SMT-Based Checking of Predicate-Qualified Types for Scala

Georg Stefan Schmid Viktor Kuncak
EPFL, Switzerland
{firstname.lastname}@epfl.ch

Abstract

We present qualified types for Scala, a form of refinement types adapted to the Scala language. Qualified types allow users to refine base types and classes using predicate expressions. We implemented a type checker for qualified types that is embedded in Scala's next-generation compiler Dotty and delegates constraint checking to an SMT solver. Our system supports many of Scala's functional as well as its object-oriented constructs. To propagate user-provided qualifier ascriptions we utilize both Scala's own type system and an incomplete, but effective qualifier inference algorithm. Our evaluation shows that for a series of examples exerting various of Scala's language features, the additional compile-time overhead is manageable. By combining these features we show that one can verify essential safety properties such as static bounds-checks while retaining several of Scala's advanced features.

Categories and Subject Descriptors D.2.4 [*Software engineering*]: Software/Program Verification

General Terms Languages, Reliability, Verification

Keywords Scala, Refinement Types

1. Introduction

This paper explores the prospect of adding refinement types to Scala to encode a larger class of specifications, preventing a broader class of run-time errors. Examples of run-time errors we aim to prevent include array bounds check exceptions. The key idea is to introduce a new class of types which refine existing Scala types (such as `Int`) using a predicate, such as `x:Int => 0 < x`. Analogously to the situation with sets in interactive proof assistants such as Isabelle [10], a predicate refining type `T` is a terminating Scala function from `T` to `Boolean`. We write such refined types as predicates in curly braces, as in `{x:Int => 0 < x}`. Additionally, we restrict the body of the function so that it can be encoded into a decidable logic supported by the satisfiability-modulo-theory (SMT) solvers we use. Our system aims to have a low annotation overhead by avoiding the need to duplicate a computation sub-language within the type language, and by extracting path conditions automatically from program control-flow to infer refined types for intermediate values.

To illustrate our system, consider the example of binary search. In 2006 Joshua Bloch drew attention [3] to a problem in the implementation of binary search for Java's Array class. Listing 1 shows a Scala implementation that suffers from the same defect. The problem with this encoding lies in the calculation of `i` on line #7.

Listing 1: A faulty implementation of binary search in Scala

```
1  abstract class IntArray(val length: Int, init: Int) {
2    def access(i: Int): Int }
3
4  def binarySearch(arr: IntArray, x: Int): Boolean = {
5    def rec(lo: Int, hi: Int): Boolean =
6      if (lo <= hi) {
7        val i = (lo + hi) / 2
8        val y = arr.access(i)
9        if (x == y)     true
10       else if (x < y) rec(lo, i-1)
11       else            rec(i+1, hi)
12     } else false
13   if (arr.length > 0) rec(0, arr.length - 1)
14   else                false
15 }
```

Listing 2: The example in our system using qualifiers

```
1  type NonNeg = { v: Int => v >= 0 }
2  def range(lo: Int, x: Int, hi: Int) = lo <= x && x <= hi
3
4  abstract class IntArray(val length: NonNeg, init: Int) {
5   def access(i: {v: Int => range(0, v, this.length-1)}):
        Int }
6
7  def binarySearch(arr: IntArray, x: Int): Boolean = {
8    def rec(lo: NonNeg)(hi: {v: Int => range(lo-1, v,
       arr.length-1)}): Boolean =
9      if (lo <= hi) {
10       val i = (lo + hi) / 2
11       val y = arr.access(i)
12       if (x == y)     true
13       else if (x < y) rec(lo)(i-1)
14       else            rec(i+1)(hi)
15     } else false
16   if (arr.length > 0) rec(0)(arr.length - 1)
17   else                false }
```

While the expression `(lo + hi)/2` correctly computes the desired middle index when the integer sum of `lo` and `hi` is up to $2^{31} - 1$, values above cause a negative result due to the bitvector semantics of Scala's `Int`, which follows the semantics of Java [7, Section 4.2.2. Integer Operations]. The halved value remains negative, and once we execute the array access on the subsequent line an `ArrayIndexOutOfBoundsException` is thrown, indicating the real problem.

In our system we can rewrite the example as shown in Listing 2. We add a type alias `NonNeg` (line #1) referring to non-negative integers. (The meaning of arithmetic operations inside qualifiers is given by bitvector arithmetic (just like in the executable code), avoiding unnecessary proliferation of concepts.) We use the type alias to constrain the permissible values of the `IntArray` class' `length` parameter (line #4), but also the `lo` parameter of the recursive search function `rec` (line #8). To statically prove the absence of `ArrayIndexOutOfBoundsExceptions`, we also strengthen the parameter type on the `access` method (line #5): Index `i` is required to be within the bounds of the array, expressed by the qualifier `range(0, v, this.length-1)`, a Scala expression which can be extracted precisely and expands to `0 <= v && v <= this.length-1`. Note how we allow the qualifier to refer to (stable) members of the `IntArray` class, making our `access` method dependent on the object instance. Finally, we change the signature of `rec` (line #8) to constrain the range of the `hi` parameter. Since `hi` must not only be within the array's bounds, but is also lower bounded by `lo-1`, we would like to mention `lo` in its qualifier. Scala only allows such *dependently typed parameters* when they refer to a preceding parameter group, so we split the signature of `rec` in two.

Given these bounds, we can be confident that our program only compiles if no `ArrayIndexOutOfBoundsException` can occur at runtime. Upon type checking the program in listing 2 we are presented with the following error message:

```
IntArray.scala:11: error: constraint violation
  Counterexample:
    hi: 2113929997
    lo: 2113929295
    i: -33554002
    x: 0
      val y = arr.access(i)
                         ^
Qualified type check failed!
```

We can see that a subtyping constraint among two qualified types was violated on line #11. The system provides us with additional information on the violated constraint and a concrete counterexample for the involved variables, leading to an out-of-bounds argument for `i` on line #11. At this point we hopefully realize our mistake and correct the offending computation on line #10, e.g. by setting `i:=lo+(hi-lo)/2`. We then re-run the compiler, which will not report any more errors, and thereby assure us that the array accesses will succeed in all program executions.

1.1 Contributions

This paper presents a prototype implementation of a refinement type system for the Scala language. We make the case that qualified types are not only an effective tool for improving code correctness in Scala, but that they can also be gainfully applied in existing codebases.
Our concrete contributions can be summarized as follows:

- We introduce **qualified types** to Scala's next-generation compiler, Dotty[1]. Qualified types allow programmers to introduce base types and classes whose values are constrained by appropriate *constraint expressions*. Our modifications allow these qualified types to benefit from the usual local type inference of Scala.
- We implement a **type checker** for qualified types in Dotty. Our implementation is largely independent of Scala's own type checker and demonstrates how qualified types can be retrofitted to support both object-oriented and functional features. In particular, we support qualified base types and classes, class-invariants on stable constructor fields, higher-order functions and simple cases of genericity.
- We provide a **simple strategy to infer qualifiers** in a limited, but useful subset of situations. We argue that idiomatic Scala in fact lends itself to this kind of inference.
- We present an initial evaluation of our system in terms of **compile-time** and **annotation overhead**. We conclude that for our benchmark covering a range of Scala's language features, the additional overhead is manageable.

Our implementation and the examples are in a fork of Dotty: `https://github.com/gsps/dotty/tree/liquidtyper` .

2. Qualified Type Checking in Scala

This section gives a high-level overview of our qualified types solution for Scala. We begin by defining the notion of a subtyping constraint among qualified types. Using several examples we then illustrate how such subtyping constraints come to be and how our system either resolves them via inference or by checking their validity in an SMT solver.

2.1 Qualified Types and Subtyping Constraints

The basic building blocks of our type system are *constraint expressions* and *qualified types*. We first *extract* constraint expressions from Scala expressions and then encode them as SMT queries in the logic of bitvectors and uninterpreted function symbols. The following grammar captures the syn-

[1] `https://github.com/lampepfl/dotty`

tax of constraint expressions:

$$
\begin{array}{rcl}
\mathit{param} & ::= & \text{IDENT} := \mathit{term} \\
\mathit{op1} & ::= & - \mid ! \\
\mathit{op2} & ::= & + \mid - \mid * \mid / \mid \% \mid \& \mid \text{`}\mid\text{`} \mid \hat{} \mid \ll \mid \gg \mid \ggg \\
\mathit{term} & ::= & \texttt{Int}\text{-LITERAL} \mid \text{IDENT} \\
 & & \mid \mathit{op1}\ \mathit{term} \mid (\ \mathit{term}\ \mathit{op2}\ \mathit{term}\ ) \\
 & & \mid (\ \text{new IDENT}\ [\langle\ \mathit{param}\ \langle\ , \mathit{param}\ \rangle^*\ \rangle])\\
 & & \mid \mathit{term}\ .\ \text{IDENT} \\
\mathit{cmp} & ::= & = \mid < \mid \leq \\
\mathit{expr} & ::= & \top \mid \bot \mid \text{IDENT} \mid \mathit{term}\ \mathit{cmp}\ \mathit{term} \\
 & & \mid (\ \mathit{expr} \wedge \mathit{expr}\ ) \mid (\ \mathit{expr} \vee \mathit{expr}\ ) \mid \neg \mathit{expr}
\end{array}
$$

These expressions encode propositions about boolean algebra, bitvector arithmetic and the stable fields of the corresponding integral types in Scala objects.

Definition 2.1. A *qualified type* Q is given by a triple (B, v, R), where B is a base type or a class, v is the *subject variable* and R is either a constraint expression P or some $\sigma\kappa$, i.e. a *qualifier variable* κ under substitution σ. We typically write $\{v : B \Rightarrow R(v)\}$ and call $R(v)$ the *qualifier* of Q.

For example, we refer to the non-negative integers between 0 and $2^{31} - 1$ by the qualified type $\{v : \mathbf{Int} \Rightarrow v \geq 0\}$ featuring the constraint expression $v \geq 0$ in its qualifier. A corresponding type alias in our system may be written `NonNeg` in Listing 2 and can then be used as a shorthand in function and method signatures, as usual. We call a qualifier that has been introduced explicitly by the user *ascribed*, resp. a *qualifier ascription*. We consider a standard Scala type T a shorthand for the trivially qualified type $\{v : T \Rightarrow \top\}$. When it is clear from the context, we may also abbreviate a qualified type $\{v : B \Rightarrow R(v)\}$ by omitting the unqualified type B and just writing $\{R(v)\}$. To talk about qualified types whose concrete qualifier is unknown, we introduce qualifier variables which are placeholders for later inferred constraint expressions. A qualified type $\{v : B \Rightarrow R(v)\}$ is called *abstract* if R is a qualifier variable and *ground* otherwise.

We also extend the standard notion of a typing environment to allow for dependent types and path-sensitivity:

Definition 2.2. A *typing environment* $\Gamma; \mathbf{\Phi}$ consists of a sequence of *bindings* Γ and a constraint expression Φ which we call the *path condition*. Each binding in Γ is of the form $x_i : Q_i$, where x_i is a (term) variable and Q_i a qualified type. A qualifier Q_i may depend on preceding bindings x_j where $j < i$, while the path condition Φ may depend on any of the bindings. We omit the path condition if it is trivial, i.e. equivalent to $\top$.

The sequence of bindings, that is, the scope available to a qualifier is established following Scala's (path-dependent) type system. In particular, qualifiers in a method's parameter group can access parameters in the preceding groups and the result type's qualifier can access all parameters. Putting all of these concepts together, we can define the subtyping constraints among qualified types.

$$
\begin{array}{rcl}
[\![\ \Gamma; \mathbf{\Phi} \vdash \{v : B_1 \Rightarrow P_1(v)\} <: \{v : B_2 \Rightarrow P_2(v)\}\ ]\!] & & \\
\quad := \quad [\![\Gamma; v' : \{v : B_1 \Rightarrow P_1(v)\}; \mathbf{\Phi}]\!] \implies P_2(v') & & \\
[\![\Gamma; \mathbf{\Phi}]\!] & := & [\![\Gamma]\!] \wedge \mathbf{\Phi} \\
[\![(\ )]\!] & := & \top \\
[\![\Gamma';\ x : \{v : B \Rightarrow P(v)\}]\!] & := & [\![\Gamma']\!] \wedge P(x)
\end{array}
$$

Figure 1: The rules for encoding subtyping constraints

Definition 2.3. A *subtyping constraint* $\Gamma; \mathbf{\Phi} \vdash Q_1 <: Q_2$ relates qualified types Q_1 and Q_2 in the context of typing environment $\Gamma; \mathbf{\Phi}$. Let $Q_1 = \{v : B_1 \Rightarrow P_1(v)\}$ and $Q_2 = \{v : B_2 \Rightarrow P_2(v)\}$. Then the subtyping constraint is *valid* iff (a) B_1 is a subtype of B_2 w.r.t. Scala's standard subtyping relation, and (b) $[\![\ \Gamma; \mathbf{\Phi} \vdash Q_1 <: Q_2\ ]\!]$ is valid, where the encoding $[\![\cdot]\!]$ into first-order logic is given by Figure 1.

2.2 Verifying a Simple Function

How does our system prove that the qualifications on each type hold? In general, we proceed in four steps: qualifier assignment, constraint generation, qualifier inference and constraint checking. Let us examine these four steps by verifying a simple function:

Listing 3: An implementation of a maximum function

```
def max(x: Int, y: Int): { v: Int =>
    v >= x && v >= y } =
  if (x > y) x else y
```

Listing 3 implements a maximum function `max(x, y)` whose result is guaranteed to be lower bounded by both of the two parameters `x` and `y`.

2.2.1 Qualifier Assignment

In order to formulate the constraints governing a program, we first need to pick a qualified type for each expression in the program. The unqualified base and class types are already given by Scala's standard type system, therefore we only need to determine appropriate qualifiers.

Using qualifier ascriptions the user typically provides ground qualifiers for a subset of expressions. The remaining qualifiers may be either ground or abstract: Wherever possible, we assign ground qualifiers; primitives, such as literals and arithmetic operations, but also variable identifiers fall into this category. Others, such as `if`-expressions, are first assigned qualifier variables. Once we have gathered relevant constraints (step two), we either infer adequate ground qualifiers or fall back to the trivial qualifier $\{\top\}$ (step three).

Ground qualifiers are generally formed by *extraction*, i.e. by mapping Scala expressions into constraint expressions. Ideally, extractions are *precise*, accurately representing the Scala expression as a constraint expression. We only allow

qualifier ascriptions that we can extract precisely, but permit *weak* extractions, i.e. sound approximations, for branching conditions and function arguments. If, during a weak extraction, we cannot represent a Scala expression as part of a constraint expression, we instead introduce a fresh, unconstrained variable of the same sort. We do allow function calls in qualifier ascriptions, if the function itself can be extracted precisely (which implies that it is pure, side-effectless and can be encoded in our logic).

Let us look at some of the qualified types that have been assigned in our example program of listing 3:

`x`	$\mapsto$	$\{v : \mathbf{Int} \Rightarrow v = x\}$
`y`	$\mapsto$	$\{v : \mathbf{Int} \Rightarrow v = y\}$
`if (x > y) x else y`	$\mapsto$	$\{v : \mathbf{Int} \Rightarrow \kappa_{\text{if}}\}$
`max`	$\mapsto$	$\{v : \mathbf{Int} \Rightarrow \top\} \Rightarrow \{v : \mathbf{Int} \Rightarrow \top\} \Rightarrow \{v : \mathbf{Int} \Rightarrow v \geq x \wedge v \geq y\}$

Identifiers A lone identifier such as the `x` in the `then`-branch is represented precisely by qualified type $\{v = x\}$. We can recover qualifications on the referenced variable `x` from the typing environment.

If expressions The result of an `if`-expression is assigned a *fresh* qualifier variable, which will later be inferred. In our example we assign κ_{if}, allowing us to capture the consequences of both branches.

Function definitions Users are expected to ascribe qualifiers to parameters occurring in a function type, otherwise the trivial qualifier is assumed. The result type, i.e. the right-most base type or class, may be either given explicitly by an ascription or inferred, again by introducing a qualifier variable. The (unqualified) type Scala reports for `max` is `Int => Int => Int`. Since we only ascribed a qualifier to the return type of `max`, the two parameters end up trivially qualified.

In later subsections we will see examples of more involved programs, some of which require yet different ways of extracting qualifiers.

2.2.2 Constraint Generation

Wherever information flows from an expression typed $\{\kappa_A\}$ to another typed $\{\kappa_B\}$, we need to ensure that qualifier κ_B subsumes qualifier κ_A. Our system therefore establishes subtyping constraints among the two. In the case of listing 3, our system produces the following constraints:

$$x : \mathbf{Int}; y : \mathbf{Int}; x > y \vdash \{v = x\} <: \{\kappa_{\text{if}}\} \quad (1)$$

$$x : \mathbf{Int}; y : \mathbf{Int}; x \leq y \vdash \{v = y\} <: \{\kappa_{\text{if}}\} \quad (2)$$

$$x : \mathbf{Int}; y : \mathbf{Int} \vdash \{\kappa_{\text{if}}\} <: \{v \geq x \wedge v \geq y\} \quad (3)$$

In total, we get three constraints: (1) requires the `then`-branch to return a subtype of the `if`'s overall type, (2) is the analogous constraint for the `else`-branch, and (3) requires the `if`'s type to be a subtype of the our function's result type.

Note that in addition to extracting qualifiers, we also extract branching conditions, which allows our analysis to be *path-sensitive*. Whenever we enter a block of code that is guarded by a test, such as an `if`'s then-branch, we strengthen the current path condition by the respective branching condition. In the example above this is reflected in constraints (1) and (2), which gained the condition $x > y$ for the then-branch, and its negation for the else-branch.

As with extraction, we will see more situations in which constraints arise in the examples below, whereas section 3.1 covers implementation details.

2.2.3 Qualifier Inference

Before we can check the generated constraints we need to make sure all involved qualifiers are ground, meaning that we need to *eliminate* qualifier variables. We first try to *infer* a (ground) qualifier for each qualifier variable. Our system does so by taking the disjunction of all qualifiers that a qualifier variable needs to accomodate, which essentially corresponds to computing the strongest postcondition. While this is precise and practical for intermediate expressions in a function's body, the approach is not always suitable for qualifiers at abstraction boundaries. In the presence of recursion, for result types of functions that involve local variables, but also for method parameters we fall back to the trivial qualifier. That is, we eliminate the concerned qualifier variable κ by assigning $\kappa := \top$. Section 3.2 goes into the details of our inference algorithm.

In the example above we can eliminate the single qualifier variable κ_{if}. Recall the two constraints (1) and (2). In order for these constraints to be valid, κ_{if} needs to be implied by either of $v = x$ and $v = y$ when taken in conjunction with their respective typing environments, $x : \mathbf{Int}; y : \mathbf{Int}; x > y$ and $x : \mathbf{Int}; y : \mathbf{Int}; x \leq y$. Keeping in mind the embedding given by figure 1, we can precisely satisfy the two constraints by setting

$$\kappa_{\text{if}} \quad := \quad (x > y \wedge v = x) \vee (x \leq y \wedge v = y).$$

Since all the other qualified types are ground, we can now proceed to checking the constraints.

2.2.4 Constraint Checking

In a final step, we put the constraints to the test. To do so, we translate them to the logic of bitvectors and uninterpreted function symbols. Several modern SMT solvers can then be used to prove the validity of the underlying constraint. Note that we only need to check constraints whose right-hand side was a ground qualified type before inference. Constraints whose rhs qualifier was (successfully) inferred are valid by construction, whereas cases in which we introduced $\top$ as a qualifier are trivially satisfied.

In our example above this only leaves the grounded version of constraint (3) to be checked. We thus arrive at

$$x : \mathbf{Int}; y : \mathbf{Int} \vdash \{(x > y \wedge v = x) \vee (x \leq y \wedge v = y)\} <: \{v \geq x \wedge v \geq y\}$$

which in turn is translated to the formula

$$(x > y \wedge v = x) \vee (x \leq y \wedge v = y) \implies v \geq x \wedge v \geq y.$$

At this point the formula will be sent to the SMT solver, which will then report it to be valid. This brings our verification of the `max` function to a successful conclusion.

2.3 Primitives

We already mentioned in subsection 2.2.1 that we also extract ground qualifiers for primitive Scala expressions. A *primitive* is a constant or a built-in method whose meaning can be captured precisely by our constraint expressions. Examples include arithmetic operations on `Int`s, such as addition, multiplication, but also comparisons, as well as the usual logical operators on `Boolean`. We also consider integral and boolean literals primitive, as well as *fresh objects* and selectors of their fields. A fresh object represents a new instance of some user-defined class and features expressions that constrain the arguments passed to the constructor. Since we do not keep track of stores, our reasoning about equality among objects is currently very conservative.

Primitives define the cornerstones of our translation to SMT formulae, since the resulting constraint expressions can be translated to SMT queries either directly (booleans and integers) or encoded using uninterpreted function symbols (objects and field selectors).

2.4 Handling of Integral Types

For Scala's `Int`s we translate operations such as $(+)$, $(*)$, $(>)$ and $(\geq)$ to their corresponding counterparts in the theory of signed bitvectors. For instance, the branching condition `x > y` from listing 3 is translated to $x >_s y$ in the theory of bitvectors, where x and y are bitvectors of width 32. We thus make use of the fact that bitvector theory in the standardized SMT-LIB format[2] naturally captures the semantics of integral types in Scala (except possibly for division and modulo by zero). More generally, this mapping of primitive types is only limited by the repertoire of the SMT solver, so our system could be easily extended to support Scala's `BigInt`s, which correspond to unbounded integers (see [2] for a discussion of supporting verification of both modular and unbounded integer arithmetic in Scala programs; our system makes use of this infrastructure).

2.5 Function Applications

Function applications[3] directly affect the first two of the four steps we discussed before: 1) We assign the qualifier of the function's result type. If we are applying a dependently typed function, however, we also need to substitute the arguments for the function's parameters in the result type. As with branching conditions, we extract argument expressions weakly. 2) To guarantee that an application is safe, we need to establish that each argument to the application is a subtype of the function's corresponding formal parameter.

To illustrate, let us consider the next listing, which takes our previous example and extends it with a `sqrt` function that requires its single argument to be non-negative.

Listing 4: A square-root function with a restricted domain

```
def max(x: Int, y: Int): { v: Int =>
                           v >= x && v >= y } =
  if (x > y) x else y

def sqrt(z: NonNeg): Double =
  scala.math.sqrt(z.toDouble)

val u: Int = ???
sqrt(max(0,u))
```

The example features two function applications on its last line: We first call `max` to determine the maximum of 0 and an arbitrary `Int` u and then apply `sqrt` to that result. The question of interest here is how we make sure that `sqrt` actually receives a non-negative argument. Let us first see the qualified types assigned to each application:

$$\texttt{sqrt(max(0,u))} \mapsto \mathbf{Double}$$
$$\texttt{max(0,u)} \mapsto \{v : \mathbf{Int} \Rightarrow [u/y][0/x]\ v \geq x \wedge v \geq y\}$$

For the application of `sqrt` we simply adopt `sqrt`'s return type `Double` with a trivial qualifier. The application of `max`, on the other hand, requires that we substitute argument expressions for parameters `x` and `y`. To construct this type, we introduce *substitutions* that replace x by 0 and y by u. In the second step we then generate the subtyping constraint $u : \mathbf{Int} \vdash \{[u/y][0/x]\ v \geq x \wedge v \geq y\} <: \{v \geq 0\}$ which holds thanks to the substitutions.

2.6 Recursive Functions

Using all the functionality presented so far, we are already able to reason about recursive functions. Note that our qualifier inference algorithm will fall back to trivial qualifiers when it encounters substitutions or cyclic dependencies and also does not eliminate intermediate variables. This means that, as for the usual Scala types, we explicitly ascribe qualifiers to the return types of recursive functions.

2.7 Object Support

Our system supports some reasoning over objects by extracting object instantiations and selections of object fields. In particular, we allow qualifications wrt. *stable fields* of an object. Roughly speaking, Dotty considers a field stable if its value does not change after initialization. During constraint checking, we introduce an uninterpreted function for

[2] `http://www.smt-lib.org`

[3] Scala ultimately also handles functions as methods and invocations thereof, which is why we mostly disregard the distinction.

each stable field and assert the corresponding qualifier, universally quantified over object instances. As we will show below, this limited functionality can be used to great effect, giving us class invariants wrt. stable constructor fields.

2.7.1 Lifting Objects to the Qualifier Level

We gain significant expressive power by allowing qualifiers to refer to the enclosing object instance. To illustrate this point, consider class `IntArray` from the motivating example, which provided static array-bounds checks on primitive Scala `Arrays`.

Listing 5: A minimal example of static array bounds checks

```
class IntArray(val length: NonNeg, init: Int) {
  def access(i: { v: Int =>
    0 <= v && v < this.length }): Int = ??? }
```

Crucially, the qualifier of `access`' parameter `i` refers back to the object instance using `this.length`. If we instantiate `IntArray` with some argument expression for `length`, that expression will flow into all the references to `this.length` by way of substitution. For instance, we might instantiate an `IntArray` of length 3 and then commit an off-by-one error, accessing the third element:

```
new IntArray(3, 0).access(3)
```

The qualified type of `new IntArray(3, 0)` then reflects the actual values, i.e. the qualifier features a *fresh object*

$$\{v = (\text{new } \mathbf{IntArray}\ [\text{length} := 3, \text{init} := 0])\}.$$

Moreover, this fresh object propagates into method selections, such as `(new ...).access`. In our example, the constraint generated for `.access(3)` includes the exact values the object was instantiated with, i.e.

$$\vdash \{v = 3\} <: \{\ [(\text{new } \mathbf{IntArray}\ [\text{length} := 3, \text{init} := 0])/\text{<this>}]\ 0 \leq v \wedge v < (\text{<this>.length})\ \},$$

which we then prove invalid during constraint checking.

2.7.2 Class Invariants for Stable Constructor Fields

Given dependent function types and the above extraction support for objects, we can encode class invariants over stable constructor fields. Listing 6 contains class `PosTuple`, which represents a tuple of two `Int`s `a` and `b` with the notable constraint that their sum must be strictly positive.

Listing 6: An `Int`-tuple whose sum is guaranteed positive

```
class PosTuple(val a: Int, val b: Int)(
  ev: { v: Unit => a + b > 0 } = ())
```

We use an *evidence* parameter `ev` whose value is trivial, but that carries a constraint in its qualified type. This enforces the invariant when instantiating new `PosTuple` objects. Instantiations such as `new PosTuple(1,-1)()` therefore result in a compile-time error, as desired. More generally, we make such evidence invariants/preconditions available to function and class bodies, by adding them to the path condition. We do so whenever a function or constructor contains a parameter whose qualifier ascription does not mention its subject variable (see definitions of `mergeSort` and `copy` in listing 10 for an example).

2.8 Interaction with Generic Types

Scala programs make heavy use of generic data types. To support the common use-case of collections, we allow generic data structures to be instantiated with qualified types. For instance, consider the following listing.

```
val nnList                  = List[NonNeg](1,2,3)
val x: NonNeg               = nnList.head
val nnListRev: List[NonNeg] = nnList.reverse
```

We first instantiate a new list of non-negative `Int`s. Note that in lines #2 and #3 we access a single element and manipulate the entire list, respectively, while maintaining the precise qualifier `NonNeg`. The next listing demonstrates that we cannot arbitrarily reinterpret the element type of lists.

Listing 7: Invalid attempt to reinterpret a list's element type

```
type Neg = { v: Int => v < 0 }
// compile-time error:
val negList: List[Neg] = nnList
```

While type checking the above snippet we generate a constraint $\vdash \{v \geq 0\} <: \{v < 0\}$, which is obviously invalid.

2.9 Higher-Order Functions (HOFs)

We also maintain the qualifiers of functions passed as arguments, including those of closures. This allows HOFs to benefit from qualified types, as well. As with unqualified HOFs, subtyping constraints are subject to the contravariance of parameter types. Below we see a HOF manipulating integers:

Listing 8: Passing an incompatible closure to a HOF

```
def h(f: Int => Int): Int = f(-1)
h((x: NonNeg) => x)  // compile-time error
```

The critical subtyping constraint in listing 8 is the one between the closure argument and parameter `f`:

$$\vdash \quad \{v \geq 0\} \Rightarrow \mathbf{Int} \quad <: \quad \mathbf{Int} \Rightarrow \mathbf{Int}$$

Our system unfolds this constraint among function types into simple constraints over qualified types. The contravariance of parameter types then gives rise to the simple constraint $\vdash \mathbf{Int} <: \{v \geq 0\}$, which is clearly invalid.

2.10 Combining Advanced Scala Features

Our final example illustrates that programmers can benefit from the additional safety afforded by qualified types, without having to give up on Scala code that uses a combination of the previously demonstrated features. We consider

the `IntArray` class from above and extend it with methods `sum`, `max` and `contains`, all of which have their usual semantics. Recall that `IntArray` ensured that any access to the underlying array was statically verified to be within bounds. We can maintain this property, while concisely implementing the new functionality.

Listing 9: A statically bounds-checked wrapper for integer arrays, providing methods for sum, max and contains

```
type NonNeg = { v: Int => v >= 0 }

def max2(x: Int, y: Int): {v: Int => v >= x && v >= y } =
  if (x > y) x else y

abstract class IntArray(val length: NonNeg, init: Int) {
  def access(i: {v: Int => 0 <= v && v < this.length}): Int

  def fold[A](f: (A, Int) => A, z: A): A = {
      def rec(i: NonNeg, acc: A): A =
        if (i < this.length)
          rec(i+1, f(acc, this.access(i)))
        else acc
    rec(0, z)
  }

  def sum: Int = this.fold[Int](_ + _, 0)
  def max: NonNeg = this.fold[NonNeg](max2, 0)

  def contains(x: Int): Boolean = {
      def check(res: Boolean, y: Int) =
        if (res) true else x == y
    this.fold[Boolean](check, false)
  }
}
```

The example's code can be found in listing 9. To begin with, all of the new functionality `IntArray` gained was implemented in a modular fashion using a *fold* over the array. We also provide the implementation of the corresponding *higher-order* method `fold`. Note that `fold` itself is both *recursive* and *polymorphic*. Among the three methods mentioned before, `max` is notable: It uses the guarantees of helper function `max2` to prove the result of a maximum over all array elements and zero to be non-negative.

3. Implementation

Our system is implemented in the context of *Dotty*, the next-generation compiler of Scala. It extends the Scala language by qualifiers that act as refinements of base and class types. These extensions go hand-in-hand with a new type checking procedure for qualified types.

Figure 2 gives an overview of the combined architecture and the intermediate steps of our type checking procedure. Only minor changes were made to the parser and the typer of Dotty itself. The bulk of qualified type checking is performed thereafter in a dedicated compiler phase, which can be divided into four distinct steps, closely following those outlined in section 2.2:

1) **Qualifier assignment** is handled in two traversals over the AST: We first index symbols and extract qualifier ascriptions along with the typing environment that each Scala expression lives in (*SymbolIndexer*). In a second traversal we assign qualifiers to the remaining expressions (*Typer*).
2) We **gather subtyping constraints** among the previously established qualified types.
3) We then apply our **qualifier inference** algorithm, hopefully discovering strong-enough qualifiers.
4) Finally, we translate all non-trivial **subtyping constraints** to logical formulae which are **tested for validity** by an SMT solver.

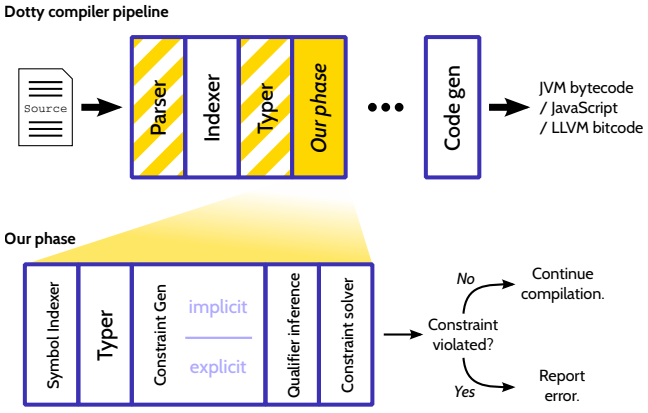

Figure 2: Our system's architecture. Shaded components we added (solid) or slightly modified (striped).

In the remainder of the section we will focus on the procedures we employed in steps two and three.

3.1 Constraint Generation

Constraints are created in one of two ways: Either *explicitly* during a traversal of the AST or *implicitly* after we recover the subtyping constraints that the Dotty typer relied on.

3.1.1 Explicit Constraint Generation

The majority of subtyping constraints are established by visiting each AST node and following the explicit rules of establishing subtyping constraints for that kind of node. For instance, in case of an application we require that the types of the arguments are subtypes of the respective formal parameters. The general rule here is that wherever information flows from one typed expression to another, we need to make sure that the former's type is a subtype of the latter's.

3.1.2 Implicit Constraint Generation

Our phase comes into play after Dotty's typer has done its job and has assigned (unqualified) types to all AST nodes. While this design allowed us to keep the qualified type checker simple and largely independent from the Scala typer, it also brings some challenges with it. As the Dotty typer performs its checks and infers missing unqualified types, it assumes that various subtyping relations hold. Since none of the qualified type checking has taken place at that point, we initially ignore the qualifiers and thereby defer the decision.

Without further handling, this would give rise to unsoundness, since programs such as listing 7 would erroneously pass qualified type checking. In order to recover implicitly relied upon subtyping constraints, we utilize Dotty's so-called `ReTyper`, which reruns typing on an already typed program. This allows us to intercept the essential subtyping checks that we would have otherwise missed, such as those concerning type parameters of collections.

3.2 Qualifier Inference

In order to save the programmer the effort of explicitly annotating many intermediate expressions we provide a simple, yet effective way of inferring qualifiers. Below we describe our algorithm and situations in which it falls back to trivial qualifiers.

3.2.1 Intuition

At the heart of our inference algorithm lies the idea that when we know *all* subtyping constraints in which a certain qualifier Q appears on the right-hand side, then we can (usually) replace that qualifier by the disjunction of all left-hand sides (along with potential path-conditions specific to each left-hand side). The intuition is simple: If we let the qualifier contain the exact union of all those values that should be included according to the subtyping constraints, then those constraints must trivially hold. This approach essentially corresponds to computing the *strongest postcondition* of all "incoming" qualifiers. Note that it also comes with certain limitations:

Recursive methods: Due to recursive calls the qualifier of a method's result type may depend on a version of itself with some substitutions. We then face the non-trivial task of unifying these qualifiers. Instead, we simply abort and require the user to provide a qualifier ascription.

Intermediate variables: If the inferred qualifier contains term-level variables that are out of scope at the position of the corresponding qualified type, we simply fall back to the trivial qualifier.

While our current algorithm only works on the basis of forward-propagation, one could also do the converse and compute the weakest precondition. Combining both forward and backward-propagation would yield a procedure analogous to bidirectional type inference [11]. This approach also naturally extends to inference using Horn clauses [13].

3.2.2 Our Qualifier Inference Algorithm

We consider qualifier inference as a problem on the constraint graph, i.e. the directed graph that is induced by all the non-trivial constraints we generated. For the treatment below, we choose to represent subtyping constraints as five-tuples. That is, if we have a subtyping constraint $\Gamma; \Phi \vdash \{\sigma_a \kappa_a\} <: \{\sigma_b \kappa_b\}$ then we represent it using the tuple $(a, b, \sigma_a, \sigma_b, \phi)$. Occurrences of ground qualifiers are handled analogously except with σ_a, σ_b set to equal the empty substitution ϵ, since substitutions on ground qualifiers can be applied right away.

Algorithm 1: Our qualifier inference algorithm.

Data: Constraint graph $\mathcal{G} = (V \cup G, C)$, Qualifier variable map *qvar*, Scopes Σ
Result: $\forall v \in V.\ \exists (v, g) \in \text{qvar}.$ g is ground qualifier
if $\mathcal{G}$ *contains a cycle of qualifier vars* **then**
  Abort and print error report affected Scala expressions
$\text{unassigned} := \lambda v.\ (\neg \exists g.\ (v, g) \in \text{qvar})$
$\forall v \in \{b \mid b \in V \wedge \text{unassigned}(b) \wedge (a, b, \sigma_a, \sigma_b, \phi) \in C \wedge \sigma_b \neq \epsilon\}.\ \text{qvar}(v) := \top$
$\text{incoming} := \lambda v.\ \{(a, \sigma_a, \phi) \mid (a, v, \sigma_a, \sigma_v, \phi) \in C\}$
$\text{numPreds} := \lambda v.\ |\{(a, v, \sigma_a, \sigma_v, \phi) \in C \mid \text{unassigned}(a)\}|$
$\text{frontier} := \{v \in V \mid \text{unassigned}(v) \wedge \text{numPreds}(v) = 0\}$
while frontier *is non-empty* **do**
  pick and remove some v from frontier
  $\text{qvar}(v) := \bigvee_{(w, \sigma_w, \phi) \in \text{incoming}(v)} (\phi \wedge \sigma_w w)$
  if *term-level variables in* $\text{qvar}(v) \not\subseteq \Sigma(v)$ **then**
    $\text{qvar}(v) := \top$
  $\text{frontier} := \{v \in V \mid \text{unassigned}(v) \wedge \text{numPreds}(v) = 0\}$
$\forall v \in V \setminus \{w \mid (w, g) \in \text{qvar}\}.\ \text{qvar}(v) := \top$

Definition 3.1. A *constraint graph* $\mathcal{G}$ is a tuple $(V \cup G, C)$ where V and G are disjoint sets of qualifiers, V being qualifier variables and G being ground qualifiers; C is a set of non-trivial subtyping constraints, represented as tuples $(a, b, \sigma_a, \sigma_b, \phi) \in C$ where a, b are qualifiers in $V \cup G$, σ_a, σ_b are substitutions and ϕ is a path condition. We consider elements in $V \cup G$ the *nodes of* $\mathcal{G}$ and each $(a, b, \sigma_a, \sigma_b, \phi) \in C$ an *edge from* a *to* b, annotated by substitutions σ_a, σ_b and path condition ϕ.

Algorithm 1 contains pseudo-code implementing our qualifier inference. The central idea is to traverse yet unassigned qualifier variables in topological order and compute the strongest postconditions for each. Before the topological traversal we check for cycles of qualifier variables in the constraint graph and abort if one is found — the user must provide more qualifier ascriptions in this case. We also assign the trivial qualifier to all qualifier variables that occur under a non-trivial substitution on the right-hand side of a constraint, thus side-stepping the unification problem. After this we perform the actual inference. Note that whenever we find that the inferred qualifier would violate the scope of the qualified type it belongs to, we fall back to the trivial qualifier. Finally, we also assign the trivial qualifier to all remaining unassigned qualifier variables — these correspond to unconstrained qualifier variables.

Our algorithm is called with a constraint graph satisfying the above two definitions. In addition, it takes a map `qvar` which relates qualifier variables to ground qualifiers, and a map of scopes Σ which relates each qualifier variable to the set of available term-level variables (iow., the scope of the qualifier). Upon completion, `qvar` is guaranteed to map every qualifier variable in V to some ground qualifier.

Lst.	Example	+/-	# Lines	AST size	Time
#2	binsearch	-	22 (2)	192 (61)	0.35
#3	max	+	4 (0)	55 (19)	0.11
#4	sqrt	+	11 (2)	87 (34)	0.17
#5	intarray	+	11 (2)	117 (35)	0.10
#6	postuple	+	3 (0)	95 (50)	0.20
	list1	+	7 (2)	60 (17)	0.07
#7	list2	-	7 (3)	70 (32)	0.11
	hofsafety1	+	6 (2)	60 (15)	0.13
#8	hofsafety2	-	6 (2)	60 (15)	0.08
#9	arrfold	+	31 (2)	261 (54)	0.27
	sumnat	+	10 (2)	77 (15)	0.23
	rational	+	6 (0)	73 (13)	0.05
	insertionsort	+	29 (5)	277 (117)	0.26
#10	mergesort	+	51 (10)	582 (303)	1.28

Figure 3: The results of our benchmarks. Column 1 refers to the corresponding listings. Columns 3-5 state whether it constitutes a valid or buggy program, the number of lines and and the size of the abstract syntax tree in the example. In parentheses we show how much larger qualified examples are when compared to their unqualified counterparts. The last column shows time spent checking qualified types in seconds averaged over 10 runs.

4. Evaluation

We evaluated our system based on a suite of benchmarks including examples presented in the preceding chapters. Below we report both annotation overhead and the additional time incurred checking qualified types. We argue that our results show the viability of qualified types in idiomatic Scala code.

4.1 Setup

We performed our benchmarks on a Lenovo ThinkPad X230 (2012) running Ubuntu 14.04 LTS. The laptop is powered by a dual-core Intel Core i5-3320M CPU @ 2.60GHz featuring 32KB L1 data and instruction caches, as well as 2x256KB on L2 and 3MB on shared L3. The laptop contains 16GB of DDR3 RAM clocked at 1600MHz, out of which we dedicated 4GB to the JVM instance running the benchmarks. Furthermore, the tests were performed on a warmed up JVM instance to avoid the high startup cost of JIT compilation. We consider this a realistic scenario, as it is common for Scala users to run the compiler in a long-lived JVM instance. As SMT solvers we used CVC4 (version 1.5-prerelease, 2016-07-31), falling back to Z3 (version 4.4.1) when CVC4 could not determine whether a query was satisfiable.

4.2 Benchmarks

The metrics we considered were: a) *Type checking runtime*, which is the additional time spent checking *qualified* types during compilation. We measured this by instrumenting the running time of our compiler phase. b) *Annotation overhead*, measured in terms of line count and size of the abstract syntax tree. Figure 3 below presents our results, reporting one example per line. In addition to the examples from section 2 we included several more programs which can be found in our code repository mentioned at the beginning.

We see that our system completes type checking for all but one of the examples in less than a second and requires only modest amounts of annotations. In our opinion this overhead is acceptable for the additional safety benefits qualified types provide. We note that among our benchmarks the `mergesort` example in listing 10 is the only one that incurs significant constraint solving overhead, i.e. CVC4 takes up almost 70% of the 1.28s needed to check `mergesort`.

With our system being mostly a prototype at the moment, no significant profiling has been done. While SMT solving time might become a bottleneck for more complex qualifiers as in `mergesort`, simple qualified type checking could benefit from optimizing the indexing and typing traversals.

5. Related Work

Our work is most closely related to *refinement types* and *LiquidTypes* in particular. Refinement types were introduced by Freeman and Pfenning in [6]. Their specific type system handles algebraic data types in *ML* and provides a way of referring to more precise subsets, i.e. refinements, of algebraic data types. They also contribute a decidable type inference mechanism by using predicate abstraction over the lattice of refinements.

More recently, Rondon, Kawaguchi and Jhala presented LiquidTypes [12], which allows refinements in the form of logically qualified base- and function types in functional languages. Their system takes qualifier *templates* and then infers refinements using predicate abstraction; it was subsequently extended with support for qualifier parametricity in *abstract refinement types* [15].

Our system is inspired by Rondon et al.'s work, sharing not only a similar syntax for qualified types, but also the ideas surrounding constraint generation, in particular that of qualifiers variables. The most striking difference between LiquidTypes and our system is that we do not provide predicate abstraction as a means of type inference, but rely on the algorithm of section 3.2 and qualifier ascriptions instead.

These projects are instances of the bigger area of *dependent typing* [16]. Dependently typed programs in their most general form allow types to be *indexed* by term-level expressions, fulfilling a purpose comparable to that of our qualifiers. In their initial paper Xi and Pfenning considered indices drawn from a decidable logic, Presburger arithmetic. There is a notable body of work on proof assistants and higher-order logics for verification, that can be characterized as or are founded upon dependently-typed languages [1, 4, 5, 9].

Leon [8] is a verification tool for Scala. Our system relies on Leon infrastructure for representing constraint expressions and interfacing with SMT solvers.

During development we also learned about a project similar to ours in the Scala space. So-called *refined* [14] is implemented entirely as a library and relies on Scala's already powerful type system, in particular its implicit resolution mechanism, to introduce refinement types. Their approach is quite different from ours: Rather than outsourcing proof obligations to an SMT solver, *refined* only allows a select subset of qualifiers for which decision procedures have been encoded by means of Scala's implicit resolution. This results in a rather compact and self-contained implementation of refinement types for Scala. It is unclear to us how well implicit-based encodings can scale to verifying large Scala programs. Similarly, there is the question of how user-friendly error and counterexample-reporting can be made.

Acknowledgments

We would like to thank our reviewers for their helpful feedback, Nicolas Voirol and Mikaël Mayer for the insightful discussions we had with them during the development of our system, and Régis Blanc in particular for his advice on using Leon's solver infrastructure. We thank Martin Odersky for many discussions and for helpful suggestions on using Dotty's ReTyper.

References

[1] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer, 1st edition, 2010. ISBN 3642058809, 9783642058806.

[2] R. Blanc and V. Kuncak. Sound reasoning about integral data types with a reusable SMT solver interface. In *Scala Symposium*, 2015.

[3] J. Bloch. Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken. `https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`, 2006. Accessed: 2016-06-23.

[4] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[5] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., 1986. ISBN 0-13-451832-2.

[6] T. Freeman and F. Pfenning. Refinement Types for ML. In *PLDI*, pages 268–277, 1991.

[7] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java® Language Specification (Java SE 8 Edition)*. Oracle, 2015. URL `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`. 2015-02-13.

[8] E. Kneuss, V. Kuncak, I. Kuraj, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[9] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[10] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994.

[11] B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, pages 252–265, 1998.

[12] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *POPL*, pages 159–169, 2008.

[13] P. Rümmer, H. Hojjat, and V. Kuncak. On recursion-free Horn clauses and Craig interpolation. *Formal Methods in System Design*, 47(1):1–25, 2015.

[14] F. S. Thomas. Refined – Simple refinement types for Scala. `http://refined.timepit.eu`, 2016. Accessed: 2016-06-23.

[15] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In *ICFP*, pages 269–282, 2014.

[16] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257, 1998.

Listing 10: The `mergesort` example

```
abstract class CheckedIArray(val length: NonNeg, init: Int) {
  def apply(i: {v: Int => range(0, v, this.length-1)}): Int
  def update(i: {v: Int => range(0, v, this.length-1)}, x:
      Int): Unit }

def mergeSort(arr: CheckedIArray, aux: CheckedIArray)(
    ev: {v: Unit => arr.length == aux.length } = ()): TUnit = {
  def merge(lo: NonNeg)(
            mid: {v: Int => range(lo+1, v, arr.length-1) },
            hi:  {v: Int => range(lo+1, v, arr.length) })(
            i:   {v: Int => range(lo, v, mid) },
            j:   { v: Int => range(mid, v, hi) },
            k:   {v: Int => range(lo, v, aux.length) }): TUnit =
    if (k < hi) {
      if (i == mid)       copy(arr)(aux, j)(hi)
      else if (j == hi) copy(arr)(aux, i)(mid)
      else if (arr(i) <= arr(j)) {
        aux(k) = arr(i)
        merge(lo)(mid, hi)(i+1, j, k+1)
      } else {
        aux(k) = arr(j)
        merge(lo)(mid, hi)(i, j+1, k+1)
      }
    }

  def copy(from: CheckedIArray)(to: CheckedIArray, i: NonNeg)(
      hi: { v: Int => range(i, v, from.length) },
      ev: { v: Unit => from.length == to.length } = ()): TUnit=
    if (i < hi) {
      to(i) = from(i)
      copy(from)(to, i+1)(hi)
    }

  def partial(lo: NonNeg)(hi: { v: Int => lo < v && v <=
      arr.length }): TUnit =
    if (hi - lo > 2) {
      val mid = lo + (hi - lo) / 2
      partial(lo)(mid)
      partial(mid)(hi)
      merge(lo)(mid, hi)(lo, mid, lo)
      copy(aux)(arr, lo)(hi)
    }

  if (arr.length > 1) partial(0)(arr.length)
}
```