International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-026-00846-0

GENERAL

Special Issues CSV 2025

Inferring contracts by abstract interpretation with application to pointer nullness analysis

Paul Robert[1,2] · **Matthieu Lemerre**[1] · **Mihaela Sighireanu**[2]

Accepted: 20 February 2026

Abstract

This paper proposes a semantic static analysis for inferring nullable or non-null pointer annotations in low level programs. The analysis is formulated on a minimalistic imperative language and it is expressed as a least fixpoint computation over pointer annotations calling a sound type-checking algorithm. Therefore, this analysis may be used for more general annotations than pointer nullability. We prove two main properties for this approach: (1) when using a sound and precise type-checker (i.e., without false alarms), it will find an annotation that guarantees the absence of run-time errors due to pointer deference, if such an annotation exists, (2) when the type-checker is only sound, the errors singled out by the analysis are real errors. We report on the implementation of this method in Codex, a static analyzer for C and binary code. Codex already provides an abstract interpretation based type-checker for a rich dependent type systems. The evaluation of our implementation on a benchmark of challenging programs shows that the inference is better than manual annotations obtained by code inspection.

Keywords Abstract interpretation · Type checking · Type inference · Pointer analysis · Pointer nullability

1 Introduction

A wide range of low-level programs rely on null values for pointers, to encode special cases or properties of data structures, like the end of a linked list or the absence of a value in an option type. This use of null pointers is the source of numerous bugs due to invalid access to memory by a null pointer dereference, and has been called "the billion dollar mistake" by Hoare [7]. To guarantee that such null-pointer dereference errors do not happen at runtime, the programmer can use static analysis. To minimize false positives and support modular analysis, the static analyzer requires developer-provided guidance. Many programming languages support such annotations to guide their type checking of static analysis, such as `@Nullable` annotations in C# or Java, of the `((nonnull))` attribute in GCC.

However, annotating a large body of code is tedious, where it is easy to make mistakes. Thus, the main goal of this paper is to define an inference method for such annotations. This problem is challenging for several reasons (i) pointer annotations can appear in numerous places (inside type definitions, function arguments and return values), (ii) adding or removing a nullable annotation can lead to new errors being detected in different parts of the code base, and (iii) there may be more than one suitable solution for such annotations.

We propose a solution based on the theory of abstract interpretation, where we view the annotation specification as a lattice. We use a least fixpoint computation to compute the specification maximizing the number of non-null annotations. This algorithm allows to transform a sound type-checking algorithm into a specification inference algorithm which is also a type-inference.

In Sect. 3, we propose a formalization of the problem explaining why a least fixpoint computation can be applied to this problem, despite the fact that type errors do not change monotonically when we ascend the lattice of annotations, and that type-checking may produce false alarms. The main result is that we can turn a precise static analysis (i.e., without false alarms) acting as an oracle for property verification into an inference algorithm able to find a set of annotation that can prove the absence of runtime errors, if such annotations exist. Paradoxically, the inference algorithm can prove absence of null pointer dereference errors by disregarding these errors (and others) when searching for annotations; the inference needs only to concentrate on type-checking errors.

✉ P. Robert
paul.robert@cea.fr

[1] CEA, LIST, Université Paris-Saclay, Palaiseau, France

[2] LMF, ENS Paris-Saclay, Université Paris-Saclay, Gif-sur-Yvette, France

Published online: 16 March 2026

```
struct node {
  struct node* next; //p1
  struct foo* data; //p2
};

struct foo {
  struct node* container; //p3
  int num;
};

struct node* //p4
  cons(int num, struct node* tail); //p5
int main(void);
```

Listing 1 Interface of a list-manipulating program

```
1  struct node* cons(int num, struct node* t) {
2    struct node* n = malloc(sizeof(struct node));
3    struct foo*  d = malloc(sizeof(struct foo));
4    assert(n != 0); assert(d != 0);
5    d->container = n; d->num = num;
6    n->next = t; n->data = d;
7  }
8
9  int main() {
10   struct node* list = 0;
11   for (int i = 0; i < 100; i++){
12     list = cons(i*i,list);
13   }
14 }
```

Listing 2 A list-manipulating program

This formalization is not specific to C or to the nullness inference problem, and may be applied to the inference of other type refinements and other typed languages.

In Sect. 4, we describe our implementation of this method for nullness inference in C and binary programs. For this, we rely on the Codex [2], which includes a type-checker for a rich dependent type system described in [15]. Interestingly, the type-checker is also based on abstract interpretation.

Section 5 reports on the results obtained by applying our analysis on a benchmark of C problems involving complex data structures. Starting from specifications where the pointer nullness is unknown, we infer automatically specifications with nullness pointer annotations, such that the number of nullable pointers is minimal while all pointer assignments type-check. The specifications inferred are usually more precise than manually written specifications based on code inspection.

We compare our method with existing approaches in Sect. 6 and we conclude by listing some limitations and ideas to palliate them in Sect. 7.

2 Overview

2.1 Inferring pointer nullness

Problem Our general goal is to extend a forward analysis based on abstract interpretation that is able to perform a semantic type checking of C programs [15], into an algorithm able to automatically infer a type specification that minimize the type-checking errors. This paper focuses on a first instance of this goal, which is the automatic inference of nullness annotations for pointer types such that type safety implies memory safety.

Consider the C program on Listing 2, whose interface (header file) is declared in Listing 1.[1] In this code, the `node` structure represents a linked list, where the content of each node (of type `struct foo`) points back to the beginning of the list.

The C type system does not distinguish nullable and non-nullable pointers, which makes it difficult for a modular analysis to assert that a program is safe. For instance, the program `p = cons(0,a); assert(p->data == a);` is safe only if `cons` returns a non-null pointer, and the second argument of `cons` can be null. The first error is a *null-pointer dereference* error (due to reading `p->data`) while the second is a *type-checking error* (if zero is not a value admissible for the second argument).

To make the program type-check, the TypedC type system (used as specification language for types in C programs) distinguishes nullable from non-nullable pointers using the following type expressions:

- $\tau?$ denotes addresses that may be null,
- $\tau+$ denotes addresses guaranteed to be non-null,
- $\tau*$ denotes addresses whose nullability is unspecified.

We distinguish the syntactic placeholder $\tau*$, which marks types targeted for inference in the source code, from the semantic value $\tau?$, which represents a nullable type in the analysis lattice.

Goal Our goal is to automatically infer the nullability of pointers whose nullability is unspecified, i.e., transform in our specification any $\tau*$ into either a $\tau?$ or a $\tau+$, such that the program type-checks and it is proved exempt of undefined behaviour (including null-pointer dereferences).

Listing 3 presents a solution to the problem, as it is inferred by our algorithm. This specification and the program in Listing 2 may be analyzed using the modular type-checking defined in [15] and implemented in Codex. On this example, Codex reports that the program has no null-pointer dereference nor any type-checking error.

Note however that other solutions for this specification are also admissible. For instance, if the type of `data` and `container` fields, as well as the return type of `cons` may be

[1] Codex can actually take as an input a specification in TypedC [15], a rich dependently-typed language, but the fragment of TypedC which is understood by the C compiler suffice for this example.

```
struct node {
  struct node? next; //p1
  struct foo+ data; //p2
};

struct foo {
  struct node+ container; //p3
  int num;
};

struct node+ //p4
  cons(int num, struct node? tail); //p5
int main(void);
```

Listing 3 Program specification allowing to prove absence of undefined behaviour in Listing 2

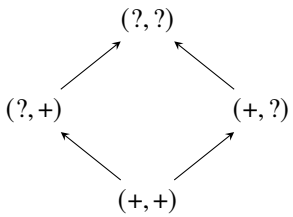

Fig. 1 The annotation lattice for the "`struct node`" fields; the inference starts at the bottom $(+,+)$ and ascends to $(?,+)$ only to fix specific type errors

changed to nullable types, CODEX would still report that the input program has no runtime errors. The nullability of the `data` field has to be specified only if the program is extended with a code that either writes 0 to this field, or dereferences pointers aliasing this field after checking their nullability.

Key ingredient 1: lattice of annotations We propose to solve this problem using an abstract interpretation approach. In particular, we represent pointer nullability as a lattice, essentially mapping each location of the specification where an unspecified pointer $\tau*$ appears (marked by `p1,...,p5` in Listing 1) to its solution, i.e., + for non-nullable, ? for nullable. The final specification corresponds to the fixpoint of a set of recursive equations; this gives a theoretical insight why several solutions may exist.

To illustrate the lattice structure, Fig. 1 depicts the search space for the two pointer fields of the type `struct node`. The analysis starts at the bottom element $(+,+)$, assuming both pointers are non-null. When the type-checker reports a contract violation (e.g., passing `NULL` to the second argument of `cons`), the algorithm relaxes the corresponding annotation by moving up an edge in the graph (i.e., changing the annotation of the second parameter to '?').

2.2 Extracting nullability information from the code

When experimenting with the TYPEDC checker to find specifications of programs that imply their memory safety, we quickly observed that solving this problem manually is a difficult task. Indeed, changing in the specification a pointer from non-null to possibly null (or the other way round) may remove some errors somewhere, but also may introduce new ones elsewhere. As a matter of fact, our type specifications are a contract about the content of a memory location; making a location nullable removes the type-checking errors when trying to write a null value to that location, but increases the set of values that can be read from that location, which opens up new execution paths that can trigger further errors. The fact that the set of errors does not change monotonically when ascending the lattice of annotations, property formalized in Proposition 1, explains why the problem is difficult.

Thus, when trying to manually find a solution, the idea is to be both guided by the errors reported by the type checker, and to react to the following patterns.

`if (p == NULL)`
`{`
`  ...`
`}`
	Explicitly testing for pointer nullness
	This generally indicates that the pointer may be null; otherwise the true branch of the conditional is dead code, which should rarely happen.

`... = (*p)...`	*Dereferencing a pointer* This indicates that the pointer cannot be null at this location, provided that the code is reachable and does not have a bug.

`p->next = 0;`
`g(0);`
`return 0;`
	Explicitly setting a pointer to 0
	Writing 0 to a pointer field, using zero as the value of a pointer parameter, or returning the null pointer, all lead to type checking errors, as the inference initializes the target type to non-null. Thus, assigning 0 creates an immediate contradiction.

Although these patterns are all interesting as heuristics in a manual method, they are not equally informative in a formal tool.

The first pattern is brittle, as there may be some code that tests for pointer nullness in situations where the pointer is never null (e.g., in an assertion checked at runtime, or if the pointer becomes non-null after refactoring). It is only a heuristic and does not provide any certain information.

The second pattern provides useful information, but this information is often difficult to exploit. Consider the following program:

```
1 int f1(int* p, int *q) {
2   if (*p) {
3     int *r = f2(q);
4     return *r;
5   }
6   else return 0;
7 }
8
```

It is complicated to assert that `q` is non-null from the dereference of `r` at line 4, due to the fact that several execution steps happen between the location where `r` is dereferenced and the location (start of function `cons`) where we want to infer a pointer property. Indeed, the only thing that we know is that `f2(q)` must return a non-null pointer when `*p` is true, which is hard to exploit in a modular analysis where neither `f2` nor the calling context is known, and would require a non-trivial trace analysis in any case. The only situation where this information is easy to exploit is when the trace is trivial, e.g., we can indeed conclude that all calls to `f1` must be such that `p` is non-null.

By contrast, exploiting the third pattern is quite easy, and can be done using a state-based analysis. For function calls and returns, it suffices to see if zero reaches an argument or return value whose type is non-null to observe that there is an error, and that this type has to be changed. It is more complicated for memory writes as we also have to track the target locations of pointers (e.g., to which fields a pointer may point), but a state-based analysis also suffices.

Key ingredient 2: support inference decisions solely on type checking errors We detect abstract execution traces that write null values to locations currently assumed to hold non-null values, and use this information to relax the corresponding annotations. Note that this verification-driven approach contrasts with standard constraint-based inference, such as the type reconstruction for machine code proposed by [16] or inference for low-level liquid types [14]. While such methods derive global systems of constraints (equations) directly from the syntactic structure of the program, our approach is driven by the semantics of the program as computed by the abstract interpreter. This allows us to leverage complex properties discovered by the analyzer—such as path sensitivity and relational invariants (e.g., a pointer is non-null only when a length variable is positive)—which are difficult to capture in purely syntactic constraint systems.

2.3 Summary of the method

Given the two key ingredients presented above, we can now provide a full overview of our solution. The idea is to compute a least fixpoint over the lattice of annotations, where each fixpoint step consists in changing the type annotations for which there is a type error.

Example Let us illustrate how the inference works on the example of Listings 1 and 2. As we are performing a least fixpoint computation using a Kleene iteration, we start the analysis with the bottom element of the lattice, in which each pointer annotation of the type specification is set to '+'. We first try to type-check both functions: there are no errors reported when analyzing the `cons` function. However, in the body of the `main` function (lines 11–12), the first iteration of the `for` loop calls `cons` with a null value for the second argument, raising a type error. To fix this error, we need to update the specification of the `cons` function to `struct node+ cons(int num, struct node? tail);` and to rerun the analysis with the new specification.

This time, analysing `cons` yields a type error: the argument `t` being annotated by '?', and the field `node->next` being annotated by '+', we would assign a possibly-null pointer to a non-null field. Again, we update the specification of the `struct node` type such that its `next` field has type `struct node?`.

Finally, we can type check the program once again, and no type errors are reported, meaning that we have reached a fixpoint. The result lattice corresponds to the specification in Listing 3.

Rationale While our type inference algorithm is a simple Kleene iteration on a simple lattice, the reasons why it works are more subtle. In particular, the number of errors is not monotonically decreasing when we ascend the lattice.

What is monotonically increasing when we ascend the lattice is the interpretation of types: a pointer $\tau?$ admits more values than a pointer $\tau+$. In turn, this leads to an increasing number of admissible execution traces. For instance, the execution trace for the `cons` function when the second argument is 0 does not exist when the argument is non-null.

Thus, our inference starts from the smallest possible set of execution traces and iteratively finds the traces that lead to a type-checking error corresponding to a contract violation for a pointer. Due to the monotonic increase of the set of traces, these type-checking errors cannot disappear unless we enlarge the interpretation of the pointer type. Thus, each step minimally increments the annotation in the lattice to get rid of all the typing errors due to non-nullable pointers that have been found, on the minimal set of traces that have to exist. When a fixpoint is reached, the algorithm has converged on the specification with a minimal number of nullable pointers (and leading to a minimal set of traces).

The algorithm remains sound when the type-checking is performed by an abstract interpreter. However, the optimality of the inferred annotation is relative to the precision of this underlying analyzer. If the analyzer reports a false alarm (due to over-approximation), our algorithm must treat it as a potential error and relax the annotation to '?' to ensure soundness. Thus, the inference yields the precise minimal

specification only if the underlying analyzer is precise for the program under analysis.

Limitations There are two main limitations of the method described above.

A first limitation has been pointed-out above: the algorithm is sensitive to the precision of the type-checking algorithm. If the type-checking issues a false alarm (thinking that zero reaches a non-null pointer when it is not the case), then correcting this false alarm will possibly wrongly increase the set of traces, leading to further false alarms.

A second limitation is that the solution found by the algorithm is the smallest solution, which may not be the desired solution, in particular for incomplete codes like libraries. Consider the following listing for a library of single linked lists, containing two functions: `add`, which adds a cell in front of a list, and `tail`, which returns the tail of the input list if it is non-null and zero otherwise.

```
struct list {
  int data;
  struct list *next;
};
struct list* add(int x, struct list* l);
struct list* tail(struct list* l);
```

```
1  struct list* add(int x, struct list* l){
2    struct list* l2 = malloc(sizeof(struct list));
3    l2->data = x;
4    l2->next = l;
5    return l2;
6  }
7  struct list* tail(struct list *l){
8    if (l==0) return 0;
9     else     return (l->next);
10 }
```

The bottom element of our lattice, where each pointer is annotated as non-null (‘+’), is the least fixpoint solution: none of the functions are called with a null list, and the `next` field is never explicitly set to null. Finally, the first branch in the body of `tail` is considered to be dead-code since the variable `l` is set to non-null by the specification in this iteration.

However, the expected solution for such a library could be a larger specification (which is also a fixpoint), such as:

```
struct list {
  int value;
  struct list? next;
};
struct list+ add(int x, struct list? l)
struct list? tail(struct list? l)
```

One should note that this problem does not happen when analyzing complete programs, i.e., the library together with the code using it. In this case, computing a fixpoint that is smaller than the one expected indicates that the client code requires a stronger contract than the one normally expected in the library, and that consequently some code in the library is dead and can be optimized away.

A practical work-around of this limitation is inferring the nullability only for those pointers that are left unspecified (instead of all pointers), i.e., pointers denoted by ‘$\tau*$’. In this case, it is possible for library code to explicitly mention that they are supporting nullable arguments.

3 Inference of type specifications

This section formalises a generic algorithm for type inference by abstract interpretation, that we apply to the inference of C pointer nullness in Sect. 4.1.

The analysis inputs a strongly typed program[2] and a specification mapping type names to their *domain*, i.e., to the set of values that the type accepts. It calls a sound type-checking analysis that signals type errors, and the locations of the program where the errors occur. These type errors are used to refine the specification, i.e., the domain of each type. In our application, the program will be in C or machine code, the specification is written in TypedC, and the sound type-checking analysis is performed by Codex; however, this formalization can be applied to other situations.

3.1 Syntax

We present our analysis on a basic imperative language with strong typing, whose syntax is given in Fig. 2. Although our analysis may be extended to an inter-procedural analysis, we choose for the sake of simplicity to avoid procedural calls in this presentation. We limit the program's control statements to assertions and (conditional) jumps. This language is representative of low-level languages like C and binary code.

A program p is a mapping from program locations (or code's addresses) in $\mathbb{N}$ to statements. The program expressions are obtained by combining variables and constants through binary operators. Assertions and conditional jumps take as a parameter a value considered as false if zero, and true otherwise.

Every type τ used belongs to a fixed universe of types *Typ*. There are two assignment statements: a classic deterministic one (given by an expression) and a non deterministic assignment x:=$\texttt{rcv}_\tau()$, where the left hand-side is assigned to a value of the given type. The latter models the fact that the program "receives" an unknown value of type τ from its environment. The counterpart of $\texttt{rcv}_\tau$ is $\texttt{send}_\tau$, representing

[2] Note that TypedC turns C and machine code into a strongly-typed language whose type safety can be verified automatically.

$x \in \mathit{Vars}$	variables
$k \in \mathbb{V}$	constants
$l \in \mathbb{N}$	program locations
$\tau \in \mathit{Typ}$	types
$\langle e \rangle \in \mathit{Exp}$	expressions
$\diamond$	effect-free binary operators
$\langle \mathit{stmt} \rangle \in \mathit{Stmt}$	statements
$p \in \mathit{Prog}$	programs with $\mathit{Prog} \triangleq [\mathbb{N} \to \mathit{Stmt}]$

$\langle e \rangle ::= x \mid k \mid \langle e \rangle \diamond \langle e \rangle$

$\langle \mathit{stmt} \rangle ::= \texttt{nop}$	*skip*
$\mid \; x := \langle e \rangle$	*assignment*
$\mid \; x := \texttt{rcv}_\tau()$	*set to random value of* τ
$\mid \; \texttt{assert}(\langle e \rangle)$	*asserts that* $\langle e \rangle \neq 0$
$\mid \; \texttt{jmp}(l)$	*unconditional jump*
$\mid \; \texttt{jmpc}(\langle e \rangle, l)$	*jump if* $\langle e \rangle \neq 0$
$\mid \; \texttt{send}_\tau(\langle e \rangle)$	*checks that* $\langle e \rangle$ *is of type* τ

Fig. 2 A strongly typed imperative language

a situation where the program sends a value of type τ to the environment. The execution of this program is correct only if the sent value belongs to the domain of τ.

The $\texttt{rcv}_\tau$ and $\texttt{send}_\tau$ represents the fundamental interactions of a program: $\texttt{send}_\tau$ corresponds to cases where the program emits a value fulfilling the contract that this value has type τ (such as when passing arguments to a function, returning a value, or writing to the heap), while $\texttt{rcv}_\tau$ represents cases where the program relies on the contract that this value has type τ (such as when receiving arguments in a function, obtaining the return value of a function call, or reading from memory).

The semantics of the language is given only in Sect. 3.3, as it depends on the domains of the types, that we formalise in the next section.

3.2 Type specifications and annotations

We are given a set of *type specifications Spec*, as provided in Listings 1 and 3. Given a specification $\theta \in \mathit{Spec}$, we can derive an *interpretation of types* as a set of values, where $(\!|\tau|\!)_\theta \in \mathcal{P}(\mathbb{V})$ represents the domain of type τ in the type specification θ. In TypedC, $(\!|\texttt{struct foo+}|\!)_\theta$ represents the set of addresses that satisfy the abstract invariant of being a valid, non-null location for a `struct foo` value. Then, $(\!|\texttt{struct foo?}|\!)_\theta$ is $(\!|\texttt{struct foo+}|\!)_\theta \cup \{0\}$. Note that the interpretation of the set of values of a type may depend on the annotation ('+' or '?') of another type; for instance, the set of admissible values for a record type depends on the interpretation of the type of each of its fields. Our results do not depend on the interpretation of a specification, so we do not detail them more.

Let us fix a specification θ and $k \geq 1$ type occurrences in θ for which type qualifiers (or refinements) should be inferred. We suppose that type qualifiers belong to a finite set Q for which a lattice $(Q, \sqsubseteq, \sqcup, \bot, \top)$ is defined. For example, the specification given in Listing 1 contains $k = 5$ occurrences of pointer types where the syntactic placeholder '$*$' should be replaced by a precise annotation in $Q = \{+, ?\}$ where $+ \sqsubseteq ?$.

We define the *annotation lattice* $\mathbb{A}$ as the set of mappings from these k positions to the set of qualifiers Q. The order relation over $\mathbb{A}$, denoted $\preccurlyeq$, is a point-wise extension of $\sqsubseteq$: for two annotations $a, a' \in \mathbb{A}$, we have $a \preccurlyeq a'$ if and only if $a(i) \sqsubseteq a'(i)$ for all positions $i \in [1..k]$. The bottom element $\bot_\mathbb{A}$ maps all positions to $\bot$ in Q. For the sake of simplicity, we represent these mappings as k-tuples where the i-th element corresponds to the i-th type occurrence to be annotated in the depth-first traversal order of the syntax-tree of θ. For instance, the annotation corresponding to the specification in Listing 3 is represented as $\langle ?, +, +, +, ? \rangle$.

Finally, we define the *instantiation function* $\hat{\cdot} : \mathbb{A} \to \mathit{Spec}$, which applies an annotation $a \in \mathbb{A}$ to the template specification θ, i.e., it qualifies the type corresponding to position i by $a(i)$.

We require that the partial order $\preccurlyeq$ is compatible with the interpretation of types in the corresponding specification; i.e., that a larger annotation leads to a larger definition of the set of values for every type.

Hypothesis 1
$\forall a_1, a_2 \in \mathbb{A} : a_1 \preccurlyeq a_2 \Rightarrow \forall \tau \in \theta : (\!|\tau|\!)_{\hat{a}_1} \subseteq (\!|\tau|\!)_{\hat{a}_2}$

In our instantiation of this model, i.e., for $Q = \{+, ?\}$, the order $\preccurlyeq$ is compatible with the interpretation of types, as changing a pointer type from non-null to possibly-null increases the set of admissible values for each type (assuming that the specification does not contain contra-variant type constructors, like function parameters).

3.3 Semantics

The semantics of our language is given in Fig. 3. A program state s is a pair of a program location $l \in \mathbb{N}$ and a store σ mapping program variables to values, i.e., $\sigma : \mathbb{X} \to \mathbb{V}$. We denote by Σ the set of all stores σ and by $\sigma[x \leftarrow v]$ the mapping behaving as σ except for x, which is mapped to v. The semantics of expressions $[\![e]\!](\sigma)$ is as expected. The interpretation of statements is also standard: the interpretation is non-deterministic (and thus returns a set of possible output states), or can lead to an error in the set Ω.

The most interesting property of our semantics is that it depends on the input specification θ. This dependency is explicit in the semantics of statements that refer to types. The set of output states for a statement $x := \texttt{rcv}_\tau()$ increases when the type τ has a larger domain, as the new values

$$
\begin{aligned}
[\![.]\!] &: \mathit{Exp} \to \Sigma \to \mathbb{V}\\
[\![.]\!]_\theta &: \mathit{Stmt} \to \Sigma \times \mathbb{N} \to \mathcal{P}(\Sigma \times \mathbb{N}) \cup \Omega\\
[\![\mathit{stmt}]\!]_\theta(\sigma,0) &\triangleq \{(\sigma,0)\}\\
[\![\mathtt{nop}]\!]_\theta(\sigma,l) &\triangleq \{(\sigma,l+1)\}\\
[\![x\mathtt{:=}e]\!]_\theta(\sigma,l) &\triangleq \{(\sigma[x \leftarrow v], l+1) \mid v \in [\![e]\!](\sigma)\}\\
[\![x\mathtt{:=rcv}_\tau()]\!]_\theta(\sigma,l) &\triangleq \{(\sigma[x \leftarrow v], l+1) \mid v \in (\!|\tau|\!)_\theta\}\\
[\![\mathtt{assert}(e)]\!]_\theta(\sigma,l) &\triangleq \{(\sigma,l+1) \mid [\![e]\!](\sigma) \neq 0\}\\
&\quad \cup \{\omega_{l,\sigma} \mid [\![e]\!](\sigma) = 0\}\\
[\![\mathtt{jmp}(l')]\!]_\theta(\sigma,l) &\triangleq \{(\sigma,l')\}\\
[\![\mathtt{jmpc}(e,l')]\!]_\theta(\sigma,l) &\triangleq \{(\sigma,l') \mid [\![e]\!](\sigma) \neq 0\}\\
&\quad \cup \{(\sigma,l+1) \mid [\![e]\!](\sigma) = 0\}\\
[\![\mathtt{send}_\tau(e)]\!]_\theta(\sigma,l) &\triangleq \{(\sigma,l+1) \mid [\![e]\!](\sigma) \in (\!|\tau|\!)_\theta\}\\
&\quad \cup \{\omega_{\tau,l,\sigma} \mid [\![e]\!](\sigma) \notin (\!|\tau|\!)_\theta\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Tr}_\theta &: \mathit{Prog} \to \Sigma \to \mathcal{P}(\mathbb{S}^c) \uplus \mathcal{P}(\mathbb{S}^e)\\
\mathbb{S}^c &\triangleq (\Sigma \times \mathbb{N})^\infty \qquad \textit{correct traces}\\
\mathbb{S}^e &\triangleq (\Sigma \times \mathbb{N})^* \times \Omega \qquad \textit{incorrect traces}\\
\mathit{Tr}_\theta(p)(\sigma_1) &\triangleq \{(\sigma_i,l_i)_{i\in\mathbb{N}^+} \mid l_1 = 1 \wedge\\
&\qquad \forall i \geq 1 : (\sigma_{i+1},l_{i+1}) \in [\![p(l_i)]\!]_\theta(\sigma_i,l_i)\}\\
&\quad \cup \{\big((\sigma_i,l_i)_{i\in[1..n]},\omega\big) \mid l_1 = 1 \wedge \omega \in \Omega \wedge\\
&\qquad \forall i \in [1..n-1] : (\sigma_{i+1},l_{i+1}) \in [\![p(l_i)]\!]_\theta(\sigma_i,l_i) \wedge\\
&\qquad \omega \in [\![p(l_n)]\!]_\theta(\sigma_n,l_n)\}
\end{aligned}
$$

Fig. 3 Semantics of the example language

can be assigned to x. The set of output states for $\mathtt{send}_\tau(e)$ changes when the type τ has a larger domain, as values previously yielding a type-checking error become accepted, which allows the execution to continue. The effect of the specification on the interpretation of these statements is at the heart of our inference method.

The error generated by a failing `assert` statement at program location l for a store σ is denoted by $\omega_{l,\sigma}$; the set of all such errors is denoted $\Omega_{\mathtt{assert}}$. The error generated by a failing $\mathtt{send}_\tau(e)$ statement at location l and for a store σ is denoted by $\omega_{\tau,l,\sigma}$; the set of such errors is denoted $\Omega_{\mathtt{send}}$. The set of errors Ω is the disjoint union of the previous sets, i.e., $\Omega \triangleq \Omega_{\mathtt{assert}} \uplus \Omega_{\mathtt{send}}$.

To simplify our proofs, we consider that correct computations are infinite; for this, we suppose that terminating programs jumps to location 0, where the semantics interprets the statement as a jump to 0. We thus partition the set of execution traces in two subsets: the subset $\mathbb{S}^c$ of infinite correct traces, and the subset $\mathbb{S}^e$ of incorrect traces, that are finite and lead to an error $\omega \in \Omega$.

We define the *collecting type checking* of a program p as the collection of errors on every execution trace of p. Formally:

$$\mathit{Typchk} : \mathit{Prog} \to \mathit{Spec} \to \mathcal{P}(\Omega)$$

$$\mathit{Typchk}(p)(\theta) = \{\omega \mid \exists t \in (\Sigma \times \mathbb{N})^* : (t,\omega) \in \mathit{Tr}_\theta(p)(\sigma_1)\}$$

```
1. x := rcv_τ1();
2. jmpc(x ≠ 0, 4);
3. send_τ2(0);
4. send_τ1(0);
5. assert(0);
6. jmp(6)
```

Fig. 4 Nullness type program

3.4 Properties

We now provide important properties relating the input specification, its annotations and the execution of the program specified. We will illustrate these properties using the program p on Fig. 4. This program employs two types τ_1 and τ_2 representing pointer types to the annotated in the input specification θ; the position of type τ_i is i.

When executing p under the specification obtained by instantiating $\bot_{\mathbb{A}} = \langle +, + \rangle$, the statement at line 4 produces a type error: since the type τ_1 is annotated with '+', 0 cannot be sent.

If we increase the annotation to $a_1 = \langle ?, + \rangle$ to fix the error produced at line 4, two things happen. First, the traces that reached line 4 now continue instead of returning an error; this leads to an `assert` error on line 5. Second, the $s := \mathtt{rcv}_{\tau_1}()$ statement on line 1 produces a trace where $\sigma(x) = 0$, and the condition on line 2 is false for this trace, making the statement on line 3 reachable; this leads to a `send` error on line 3, since τ_2 is annotated with '+'.

This example demonstrates that whilst increasing the annotation may eliminate some existing $\Omega_{\mathtt{send}}$ errors, the `rcv` statements may also produce traces that reach new statements and thus may introduce new (`assert` or `send`) errors. Therefore:

Proposition 1
Let $p \in \mathit{Prog}$ and $a_1, a_2 \in \mathbb{A}$ such that $a_1 \preccurlyeq a_2$. In general, $\mathit{Typchk}(p)(\hat{a}_1)$ and $\mathit{Typchk}(p)(\hat{a}_2)$ are not comparable.

While the number of errors, or locations where these errors appear, do not monotonically increase or decrease, the set of correct traces (that never encounter any error and are thus infinite) increases when the annotations increase.

Proposition 2
Let $p \in \mathit{Prog}$ and $a_1, a_2 \in \mathbb{A}$. If $a_1 \preccurlyeq a_2$ then $\mathit{Tr}_{\hat{a}_1}(p)(\sigma) \cap \mathcal{P}(\mathbb{S}^c) \subseteq \mathit{Tr}_{\hat{a}_2}(p)(\sigma) \cap \mathcal{P}(\mathbb{S}^c)$.

The above property follows from the following lemma:

Lemma 1
Let $\mathit{stmt} \in \mathit{Stmt}$, $l \in \mathbb{N}$, $\sigma \in \Sigma$ and $a_1, a_2 \in \mathbb{A}$. If $a_1 \preccurlyeq a_2$ then:

$$[\![\mathit{stmt}]\!]_{\hat{a}_1}(\sigma,l) \setminus \Omega \;\subseteq\; [\![\mathit{stmt}]\!]_{\hat{a}_2}(\sigma,l) \setminus \Omega$$

Proof
The proof is by case analysis over *stmt*. The only statements capable of generating type errors, and thus relevant to the proof are $x := \texttt{rcv}_\tau()$ and $\texttt{send}_\tau(e)$. There, the proof directly stems from Hypothesis 1. □

Furthermore, an incorrect trace t is either preserved or it ends with an error in $\Omega_{\texttt{send}}$ produced by a $\texttt{send}_\tau(e)$ statement and τ is chosen to increase its type qualifier. In this case, the interpretation of τ will have a larger domain, so the trace t could be reproduced and it executes again the $\texttt{send}_\tau(e)$ statement; this may lead to a correct trace or to a trace that ends with another error. Formally:

Proposition 3
Let $p \in Prog$ and $a_1, a_2 \in \mathbb{A}$ such that $a_1 \preccurlyeq a_2$. Then:

$$\forall t \in Tr_{\hat{a}_1}(p)(\sigma) \cap \mathcal{P}(\mathbb{S}^e) : \\ \exists t' \in Tr_{\hat{a}_2}(p)(\sigma), t \text{ is a prefix of } t'.$$

Proof
Given $t \in Tr_{\hat{a}_1}(p)(\sigma) \cap \mathcal{P}(\mathbb{S}^e)$, we may express t as a finite sequence of size $n \in \mathbb{N}$ of states leading to an error, i.e., $t = ((\sigma_i, l_i)_{i \in [1..n]}, \omega)$.

From Lemma 1, and given that the execution in (σ_i, l_i) for $i \in [1..n]$ resulted in a non-erroneous state, we can conclude that $(\sigma_i, l_i)_{i \in [1..n]}$ also exists in $Tr_{\hat{a}_2}(p)(\sigma)$. Now, let us consider the two possible cases for ω:

- $\omega = \omega_{\tau, l_n, \sigma_n}$ with $p(l_n) = \texttt{send}_\tau(e)$. Then, due to Hypothesis 1, either we have both $[\![e]\!](\sigma_n) \notin (\!|\tau|\!)_{\hat{a}_1}$ and $[\![e]\!](\sigma_n) \notin (\!|\tau|\!)_{\hat{a}_2}$, and the trace t still ends with an error under $\hat{a}_2$; or we have $[\![e]\!](\sigma_n) \notin (\!|\tau|\!)_{\hat{a}_1}$ and $[\![e]\!](\sigma_n) \in (\!|\tau|\!)_{\hat{a}_2}$, so the trace under $\hat{a}_2$ no longer ends with an error. Either way, there exists t' in $Tr_{\hat{a}_2}(p)(\sigma)$ that is either equal to t, or continue the execution after $(\sigma_i, l_i)_{i \in [1..n]}$.
- $\omega = \omega_{\texttt{assert}, l_n}$. In this case, the interpretation of $[\![\texttt{assert}(e)]\!]_{\hat{a}_2}(\sigma_n, l_n)$ will not change and will produce the same error $\omega_{\texttt{assert}, l_n}$, so $t \in Tr_{\hat{a}_2}(p)(\sigma)$. □

To summarize, for any program trace that stops with an error in $\Omega_{\texttt{send}}$, increasing a type qualifier in the input specification may only produce a larger set of traces that execute for longer.

3.5 Two distinct classes of errors

Now equipped with our formalization, we can explore the fundamental distinction between elements of $\Omega_{\texttt{assert}}$ and $\Omega_{\texttt{send}}$, briefly discussed in Sect. 2.2. Of course, by definition, the former are produced by an `assert` statement, while the second by a $\texttt{send}_\tau$ statement. Moreover, they also differ in how we can refine an annotation to make the corresponding error disappear.

Let us start with $\Omega_{\texttt{assert}}$ errors. These errors come from `assert` statements, whose semantics cannot be changed by changing the type specification. Thus, the only way to suppress such an error by changing the specification is to remove the problematic trace that leads to this error. Due to the monotonicity of the semantics, this means that we have to restrict the domain of the types (i.e., move down in the annotation lattice $\mathbb{A}$). Note that, even with a concrete execution trace, finding how to update the annotation is non-trivial. For instance, the trace may depend on the interpretation of several types, and there would be no way to know which one is responsible for the error. This is even more difficult when the type checking is done by sound static analysis, as this requires a trace analysis.

A practical way to solve this problem is to perform instead an ascending iteration, starting from the bottom element $\bot_{\mathbb{A}}$ in the lattice of annotations, which corresponds to the smallest set of traces. After each type-checking phase, we try to minimally ascend in the lattice of annotations, to maximize the chance that there are no $\Omega_{\texttt{assert}}$ errors. If we find such an error nonetheless, it means that there is an inconsistency in the program expectation or an imprecision in the static analysis used for type-checking.

Let us now examine an $\Omega_{\texttt{send}}$ error, corresponding to statement $\texttt{send}_\tau(e)$ where e evaluates to a value $v \notin (\!|\tau|\!)_{\hat{a}}$. Such an error may be raised by impossible traces, introduced by some imprecision in the analysis algorithm. Similarly to $\Omega_{\texttt{assert}}$ errors, we should handle such cases by minimal increases in the annotations lattice. The interesting case is when the error is caused by the fact that $(\!|\tau|\!)_{\hat{a}}$ is too small; then we should increase a to a' such that $v \in (\!|\tau|\!)_{\hat{a}'}$. Finding how to do this is much easier: we know the problematic value v and the type τ; then $a' = a[i \leftarrow q']$ where i is the position of τ in a and $q' \in Q$ includes v in its concretization, such that $v \in (\!|\tau|\!)_{\hat{a}'}$. If we perform a type-checking based on abstract interpretation then the set of values $v \notin (\!|\tau|\!)_{\hat{a}}$ may be obtained from the state information and thus may be used to obtain a'.

To summarize, $\Omega_{\texttt{assert}}$ errors can only be solved by decreasing iteration, require a trace analysis where the relation between the error and the problematic part of the specification is difficult to establish. On the other hand, $\Omega_{\texttt{send}}$ errors can also be solved by an ascending iteration, and the relation between the error and the problematic part of the specification is more explicit. In particular, it only requires the inspection of the last state of the execution (and thus only needs a state-based analysis). This explains why an ascending iteration, focusing only on $\Omega_{\texttt{send}}$ errors, is a preferable approach, and that we focus on it for the rest of the paper.

3.6 From type-checking to type inference

We now describe how to turn type-checking into a type inference, by minimal increase steps in the annotation lattice.

In this section, we focus on the ideal (but not implementable in general) case where we can produce the (possibly infinite) set of traces that lead to an error.

Given an annotation $a \in \mathbb{A}$, we can use collecting type-checking to compute $Typchk(p)(\hat{a}) \in \mathcal{P}(\Omega)$, the set of errors raised with the specification $\hat{a}$.

To identify which type qualifiers should be modified, we do not need all the errors: we only need to collect the pairs $(v, \tau) \in \mathbb{V} \times Typ$ of value and type such that $v \notin (\!|\tau|\!)_{\hat{a}}$. For this, we define a helper function $collect_p$ as follows:

$$collect_p : \mathcal{P}(\Omega) \to \mathcal{P}(\mathbb{V} \times Typ)$$
$$collect_p(S) \triangleq \{(\sigma(e), \tau) \mid \exists \omega_{\tau,l,\sigma} \in S : p(l) = \texttt{send}_\tau(e)\}$$

To find the smallest increase to the annotation lattice such that the set of values returned by $collect_p$ is covered, we suppose given a solver function such that:

$$solver : \mathcal{P}(\mathbb{V} \times Typ) \times \mathbb{A} \to \mathbb{A}$$
$$solver(S, a) \triangleq \inf\{a' \in \mathbb{A} \mid a \preccurlyeq a' \wedge \forall (v, \tau) \in S : v \in (\!|\tau|\!)_{\hat{a}'}\}$$

Note that *solver* is well-defined since $\top_{\mathbb{A}}$ (all positions mapped to $\top$ of $\mathcal{Q}$) satisfies the condition of the set.
If $\mathbb{A}$ is a complete lattice, this is equivalently defined as

$$solver(S, a) \triangleq a \sqcup \left(\bigsqcup_{(v,\tau) \in S} \min\{a' \in \mathbb{A} : v \in (\!|\tau|\!)_{\hat{a}'}\} \right)$$

We can now define our transfer function $F_p : \mathbb{A} \to \mathbb{A}$ as:

$$F_p(a) \triangleq solver(collect_p(\Omega_{\texttt{send}} \cap Typchk(p)(\hat{a})), a)$$

This function is used to iteratively fix errors in $\Omega_{\texttt{send}}$ that may be fixed, as stated by the following property.

Lemma 2
The `lfp` F_p *is the smallest element in the annotation lattice whose instantiation fixes all the fixable* `send` *errors.*

Proof
We proceed in three steps to show existence, constraint satisfaction, and minimality.

- **(1) Existence and termination.** By definition of the *solver* function, for any annotation a, the function F_p returns an annotation greater than or equal to a to solve the fixable `send` errors; thus F_p is extensive ($a \sqsubseteq F_p(a)$). Then, any Kleene sequence defined by $a_0 = \bot_{\mathbb{A}}$ and $a_{n+1} = F_p(a_n)$ forms an ascending chain. Since the annotation lattice $\mathbb{A}$ is finite (as a product of finite lattice $\mathcal{Q}$ over a finite set of program locations), this chain has finite height and must stabilize at a fixpoint, denoted as lfp F_p.
- **(2) Constraint satisfaction.** At the fixpoint, $F_p(\text{lfp } F_p) = \text{lfp } F_p$. Since *solver* returns the minimal increment required to satisfy the constraints collected by $collect_p$, reaching a fixpoint implies that the set of collected, fixable errors from $\Omega_{\texttt{send}}$ is empty. In other words, the result satisfies all fixable type constraints.
- **(3) Minimality.** Let a' be any annotation that fixes all the fixable errors in $\Omega_{\texttt{send}}$. We show by induction that $a_n \sqsubseteq a'$ for all $n \in \mathbb{N}$.
  - **Base case:** $a_0 = \bot_{\mathbb{A}} \sqsubseteq a'$ holds trivially.
  - **Inductive step:** Assume $a_n \sqsubseteq a'$. Since a' is a valid annotation (containing no fixable `send` errors), it must satisfy the type constraints for all values sent in the program. Specifically, for any error (v, τ) collected at step a_n, we must have $v \in (\!|\tau|\!)_{\hat{a}'}$ (otherwise a' would not be a valid solution).

    By definition, $a_{n+1} = F_p(a_n)$ is the *least* lattice element greater than a_n that satisfies these constraints. Since a' is one such element greater than a_n, it follows that $a_{n+1} \sqsubseteq a'$.

  Thus, the limit satisfies lfp $F_p \sqsubseteq a'$. □

The above property allows to prove one of the most important results of this paper: if there exists an annotation such that the type-checking is successful, the least fixpoint of F_p computes one of such annotations (as we will see in Proposition 4).

It is worth noting that we prove minimality via explicit induction rather than invoking the Kleene's theorem. As established in Property 1, the set of type errors does not strictly decrease when ascending the lattice, and the transfer function F_p is not guaranteed to be monotonic in the general case. Our manual inductive proof demonstrates that the algorithm nevertheless converges to the least solution by relying on the local precision of the *solver* and the extensivity of F_p, without requiring global monotonicity.

Theorem 1
Given a program p, if there exists an annotation a such that $Typchk(p)(\hat{a}) = \emptyset$ *then* $Typchk(p)(\texttt{lfp } F_p) = \emptyset$.

Proof
If a allows the program to type-check without any error, the program type-checks without any `send` error, i.e., $Typchk(p)(\hat{a}) \cap \Omega_{\texttt{send}} = \{\}$. Thus, a is a fixpoint of F_p.

`lfp` F_p is another fixpoint of F_p, and `lfp` $F_p \sqsubseteq a$, which implies that there are fewer traces corresponding to `lfp` F_p, and thus fewer errors. Given that the traces for $\hat{a}$ do not lead to any errors, so does the traces for `lfp` F_p. □

The above theorem characterises a solution (the minimal solution) to the type inference problem, if it has one. Paradoxically, finding a solution that proves the absence of null pointer dereference requires to disregard the null pointer dereference errors!

3.7 Computing $\texttt{lfp}\ F_p$ in practice

When the set of possible execution traces for a program p is finite, $\texttt{lfp}\ F_p$ can be obtained by a Kleene iteration; note that for lattices of qualifiers of finite height such as the lattice $Q = \{+, ?\}$ of nullness qualifiers, the iteration terminates, without widening, in a finite number of steps. Unfortunately, in most programs the set of possible execution traces is infinite or too large.

Testing methods Note first that there is no need to find all the errors and execution traces to start ascending the lattice of annotations; it suffices to find one concrete trace that leads to an error. This implies that testing methods, like fuzzing, symbolic execution or bounded model-checking, can be used to infer nullness annotations instead of a type-checking analysis. Iterations of this method guarantee to find an annotation a such that $a \sqsubseteq \texttt{lfp}\ F_p$. Notice that all the unfixable errors found when we reach a are then real errors.

Lifting a sound type checking We say that a function $F_p^\sharp : \mathbb{A} \to \mathbb{A}$ is a sound over-approximation of F_p if for all $a \in \mathbb{A}$ we have $F_p(a) \sqsubseteq F_p^\sharp(a)$. Given that $F_p^\sharp$ is computable and $\mathbb{A}$ is a lattice of finite height, then $\texttt{lfp}\ F_p^\sharp$ can be computed by Kleene iteration.

Let us now see how $F_p^\sharp$ can be implemented. We first define an abstract set of $\Omega_{\texttt{send}}$ errors: $\Omega_{\texttt{send}}^\sharp \triangleq \{\omega_{\tau,l,\sigma^\sharp} \mid \tau \in Typ, l \in \mathbb{N}, \sigma^\sharp \in \Sigma^\sharp\}$. Each abstract error represents a set of concrete errors, that is defined using the following concretization function:

$$\gamma_{\Omega_{\texttt{send}}^\sharp} : \Omega_{\texttt{send}}^\sharp \to \mathcal{P}(\Omega_{\texttt{send}})$$
$$\gamma_{\Omega_{\texttt{send}}^\sharp}(\omega_{\tau,l,\sigma^\sharp}) \triangleq \{\omega_{\tau,l,\sigma} \mid \sigma \in \gamma_\Sigma(\sigma^\sharp)\}$$

where $\gamma_\Sigma \in \Sigma^\sharp \to \mathcal{P}(\Sigma)$ represents the concretization of the abstract stores.

A sound type-checking algorithm is a function

$$Typchk^\sharp : Prog \to \mathbb{A} \to \mathcal{P}(\Omega_{\texttt{send}}^\sharp)$$

that over-approximates the collecting type-checking, i.e., for all program p and annotation a:

$$Typchk(p)(a) \subseteq \bigcup \{\gamma_{\Omega_{\texttt{send}}^\sharp}(\omega) \mid \omega \in Typchk^\sharp(p)(a)\}$$

The implementation of $F_p^\sharp$ then consists in finding an (ideally, minimal) increment to an annotation in the lattice that removes all the errors found by $Typchk^\sharp$, i.e., it returns the smallest $a' \in \mathbb{A}$ such that

$$solver\big(collect_p\big(\bigcup \{\gamma_{\Omega_{\texttt{send}}^\sharp}(\omega) \mid \omega \in Typchk^\sharp(p)(a)\}\big)\big) \preccurlyeq a'$$

It is easy to see that, if during the Kleene iteration computing $\texttt{lfp}\ F_p^\sharp$, the computation of $F_p^\sharp$ is precise, then the analysis converges to a solution to the nullness inference problem, i.e., to an annotation whose instantiation allows proving the absence of errors:

Proposition 4
If $\forall n \geq 1 : (F_p^\sharp)^{(n)}(a) = F_p^{(n)}(a)$, *then the analysis converges to* $\texttt{lfp}\ F_p^\sharp = \texttt{lfp}\ F_p$

This means that our inference algorithm succeeds in finding a suitable annotation and proves the absence of errors if (i) such annotation exists, and (ii) the type-checking analysis is sufficiently precise to not spuriously perform increments of annotations.

Note that having $F_p^\sharp(a) = F_p(a)$ does not mean that the analysis is fully exact (e.g., computes the exact set of errors, which is, in general, impossible). It suffices that (i) $F_p^\sharp$ does not raise any false alarms for a type-checking error and (ii) for the type-checking errors raised, it correctly infers how to change the annotation to fix the alarm; the latter condition is relatively easy in the case of nullability annotations. Thus, given a precise analysis that is generally able to prove the absence of type-checking errors when there are none, we obtain an inference algorithm that has a good chance of success.

Note that even if $F_p^\sharp$ is sometimes imprecise, we have that $\texttt{lfp}\ F_p \preccurlyeq \texttt{lfp}\ F_p^\sharp$, and thus $\texttt{lfp}\ F_p^\sharp$ also fixes all the errors in $\Omega_{\texttt{send}}$ that have to be fixed and can be fixed. Thus, even in the case where the type-checking returns an over-approximation, there is a chance that the returned annotation contains no `send` error, which is a necessary condition for finding an annotation leading to no error at all.

3.8 Confluence and monotonicity hypothesis

While the *solver* function may process collected errors in a non-deterministic order, this choice does not affect the precision of the final result under the assumption that the underlying abstract interpreter is monotonic. Using the standard result from [3], any chaotic iteration sequence for a monotonic function converges to the unique least fixpoint, independent of the update order.

In practice, the monotonicity hypothesis is often violated when the underlying analyzer employs widening operators (e.g., in polyhedral or interval domains) to ensure convergence in infinite domains. Because widening is sensitive to the sequence of abstract states, the specific order in which errors are resolved could theoretically alter the widening thresholds, leading to different—though still valid—fixpoints.

Nevertheless, because our annotation lattice for pointer type nullness is finite and our transfer function is extensive (meaning annotations only increase), the termination is unconditionally guaranteed. Thus, even in the presence of non-monotonic widening, the algorithm always converges to a sound annotation that satisfies the type safety requirements.

4 Application to C code analysis

The method formalized has been implemented in CODEX [2], an analyzer for C and binary code. We employ as *Typchk*$^\sharp$ the type-checking analysis for the dependent type system defined in [15]. Whereas the implementation itself is tightly coupled with CODEX, we discuss below some implementation details and improvements that could be used by other analyzers based on our method.

4.1 Analyzing C code

The input language for which we defined the analysis includes only the essential constructs necessary for the formalization of our main results, namely, the interaction of the program with an "external environment" (e.g., other procedures) using a *contract* [12] that limits the set of values that can be exchanged. However, this language is representative of the use of contracts in other languages, such as C.

In our TYPEDC specification language, the contract is specified as type annotations applied on the interface declaring type definitions and function prototypes usable by other compilation units; for instance, Listing 3 is the result of annotating the standard C interface (header) file of Listing 1. The constructs of the C language can be viewed and analyzed as a sequence of constructs of our simplify language. For instance, calling a function `f` with prototype τ_r `f`$(\tau_1, \ldots, \tau_n)$ can be understood as `send`-ing a tuple value with type $\tau_1 \times \cdots \times \tau_n$, and then `rcv`-ing a value of type τ_r.[3] The analysis of the function `f` can be seen as `rcv`-ing a value of type $\tau_1 \times \cdots \times \tau_n$ as argument, and then `send`-ing the return value of type τ_r. Writing a pointer of type τ`+` can be seen as `send`-ing a value of type τ`+`, while dereferencing a pointer can be seen as `rcv`-ing a value of type τ`+`. Dereferencing a pointer for reading and writing can be seen as first `assert`-ing that its value is not 0, then receiving or sending a τ-value. The other C constructs (assignments, conditional jumps) also have standard equivalents in our simplified language.

[3] Note that we send a tuple of arguments instead of sending the arguments one by one. This is due to a subtle problem that appears when using dependent types in functions' specification: the specification may relate the types of different arguments and the type of the result and this relation may be exploited by the analysis. This is why our real analysis does not sequence the analysis of calls into separate `send` and `rcv` phases.

An important point is that the analysis of our formalization is a whole-program analysis over a single program, while the presentation in this section sees each function as a separate program. This only changes the fixpoint computation as the increases on the annotation lattice $\mathbb{A}$ must be done by observing the errors on every function.

4.2 Implementation details

The implementation of the fixpoint computation for the inference of annotations consists of only 200 LOC of OCaml, which is rather modest compared with the type-checker of CODEX.

Although the method presented allows to increase (from '+' to '?') several type qualifiers at once without compromising the precision of the inference, our implementation does only one change by iteration of $F_p^\sharp$. This implementation choice eases the debugging and the logging of annotation changes, but it increases the number of iterations (see Table 1) required to reach the fixpoint. The type-checker of CODEX being very fast, this design choice does not have a negative impact on the time performances of the inference.

An important aspect of C (and binary) code analysis that is simplified in our formal language is the identification of the type being sent (with the `send` function in the formalization). While this is relatively easy in the case of function calls and returns, it is more difficult to identify the type of the pointer being written to in presence of pointer arithmetics and discrimination between variants of a union type [15]. Another difficulty is that the domain for one type may depend on the definition of other types, which makes it harder to identify how to update the annotation. A part of our implementation effort has been devoted to instrument the type-checking to produce typing trees that provide a concise explanation of why an abstract value is not included in the domain of a type. For instance, when writing a record value with 8 fields, it is possible that the error only concerns 2 of these 8 fields; thus, the increment to the annotation should only apply to these 2 types. As a side effect, these typing trees improve the usability of the type-checker, which has become better at explaining its type-checking errors.

5 Evaluation

We evaluated our implementation on the benchmark of programs analyzed by CODEX. We present in Table 1, the results on some case studies of this benchmark, among which are OS kernels (e.g., Contiki [4]) or kernel's fragments (RBTreeLinux), code with complex shapes for data structures (e.g., Graph, RBTree), and programs from standard benchmark for tools checking spatial memory safety (here Olden [11] bench for case studies bisort, health, perimeter and tsp). We selected

Table 1 Experimental results

Case studies	#LoC	#Fun	#Spec lines	#Alarms				#Iter	Δ τ?
				lower	advised	infer	true		
Contiki	329	12	14	16	2	0	0	7	−3
RBTreeLinux	1111	2	17	8	8	8	0	0	0
Graph	155	7	14	2	0	0	0	2	−10
Kennedy	197	6	24	6	0	0	0	1	−1
RBtree	978	7	18	56	16	16	0	0	−3
bisort	356	11	29	9	0	0	0	1	0
health	485	13	57	16	3	3	0	13	−1
perimeter	486	12	41	13	1	1	0	3	0
tsp	617	12	32	3	0	0	0	6	0

these benchmarks because they intensively use pointer types which are difficult to specify manually without a careful code inspection. Table 1 provides the characteristics of the C code for each case study: the number of lines in column `#LoC`, the number of analyzed functions in column `#Fun`. The original specification for each case study has been manually written and includes, in addition to pointer types, dependent types defined in [15] involving numerical predicates, parameterized type and existential quantifiers. The number of lines of existing specifications is given in column `#Spec lines`. In Table 1, we identify three versions of these specifications: "`lower`" where all pointer annotations are '+', "`advised`" where pointer annotations have been manually written after code inspection by experimented users and "`infer`" where pointer annotations have been inferred by our method.

The number of alarms obtained by analyzing the case studies with each version of the specification is given in the respective columns under `#Alarms`. The column `#Alarms`:true identifies the true errors among the alarms obtained when "`infer`" specification is used as input.

Our inference method starts from the "`lower`" specification to benefit from the dependent types specified, so all pointer annotations are set to '+'. Column #Iter provides the number of iterations (equal to the number of annotation changed from '+' to '?' in our implementation) needed to reach a fixpoint. Due to the time performances of the CODEX's type-checking, the time spent is less than 10 seconds for each of these benchmarks on an Intel Ultra 7 165H with 5 GHz laptop.

The difference between the number of qualifiers '?' in the "`infer`" version of the specification and the "`advised`" version is given in column Δ τ?. The results obtained are either 0 (same number of '?') or negative, which means that the inferred annotations are smaller or equal than the manual ones. The number of alarms obtained with the inferred specification is always equal to the one obtained with the advised specification, except for Contiki; all these alarms are false positive.

The Graph case study is interesting because it reveals a limitation of our method. It is a library implementing operations on graphs (building, traversing) and it does not have a main program calling these operations. Therefore, we are not able to infer any meaningful nullness information, whereas an educated user was able to qualify 10 more pointer types as nullable by inspecting the code. This case study illustrates the limitation of the least fixed point approach on an unannotated incomplete program, which starts with all pointers being non-null and thus depends on the calling context for functions to introduce nullable pointer qualifiers.

6 Related work

6.1 Pluggable types and Java inference

A closely related line of work is the *Checker Framework* [13], which introduced pluggable type systems for Java to support nullness checking via annotations. Kellogg et al. [10] extends this work to automatically infer these qualifiers. Our approach shares the lattice-theoretic foundation of these works—ascending the lattice until a fixpoint is reached. We extend this work in several aspects. First, our framework is semantic rather than type-theoretic, allowing us to prove that our collecting inference computes an optimal solution when it exists (unlike Kellogg et al. [10], which prioritizes preserving unspecified qualifiers over finding a minimal fixpoint). Second, our algorithm is analyzer-agnostic: it treats the underlying type verifier as a black box rather than being tied to a specific compiler infrastructure.

Crucially, the differences between Java and C necessitate distinct design choices. A common criticism of variable-based analysis in Java is the lack of context sensitivity for

polymorphic containers (e.g., a single `List` class used everywhere) [13]. In contrast, C systems code is typically monomorphic (e.g., specialized `task_list` vs. `int_node` structs), allowing our variable-based approach to achieve high precision without the complexity of polymorphic analysis. Similarly, while Java requires "Raw" types to safely track fields during object initialization, C treats uninitialized memory as undefined values rather than `null`. Our underlying analyzer handles initialization failures orthogonally, removing the need to model initialization states within the nullability lattice itself.

6.2 Precise semantic analysis

Beyond pluggable types, deep semantic analyses have been proposed to verify null-safety with high precision. Hubert et al. [8] defines a semantic analysis for the simultaneous automatic checking of absence of null pointer dereference (similar to the system of Fähndrich and Leino [6]) and the inference of nullability annotations. Like us, their work is based on program semantics and performs an ascending lattice iteration. Spoto [17] pushes this further by constructing a specialized, monolithic abstract domain (using binary-decision diagrams) to track nullability and aliasing. While these works achieve high precision by building dedicated domains, our work differs in modularity: we provide a meta-algorithm that wraps any existing sound analyzer (treating it as an oracle) to turn it into an inference tool. We extend Hubert et al. [8] by formalizing this independence and studying the integration of descending iterations.

Karimipour et al. [9] developed an automatic inference algorithm for pointer nullness annotation, which also treats the type checker as a black box. Their technique searches for an annotation which minimizes the amount of errors reported by the checker (including null pointer dereferences). Because the search focuses on minimizing the amount of errors, there is no theoretical guarantee that the inferred annotations are correct. Indeed, this work focuses in applicability to large code base, not in guaranteeing the absence of errors.

A variant of the more complex descending iteration method of Sect. 3.5, starting from assertions to infer necessary invariants using a trace analysis, has been proposed by Bouaziz et al. [1]. Their goal is to address usability issue of the ascending method, which is that the location of a failed assertion may not be responsible for the failure. Their analysis can generate necessary preconditions and object invariant that help finding the code responsible for breaking the invariant. However, this method cannot be used to prove the absence of errors if it is possible.

Estep et al. [5] address nullness inference in the framework of gradual typing, where nullability information can be partially inferred statically and partially enforced via runtime checks. Their system refines types with nullability annotations and uses constraint solving to propagate annotations, inserting dynamic checks when uncertainty remains. Like us, their approach can leverage a static type checker implemented as an abstract interpretation. This partially dynamic approach trades completeness for flexibility: it ensures soundness through a hybrid type-checking (static/dynamic combination), while our approach seeks to discharge all nullness obligations statically within an abstract type checker.

7 Conclusion and future work

We formalized a modular method for inferring pointer nullness in low-level programs based on a type-checking analysis over a rich type system. We demonstrated that our method can automatically infer nullability annotations for pointer types that prove the absence of runtime errors if such annotations exist, and if the type-checking analysis is sufficiently precise. The experimental results demonstrate its efficiency in inferring precise specification of challenging C benchmarks.

We think that the method proposed may be applied to other type annotations than pointer nullness due to the generality of our formalization. Extensions to more complex annotations, such as refinement types [18] (refining dependent types using predicates) using existential quantifications, is the subject of future work. The type system supported by our type-checking, Typed C, can express complex relational properties (e.g., in the nodes of a balanced tree, the pointers to both children must be simultaneously null or non-null) which are challenging to infer.

Our method strongly depends on the precision of the type-checking algorithm, which may return false positive alarms and thus forcing the inference algorithm to ascend the annotation lattice. A careful (automatic) analysis of proof trees produced by the type-checking may help to direct the inference to detect the real cause of an error and to ignore the false positive, possibly using automatic methods to refine the analysis precision or using testing methods that could ensure the existence of an type error.

Finally, the main hypothesis of our formalization is that the domains for types increase when we ascend in the lattice of annotations. This hypothesis may not be true in the presence of contravariant type constructors. For instance, functions that accept both non-null and the null pointers can be used when a function that only accepts non-null pointers is expected; thus, the set of functions accepting arbitrary pointer is included in the set of pointers accepting only non-null pointers. More work is needed to formalize and implement a solution that can infer optimal annotations in the presence of contravariant type constructors.

Funding information Open access funding provided by Université Paris-Saclay.

References

1. Bouaziz, M., Logozzo, F., Fähndrich, M.: Inference of necessary field conditions with abstract interpretation. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012, Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 173–189. Springer, Berlin (2012). ISBN 978-3-642-35181-5, 978-3-642-35182-2. https://doi.org/10.1007/978-3-642-35182-2_13
2. CODEX: (2025). https://codex.top
3. Cousot, P., Cousot, R.: Automatic synthesis of optimal invariant assertions: mathematical foundations. ACM SIGPLAN Not. **12**(8), 1–12 (1977)
4. Dunkels, A.: Contiki-os (2023). https://github.com/contiki-ng/contiki-ng/blob/master/os/lib/list.c
5. Estep, S., Wise, J., Aldrich, J., Tanter, É., Bader, J., Sunshine, J.: Gradual program analysis for null pointers. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark, Virtual Conference. LIPIcs, vol. 194, pp. 3:1–3:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPICS.ECOOP.2021.3
6. Fähndrich, M., Leino, K.R.M.: 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, pp. 302–312. Association for Computing Machinery, New York (2003). ISBN 1581137125. https://doi.org/10.1145/949305.949332
7. Hoare, T.: Null references: the billion dollar mistake. In: InfoQ Conference (2009)
8. Hubert, L., Jensen, T., Pichardie, D.: Semantic foundations and inference of non-null annotations. In: Barthe, G., De Boer, F.S. (eds.) Formal Methods for Open Object-Based Distributed Systems, vol. 5051, pp. 132–149. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-68863-1_9. ISBN 978-3-540-68862-4, 978-3-540-68863-1. http://link.springer.com/10.1007/978-3-540-68863-1_9
9. Karimipour, N., Pham, J., Clapp, L., Sridharan, M.: Practical inference of nullability types. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1395–1406. ACM, New York (2023). ISBN 979-8-4007-0327-0. https://doi.org/10.1145/3611643.3616326. https://dl.acm.org/doi/10.1145/3611643.3616326
10. Kellogg, M., Daskiewicz, D., Duc Nguyen, L.N., Ahmed, M., Ernst, M.D.: Pluggable type inference for free. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE, Luxembourg, Luxembourg, Sept. 2023, pp. 1542–1554. IEEE (2023). ISBN 979-8-3503-2996-4. https://doi.org/10.1109/ASE56229.2023.00186
11. Machiry, A., Kastner, J.H., McCutchen, M., Eline, A., Headley, K., Hicks, M.: C to checked C by 3c (with appendices) (2022). CoRR arXiv:2203.13445. https://doi.org/10.48550/arXiv.2203.13445
12. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
13. Papi, M.M., Ali, M., Correa, T.L. Jr., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: Ryder, B.G., Zeller, A. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, pp. 201–212. ACM (2008). https://doi.org/10.1145/1390630.1390656
14. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL, pp. 131–144. ACM, New York (2010). https://doi.org/10.1145/1706299.1706316
15. Simonnet, J., Lemerre, M., Sighireanu, M.: A dependent nominal physical type system for static analysis of memory in low level code. Proc. ACM Program. Lang. **8**(OOPSLA2), 30–59 (2024). https://doi.org/10.1145/3689712
16. Smith, I.: Binsub: the simple essence of polymorphic type inference for machine code. In: Giacobazzi, R., Gorla, A. (eds.) Static Analysis - 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20-22, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14995, pp. 425–450. Springer (2024). https://doi.org/10.1007/978-3-031-74776-2_17
17. Spoto, F.: Precise null-pointer analysis. Softw. Syst. Model. **10**(2), 219–252 (2011)
18. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Appel, A.W., Aiken, A. (eds.) Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '99, San Antonio, TX, USA, January 20-22, 1999, pp. 214–227. ACM (1999). https://doi.org/10.1145/292540.292560