Chartreux: Towards Verified Dataflow Analyses for Kotlin

Jacopo Moretti[1][0009-0000-3662-1683], and Marcin Wojnarowski[1][0009-0003-7962-3904]

[1] JetBrains München
firstname.lastname@jetbrains.com

Abstract. The Kotlin programming language differs from others through its use of dataflow analysis not just as a way to optimize the generated code, but also as a complement to type analysis, to decide program validity and increase the expressive power of the language. These analyses include flow sensitive type refinement of expressions ("smart casts"), and more recently, function contracts on lambda parameters. While useful, the implementation of these analyses carries no guarantees, and the soundness of their behavior is not immediately obvious. To aid the understanding of current analyses, and the design of new ones, we present Chartreux, a generic Lean framework for specifying and verifying dataflow analyses on control flow graphs. We showcase how Chartreux allows language designers to verify the correctness of their analyses with respect to arbitrary language properties, by applying it to analyses and techniques from the Kotlin compiler.

Keywords: Dataflow Analysis · Program Verification · Kotlin

1 Introduction

Static analysis is a class of techniques used to analyze programs to obtain guarantees on their behavior without having to execute them. Foundational work by Cousot & Cousot [1] developed the theory of Abstract Interpretation, unifying many static analysis approaches under one mathematical umbrella. Today, a practical instantiation of abstract interpretation, called dataflow analysis, is heavily employed in many modern industrial compilers such as LLVM, GCC, or V8.

Consider the following Kotlin [2] program:

```
val a = 5
if (a + 2 == 7) {
  println("It is seven!")
} else {
  println("Oh no!")
}
```

By performing a constant propagation dataflow analysis, we learn that the condition always holds true (and has no side-effects). Therefore, we can eliminate the `if` statement completely reducing this program to just `println("It is seven!")`. This classic analysis is an example of using analyses to perform program optimization. But this is only one example use-case for dataflow analyses; they are also widely used for identifying potential runtime errors.

The Kotlin compiler leverages this latter capability significantly, relying on dataflow analyses to help the type checker decide whether a program should be

accepted. In Kotlin, this feature is known as "smart casts" [3]. Similar approaches are also used extensively in languages such as Typed Racket (called "occurrence typing" [4]) and TypeScript ("narrowing" [5]). For instance, Kotlin accepts the following program by feeding the type checker flow-sensitive information coming from a nullability analysis:

```
fun f(x: Int?): Int {
  if (x != null) {
    // Addition works because we know `x` is not null here!
    return x + 1
  }
  return 0
}
```

But Kotlin's smart casts go far beyond such simple flow typing. It can also use boolean variables as witnesses for nullability facts, rewrite program's control flow graph (CFG) depending on user provided function contracts, and more. Unfortunately in practice these analyses quickly get hairy with the introduction of things such as mutability, aliasing, or exceptions. Without a formal understanding of the problem, the analyses can contain subtle unsoundness issues, leading to incorrect conclusions about program's behavior. Since Kotlin relies on these analyses to prevent a class of runtime errors, we want to be very sure they are indeed correct. Additionally, a formal verification serves as a tool for surfacing these subtle edge cases.

While verifying static analyzers is not a new concept, existing tools are ill-suited for our needs. Verasco [6] is a Rocq-mechanized static analyzer for CompCert's [7] subset of C. It establishes the absence of runtime errors by means of many dataflow analyses. It is however specific to that single language and therefore not directly usable for Kotlin. On the other hand [8] does provide a generic formalization of abstract interpreters based on interaction trees. Unfortunately these constructions are highly complex requiring the user to understand monadic handling, flow combinators, and more. We have found that existing solutions are either focused on a specific language or too complex requiring the user to prove many obligations.

Motivated by our need of creating a playground for verifying Kotlin's dataflow analyses, we present `Chartreux`: a language-agnostic framework for specifying and verifying CFG-based analyses in Lean. We instantiate it on analyses drawn directly from the Kotlin compiler by which we establish their soundness and derive proper guarantees for language constructs that rely on them.

By providing a low-obligation framework for CFG-based analyses, we help the Kotlin language team by verifying the implementation of analyses currently used in production, as well as aid and accelerate the development of novel analyses. By being language agnostic we allow for experiments on smaller toy languages.

Our main contributions are the following:

(i) a framework for describing languages and analyses in Lean, with guarantees of correctness of said analyses;
(ii) mechanized soundness proofs for two analyses that mirror Kotlin language constructs, namely nullability with consequents (modelling Kotlin's flow-sensitive typing with implications) and calls-in-place lambda contracts.

Section 2 dives deep into relevant Kotlin language features which are used for smart-casts. Section 3 presents the framework's design and implementation, as well as showing its theoretical guarantees. In Section 4 we validate the framework by instantiating it on analyses drawn from the Kotlin compiler, before commenting on related work and next steps in Section 5.

2 The Kotlin language

Kotlin [2] is a statically typed language created and developed at JetBrains. It is designed as a modern language for the Java Virtual Machine, incorporating both the object oriented and functional paradigm. Kotlin powers its most unique features by the use of dataflow analysis in the type checking phase, yielding a flow-sensitive typing that allows writing more expressive programs while maintaining safety.

Nullability analysis. Classically, in strongly typed languages, bindings keep their type throughout execution. This is not the case in Kotlin: a binding can be *refined* to an instance of a more precise type, if the control flow around it provides enough information to do so, for example in determining the legality of accesses to potentially nullable values. In particular, this goes beyond simple `if (v != null)` checks: the analysis builds a system of implications that tie boolean values with the names that are impacted, in order to determine the legality of every operation.

```
fun calculateTax(cost: Double?): Double? {
  val valid = cost != null && /*1*/cost > 0.0
  /*2*/

  Telemetry.log("calculateTax_valid", valid)

  if (!valid) return null

  /*3*/
  return cost * 0.23
}
```

Listing 1. Kotlin example showcasing different features of nullability analysis.

Consider Listing 1, which the Kotlin compiler automatically accepts. The function receives a cost which can potentially be `null`. In `valid` we store whether the cost is not null and strictly positive. We log `valid` and exit early if it is `false`. Finally, we compute the tax if everything went well. There are a few interesting program points which we now elaborate on:

1. Since `&&` is short-circuiting, this point is reached only if `cost != null` succeeded. This means Kotlin can refine `cost` to be non-null and allow for the `>` comparison.
2. Kotlin creates two dataflow implications. If `valid` is `true` it means that `cost != null && cost > 0.0` holds. Hence, `cost` is not `null`. On the other hand, if it is `false` we do not gain any information about the nullability of `cost`.
3. Since we have previously exited early on the condition of `!valid` it means that here `valid` is `true`. Using the implication from the previous point Kotlin learns that `cost` cannot be `null`, allowing us to perform the multiplication!

This system allows the user to write more complex programs while still maintaining compiler-proven `null` safety. However, under conditions of mutability or potential aliasing, the analysis can be rendered unsound, by the introduction of slight mismatches between the transfer functions and the language semantics: the correctness of this witnessing behavior is not immediate and unique.

Function contracts. Another use of dataflow analysis in the Kotlin compilation process is *function contract checking*. In Kotlin, functions are first-class, which means they can be passed to other functions as parameters and returned like regular values. These lambda parameters allow the user to write code following a more

functional style, but the interaction with the pre-existing dataflow system is not intuitive, and leads to many corner cases.

Consider the following program, where `run` is a standard library function that takes a single function as parameter.

```
fun main() {
  val x: Int
  run({ x = 42 })
  println(x)
}
```

In Kotlin, `{ x = 42 }` is the syntax for a parameterless (closure) lambda with the body of `x = 42`. A `val` in Kotlin denotes an immutable binding that can be assigned to only once and cannot be used before being initialized to some value.

Without any guarantees on the behavior of `run`, we hit two compilation errors:

1. This lambda cannot be passed because `run` might potentially call it multiple times, causing the variable `x` to be initialized more than once;
2. The call to `println` fails, because `run` might potentially never call the lambda and so `x` is never initialized.

Without specifying *how* `run` utilizes its lambda parameter, the call behaves as an opaque system, and the initialization analysis has nothing to latch onto to gain information: the code cannot compile. However, now consider the following specification for `run`.

```
fun run(f: () -> Unit): Unit {
  contract {
    callsInPlace(f, InvocationKind.EXACTLY_ONCE)
  }
  f()
}
```

The contract becomes part of the signature of `run` itself. Here, it specifies that `run` calls `f` exactly once before returning, which gives the analysis enough information to pass: we now have a guarantee that `f` initializes `x` exactly once.

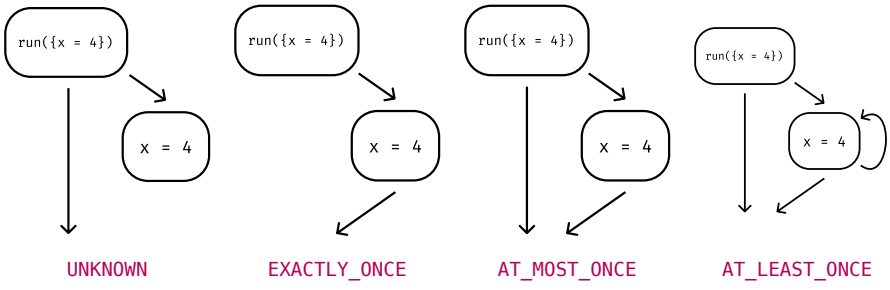


Fig. 1. Sketch of the Kotlin CFG for lambda parameters in the presence of contracts.

Contracts exist for lambda parameters called `EXACTLY_ONCE`, `AT_LEAST_ONCE` and `AT_MOST_ONCE`, as well as a default `UNKNOWN` value that falls back to the default behavior. In practice, the analysis phase implements these checks in two different passes, working together to compile the necessary information:

1. a simple dataflow analysis on function bodies checks that they respect the contract declared in their signature;
2. at call site, the CFG is rewritten according to rules exemplified in Fig. 1: edges are added based on the contract, propagating the side effects of the call to the local analysis.

These rewrites increase the precision of the analysis by inlining the lambda body with edges that congruous with the function's contract. These new edges make intuitive sense with the language's semantics, but they are far from trivial: it's not clear on first glance why the result reached after considering them would be sound. Additionally, the rewrites rely on the correctness of the counting analysis, making the machinery around them brittle. Function contracts have been an experimental Kotlin feature since 2018, with their design being the subject of large amounts of scrutiny and discussion.

While type systems have systematic ways to study their interaction with semantics, these dataflow techniques, in the formulation they usually have in production compilers, have not enjoyed the same level of study. Our framework reduces this gap by obtaining proper, workable guarantees and by allowing for specifying these analyses and relating them back to program executions.

Let us now present our framework.

3 Design and implementation

Before we can return to Kotlin and the analyses that we formalized, we describe the framework in Lean alongside the obligations that are put on the user. Once the language designer fulfills them, they obtain a theorem on the correctness of their analysis, a relation between the abstract state computed by the algorithm and the concrete state arising from execution. We call this relation "coherence", and denote it by $\mathsf{Coh} \subseteq L \times S$, where L, S are the abstract and concrete state spaces respectively. It serves as a stuttering simulation invariant between the abstract and the concrete. As we will see later, this coherence relation is chosen by the user depending on her goals, this allows full flexibility in the theorems one wants to prove, while maintaining the proof burden to a minimal amount.

Classical abstract interpretation methods would establish a connection between abstract and concrete states through a Galois connection: given spaces L, S, one would have to determine functions $\alpha : S \to L, \gamma : L \to S$ to obtain preorders on both, which encode precisely the information that the analysis should provide. While some presentations make this more suitable for formal methods work (like Pichardie's γ-only formulation [9]), finding a good α, γ pair requires an uncomfortable amount of backwards thinking about the guarantees that one wishes to achieve, encoding a relation between two spaces S and L as a pair of two projections instead of phrasing it directly. Coh works exactly as that direct relation, and the framework is built around making it easy for the user to work from this relation towards a full proof of correctness.

In the rest of this section we explain the most important components of the mechanized framework in Lean, while highlighting the obligations put on a user. We conclude it with a proof of Theorem 3, certifying the soundness of an analysis.

3.1 CFGs

A CFG is represented as a list of abstract nodes, and a list of edges between them. We require an API to extract source and destination nodes from an edge (`srcOf` and

`dstOf` respectively), as well as well-formedness conditions on both. We additionally distinguish a special `entry` node to start the analysis from.

```
class AnalysisCFG (Node Edge : Type)
    [DecidableEq Node] [DecidableEq Edge] where
  nodes : List Node
  edges : List Edge
  entry : Node
  srcOf : Edge -> Node
  dstOf : Edge -> Node
  entry_mem : entry ∈ nodes
  srcOf_mem :
    ∀ e ∈ edges, srcOf e ∈ nodes
  dstOf_mem :
    ∀ e ∈ edges, dstOf e ∈ nodes
```

The well-formedness conditions could instead be enforced with the use of a dependent type `NodeOf g` carrying its membership proof, but that proved to be more burdensome for the user, and the more explicit formulation was maintained. The `Node` and `Edge` types are deliberately abstract to allow full flexibility in the end user's choice of model.

3.2 Finite height bounded semilattices

The properties of finite-height, bounded semilattices lie at the core of monotone frameworks, and most of the properties we provide, like termination and soundness, follow directly from them. Analyses in our framework are stated over join-semilattices (isomorphic to their meet counterpart), so we describe them in detail.

We say that a type is of *finite height* with the following typeclass:

```
class FiniteHeight (L : Type) [Max L] where
  remainingHeight : L -> Nat
  height_join : ∀ a b, a ⊔ b ≠ a -> remainingHeight (a ⊔ b) <
remainingHeight a
```

We require every chain in L to have a decreasing bounded measure, meaning that any join between two elements strictly decreases the measure of its operands, if it is not equal to them. The finite-heightness requirement on lattices is required to guarantee termination. While all of the analyses we consider are defined over lattices of finite height, other techniques to establish termination exist, such as widening; we will not consider them here as these are not used in Kotlin.

With this definition, we can define semilattices. Consider a join operation $\cdot \sqcup \cdot : L \times L \to L$ over elements of type L. This operation induces an ordering. For all $a, b \in L$, $a \sqsubseteq b \equiv a \sqcup b = b$.

We say that a lattice is *bounded* if there exists a distinguished element $\bot : L$ such that $\forall a \in L, \bot \sqsubseteq a$.

With these definitions in hand, modelling the **F**inite **H**eight **B**ounded **S**emi**lattice** is relatively straightforward:

```
class FHBSLattice (L : Type) [Max L] [Bot L] [FiniteHeight L] where
  join_comm : ∀ a b : L, a ⊔ b = b ⊔ a
  join_assoc : ∀ a b c : L, (a ⊔ b) ⊔ c = a ⊔ (b ⊔ c)
  join_idem : ∀ a : L, a ⊔ a = a
  bot_le : ∀ a : L, ⊥ ⊑ a
```

Generic `FHBSLattices`. There are a few very common shapes of lattices that appear when designing analyses. We provide templates for those and prove that they too are `FHBSLattice`:

- **The product lattice** – given two `FHBSLattices` L_1 and L_2, $L_1 \times L_2$ is a `FHBSLattice` with $(a_1, a_2) \sqcup (b_1, b_2) = (a_1 \sqcup_1 b_1, a_2 \sqcup_2 b_2)$.
- **The boolean lattice** – $\{\bot, \top\}$ is a lattice with $a \sqcup b = \begin{cases} \bot \text{ if } a{=}b{=}\bot \\ \top \text{ otherwise} \end{cases}$.
- **The domain lattice** – given a `FHBSLattice` L_1 and $n \in \mathbb{N}$, the map Fin $n \to L_1$ is a `FHBSLattice` with $m_1 \sqcup m_2 = \lambda i.\ m_1(i) \sqcup_1 m_2(i)$. Here, Fin n represents a finite type of size exactly n.
- **The powerset lattice** – given an $n \in \mathbb{N}$, the powerset lattice is represented using the domain lattice with n and the boolean lattice. This encodes the lattice of characteristic functions of a set. One can see that this induces the join to be set union and the order to be the subset relation.

We end up making extensive use of the domain lattice during our formalization, as the basis of the representation of a finite number of variables in a program.

3.3 Language semantics and dataflow analysis

Our framework is only as useful as it allows us to relate abstract structures to real-world constructs. We do so by allowing the user to define their language semantics on the CFG they provide:

```
class LangSem (Edge Node State : Type) where
  LStep : Edge -> State -> State -> Prop
  LStutter : Node -> State -> State -> Prop
  IsInit : State -> Prop
```

The framework requires a stepping relation on the edges of the CFG, to relate input and output states. Since the abstract and concrete semantics might not always be completely synchronized, we allow the user to provide an `LStutter` condition for a given node, to allow the concrete semantics to take a step that has no corresponding CFG transition. Finally, we ask of the user to define a predicate characterizing all possible initial states. On these semantics we define a reachability condition over states, as the reflexive transitive closure of `LangSem.LStep` and `LangSem.LStutter` starting from any state such that `LangSem.IsInit`.

The desired properties of `LangSem` are expressed in terms of a dataflow analysis (DFA). To specify an analysis it is enough to provide the transfer functions and the starting lattice value.

```
structure DFA (Node Edge : Type) where
  L : Type
  nodeTransfer : Node -> L -> L
  edgeTransfer : Edge -> L -> L
  entry        : L
```

We additionally define `transferAlong`, which composes the edge and node transfer functions, computing the in-state of the destination node from the in-state of a source node.

```
def DFA.transferAlong (A : DFA Node Edge) (e : EdgeOf g) (ℓ : A.L) : A.L :=
  A.edgeTransfer e (A.nodeTransfer (g.srcOf e) ℓ)
```

With this, we are ready to state the required properties of the dataflow analysis with regards to some language semantics.

```
structure DFASemantics (A : DFA Node Edge) where
  Coh : A.L -> State -> Prop
  preserve_entry :
    ∀ {σ : State}, LangSem.IsInit σ -> Coh A.entry σ
  preserve_step :
    ∀ {e : Edge} {σ σ' : State} {ℓ : A.L},
      LangSem.LStep e σ σ' -> Coh ℓ σ -> Coh (A.transferAlong g e ℓ) σ'
  preserve_stutter :
    ∀ {n : Node} {σ σ' : State} {ℓ : A.L},
      LangSem.LStutter g n σ σ' -> Coh ℓ σ -> Coh ℓ σ'
```

This is where the user finally defines their Coh relation, and where they are required to show that:

1. Coh relates the abstract entry value to any initial concrete state;
2. Coh is preserved when a related abstract transfer application and concrete step along the CFG take place;
3. Coh is preserved when the concrete semantics take a step on a node on which the semantics stutter.

These simple definitions are enough for users to define complex analyses, and establish non-trivial correctness relations for them. These obligations seem heavy, but they allow the framework to run the analysis algorithm and obtain guarantees on the result. Let us discuss what those guarantees are, and how they are established. w

3.4 Kildall's worklist algorithm

To drive our analyses, we implement and verify the worklist algorithm on `FHBSLattices` as described in [10]. Provided a CFG $g = (N, E)$ and a DFA with transfer functions f_n, f_e, the algorithm runs to refine $\rho : N \to L$ into its post-fixpoint:

$$\forall n \in N, \forall s \in \mathsf{succ}_G(n), \quad f_n(\rho(n)) \sqsubseteq \rho(s)$$

The algorithm works by maintaining a worklist l of nodes to process. We initialize ρ to ρ_0, mapping every node into $\bot$, and l to N

Let $(\rho, l) \rightsquigarrow (\rho', l')$ denote a single "step" (an iteration) of the algorithm. When processing a node $n \in l$, ρ' is computed by joining the results of the $f_{(n',n)}$ on all of its incoming edges, before applying the corresponding f_n and joining it with the current state $\rho(n)$. Mathematically, this yields:

$$\rho'(n) = \rho(n) \sqcup f_n \left(\bigsqcup_{n' \in \mathrm{pred}_G(n)} f_{(n',n)}(\rho(n')) \right)$$

If $\rho'(n)$ is different from $\rho(n)$, we append the successors of n to the worklist, since they were affected by the change. This process continues until the worklist is empty. This procedure might be simple, but it is enough to compute a post-fixpoint solution, and to guarantee termination on finite height domains. We sketch the proofs of those properties here:

Theorem 1. The worklist algorithm terminates.

Proof. We argue that the measure $(h, l) \in \mathbb{N} \times \mathbb{N}$ decreases with every iteration, where h is the sum of the remaining height of $\rho(n)$ for every $n \in N$, and l is the length of the worklist. Let $\rho(n), \rho'(n)$ be the abstract values at n before and after an iteration. Since $a \sqsubseteq a \sqcup b$, we know that $\rho(n) \sqsubseteq \rho'(n)$. We therefore have two

cases: if $\rho(n) = \rho'(n)$ there was no change, so l decreases. Otherwise, a better result was found, so h decreases.

This argument only holds because of the finite height of our lattice, which gives us a lower bound we can decrease towards. To show that the result of this algorithm is a post-fixpoint, we borrow a technique from [11] to establish invariants over the algorithm's runtime.

We say that a property P over an intermediate analysis result ρ and a worklist l is *inductive* if $P(\rho_0, l_0)$ and $P(\rho, l) \to ((\rho, l) \rightsquigarrow (\rho', l')) \to P(\rho', l')$ where $\rightsquigarrow$ denotes a single iteration of the algorithm as before. We denote the final result of the algorithm with $\overline{\rho}$. Then, we can show the following intermediate result:

Lemma 1. Let P be an inductive property. Then, $P(\overline{\rho}, [\,])$.
Proof. By induction on the iterations of the algorithm.

Using this result, the proof of soundness is relatively straightforward.

Theorem 2. For any $n \in N$, and any transfer function f_n, the result of the worklist algorithm $\overline{\rho}$ is a post-fixpoint.
Proof. It is easy to check that $P(\rho, l) =$ "ρ is a post-fixpoint for all $n \notin l$" is inductive. We apply Lemma 1 with P, concluding the proof.

Interestingly, this soundness result does not require the transfer functions in the analysis to be monotonic. Monotonic transfer functions allow us to show that $\overline{\rho}$ is a *least* post-fixpoint, a result that is mechanized in our framework but that goes unused in our case studies.

Using Theorem 2, we can finally state and prove the main theorem of the framework, Theorem 3.

Theorem 3. Let $g = (N, E)$ be a CFG with semantics relating states in S. Given an analysis A with transfer functions over a bounded semilattice L, and a relation $\mathsf{Coh} \subseteq L \times S$ that's preserved under semantics steps, the post-fixpoint $\rho : N \to L$ of the transfer functions from A on g is such that for all reachable states $\sigma : S$ at node $n \in N$, $\mathsf{Coh}(\rho(n), \sigma)$ holds.
Proof. By induction on the steps, using `DFASemantics`' initialization and preservation properties, together with the soundness of the Kildall algorithm from Theorem 2.

With the guarantees established, we move onto discussing our application of the framework, showing its full power in action.

4 Application

We implement our analyses on Duke, an untyped imperative language designed to be a simple subset of Kotlin. It contains the required language constructs for the nullability analysis.

Values Val are represented by potentially nullable integers. Values which are either null or the integer 0 are considered falsy. Strictly positive integers are considered truthy. We can further compose values into expressions with variables and simple operators. Statements consist of variable declarations, assignments, if- and while-statements; its entire syntax is defined in Fig. 2. By allowing values to be nullable we model Kotlin's approach to flow typing, including the use of *consequents* which we will present shortly.

$$
\begin{aligned}
\oplus &::= + \mid - \mid * \mid < \mid = \mid \wedge \\
e &::= \text{null} \mid \text{Int } n \mid \text{Var } x \mid \text{isnull } e \mid !e \mid e \oplus e \\
s &::= \text{var } x := e \mid x = e \mid \text{if } (e)\ \{s\}\ \text{else}\ \{s\} \\
&\quad \mid \text{while } (e)\ \{s\} \mid \text{skip} \mid s; s
\end{aligned}
$$

Fig. 2. Syntax of Duke

To define the concrete semantics of Duke, we must first present its CFG construction.

We represent nodes with simple identifiers NodeID, where edges are pairs of such NodeIDs. We use NodeKind to differentiate kinds of nodes in the CFG, corresponding to program constructs:

$$\mathsf{NodeKind} := x = e \mid \text{assume } e \mid \text{skip}$$

The assignment node simply binds the value of the expression e to the variable named x. It serves both declarations and mutations. The synthetic assume node is inserted after conditionals to indicate which branch was taken. The abstract interpreter is expected to assume that e holds when traversing an assume node. This information could be instead stored in edges, but having it in the nodes leads to a more convenient formalization. Finally, skip, is a node with no action attached to it.

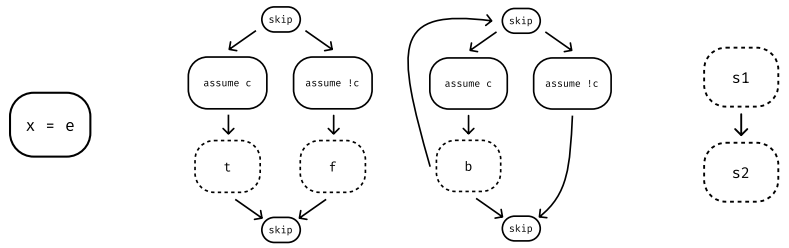


Fig. 3. Statement to CFG translation for Duke. Dotted nodes represent subgraphs.

In Fig. 3 we show the translation of Duke's statements into its CFG. A conditional over an expression c always create two edges from a skip node into nodes for assume c and assume $!c$.

To define the concrete semantics on CFGs we first present the big-step reduction for expressions into values in Fig. 4. We represent the environment map with σ : Var $\rightharpoonup$ Val. We denote by $\ulcorner\oplus\urcorner$ the interpretation of the binary operator $\oplus$. Notice that we can have stuck terms in three cases:

1. if either operand of $\oplus$ is null, or
2. if the operand of ! is null, or
3. if for Var x, x is not defined in σ.

With that we define the concrete semantics of statements directly on the CFG as seen in Fig. 5. The semantics are expressed as reductions on $(n, \sigma) \in N \times (\text{Var} \rightharpoonup \text{Val})$. They are designed to integrate pretty naturally with the existing features of the

`Chartreux` framework, all while remaining the fundamental ground-truth semantics for this language.

We again encounter reductions which can get stuck, either because an expression reduction failed or because an assume node contains a falsy expression. It will be precisely the role of the dataflow analysis to show that properly formed programs cannot get stuck during execution.

When integrating all of the above with `Chartreux`, we can think about what steps are "loud" and which are "stutter": since CFG reductions are the ground truth `LStep` semantics, we do not have to define anything as stutter as there is no correspondence to be drawn. The concrete state `State` simply becomes Var ⇀ Val.

$$\frac{}{\sigma \vdash \text{null} \Downarrow \text{null}}\,\text{null} \qquad \frac{}{\sigma \vdash \text{Int } n \Downarrow \text{Int } n}\,\text{int} \qquad \frac{\sigma(x) = v}{\sigma \vdash \text{Var } x \Downarrow v}\,\text{var}$$

$$\frac{\sigma \vdash e \Downarrow \text{null}}{\sigma \vdash \text{isnull } e \Downarrow \text{Int } 1}\,\text{isnullT} \qquad \frac{\sigma \vdash e \Downarrow v \quad v \neq \text{null}}{\sigma \vdash \text{isnull } e \Downarrow \text{Int } 0}\,\text{isnullF}$$

$$\frac{\sigma \vdash e \Downarrow \text{Int } n \quad n \neq 0}{\sigma \vdash !e \Downarrow \text{Int } 0}\,\text{notT} \qquad \frac{\sigma \vdash e \Downarrow \text{Int } 0}{\sigma \vdash !e \Downarrow \text{Int } 1}\,\text{notF}$$

$$\frac{\sigma \vdash e_1 \Downarrow \text{Int } n_1 \quad \sigma \vdash e_2 \Downarrow \text{Int } n_2 \quad v = n_1 \ulcorner\oplus\urcorner n_2}{\sigma \vdash e_1 \oplus e_2 \Downarrow v}\,\text{binop}$$

Fig. 4. Big-step semantics for Duke expressions

$$\frac{\text{kind}(n) = \text{skip} \quad n' \in \text{succ}(n)}{(n, \sigma) \to (n', \sigma)}\,\text{skip}$$

$$\frac{\text{kind}(n) = (x = e) \quad \sigma \vdash e \Downarrow v \quad n' \in \text{succ}(n)}{(n, \sigma) \to (n', \sigma[x \mapsto v])}\,\text{assign}$$

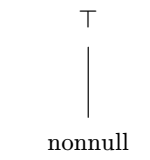
$$\frac{\text{kind}(n) = \text{assume } e \quad \sigma \vdash e \Downarrow \text{Int } m \quad m \neq 0 \quad n' \in \text{succ}(n)}{(n, \sigma) \to (n', \sigma)}\,\text{assume}$$

Fig. 5. Duke CFG Semantics. kind : NodeID → NodeKind identifies node IDs with their kinds.

4.1 Nullability with consequents

The point of nullability analysis is to statically determine problematic usages variables which have the value null. We now present the entire nullability analysis with boolean witnesses which implies the absence of stuck terms.

We first need a few lattices. First, AbsVal:



Which will abstract whether some variable is definitely not null (nonnull) or we don't know enough about it ($\top$). This is of finite height since it has a finite amount of elements. The join is defined as follows:

$$\begin{aligned} \text{nonnull} \sqcup a = a \sqcup \text{nonnull} &:= a \\ \top \sqcup \top &:= \top \end{aligned}$$

The bottom element of AbsVal is nonnull. The join (and the induced order) fulfills the requirements of being a finite height lattice.

Next, we use the domain lattice template to create $\mathsf{AbsConseq} : \mathrm{Var}^n \to \mathsf{AbsVal}$. Since AbsVal is a finite height lattice then so is AbsConseq. It stores the consequent of the implication that if a variable is truthy, then the assertions in AbsConseq hold. For example, in the program $x = !(\text{isnull } y) \wedge !(\text{isnull } z)$ not only do we know that x is nonnull, but also that it is a witness of y and z being nonnull. If x evaluates to a truthy value, we will know both y and z is not null. This makes our domain relational.

Finally, since we want to store nullability information for each variable separately, we wrap AbsVal and AbsConseq in a domain over all variables, $\mathsf{AbsFact} : \mathrm{Var}^n \to \mathsf{AbsVal} \times \mathsf{AbsConseq}$. Since both AbsVal and AbsConseq are finite height lattices, so is $\mathsf{AbsVal} \times \mathsf{AbsConseq}$. Similarly, so is the entire AbsFact.

To illustrate this lattice, consider these AbsFact instances expressing the knowledge we have at the individual points in Listing 1:

1. $\mathit{cost} \mapsto (\text{nonnull}, \{\mathit{cost} \mapsto \top, \mathit{valid} \mapsto \top\})$
   $\mathit{valid} \mapsto (\top, \{\mathit{cost} \mapsto \top, \mathit{valid} \mapsto \top\})$
2. $\mathit{cost} \mapsto (\top, \{\mathit{cost} \mapsto \top, \mathit{valid} \mapsto \top\})$
   $\mathit{valid} \mapsto (\text{nonnull}, \{\mathit{cost} \mapsto \text{nonnull}, \mathit{valid} \mapsto \top\})$
2. $\mathit{cost} \mapsto (\text{nonnull}, \{\mathit{cost} \mapsto \top, \mathit{valid} \mapsto \top\})$
   $\mathit{valid} \mapsto (\text{nonnull}, \{\mathit{cost} \mapsto \text{nonnull}, \mathit{valid} \mapsto \top\})$

To define the CFG transfer functions, we first need to define abstract evaluation of expressions:

$$\begin{aligned} \text{aeval} : \mathsf{AbsFact} \times \text{Expr} &\to \mathsf{AbsVal} \\ \text{aeval}(\rho, \text{null}) &:= \top \\ \text{aeval}(\rho, \text{Int } n) &:= \text{nonnull} \\ \text{aeval}(\rho, \text{Var } x) &:= \text{fst}(\rho(x)) \\ \text{aeval}(\rho, \text{isnull } e) &:= \text{nonnull} \\ \text{aeval}(\rho, !e) &:= \text{nonnull} \\ \text{aeval}(\rho, e_1 \oplus e_2) &:= \text{aeval}(\rho, e_1) \sqcup \text{aeval}(\rho, e_2) \end{aligned}$$

We additionally need a way to extract consequent information from an expression. We want to answer the question "given an expression e, what nullability information do we gain when assuming e is truthy?".

$$\text{exprConseq} : \mathsf{AbsFact} \times \text{Expr} \to \mathsf{AbsConseq}$$

$$\text{exprConseq}(\rho, \text{Var } x) \coloneqq \lambda y. \begin{cases} \text{nonnull} & \text{snd}(\rho(x))(y) = \text{nonnull} \\ \top & \text{otherwise} \end{cases}$$

$$\text{exprConseq}(\rho, \text{!isnull (Var } x)) \coloneqq \lambda y. \begin{cases} \text{nonnull} & x = y \\ \top & \text{otherwise} \end{cases}$$

$$\text{exprConseq}(\rho, e_1 \wedge e_2) \coloneqq \lambda y. \begin{cases} \text{nonnull} & \text{exprConseq}(\rho, e_1)(y) = \text{nonnull} \vee \text{exprConseq}(\rho, e_2)(y) = \text{nonnull} \\ \top & \text{otherwise} \end{cases}$$

$$\text{exprConseq}(\rho, e) \coloneqq \lambda\_.\top$$

When looking at a variable we consult the consequents that variable holds. If we assume !isnull (Var x) holds, it of course means that x is not null. In case of a conjunction we consider both sides holding true. Finally, for all other cases there is nothing we can say.

This allows us to define the node transfer function:

$$f : \mathsf{NodeKind} \to \mathsf{AbsFact} \to \mathsf{AbsFact}$$

$$f(x = e)(\rho) \coloneqq \lambda y. \begin{cases} (\text{aeval}(\rho, e), \text{exprConseq}(\rho, e)) & x = y \\ (\text{fst}(\rho(y)), \lambda\_.\top) & \text{otherwise} \end{cases}$$

$$f(\text{assume } e)(\rho) \coloneqq \lambda y. \begin{cases} (\text{nonnull}, \text{snd}(\rho(y))) & \text{exprConseq}(\rho, e)(y) = \text{nonnull} \\ \rho(y) & \text{otherwise} \end{cases}$$

$$f(\text{skip})(\rho) \coloneqq \rho$$

Where fst and snd are standard projection functions to the first and second element respectively. In the case of assignment $x = e$, we assign the AbsVal to x from the abstract evaluation of e and we extract the consequents coming from e. For all other variables in the program we keep their AbsVal but invalidate all consequents since they might have been invalidated due to the mutation of x. The transition for assume e extracts the consequents from e and immediately assumes them to hold.

Correctness. For an abstract state $\rho : \mathsf{AbsFact}$ and a concrete state $\sigma : \text{State}$, we define the following coherence relation:

$$\begin{aligned} \mathsf{Coh}(\rho, \sigma) = \forall x \in \text{dom}(\rho).\ & [\text{fst}(\rho(x)) = \text{nonnull} \to \sigma(x) \neq \text{null}] \wedge \\ & [\forall y \in \text{dom}(\rho).\ \text{snd}(\rho(x))(y) = \text{nonnull} \wedge \sigma(x) = \text{Int } n \wedge n \neq 0 \to \sigma(y) \neq \text{null}] \end{aligned}$$

The first half of the coherence is the correctness property of the analysis. The second half of the correspondence gives a denotation for the consequents, needed for proving step preservation. Then, once the framework obligations are fulfilled, we combine the nullability analysis with a standard initialization analysis, leading to the proof of the safety theorem of Duke:

Theorem 4. Let g be a program such that successful initialization and nullability analyses have been carried out. Let (n, σ) be a reachable execution configuration when running g. Then, either:

- n is the exit node of g;
- n is an assume node whose condition is false;
- there exists a node n' and a state σ' such that $(n, \sigma) \to (n', \sigma')$.

At exit nodes we naturally cannot take a step, and since an assume c node is always paired with assume !c, one of them must be necessarily false.

4.2 Function contract checking

The second problem concerns the CFG rewrites induced by calls-in-place contracts. At a call site, the compiler analyzes the body of a lambda as though it occurred directly in the caller. The edges around this copy encode whether the lambda may be skipped or repeated. This makes information produced inside the lambda available to the caller's dataflow analysis, but introduces a proof obligation: executions that enter another function and invoke a closure must be represented by paths through the local rewritten graph.

We study this problem in *FDuke*, an extension of Duke with function calls and lambda invocation. Its statements have two additional forms:

$$s ::= \ldots \mid \text{call } f\,\{s\} \mid \text{invoke}$$

The statement call $f\,\{s\}$ passes s as the closure argument of f, while invoke executes the current function's closure argument. Lambdas have no parameters of their own, but execute in the store captured at the call site. A program consists of a main statement and a finite function environment $\Phi : \mathrm{Id} \rightharpoonup (\kappa \times \mathrm{Stmt})$. The invocation kind $\kappa \in \{\varepsilon, 1, +, ?\}$ denotes, respectively, an unknown number of invocations, exactly one, at least one, or at most one.

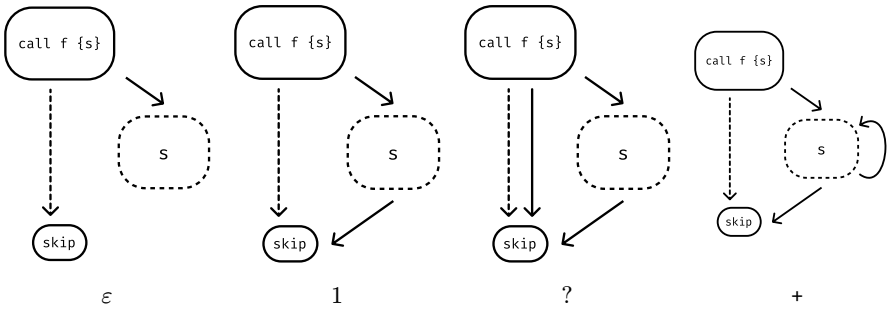


Fig. 6. Contract-dependent CFG fragments in FDuke. The lambda body $\{s\}$ is copied into the caller for analysis; the dotted arrow depicts the CEK machine's call/return control transfer and is not an edge of the analysis graph.

For a call node with lambda entry en_L, lambda exit ex_L, and return node r, the lowering emits the fragments in Fig. 6. The resulting paths execute the inlined body any number of times admitted by the contract:

$$\begin{aligned} \mathrm{admits}(\varepsilon, k) &:= \top \\ \mathrm{admits}(1, k) &:= (k = 1) \\ \mathrm{admits}(+, k) &:= (1 \le k) \\ \mathrm{admits}(?, k) &:= (k \le 1). \end{aligned}$$

The lowering also records some metadata for the callsite: the call node, callee, lambda identifier, invocation kind, inline entry and exit, return node, and source lambda body. The dotted arrow in Fig. 6 represents how the concrete CEK machine uses this metadata to enter the callee and eventually return to r; it is not an additional CFG edge.

$$\frac{\mathrm{kind}(n) = \text{call } f\,\{\ell\} \qquad r \in \mathrm{succ}_{\mathrm{summary}}(n)}{\langle n, \sigma, \rho, K\rangle \to \langle \mathrm{en}_f, \cdot, \mathrm{some}(C), \text{call } r\,\{\rho\} :: K\rangle}\ \text{call}$$

$$\frac{\mathrm{kind}(n) = \text{invoke} \qquad \rho = \mathrm{some}(\langle \mathrm{en}_L, \mathrm{ex}_L, \rho_d, \sigma_d\rangle) \qquad r \in \mathrm{succ}(n)}{\langle n, \sigma, \rho, K\rangle \to \langle \mathrm{en}_L, \sigma_d, \rho_d, \mathrm{inv}(r, \rho, \sigma) :: K\rangle}\ \text{invoke}$$

$$\frac{\mathrm{succ}(n) = \varnothing}{\langle n, \sigma, \mathrm{some}(\langle \mathrm{en}_L, \mathrm{ex}_L, \rho_c, \sigma_c\rangle), \text{call } r\,\{\rho'\} :: K\rangle \to \langle r, \sigma_c, \rho_c, K\rangle}\ \text{return-call}$$

$$\frac{\mathrm{succ}(n) = \varnothing}{\langle n, \sigma, \rho', \mathrm{inv}\big(r, \mathrm{some}(\langle \mathrm{en}_L, \mathrm{ex}_L, \rho_d, \sigma_d\rangle), \sigma_f\big) :: K\rangle \to \langle r, \sigma_f, \mathrm{some}(\langle \mathrm{en}_L, \mathrm{ex}_L, \rho', \sigma\rangle), K\rangle}\ \text{return-invoke}$$

Fig. 7. Function and invocation semantics

Concrete semantics. Lowering an FDuke program produces a family of CFGs: one graph for `main`, one for each function body, and one for every lambda literal. The concrete semantics is a CEK machine over this family. Its configurations have the form $\langle n, \sigma, \rho, K\rangle$, where n identifies both a graph and a node, σ is the current store, ρ is an optional closure, and K is a stack of continuations. A closure records a lambda's entry and exit nodes, its captured closure, and its captured store.

The relevant rules are shown in Fig. 7. Calling f saves the lambda and the caller's store in a closure, starts the function body with an empty store, and pushes the return point. An invoke starts the saved lambda in its captured store and pushes the function's current store. Returning from the lambda updates the closure with the lambda's resulting store and resumes the function; returning from the function restores this store in the caller. These rules give calls their ground-truth behavior independently of the CFG rewrite.

Checking contracts. Before relying on a contract, we verify it with a counting analysis. Its domain is $\mathsf{Cnt} = \mathcal{P}(\{0, 1, \omega\})$, where ω represents two or more invocations. It is a finite-height powerset lattice ordered by inclusion, with union as join and the empty set as $\bot$. The entry value is $\{0\}$. At an invoke node, every possible count is incremented according to $0 + 1 = 1$, $1 + 1 = \omega$, and $\omega + 1 = \omega$; all other transfer functions are the identity.

The concrete semantics used for this analysis pairs the store with a ghost natural-number counter. Its coherence relation states that the abstraction of the concrete counter belongs to the current element of Cnt. Incrementing the counter and applying the transfer function preserve coherence, so Theorem 3 establishes that every concrete invocation count reaching a node is included in the computed result.

Let $\overline{\rho}_f$ be a post-fixpoint obtained for the body of f, and let n_f be its exit node. The declared contract is accepted when:

$$\begin{aligned} 1 &\leftrightarrow \overline{\rho}_f(n_f) = \{1\} \\ + &\leftrightarrow 0 \notin \overline{\rho}_f(n_f) \\ ? &\leftrightarrow \omega \notin \overline{\rho}_f(n_f). \end{aligned}$$

No restriction is imposed for ε. We write $\mathsf{FamilyChecked}(p)$ when every function CFG in p's generated family has such a post-fixpoint satisfying its declared contract. Checkedness is required for the whole family, since a function may call another function whose invocation of its own lambda affects the first function's count.

Correctness of the rewrite. The proof relates the interprocedural CEK semantics directly to the structural analysis graph G_κ. A step at an ordinary node has its usual store effect, while a step leaving a call node is store-preserving. The effect of a complete call is therefore represented by traversing the copied lambda body zero or more times, as allowed by its contract fragment.

This direct account depends on certified lowering. For every call, the builder records a *call gadget* containing the call and return nodes, inline-body endpoints, lambda identifier, invocation kind, and source body. It proves that all edges prescribed by that invocation kind are present. A certificate-bearing family additionally connects the inline copy to the separately generated lambda component, including the node offset, shifted lambda identifiers, edges, and nested call gadgets. Erasing the proofs recovers the execution-facing CFG family and preserves both component and gadget lookup. Consequently the projection never infers provenance from an arbitrary graph after the fact.

Theorem 5. Let p be an FDuke program such that $\mathsf{FamilyChecked}(p)$. Every CEK execution that starts and ends in the same component with an empty continuation and starts without a closure is represented by a sequence of store steps in that component's structural graph G_κ, with the same initial and final stores.

The Lean theorem `balanced_project` proves a stronger statement by strong induction on the length of a *balanced* CEK run: throughout the run, the continuation has a fixed stack as suffix, and the endpoints have exactly that stack. Its conclusion simultaneously retains:

1. a bounded chain describing every completed invocation of the ambient, evolving closure, including each lambda-body CEK run;
2. the exact number q of links in that chain;
3. a run of a ghost-counting semantics from $(\sigma, 0)$ to (σ', q) on G_κ and
4. the required store-only run from σ to σ' on G_κ.

The bounds make every recursively projected lambda or callee body strictly shorter than the enclosing run. Closure and continuation well-formedness keep all referenced components resolvable, while a fixed pair of lambda endpoints ensures that an evolving closure still denotes the same body. This strengthening is what makes nested calls and invocations compositional. Two projected segments compose by concatenating their closure chains and store runs; the second ghost-counted run is shifted by the first segment's final count.

Assignment, assumption, and skip steps prepend the corresponding graph edge and use the induction hypothesis on the tail. A return that would pop below the fixed continuation suffix is impossible. The two interprocedural cases require explicit CEK-run decomposition:

1. For invoke, the run is split at the matching return from the lambda. The completed lambda body becomes one link in the ambient closure chain, its captured environment and store are updated, and the strictly shorter caller tail is projected recursively. The invoke node increments the ghost counter once but is a store-identity step in the store-only projection.
2. For `call`, the run is split at the matching return from the callee into the completed callee body and the caller tail. Projecting the callee records the exact

number k of times it invoked its callback. Each recorded standalone lambda run is recursively projected and shifted into the certified inline copy at the caller's call site, including nested gadgets. Counting soundness puts the abstraction of k in the callee's exit result; family checkedness supplies its verdict, from which admits(κ, k) follows. The certified gadget is then replayed: $k = 0$ uses the bypass, $k > 0$ uses the entry and exit edges, and all iterations after the first use the back-edge. Finally this call path is composed with the recursively projected caller tail.

Specializing the strengthened theorem to an empty continuation and no initial closure yields Theorem 5 (the Lean theorem `cek_projection`). The last correctness layer is deliberately independent of programs and contracts. The store-step semantics used by the projection and the semantics expected by an arbitrary analysis have the same structural graph and loud steps, so reachability transports edge by edge. Applying the generic post-fixpoint soundness theorem Theorem 3 then gives the public Lean theorem `kappa_rewrite_correct`.

Theorem 6. Let p be an FDuke program with a checked function family, and let g be one of its CFG components. Suppose an analysis A over G_κ has a coherence relation preserved by its transfer functions, and let ρ be a post-fixpoint whose entry contains the analysis entry value. For every state σ reached at node n by a balanced CEK execution of p from the entry of g, $\mathsf{Coh}(\rho(n), \sigma)$ holds.

The theorem is independent of the particular dataflow domain. In particular, the initialization and nullability analyses used for Duke also run on the rewritten FDuke graph, with the same correctness guarantees as in the environment without function calls.

5 Discussion

5.1 Future work

This tool is under very active development. The end-to-end function-contract derivation presented here is mechanized, while `Chartreux` as a framework has multiple paths for improvement. The integration of a WTO-based solver [11] could provide better efficiency and accuracy of computed fixpoints. Additionally, it would be beneficial if the framework provided primitives for the handling of function calls, given the current burden on the users. Lastly, the framework could provide alternative formalization paths like allowing for non-finite height lattices with a widening operator, or letting the user define correctness by means of a Galois connection whenever more convenient.

On the other hand, a more complete formalization of Kotlin's analyses requires extending the verified function contract model with more realistic language constructs. In a real-world compiler, aliasing, exceptions, and other runtime dangers are real sources of hidden unsoundness, and our framework can be very useful in the study of such pitfalls. Additionally, formalizing part of Kotlin, and showing correspondence between regular small-step semantics and the CFG execution will increase our confidence that the CFG semantics are fully representative.

5.2 Conclusion

We describe the design and implementation of a foundational, language-agnostic, formally verified framework for exploration, specification, and verification of dataflow analyses that *properly capture the semantics of the enclosed language*. The framework is minimal and low obligation. We apply it to modern analyses to obtain

formal guarantees and increased understanding of the Kotlin language. We trace a clear path of future work to graduate this project from a proof of concept to a versatile, extensible and easy-to-use tool, as a companion to aid the tasks of analysis engineers.

Acknowledgements We deeply thank Kameliya Golova for her mentoring throughout this work and for providing invaluable feedback on the paper.

Artifact availability The framework together with Duke/FDuke and their analyses are available at https://github.com/korrektness/chartreux.

Bibliography

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. Association for Computing Machinery, Los Angeles, California (1977). https://doi.org/10.1145/512950.512973.
2. JetBrains: Kotlin Programming Language, (2011).
3. JetBrains s.r.o.: Type checks and casts | Kotlin Documentation, https://kotlinlang.org/docs/typecasts.html#smart-casts, (2026).
4. The Racket Team: The Typed Racket Guide, https://docs.racket-lang.org/ts-guide/occurrence-typing.html, (2026).
5. Microsoft Corporation: TypeScript: Documentation - Narrowing, https://www.typescriptlang.org/docs/handbook/2/narrowing.html, (2026).
6. Jourdan, J.-H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A Formally-Verified C Static Analyzer. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 247–259. Association for Computing Machinery, Mumbai, India (2015). https://doi.org/10.1145/2676726.2676966.
7. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert – A Formally Verified Optimizing Compiler. In: ERTS 2016: Embedded Real Time Software and Systems. SEE (2016).
8. Michelland, S., Zakowski, Y., Gonnord, L.: Abstract Interpreters: A Monadic Approach to Modular Verification. Proc. ACM Program. Lang. 8, (2024). https://doi.org/10.1145/3674646.
9. Pichardie, D.: Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés, http://www.theses.fr/2005REN1S183, (2005).
10. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 194–206. Association for Computing Machinery, Boston, Massachusetts (1973). https://doi.org/10.1145/512927.512945.
11. La Spina, R., Demange, D., Blazy, S.: Formal Verification of WTO-based Dataflow Solvers. In: Programming Languages and Systems. pp. 1–27. , Hamilton, Canada (2025).