. 4th. Geneva, Switzerland: International Organization for Standardization, Dec. 15, 2014.

[11]. Gerwin Klein. "Verified Java Bytecode Verification". PhD thesis. Institut für Informatik, Technische Universität München, 2003. url: http://www4.in. tum.de/~kleing/diss/.

[12]. 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.

[13]. 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. url: http://digbib.ubka.unikarlsruhe.de/volltexte/1000028867.

[14]. Bertrand Meyer. *Targeted expressions: safe object creation with void safety*. http://se.ethz.ch/~meyer/publications/online/targeted.pdf. July 2012. (Visited on 2016-09-24).

[15]. 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.

[16]. Benjamin Morandi et al. "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. url: http://dx.doi.org/10.1109/ACSD.2013.21.

[17]. Robert Morgan. *Building an Optimizing Compiler*. Newton, MA, USA: Digital Press, 1998. isbn: 1-55558-179-X.

[18]. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. isbn: 1-55860320-4.

[19]. Scott Owens et al. "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: 10.1007/978-3-662-494981_23. url: http://dx.doi.org/10.1007/978-3-662-49498-1_23.

[20]. Jeremy Siek. *Big-step, diverging or stuck?* http://siek.blogspot.ch/2012/07/big-step-diverging-or-stuck.html. July 2012. (Visited on 2016-09-15).

[21]. Jeremy Siek. *Type Safety in Three Easy Lemmas*. http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html. May 2013. (Visited on 2016-09-15).

[22]. 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*. Ed. by . OOPSLA '11. Portland, Oregon, USA: ACM,2011,pp.1013–1032.isbn:978-1-4503-0940-0.doi:10.1145/2048066.2048142.